C++ ライブラリ・リファレンス

Sun WorkShop 6

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

Part No. 806-4840-01
2000 年 6 月 Revision A

原典： *C++ Library Reference Manual*
Part No: 806-3569-10
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

製品名の変更について iii

はじめに xi

1. C++ **ライブラリについて** 1

マニュアルページ 1

その他のライブラリ 2

Tools.h++ **ライブラリ** 3

Sun WorkShop Memory Monitor 3

2. **複素数ライブラリ** 5

複素数ライブラリ 5

複素数ライブラリの使用方法 6

`complex` **型** 6

`complex` **クラスのコンストラクタ** 6

算術演算子 7

数学関数 8

エラー処理 10

入出力 11

混合算術演算 12

効率 13

複素数ライブラリに関するマニュアルページ 14

3. 従来型の iostream ライブラリ 15

定義済みの iostream 16

iostream 操作の基本構造 16

従来型の iostream ライブラリの使用 17

iostream を使用した出力 18

iostream を使用した入力 22

ユーザー定義の抽出演算子 23

char* の抽出子 24

1 文字の読み込み 24

バイナリ入力 25

入力データの先読み 25

空白の抽出 25

入力エラーの処理 26

iostream と stdio の併用 26

iostream の作成 27

クラス fstream を使用したファイル操作 27

iostream の代入 31

フォーマットの制御 32

マニピュレータ 32

引数なしのマニピュレータの使用法 34

引数付きのマニピュレータの使用法 35

strstream： 配列用の iostream 37

stdiobuf: 標準入出力ファイル 用の iostream 37

streambuf 37

streambuf の機能 38

streambuf の使用 38
iostream に関するマニュアルページ 39
iostream の用語 42

4. マルチスレッド環境での従来型の iostream ライブラリの使用 45

マルチスレッド 45
「MT-安全」の iostream ライブラリの構成 46
公開変換ルーチン 48
「MT-安全」の libC ライブラリのコンパイルとリンク 49
「MT-安全」の iostream ライブラリの制限 49
パフォーマンス 51
iostream ライブラリのインタフェースの変更 53
新しいクラス 53
新しいクラス階層 53
新しい関数 54
大域データと静的データ 56
順次実行 57
オブジェクトのロック 57
stream_locker クラス 58
「MT-安全」のクラス 59
オブジェクトの破棄 60
アプリケーションの 61

5. C++ 標準ライブラリ 65

C++ 標準ライブラリのヘッダーファイル 66
5.2 C++ 標準ライブラリのマニュアルページ 68

索引 85

表目次

表 1-1 Sun WorkShop Memory Monitor のマニュアルページ 4

表 2-1 複素数ライブラリの関数 8

表 2-2 複素数の数学関数と三角関数 9

表 2-3 複素数ライブラリ関数 11

表 2-4 complex 型に関するマニュアルページ 14

表 3-1 iostream ルーチンのヘッダーファイル 17

表 3-2 iostream の定義済みマニピュレータ 33

表 3-3 iostream に関するマニュアルページの概要 40

表 3-4 iostream の用語 42

表 4-1 コアクラス 46

表 4-2 再入可能な公開関数 48

表 5-1 C++ 標準ライブラリのヘッダーファイル 66

表 5-2 C++ 標準ライブラリ用のマニュアルページ 68

はじめに

このマニュアルでは、以下を含む C++ ライブラリについて説明します。

- Tools.h++ クラスライブラリ
- Sun WorkShop Memory Monitor
- 複素数ライブラリ

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「C++ ライブラリについて」では、C++ ライブラリの概要を説明しています。

第 2 章「複素数ライブラリ」では、複素数ライブラリの算術演算子と算術関数について説明しています。

第 3 章「従来型の iostream ライブラリ」では、C++ で使用される従来型の入出力機能について説明しています。

第 4 章「マルチスレッド環境での従来型の iostream ライブラリの使用」では、マルチスレッド環境で `iostream` ライブラリを使用して入出力処理を行う方法について説明しています。

第 5 章「C++ 標準ライブラリ」では、標準ライブラリの概要を簡単に示します。

書体と記号について

このマニュアルで使用している書体と記号について説明します。

表 P-1　このマニュアルで使用している書体と記号

書体または記号	意味	例
`AaBbCc123`	コマンド名、ファイル名、ディレクトリ名、画面上のコンピュータ出力、コーディング例。	`.login` ファイルを編集します。 `ls -a` を使用してすべてのファイルを表示します。 `machine_name% You have mail.`
`AaBbCc123`	ユーザーが入力する文字を、画面上のコンピュータ出力と区別して表わします。	`machine_name% su` `Password:`
AaBbCc123 または **ゴシック**	コマンド行の可変部分。実際の名前または実際の値と置き換えてください。	`rm` *filename* と入力します。 `rm` **ファイル名** と入力します。
『 』	参照する書名を示します。	『SPARCstorage Array ユーザーマニュアル』
「 」	参照する章、節、または、強調する語を示します。	第 6 章「データの管理」を参照してください。 この操作ができるのは、「スーパーユーザー」だけです。
\	枠で囲まれたコード例で、テキストがページ行幅を超える場合、バックスラッシュは、継続を示します。	`machinename% grep '^#define \` `  XV_VERSION_STRING'`
➤	階層メニューのサブメニューを選択することを示します。	作成： 「返信」➤「送信者へ」

シェルプロンプトについて

シェルプロンプトの例を以下に示します。

表 P-2　シェルプロンプト

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

関連マニュアル

以下の方法で、関連マニュアルにアクセスすることができます。

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

  `http://docs.sun.com`

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

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

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

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

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

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

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

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

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

マニュアルコレクション	マニュアルタイトル	内容の説明
	Fortran プログラミングガイド	入出力、ライブラリ、プログラム分析、デバッグおよびパフォーマンスに関連する内容を記述しています。
	Fortran ユーザーズガイド	コマンド行オプションとコンパイラの使い方についての情報を記載しています。
	FORTRAN 77 言語リファレンス	Fortran 77 言語の包括的な参照情報を記載しています。
	Fortran 95 区間演算プログラミングリファレンス	Fortran 95 コンパイラによってサポートされる組み込み `INTERVAL` データについて説明しています。
Forte TeamWare 6 / Sun WorkShop TeamWare 6	Sun WorkShop TeamWare ユーザーズガイド	Sun WorkShop TeamWare コード管理ツールの使用方法について説明しています。
Forte Developer 6/ Sun WorkShop Visual 6	Sun WorkShop Visual ユーザーズガイド	C++ と Java™ の GUI (グラフィカルユーザーインタフェース) を Sun WorkShop Visual を使用して作成する方法について説明しています。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。
	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。

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
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
`varargs`	変更可能な引数のリストを処理
`version`	オブジェクトファイルまたはバイナリファイルのバージョン識別情報の表示
`yacc`	文脈自由文法を、LALR(1) 構文解析アルゴリズムを実行する単純オートマトン用の一連の表に変換

README (最新情報) ファイル

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

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

README ファイルを表示するには次のように入力します。

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

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

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

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

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

市販の書籍

C++ について書かれている書籍の一部を紹介します。

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

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

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

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

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

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

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

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

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

第1章

C++ ライブラリについて

C++ クラスライブラリとは、再利用可能なコードをモジュール単位で集めたものです。クラスライブラリを使用すると、既存のテスト済みコードをプログラムに組み込むことができます。

C++ ライブラリは、1 つ以上のヘッダーファイルと 1 つのオブジェクトファイルで構成されています。ヘッダーファイルには、クラスの定義など、ライブラリ関数を使用するのに必要なさまざまな定義が入っています。オブジェクトファイルには、コンパイル済みの関数とデータが入っており、それをユーザーが作成したプログラムとリンクすることで実行可能プログラムが作成されます。

本書では、C++ コンパイラで提供される次の 3 つのクラスライブラリについて説明します。

- 複素数ライブラリ — 第 2 章「複素数ライブラリ」で説明します。
- `iostream` ライブラリ — 第 3 章「従来型の iostream ライブラリ」で説明します。
- C++ 標準ライブラリ — 第 5 章「C++ 標準ライブラリ」で説明します。

共有ライブラリと静的ライブラリの構築方法、ライブラリの使用方法については、『C++ ユーザーズガイド』を参照してください。

マニュアルページ

このマニュアルで説明するライブラリに関連するマニュアルページは、次の場所にあります。

install-directory`/SUNWspro/man/man3`

install-directory`/SUNWspro/man/man3CC4`

install-directory/SUNWspro/man/man3c++

install-directory はインストールディレクトリです。デフォルトでは /opt です。日本語化されているマニュアルページは *install-directory*/SUNWspro/man/ja 以下にあります。

上記のマニュアルページを参照するには、MANPATH の指定に *install-directory*/SUNWspro/man が含まれていなければなりません。MANPATH の設定手順については、「はじめに」の xi ページの「Sun WorkShop 開発ツールへのアクセス方法」を参照してください。

Sun WokShop Compiler C++ ライブラリのマニュアルページを表示するには、次のように入力します *(library-name* には、ライブラリ名を入力します)。

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

Sun WorkShop C++ コンパイラの互換モードライブラリのマニュアルページを表示するには、次のように入力します。

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

あるいは、ブラウザでマニュアルページにアクセスするには、次のファイルを開きます。

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

デフォルトのインストールディレクトリ *(install-directory)* は /opt です。

その他のライブラリ

今回のリリースでは、complex、iostreams および C++ 標準 ライブラリに加えて、Tools.h++ ライブラリ、Sun WorkShop Memory Monitor ライブラリが含まれています。

Tools.h++ ライブラリ

Tools.h++ は、RogueWave の C++ 基礎クラスライブラリです。このリリースにはバージョン 7 の Tools.h++ ライブラリが含まれています。Tools.h++ライブラリの詳細については、次のマニュアルを参照してください。

- 『C++ ユーザーズガイド』
- 『Tools.h++ 7 ユーザーズガイド』
- 『Tools.h++ 7 クラスライブラリ・リファレンスマニュアル』

Sun WorkShop Memory Monitor

Sun WorkShop Memory Monitor には、メモリーリークやメモリーの断片化、メモリーの早期解放の問題を自動的に報告し、解決する機能があります。Sun WorkShop Memory Monitor には、次の 3 つの操作モードがあります。

- デバッグ
- 実装
- ガベージコレクション

使用されるモードは、アプリケーションのリンク先のライブラリによって決まります。

Sun WorkShop Memory Monitor の構成要素は次のとおりです。

- `libgc` - ガベージコレクションと実装モードで使用されるライブラリ
- `libgc_dbg` - デバッグモードで使用されるライブラリ
- `gcmonitor` - デバッグモードで使用されるデーモン

Sun WorkShop Memory Monitor に関する文書を表示するには、Memory Monitor を起動するか、ブラウザで次のファイルを開きます。

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

install-directory は、Sun WorkShop インストールディレクトリのパスに置き換えてください。デフォルトのインストールでは、*install-directory* は `/opt` です。

Sun WorkShop Memory Monitor のマニュアルページは、次の場所にあります。

install-directory`/SUNWspro/man/man1`

install-directory/SUNWspro/man/man3

install-directory は、デフォルトでは /opt です。日本語化されているマニュアルページは *install-directory*/SUNWspro/man/ja 以下にあります。

表 1-1 Sun WorkShop Memory Monitor のマニュアルページ

マニュアルページ	概要
gcmonitor	Sun WorkShop Memory Monitor の Web インタフェース
gcFixPrematureFrees	Sun WorkShop Memory Monitor による、早期メモリー解放の修正を有効または無効にする。
gcInitialize	起動時に Sun WorkShop Memory Monitor を構成する。

第2章

複素数ライブラリ

下の例のように、複素数には「実部」と「虚部」があります。

```
3.2 + 4i
1 + 3i
1 + 2.3i
```

通常は、`0+3i` のように完全に虚部だけのものは通常 `3i` と書き、`5+0i` のように完全に実部だけのものは通常 `5` と書きます。データ型 `complex` を使用すると複素数を表現することができます。

注 – 複素数ライブラリ (`libcomplex`) は互換モードでのみ使用できます (`-compat[=4]`)。標準モード (デフォルトのモード) では、同様の機能を持つ複素数クラスが C++ 標準ライブラリ (`libCstd`) に含まれています。

複素数ライブラリ

複素数ライブラリは、新しいデータ型として複素数データ型を実装します。このライブラリには以下が含まれています。

- 演算子
- 数学関数 (組み込み数値型用に定義されている関数)
- 拡張機能 (複素数の入出力を可能にする `iostream` 用)
- エラー処理機能

複素数には、実部と虚部による表現方法の他に、絶対値と偏角による表現方法があります。複素数ライブラリには、実部と虚部によるデカルト表現と、絶対値と偏角による極座標表現とを互いに変換する関数も提供しています。

共役複素数は、虚部の符号が反対の複素数です。

複素数ライブラリの使用方法

複素数ライブラリを使用する場合は、プログラムにヘッダーファイル `complex.h` をインクルードし、`-lcomplex` オプションまたは `-library=complex` オプションを使用してリンクしてください。

`complex` 型

複素数ライブラリでは、クラス `complex` が 1 つだけ定義されています。クラス `complex` のオブジェクトは、1 つの複素数を持つことができます。複素数は次の 2 つの部分で構成されています。

- 実部
- 虚部

```
class complex {
    double re, im;
};
```

各部の数値は `double` 型で入っています。上は、クラス `complex` の定義の例です。

クラス `complex` のオブジェクトの値は、1 組の `double` 型の値です。最初の値が実部を表し、 2 番目の値が虚部を表します。

`complex` クラスのコンストラクタ

`complex` クラスには、2 つのコンストラクタがあります。以下は、その定義です。

```
complex::complex(){ re=0.0; im=0.0; }
complex::complex(double r, double i = 0.0)  { re=r; im=i; }
```

複素数の変数を引数なしで宣言すると、最初のコンストラクタが使用され、実部も虚部もゼロで初期化されます。次の例では、実部も虚部もゼロの複素数の変数が生成されます。

```
complex aComp;
```

引数は 1 つまたは 2 つ指定することができ、どちらの場合も 2 番目のコンストラクタが使用されます。次の例のように、引数を 1 つだけ指定した場合は、その値は実部の値とみなされ虚部はゼロに設定されます。

```
complex aComp(4.533);
```

上の例では、次の値を持つ複素数の変数が生成されます。

```
4.533 + 0i
```

次の例のように、引数を 2 つ指定した場合は、最初の値が実部、2 番目の値が虚部となります。

```
complex aComp(8.999, 2.333);
```

上の例では、次の値を持つ複素数の変数が生成されます。

```
8.999 + 2.333i
```

また、複素数ライブラリが提供する `polar` 関数を使用して複素数を生成することもできます (8 ページの「数学関数」を参照してください)。`polar` 関数は、指定した 1 組の極座標値 (絶対値と偏角) を使用して複素数を作成します。

`complex` 型にはデストラクタはありません。

算術演算子

複素数ライブラリでは、すべての基本算術演算子が定義されています。特に、次の 5 つの演算子は通常の型の演算と同様に使用することができ、優先順序も同じです。

```
+   -   /   *   =
```

演算子 - は、通常の型の場合と同様に 2 項演算子としても単項演算子としても使用できます。

この他、次の演算子の使用方法も通常の型で使用する演算子と同様です。

```
+=   -=   *=   /=
```

ただし、最後の 4 つの演算子では、式の中で使用できる値は生成されません。したがって、次のように使用することはできません。

```
complex a, b;
...
if ((a+=2)==0) {...}; // 誤り
b = a *= b; // 誤り
```

また、等しいか否かを判定する次の 2 つの演算子は、通常の型で使用する演算子と同様に使用することができます。

```
==   !=
```

算術式で実数と複素数が混在しているときは、 C++ では複素数のための演算子関数が使用され、実数は複素数に変換されます。

数学関数

複素数ライブラリには、多くの数学関数が含まれています。複素数に特有のものもあれば、 C の標準数学ライブラリの関数と同じで複素数を対象にしたものもあります。

これらの関数はすべて、あらゆる可能な引数に対して結果を返します。関数が数学的に正しい結果を返せないような場合は、`complex_error` を呼び出して、何らかの適切な値を返します。たとえば、オーバーフローが実際に起こるのを避けるために `complex_error` を呼び出してメッセージを出します。次の表で複素数ライブラリの関数を説明します。

表 2-1　複素数ライブラリの関数

複素数ライブラリ関数	内容
`double abs(const complex)`	複素数の絶対値を返します。
`double arg(const complex)`	複素数の偏角を返します。
`complex conj(const complex)`	引数に対する共役複素数を返します。

表 2-1　複素数ライブラリの関数

複素数ライブラリ関数	内容
`double imag(const complex&)`	複素数の虚部を返します。
`double norm(const complex)`	引数の絶対値の 2 乗を返します。`abs` より高速ですが、オーバーフローが起きやすくなります。絶対値の比較に使用します。
`complex polar(double mag,double ang=0.0)`	複素数の絶対値と偏角を表す一組の極座標を引数として受け取り、それに対応する複素数を返します。
`double real(const complex&)`	複素数の実部を返します。

表 2-2　複素数の数学関数と三角関数

複素数ライブラリ関数	内容
`complex acos(const complex)`	引数が余弦となるような角度を返します。
`complex asin(const complex)`	引数が正弦となるような角度を返します。
`complex atan(const complex)`	引数が正接となるような角度を返します。
`complex cos(const complex)`	引数の余弦を返します。
`complex cosh(const complex)`	引数の双曲線余弦を返します。
`complex exp(const complex)`	`e**x` を計算します。ここで `e` は自然対数の底で、 `x` は関数 `exp` に渡された引数です。
`complex log(const complex)`	引数の自然対数を返します。
`complex log10(const complex)`	引数の常用対数を返します。

`complex pow(double b, const complex exp)`
`complex pow(const complex b, int exp)`
`complex pow(const complex b, double exp)`
`complex pow(const complex b, const complex exp)`
	この関数は引数を 2 つ持ちます。 `pow(`*b*, *exp*`)` とすると、 *b* の *exp* 乗が計算されます。

`complex sin(const complex)`	引数の正弦を返します。
`complex sinh(const complex)`	引数の双曲線正弦を返します。
`complex sqrt(const complex)`	引数の平方根を返します。
`complex tan(const complex)`	引数の正接を返します。
`complex tanh(const complex)`	引数の双曲線正接を返します。

エラー処理

複素数ライブラリでは、エラー処理が次のように定義されています。

```
extern int errno;
class c_exception { ... };
int complex_error(c_exception&);
```

外部変数 errno は C ライブラリの大域的なエラー状態です。errno は、標準ヘッダー errno.h (perror(3) のマニュアルページを参照) にリストされている値を持ちます。 errno には、多くの関数でゼロ以外の値が設定されます。

ある特定の演算でエラーが起こったかどうか調べるには、次のようにしてください。

1. **演算実行前に errno をゼロに設定する。**
2. **演算終了後に値を調べる。**

関数 complex_error は c_exception 型の参照引数を持ち、次に示す複素数ライブラリ関数に呼び出されます。

- exp
- log
- log10
- sinh
- cosh

デフォルトの complex_error はゼロを返します。ゼロが返されたということは、デフォルトのエラー処理が実行されたということです。ユーザーは独自の complex_error 関数を作成して、別のエラー処理を行うことができます。エラー処理については、cplexrr(3CC4) のマニュアルページで説明しています。

デフォルトのエラー処理については、cplxtrig(3CC4) と cplxexp(3CC4) のマニュアルページで説明しています。次に概要を示します。

表 2-3　複素数ライブラリ関数

複素数ライブラリ関数	デフォルトエラー処理
exp	オーバーフローが起こった場合は errno を ERANGE に設定し、最大複素数を返します。
log、log10	引数がゼロの場合は errno を EDOM に設定し、最大複素数を返します。
sinh、cosh	引数の虚部によりオーバーフローが起こる場合は複素数ゼロを返します。引数の実部によりオーバーフローが起こる場合は最大複素数を返します。どちらの場合も errno は ERANGE に設定されます。

入出力

複素数ライブラリでは、次の例に示す複素数のデフォルトの抽出子と挿入子が提供されています。

```
ostream& operator<<(ostream&, const complex&); //挿入子
istream& operator>>(istream&, complex&) //抽出子
```

抽出子と挿入子の基本的な説明については、16 ページの「iostream 操作の基本構造」と 18 ページの「iostream を使用した出力」を参照してください。

入力の場合、複素数の抽出子 >> は、(括弧の中にあり、コンマで区切られた) 一組の値を入力ストリームから抽出し、複素数オブジェクトに読み込みます。最初の値が実部の値、 2 番目の値が虚部の値となります。たとえば、次のような宣言と入力文がある場合、

(3.45,5) と入力すると、複素数 x の値は 3.45+5.0i となります。抽出子の場合はこの反対になります。complex x(3.45,5), cout << x の場合は、(3.45,5) と表示されます。

```
complex x;
cin >> x;
```

入力データは、通常括弧の中でコンマで区切られた一組の値で、スペースは入れても入れなくてもかまいません。値を 1 つだけ入力したとき (括弧とスペースは入力してもしなくても同じ) は、抽出子は虚部をゼロとします。シンボル i を入力してはいけません。

挿入子は、複素数の実部と虚部をコンマで区切り、全体を括弧で囲んで挿入します。シンボル i は含まれません。2 つの値は double 型として扱われます。

混合算術演算

complex 型は、組み込みの算術型と混在した式でも使用できるように定義されています。混合算術演算においては、算術型は自動的に complex 型に変換されます。算術演算子のすべてと数学関数のほとんどに対して、complex 型を使用できるバージョンが提供されています。次の例で考えてみます。

```
int i, j;
double x, y;
complex a, b;
a = sin((b+i)/y) + x/j;
```

b+i という式は混合算術演算です。整数 i は、コンストラクタ complex::complex(double,double=0) によって、complex 型に変換されます (このとき、まず整数から double 型に変換されます)。b+i の計算結果を double 型の y で割っているので、y もまた complex 型に変換され、複素数除算演算が使用されます。商もまた complex 型ですので、複素数の正弦関数が呼び出され、その結果も complex 型になります。以下も同様です。

ただし、すべての算術演算と型変換が暗黙に行われるわけではありませんし、定義されていないものもあります。たとえば、複素数は数学的な意味での大小関係が決められないので、比較は等しいか否かの判定しかできません。

```
complex a, b;
a == b // OK
a != b // OK
a <  b // エラー：演算子 < は complex 型に使用できない
a >= b // エラー：演算子 >= は complex 型に使用できない
```

同様に、complex 型からそれ以外の型への変換もはっきりした定義ができないので、そのような自動変換は行われません。変換するときは、実部または虚部を取り出すのか、または絶対値を取り出すのかを指定する必要があります。

```
complex a;
double f(double);
f(abs(a)); // OK
f(a);      // エラー： f(complex) は、引数の型が一致していない
```

効率

クラス complex は効率も考慮して設計されています。

非常に簡単な関数が inline で宣言されており、関数呼び出しのオーバーヘッドをなくしています。

効率に差があるものは、関数が多重定義されています。たとえば、pow 関数には引数が complex 型のものの他に、引数が double 型と int 型のものがあります。その方が double 型と int 型の計算がはるかに簡単になるからです。

complex.h をインクルードすると、C の標準数学ライブラリヘッダー math.h も自動的にインクルードされます。C++ の多重定義の規則により、次のように最も効率の良い式の評価が行われます。

```
double x;
complex x = sqrt(x);
```

この例では、標準数学関数 sqrt(double) が呼び出され、その計算結果が complex 型に変換されます。最初に complex 型に変換され、sqrt(complex) が呼び出されるのではありません。これは、多重定義の解決規則から決まる方法で、最も効率の良い方法です。

複素数ライブラリに関するマニュアルページ

このほか、複素数ライブラリに関連するマニュアルページには以下のものがあります。

表 2-4 `complex` 型に関するマニュアルページ

マニュアルページ	概要
`cplx.intro(3CC4)`	複素数ライブラリ全体の紹介
`cartpol(3CC4)`	直角座標と極座標の関数
`cplxerr(3CC4)`	エラー処理関数
`cplxexp(3CC4)`	指数、対数、平方根の関数
`cplxops(3CC4)`	算術演算子関数
`cplxtrig(3CC4)`	三角関数

第3章

従来型の iostream ライブラリ

C++ も C と同様に組み込み型の入出力文はありません。その代わりに、入出力機能はライブラリで提供されています。Sun WorkShop 6 C++ コンパイラでは iostream クラスに対して、従来型の実装と ISO 標準の実装を両方とも提供しています。

- 互換モード (-compat[=4]) では、従来型の iostream クラスは libC に含まれています。
- 標準モード (デフォルトのモード) では、従来型の iostream クラスは libiostream に含まれています。従来型の iostream クラスを使用したソースコードを標準モードでコンパイルするときは、libiostream を使用します。従来型の iostream の機能を標準モードで使用するには、iostream.h ヘッダーファイルをインクルードし、-library=iostream オプションを使用してコンパイルします。
- 標準の iostream クラスは標準モードだけで使用でき、C++ 標準ライブラリ libCstd に含まれています。

この章では、従来型の iostream ライブラリの概要と使用例を説明します。この章では、iostream ライブラリを完全に説明しているわけではありません。詳細は、iostream ライブラリのマニュアルページを参照してください。従来型の iostream のマニュアルページを表示するには、次のように入力します (*name* にはマニュアルページのトピック名を入力)。

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

定義済みの iostream

定義済みの iostream には、次のものがあります。

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

定義済み iostreams は、cerr を除いて完全にバッファリングされています。18 ページの「iostream を使用した出力」と 22 ページの「iostream を使用した入力」を参照してください。

iostream 操作の基本構造

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

- 標準入力
- 標準出力
- 標準エラー
- ファイル
- 文字型配列

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

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

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

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

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

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

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

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

従来型の iostream ライブラリの使用

従来型の iostream ライブラリルーチンを使用するには、ライブラリの使用部分に対応するヘッダーファイルをインクルードしなければなりません。次の表で各ヘッダーファイルについて説明します。

表 3-1 iostream ルーチンのヘッダーファイル

ヘッダーファイル	内容
iostream.h	iostream ライブラリの基本機能の宣言。
fstream.h	ファイルに固有の iostream と streambuf の宣言。この中で iostream.h をインクルードします。
strstream.h	文字型配列に固有の iostream と streambuf の宣言。この中で iostream.h をインクルードします。

表 3-1 iostream ルーチンのヘッダーファイル

ヘッダーファイル	内容
iomanip.h	マニピュレータ値の宣言。マニピュレータ値とは iostream に挿入または iostream から抽出する値で、特別の効果を引き起こします。
stdiostream.h	(旧形式) 標準入出力の FILE 使用のための iostream と streambuf の宣言。この中で iostream.h をインクルードします。
stream.h	(旧形式) C++ バージョン 1.2 の旧形式ストリームと互換性を保つための宣言。この中で iostream.h、fstream.h、iomanip.h、stdiostream.h をインクルードします。

これらのヘッダーファイルすべてをプログラムにインクルードする必要はありません。自分のプログラムで必要な宣言の入ったものだけをインクルードします。互換モード (-compat[=4]) では、従来型の iostream ライブラリは libC の一部であり、CC ドライバによって自動的にリンクされます。標準モード (デフォルトのモード) では、従来型の iostream ライブラリは libiostream に含まれています。

iostream を使用した出力

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

```
cout << someValue;
```

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

iostream ライブラリでは、ユーザー定義型については検知しません。したがってユーザー定義型を出力したい場合は、ユーザーが自分で挿入子を正しく定義する、すなわち << 演算子を多重定義する必要があります。

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

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

上の例では、2 つの値の間に空白が入りません。空白を入れたい場合は、次のようにします。

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

<< 演算子は、組み込みの左シフト演算子と同じ優先順位を持ちます。他の演算子と同様に、括弧を使用して実行順序を指定することができます。実行順序をはっきりさせるためにも、括弧を使用するとよい場合がよくあります。次の 4 つの文のうち、最初の 2 つは同じ結果になりますが、後の 2 つは異なります。

```
cout << a+b;       // + は << より優先順位が高い
cout << (a+b);
cout << (a&y);     // << は & より優先順位が高い
cout << a&y;       // (cout <<a) & y となっておそらくエラーになる
```

ユーザー定義の挿入演算子

次のコーディング例では string クラスを定義しています。

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

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

public:
    (さまざまな関数定義)

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

この例では、挿入演算子と抽出演算子をフレンド定義しておく必要があります。string クラスのデータ部が非公開だからです。

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

上は、string クラスに対して多重定義された演算子関数 operator<< の定義です。

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

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

出力エラーの処理

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

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

一般的なエラー処理方法は、メインのどこかで定期的に出力ストリームの状態を検査する方法です。そこで、エラーが起こっていることが分かれば、何らかの処理を行います。この章では、文字列を出力してプログラムを中止させる関数 error をユーザーが定義しているものとして説明します。関数 error はユーザー定義の関数で、定義済みの関数ではありません。関数 error の内容については、「入力エラーの処理」を参照してください。iostream の状態を調べるには、演算子 ! を使用します。次の例に示すように、iostream がエラー状態の場合はゼロ以外の値を返します。

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

エラーを調べるにはもう 1 つの方法があります。ios クラスでは、operator void *() が定義されており、エラーが起こった場合は NULL ポインタを返します。したがって、次の文でエラーを検査することができます。

```
if (cout << x) return ; // 正常終了のときのみ返す
```

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

```
if ( cout.good() ) return ; // 正常終了のときのみ返す
```

エラービットは次のような列挙型で宣言されています。

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

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

出力のフラッシュ

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

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

flush は、マニピュレータと呼ばれるタイプのオブジェクトの 1 つです。マニピュレータを iostream に挿入すると、その値が出力されるのではなく、何らかの効果が引き起こされます。マニピュレータは実際には関数で、ostream& または istream& を引数として受け取り、そのストリームに対する何らかの動作を実行した後にその引数を返します (32 ページの「マニピュレータ」を参照してください)。

バイナリ出力

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

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

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

iostream を使用した入力

iostream を使用した入力は、iostream を使用した出力と同じです。入力には、抽出演算子 >> を使用します。次の例のように、挿入演算子と同様に繰り返し指定することができます。

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

この例では、標準入力から 2 つの値が取り出されます。他の多重定義演算子と同様に、使用される抽出子の機能は a と b の型によって決まります (a と b の型が異なれば、別の抽出子が使用されます)。入力データのフォーマットとその制御方法についての詳細は、ios(3CC4) のマニュアルページを参照してください。通常は、先頭の空白文字 (スペース、改行、タブ、フォームフィードなど) は無視されます。

ユーザー定義の抽出演算子

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

クラス string の抽出演算子は次のコーディング例のように定義します。

コード例 3-1 string の抽出演算子

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

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

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

char* の抽出子

この定義済み抽出子は問題が起こる可能性があるため、ここで説明しておきます。この抽出子は次のように使用します。

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

上の例で、抽出子は先頭の空白を読み飛ばし、次の空白文字までの文字列を抽出して x にコピーします。次に、文字列の最後を示す NULL 文字 (0) を入れて文字列を完成します。ここで、入力文字列が指定した配列からあふれる可能性があることに注意してください。

さらに、ポインタが、割り当てられた記憶領域を指していることを確認する必要があります。次に示すのは、よく発生するエラーの例です。

```
char * p; // 初期化されていない
cin >> p;
```

入力データが格納される場所が特定されていません。これによって、プログラムが異常終了することがあります。

1 文字の読み込み

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

```
char c;
cin.get(c); // 入力に失敗した場合は、c は変更なし

int b;
b = cin.get(); // 入力に失敗した場合は、b を EOF に設定
```

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

空白だけを読み飛ばして、タブや改行などその他の文字を取り出すようにするには、次のようにします。

```
int a;
do {
    a = cin.get();
while( a == ' ' );
```

バイナリ入力

メンバー関数 write で出力したようなバイナリの値を読み込むには、メンバー関数 read を使用します。次の例では、メンバー関数 read を使用して x のバイナリ形式の値をそのまま入力します。次の例は、先に示した関数 write を使用した例と反対のことを行います。

```
cin.read((char*)&x, sizeof(x));
```

入力データの先読み

メンバー関数 peek を使用するとストリームから次の文字を抽出することなく、その文字を知ることができます。使用例を次に示します。

```
if (cin.peek() != c) return 0;
```

空白の抽出

デフォルトでは、iostream の抽出子は先頭の空白を読み飛ばします。skip フラグをオフにすれば、先頭の空白を読み飛ばさないようにすることができます。次の例では、cin の先頭の空白の読み飛ばしをいったんオフにし、後にオンに戻しています。

```
cin.unsetf(ios::skipws);     // 先頭の空白の読み飛ばしをオフに設定
. . .
cin.setf(ios::skipws);       // 先頭の空白の読み飛ばしをオンに再設定
```

iostream のマニピュレータ ws を使用すると、空白の読み飛ばしが現在オンかオフかに関係なく、iostream から先頭の空白を取り除くことができます。次の例では、iostream istr から先頭の空白が取り除かれます。

```
istr >> ws;
```

入力エラーの処理

通常は、第 1 引数が非ゼロのエラー状態にある場合、抽出子は入力ストリームからのデータの抽出とエラービットのクリアを行わないでください。データの抽出に失敗した場合、抽出子は最低 1 つのエラービットを設定します。

出力エラーの場合と同様、エラー状態を定期的に検査し、非ゼロの状態の場合は処理の中止など何らかの動作を起こす必要があります。演算子 ! は、iostream のエラー状態を検査します。たとえば次のコーディング例では、英字を入力すると入力エラーが発生します。

```
#include <unistd.h>
#include <iostream.h>
void error (const char* message) {
    cout << message << "\n" ;
    exit(1);
}
main() {
    cout << "Enter some characters: ";
    int bad;
    cin >> bad;
    if (!cin) error("aborted due to input error");
    cout << "If you see this, not an error." << "\n";
    return 0;
}
```

クラス ios には、エラー処理に使用できるメンバー関数があります。詳細はマニュアルページを参照してください。

iostream と stdio の併用

C++ でも stdio を使用することができますが、プログラムで iostream と stdio とを標準ストリームとして併用すると、問題が起こる場合があります。たとえば stdout と cout の両方に書き込んだ場合、個別にバッファリングされるため出力結

果が設計したとおりにならないことがあります。stdin と cin の両方から入力した場合、問題はさらに深刻です。個別にバッファリングされるため、入力データが使用できなくなってしまいます。

標準入力、標準出力、標準エラーに関するこのような問題を解決するためには、入出力に先立って次の命令を実行します。次の命令で、すべての定義済み iostream が、それぞれ対応する定義済みの標準入出力の FILE に結合されます。

```
ios::sync_with_stdio();
```

このような結合を行うと、定義済みストリームが結合されたものの一部となってバッファリングされなくなってかなり効率が悪くなるため、デフォルトでは結合されていません。同じプログラムでも、stdio と iostream を別のファイルに対して使用することはできます。すなわち、stdio ルーチンを使用して stdout に書き込み、iostream に結合した別のファイルに書き込むことは可能です。また stdio FILE を入力用にオープンしても、stdin から入力しない限りは cin からも読み込むことができます。

iostream の作成

定義済みの iostream 以外のストリームへの入出力を行いたい場合は、ユーザーが自分で iostream を生成する必要があります。これは一般には、iostream ライブラリで定義されている型のオブジェクトを生成することになります。ここでは、使用できるさまざまな型について説明します。

クラス fstream を使用したファイル操作

ファイル操作は標準入出力の操作に似ています。ifstream、ofstream、fstream の 3 つのクラスはそれぞれ、istream、ostream、iostream の各クラスから派生しています。この 3 つのクラスは派生クラスなので、挿入演算と抽出演算、および、その他のメンバー関数を継承しており、ファイル使用のためのメンバーとコンストラクタも持っています。

fstream のいずれかを使用するときは、fstream.h をインクルードしなければなりません。入力だけ行うときは ifstream、出力だけ行うときは ofstream、入出力を行うときは fstream を使用します。コンストラクタへの引数としてはファイル名を渡します。

thisFile というファイルから thatFile というファイルへのファイルコピーを行うときは、次のコーディング例のようになります。

```
ifstream fromFile("thisFile");
if      (!fromFile)
        error("unable to open 'thisFile' for input");
ofstream toFile ("thatFile");
if      ( !toFile )
        error("unable to open 'thatFile' for output");
char c ;
while (toFile && fromFile.get(c)) toFile.put(c);
```

このコードでは次のことを実行します。

- fromFile という ifstream オブジェクトをデフォルトモード ios::in で生成し、それを thisFile に結合します。thisFile をオープンします。
- 生成した ifstream オブジェクトのエラー状態を調べ、エラーであれば関数 error を呼び出します。関数 error は、プログラムのどこか別のところで定義されている必要があります。
- toFile という ofstream オブジェクトをデフォルトモード ios::out で生成し、それを thatFile に結合します。
- fromFile と同様に、toFile のエラー状態を検査します。
- データの受け渡しに使用する char 型変数を生成します。
- fromFile の内容を一度に 1 文字ずつ toFile にコピーします。

注 – ファイルの内容を一度に 1 文字ずつコピーすることは実際にはあまり行われません。このコードは fstream の使用例として示したにすぎません。実際には、入力ストリームに関係付けられた streambuf を出力ストリームに挿入するのが一般的です。37 ページの「streambuf」と sbufpub(3CC4) のマニュアルページを参照してください。

オープンモード

オープンモードは、列挙型 open_mode の各ビットの OR をとって設定します。open_mode は、ios クラスの公開部で次のように定義されています。

```
enum open_mode {binary=0, in=1, out=2, ate=4, app=8, trunc=0x10,
     nocreate=0x20, noreplace=0x40};
```

注 – UNIX では binary フラグは必要ありませんが、binary フラグを必要とするシステムとの互換性を保つために提供されています。移植可能なコードにするためには、バイナリファイルをオープンするときに binary フラグを使用する必要があります。

入出力両用のファイルをオープンすることができます。たとえば次のコードでは、someName という入出力ファイルをオープンして、fstream 変数 inoutFile に結合します。

```
fstream inoutFile("someName", ios::in|ios::out);
```

ファイルを指定しない fstream の宣言

ファイルを指定せずに fstream の宣言だけを行い、後にファイルをオープンすることもできます。次の例では出力用の ofstream toFile を作成します。

```
ofstream toFile;
toFile.open(argv[1], ios::out);
```

ファイルのオープンとクローズ

fstream をいったんクローズし、また別のファイルでオープンすることができます。たとえば、コマンド行で与えられるファイルリストを処理するには次のようにします。

```
ifstream infile;
for (char** f = &argv[1]; *f; ++f) {
   infile.open(*f, ios::in);
   ...;
   infile.close();
}
```

ファイル記述子を使用したファイルのオープン

標準出力は整数 1 などのようにファイル記述子が分かっている場合は、次のようにファイルをオープンすることができます。

```
ofstream outfile;
outfile.attach(1);
```

fstream のコンストラクタにファイル名を指定してオープンしたり、open 関数を使用してオープンしたファイルは、fstream が破壊された時点 (delete するか、スコープ外に出る時点) で自動的にクローズされます。attach で fstream に結合したファイルは、自動的にはクローズされません。

ファイル内の位置の再設定

ファイル内の読み込み位置と書き込み位置を変更することができます。そのためには次のようなツールがあります。

- streampos は、iostream 内の位置を記憶しておくためのデータ型です。
- tellg (tellp) は istream (ostream) のメンバー関数で、現在のファイル内の位置を返します。istream と ostream は fstream の親クラスですので、tellg と tellp も fstream クラスのメンバー関数として呼び出すことができます。
- seekg (seekp) は istream (ostream) のメンバー関数で、指定したファイル内の位置を探し出します。

- 列挙型 seek_dir は、seek での相対位置を指定します。

```
enum seek_dir { beg=0, cur=1, end=2 }
```

fstream aFile の位置再設定の例を次に示します。

```
streampos original = aFile.tellp();      // 現在の位置の保存
aFile.seekp(0, ios::end);                // ファイルの最後に位置を再設定
aFile << x;                              // データをファイルに書き込む
aFile.seekp(original);                   // 元の位置に戻る
```

seekg (seekp) は、1 つまたは 2 つの引数を受け取ります。引数を 2 つ受け取るときは、第 1 引数は、第 2 引数で指定した seek_dir 値が示す位置からの相対位置となります。次に例を示します。この例では、ファイルの最後から 10 バイトの位置に設定されます。

```
aFile.seekp(-10, ios::end);
```

一方、次の例では現在位置から 10 バイト進められます。

```
aFile.seekp(10, ios::cur);
```

注 – テキストストリーム上での任意位置へのシーク動作はマシン依存になります。ただし、以前に保存した streampos の値にいつでも戻ることができます。

iostream の代入

iostream では、あるストリームを別のストリームに代入することはできません。

ストリームオブジェクトをコピーすると、出力ファイル内の現在の書き込み位置ポインタなどの位置情報が二重に存在するようになり、それを個別に変更できるという状態が起こります。これは、ストリーム操作を混乱させる可能性があります。

フォーマットの制御

フォーマットの制御については、ios(3CC4) のマニュアルページで詳しく説明しています。

マニピュレータ

マニピュレータとは、iostream に挿入したり、iostream から抽出したりする値で特別な効果を持つもののことです。

引数付きマニピュレータとは、 1 つ以上の追加の引数を持つマニピュレータのことです。

マニピュレータは通常の識別子であるため、マニピュレータの定義を多く行うと可能な名前を使いきってしまうので、iostream では考えられるすべての機能に対して定義されているわけではありません。マニピュレータの多くは、この章の別の箇所でメンバー関数とともに説明しています。

定義済みマニピュレータは 13 個あり、それぞれについては 33 ページの表 3-2 「iostream の定義済みマニピュレータ」 で説明します。表 3-2 で使用している文字の意味は次のとおりです。

- i は long 型です。
- m は int 型です。
- c は char 型です。
- istr は 入力ストリームです。

- `ostr` は 出力ストリームです。

表 3-2 `iostream` の定義済みマニピュレータ

	定義済みマニピュレータ	内容
1	`ostr << dec, istr >> dec`	基数が 10 の整数変換を指定します。
2	`ostr << endl`	復帰改行文字 `('\n')` を挿入して、`ostream::flush()` を呼び出します。
3	`ostr << ends`	NULL (0) 文字を挿入。 `strstream` 使用時に利用します。
4	`ostr << flush`	`ostream::flush()` を呼び出します。
5	`ostr << hex, istr >> hex`	基数が 16 の整数変換を指定します。
6	`ostr << oct, istr >> oct`	基数が 8 の整数変換を指定します。
7	`istr >> ws`	最初に空白以外の文字が見つかるまで (この文字以降は `istr` に残る)、空白を取り除きます (空白を読み飛ばす)。
8	`ostr << setbase(n), istr >> setbase(n)`	基数が `n` (0, 8, 10, 16 のみ) の整数変換を指定します。
9	`ostr << setw(n), istr >> setw(n)`	`ios::width(n)` を呼び出します。フィールド幅を `n` に設定します。
10	`ostr << resetiosflags(i), istr >> resetiosflags(i)`	`i` のビットセットに従って、フラグのビットベクトルをクリアします。
11	`ostr << setiosflags(i), istr >> setiosflags(i)`	`i` のビットセットに従って、フラグのビットベクトルを設定します。
12	`ostr << setfill(c), istr >> setfill(c)`	埋め込み文字 (フィールドのパディング用文字) を `c` とします。
13	`ostr << setprecision(n), istr >> setprecision(n)`	浮動小数点型データの精度を `n` 桁にします。

定義済みマニピュレータを使用するには、プログラムにヘッダーファイル `iomanip.h` をインクルードする必要があります。

ユーザーが独自のマニピュレータを定義することもできます。マニピュレータには次の 2 つの基本タイプがあります。

- 引数なしのマニピュレータ
  istream&、ostream&、ios& のどれかを引数として受け取り、ストリームの操作が終わるとその引数を返します。
- 引数付きのマニピュレータ
  istream&、ostream&、ios& のどれかと、その他もう 1 つの引数 (追加の引数) を受け取り、ストリームの操作が終わるとストリーム引数を返します。

以下に、それぞれのタイプのマニピュレータの例を示します。

引数なしのマニピュレータの使用法

引数なしのマニピュレータは、次の 3 つを実行する関数です。

1. ストリームの参照引数を受け取ります。
2. そのストリームに何らかの処理を行います。
3. その引数を返します。

iostream では、このような関数 (へのポインタ) を使用するシフト演算子がすでに定義されていますので、関数を入出力演算子シーケンスの中に入れることができます。シフト演算子は、値の入出力を行う代わりに、その関数を呼び出します。tab を ostream に挿入する tab マニピュレータの例を示します。

```
ostream& tab(ostream& os) {
    return os << '\t' ;
    }
...
cout << x << tab << y ;
```

次のコードは、上の例と同じ処理をより洗練された方法で行います。

```
const char tab = '\t';
...
cout << x << tab << y;
```

次に示すのは別の例で、定数を使用してこれと同じことを簡単に実行することはできません。入力ストリームに対して、空白の読み飛ばしのオン、オフを設定したいと仮定します。

ios::setf と ios::unsetf を別々に呼び出して、skipws フラグをオンまたはオフに設定することもできますが、次の例のように 2 つのマニピュレータを定義して設定することもできます。

```
#include <iostream.h>
#include <iomanip.h>
istream& skipon(istream &is) {
        is.setf(ios::skipws, ios::skipws);
        return is;
}
istream& skipoff(istream& is) {
        is.unsetf(ios::skipws);
        return is;
}
...
int main ()
{
   int x,y;
   cin >> skipon >> x >> skipoff >> y;
   return 1;
}
```

引数付きのマニピュレータの使用法

iomanip.h に入っているマニピュレータの 1 つに setfill があります。setfill は、フィールド幅に詰め合わせる文字を設定するマニピュレータで、次の例に示すように定義されています。

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

//非公開のマニピュレータ
static ios& sfill(ios& i, int f) {
          i.fill(f);
          return i;
}

//公開の適用子
smanip_int setfill(int f) {
        return smanip_int(sfill, f);
}
```

引数付きマニピュレータは、2 つの部分から構成されます。

1 つはマニピュレータ部分で、これは引数を 1 つ追加します。この前の例では、int 型の第 2 引数があります。このような関数に対するシフト演算子は定義されていませんので、このマニピュレータ関数を入出力演算子シーケンスに入れることはできません。そこで、マニピュレータの代わりに補助関数 (適用子) を使用する必要があります。

もう 1 つは適用子で、これはマニピュレータを呼び出します。適用子は大域関数で、そのプロトタイプをヘッダーファイルに入れておきます。マニピュレータは通常、適用子の入っているソースコードファイル内に静的関数として作成します。マニピュレータは適用子からのみ呼び出されるので、静的関数にして、大域アドレス空間にマニピュレータ関数名を入れないようにします。

ヘッダーファイル iomanip.h には、さまざまなクラスが定義されています。各クラスには、マニピュレータ関数のアドレスと 1 つの引数の値が入っています。iomanip クラスについては、manip(3CC4) のマニュアルページで説明しています。この前の例では、 smanip_int クラスを使用しており、ios で使用できます。ios で使用できるということは、istream と ostream でも使用できるということです。この例ではまた、int 型の第 2 引数を使用しています。

適用子は、クラスオブジェクトを作成してそれを返します。この前の例では、smanip_int というクラスオブジェクトが作成され、そこにマニピュレータと、適用子の int 型引数が入っています。ヘッダーファイル iomanip.h では、このクラスに対するシフト演算子が定義されています。入出力演算子シーケンスの中に適用子関数 setfill があると、その適用子関数が呼び出され、クラスが返されます。シフト演算子はそのクラスに対して働き、クラス内に入っている引数値を使用してマニピュレータ関数が呼び出されます。

次の例では、マニピュレータ print_hex は以下のことを行います。

- 出力ストリームを 16 進モードする。
- long 型の値をストリームに挿入する。
- ストリームの変換モードを元に戻す。

この例は出力専用のため、omanip_long クラスが使用されています。また、int 型でなく long 型でデータを操作します。

```
#include <iostream.h>
#include <iomanip.h>
static ostream& xfield(ostream& os, long v) {
        long save = os.setf(ios::hex, ios::basefield);
        os << v;
        os.setf(save, ios::basefield);
        return os;
    }
omanip_long print_hex(long v) {
       return omanip_long(xfield, v);
   }
```

strstream：配列用の iostream

strstream(3CC4) のマニュアルページを参照してください。

stdiobuf: 標準入出力ファイル 用の iostream

stdiobuf(3CC4) のマニュアルページを参照してください。

streambuf

入力や出力のシステムは、フォーマットを行う iostream と、フォーマットなしの文字ストリームの入力または出力を行う streambuf からなります。

通常は iostream を通して streambuf を使用するので、streambuf の詳細を知る必要はありません。ただし、効率をよくするため、または iostream に組み込まれているエラー処理やフォーマットのためなどに必要な場合は、直接 streambuf を使用することができます。

streambuf の機能

streambuf は文字シーケンス (文字ストリーム) と、シーケンス内を指す 1 つまたは 2 つのポインタとで構成されています。各ポインタは文字と文字の間を指しています。実際には文字と文字の間を指しているわけではありませんが、このように考えておくと理解しやすくなります。streambuf ポインタには次の種類があります。

- *put* ポインタ
  次に streambuf から渡す文字の直前を指します。
- *get* ポインタ
  次に streambuf から取り出す文字の直前を指します。

streambuf は、このどちらかのポインタ、または両方のポインタを持ちます。

ポインタの位置

ポインタ位置の操作とシーケンスの内容の操作にはさまざまな方法があります。文字列の操作時に両方のポインタが移動するかどうかは、使用される streambuf の種類によって違います。一般に、キュー形式の streambuf の場合は、get ポインタと put ポインタは別々に移動し、ファイル形式の streambuf の場合は、get ポインタと put ポインタは同時に移動します。キュー形式ストリームの例としては strstream があり、ファイル形式ストリームの例としては fstream があります。

streambuf の使用

ユーザーは streambuf オブジェクト自体を作成することはなく、streambuf クラスから派生したクラスのオブジェクトを作成します。その例として、filebuf と strstreambuf とがあります。この 2 つについてはそれぞれ filebuf(3CC4) および ssbuf(3CC4) のマニュアルページを参照してください。より高度な使い方として、独自のクラスを streambuf から派生させて特殊デバイスのインタフェースを提供したり、基本的なバッファリング以外のバッファリングを行なったりすることができます。sbufpub(3CC4) と sbufprot(3CC4) のマニュアルページでは、それらの方法について説明しています。

ユーザー用の特殊な streambuf を作成するとき以外にも、上に示したマニュアルページで説明しているように、iostream と結合した streambuf にアクセスして公開メンバー関数を使用したい場合があります。また、各 iostream には、streambuf へのポインタを引数とする定義済みの挿入子と抽出子があります。streambuf を挿入したり抽出したりすると、ストリーム全体がコピーされます。

次の例では、先に説明したファイルコピーとは違う方法でファイルをコピーしています。簡単にするため、エラー検査は省略しています。

```
ifstream fromFile("thisFile");
ofstream toFile ("thatFile");
toFile << fromFile.rdbuf();
```

入力ファイルと出力ファイルは、以前の例と同じ方法でオープンします。各 iostream クラスにはメンバー関数 rdbuf があり、それに結合した streambuf オブジェクトへのポインタを返します。fstream の場合、streambuf オブジェクトは filebuf 型です。fromFile に結合したファイル全体が toFile に結合したファイルにコピー (挿入) されます。最後の行は次のように書くこともできます。

```
fromFile >> (streambuf*)toFile.rdbuf();
```

上の書き方では、ソースファイルが抽出されて目的のところに入ります。どちらの書き方をしても、結果はまったく同じになります。

iostream に関するマニュアルページ

C++ では、iostream ライブラリの詳細を説明する多くのマニュアルページがあります。次に、各マニュアルページの概要を示します。

従来型の iostream ライブラリのマニュアルページを表示するには、次のように入力します (*name* には、マニュアルページのトピック名を入力)。

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

表 3-3 iostream に関するマニュアルページの概要

マニュアルページ	概要
filebuf	streambuf から派生し、ファイル処理のために特殊化された filebuf クラスの公開インタフェースを詳細に説明します。streambuf クラスから継承した機能の詳細については、sbufpub(3CC4) と sbufprot(3CC4) のマニュアルページを参照してください。filebuf クラスは、fstream クラスを通して使用します。
fstream	istream、ostream、iostream をファイル処理用に特殊化した ifstream、ofstream、fstream の各クラスの特殊化されたメンバー関数を詳細に説明します。
ios	iostream の基底クラスである ios クラスの各部を詳細に説明します。すべてのストリームに共通の状態データについても説明します。
ios.intro	iostream を紹介し、概要を説明します。
istream	次の各項目を詳細に説明します。 - istream クラスに対するメンバー関数で、streambuf から取り 出した文字の解釈をサポートする - 入力のフォーマット - ostream クラスで記述されている位置決め関数 - その他の istream に関連した関数 - istream に関連したマニピュレータ
manip	iostream ライブラリで定義されている入出力マニピュレータを説明します。
ostream	次の各項目を詳細に説明します。 - ostream クラスに対するメンバー関数で、streambuf に書き込まれた文字の解釈をサポートする - 出力のフォーマット - ostream クラスで記述されている位置決め関数 - その他の ostream に関連した関数 - ostream に関連したマニピュレータ
sbufprot	streambuf(3CC4) クラスから派生したクラスをコーディングするプログラマに必要なインタフェースを説明します。公開関数のいくつかは、このマニュアルページでは説明しないため、sbufpub(3CC4) のマニュアルページも参照してください。

表 3-3 iostream に関するマニュアルページの概要 (続き)

マニュアルページ	概要
sbufpub	streambuf クラスの公開インタフェース、特に streambuf の公開メンバー関数について詳細に説明します。このマニュアルページには、streambuf 型のオブジェクトを直接操作したり、 streambuf から派生したクラスが継承している関数を探し出したりするのに必要な情報が含まれています。streambuf からクラスを派生する場合は、sbufprot(3CC4) のマニュアルページも参照してください。
ssbuf	streambuf から派生し、文字型配列処理用に特殊化された strstreambuf クラスの公開インタフェースを詳細に説明します。streambuf クラスから継承する機能の詳細については、sbufpub(3CC4) のマニュアルページを参照してください。
stdiobuf	streambuf から派生し、標準入出力の FILE 処理のために特殊化された stdiobuf クラスについて最小限の説明をします。streambuf クラスから継承する機能の詳細については、sbufpub(3CC4) のマニュアルページを参照してください。
strstream	strstream の特殊化されたメンバー関数を詳細に説明します。これらの関数は、iostream クラスから派生した一連のクラスで実装され、文字型配列処理用に特殊化されています。

iostreamの用語

iostream ライブラリの説明では、一般のプログラミングに関する用語と同じでも意味が異なる語を多く使用します。次の表では、それらの用語が iostream ライブラリの説明で使用される場合の意味を定義します。

表 3-4 iostream の用語

用語	意味
バッファ	バッファには、2 つの意味があります。1 つは iostream パッケージに固有のバッファで、もう 1 つは入出力一般に適用されるバッファです。 iostream ライブラリに固有のバッファは、streambuf クラスで定義されたオブジェクトです。 一般にいうバッファは、入出力データを効率よく転送するために使用するメモリーブロックを指します。バッファリングされた入出力の場合は、バッファがいっぱいになるか、バッファが強制的にフラッシュされるときまで、データの転送は行われません。 「バッファリングなしのバッファ」とは、上で定義した一般にいうバッファがない streambuf を指します。この章では streambuf を指すバッファという語を使用しないようにしていますが、マニュアルページや他の C++ のマニュアルでは、streambuf の意味でバッファという語を使用しています。
抽出	iostream から入力データを取り出す操作を抽出といいます。
fstream	ファイル用に特殊化された入出力ストリームです。特に courier のようにクーリエフォントで印刷されている場合は、iostream クラスから派生した fstream クラスを指します。
挿入	iostream に出力データを送り込む操作を挿入といいます。
iostream	一般には、入力ストリームまたは出力ストリームです。

表 3-4　iostream の用語 (続き)

用語	意味
iostream ライブラリ	ファイル iostream.h、fstream.h、strstream.h、iomanip.h、ライブラリ stdiostream.h をインクルードすることにより使用できるライブラリです。iostream はオブジェクト指向のライブラリですので、ユーザーが必要に応じて拡張することができます。そのため、iostream ライブラリを使用して実行できるすべての機能があらかじめ定義されているわけではありません。
ストリーム	一般に、iostream、fstream、strstream、またはユーザー定義のストリームをいいます。
streambuf	文字シーケンスの入ったバッファで、put ポインタまたは get ポインタ (またはその両方) を持ちます。 courier のようにクーリエフォントで印刷されている場合は、streambuf という特定のクラスを意味します。その他のフォントで印刷されている場合は一般に streambuf クラスのオブジェクト、または streambuf の派生クラスを意味します。ストリームオブジェクトは必ず、streambuf から派生した型のオブジェクト (またはそのオブジェクトへのポインタ) を持っています。
strstream	文字型配列処理用に特殊化した iostream です。courier のようにクーリエフォントで印刷されている場合は、strstream という特定のクラスを意味します。

第4章

マルチスレッド環境での従来型の iostream ライブラリの使用

本章では、マルチスレッド環境における入出力のために、libC ライブラリと libisotream ライブラリの iostream クラスを使用する方法を説明します。また、iostream クラスから派生クラスを作成して、ライブラリ機能を拡張する例も示します。ただし、ここで説明する内容は、マルチスレッド環境で実行する C++ コードを書くための解説ではありません。

この章の説明は、古い iostream (libC および libiostream) に対してのみ該当し、C++ 標準ライブラリに組み込まれている新しい iostream (libCstd) には該当しません。

マルチスレッド

マルチスレッド (MT) は、マルチプロセッサ上で実行するアプリケーションを高速化する強力な機能です。マルチスレッド機能を使用することにより、マルチプロセッサとシングルプロセッサの両方のアプリケーションの構造を簡素化することができます。iostream ライブラリが修正されたため、マルチスレッド環境で実行するアプリケーション、すなわち Solaris オペレーティング環境の Solaris 2.6、7、8 のいずれかで実行する際にマルチスレッド機能を利用するプログラムで、libC ライブラリのインタフェースを使用することができるようになりました。iostream インタフェースは変更されましたが、旧バージョンのシングルスレッド機能だけを使用するアプリケーションには影響ありません。

マルチスレッド環境で正しく実行するためには、ライブラリが 「MT-安全」として定義されていなければなりません。そのためには、一般にすべての公開関数が再入可能になっていなければなりません。libC ライブラリでは、複数のスレッドに共有され

るオブジェクト (C++ クラスのインスタンス) の状態を変更しようとするマルチスレッドに対して、保護機能が提供されています。ただし、iostream オブジェクトの「MT-安全」のスコープは、オブジェクトの公開メンバー関数の実行中に限られます。

注意 – アプリケーションが、libC ライブラリの「MT-安全」オブジェクトを使用しているというだけで、自動的にマルチスレッド環境での安全性が保証されるわけではありません。アプリケーションが「MT-安全」となるのは、マルチスレッド環境での実行が想定されているアプリケーションの場合だけです。

「MT-安全」の iostream ライブラリの構成

「MT-安全」の iostream ライブラリの構成は、「MT-安全ではない」(マルチスレッド環境では使用できない) iostream ライブラリの構成と少し違います。「MT-安全」の iostream ライブラリから公開されているインタフェースは、iostream クラスとその基底クラスすべての公開および限定公開メンバー関数しか参照しないことは他のバージョンと同じですが、クラス階層が異なります。詳しくは、53 ページの「iostream ライブラリのインタフェースの変更」を参照してください。

本来のコアクラスは、名前に接頭語 unsafe_ が付けられました。表 4-1 に、iostream パッケージのコアクラスを示します。

表 4-1 コアクラス

クラス	内容
stream_MT	MT-安全の基底クラス
streambuf	バッファーの基底クラス
unsafe_ios	エラー状態やフォーマット状態などの、さまざまなストリームクラスに共通の状態変数を含むクラス

表 4-1　コアクラス　(続き)

クラス	内容
unsafe_istream	streambuf から取り出した文字シーケンスのフォーマット付きまたはフォーマットなし変換をサポートするクラス
unsafe_ostream	streambuf に格納される文字シーケンスのフォーマット付きまたはフォーマットなし変換をサポートするクラス
unsafe_iostream	入出力両方向の操作のために unsafe_istream クラスと unsafe_ostream クラスを結合するクラス

「MT-安全」の各クラスは、基底クラス stream_MT から派生します。streambuf を除く「MT-安全」クラスもまた、すべて名前が unsafe_ で始まる既存の基底クラスから派生します。次に例を示します。

```
class streambuf: public stream_MT { ... };
class ios: virtual public unsafe_ios, public stream_MT { ... };
class istream: virtual public ios, public unsafe_istream { ... };
```

stream_MT クラスは、それぞれの iostream クラスを「MT-安全」にするために必要な相互排他ロック機能を提供します。また動的にロックとロック解除を行う機能も提供し、「MT-安全」特性の動的変更を可能にします。入出力変換とバッファー管理の基本機能は、名前が unsafe_ で始まるクラスにまとめられており、「MT-安全」にするためにライブラリに追加した機能は、その派生クラスに限られます。「MT-安全」の各クラスに入っている公開および限定公開メンバー関数は、名前が unsafe_ で始まる基底クラスと同じです。「MT-安全」のクラスの各メンバー関数はラッパーとして働き、オブジェクトのロック後、名前が unsafe_ で始まる基底クラスの同じ関数を呼び出し、最後にそのオブジェクトのロックを解除します。

注 – streambuf クラスは、名前が unsafe_ で始まるクラスから派生したクラスではありません。streambuf クラスの公開および限定公開メンバー関数は、ロック機能により再入可能になります。_unlocked という接尾辞の付いたロックなしのものも提供されています。

公開変換ルーチン

iostream インタフェースには、マルチスレッド環境で使用しても安全で、かつ再入可能な公開関数がいくつか追加されています。各関数の追加引数として、ユーザー定義バッファーを指定します。次に関数の内容を示します。

表 4-2　再入可能な公開関数

関数	内容
`char *oct_r (char *buf, int buflen, long num, int width)`	数値を 8 進表示した ASCII 文字列へのポインタを返します。幅がゼロ以外の場合は、フォーマット用のフィールド幅とみなされます。戻り値がユーザー提供バッファーの先頭を指していることは保証されません。
`char *hex_r (char *buf, int buflen, long num, int width)`	数値を 16 進表示した ASCII 文字列へのポインタを返します。幅がゼロ以外の場合は、フォーマット用のフィールド幅とみなされます。戻り値がユーザー提供バッファーの先頭を指していることは保証されません。
`char *dec_r (char *buf, int buflen, long num, int width)`	数値を 10 進表示した ASCII 文字列へのポインタを返します。幅がゼロ以外の場合は、フォーマット用のフィールド幅とみなされます。戻り値がユーザー提供バッファーの先頭を指していることは保証されません。
`char *chr_r (char *buf, int buflen, int chr, int width)`	文字列 chr が含まれた ASCII 文字列へのポインタを返します。幅がゼロ以外の場合は、文字列には width 個のスペースに続いて chr が含まれます。戻り値がユーザー提供バッファーの先頭を指していることは保証されません。
`char *form_r (char *buf, int buflen, const char *form,...)`	フォーマット文字列 format に続く引数を sprintf でフォーマットした文字列へのポインタを返します。バッファーには、フォーマット済み文字列を入れるための十分な長さが必要です。

注意 – 旧バージョンの libC ライブラリとの互換性を保つために提供されている iostream ライブラリの公開変換ルーチン (oct、hex、dec、chr、form) は、「MT-安全」ではありません。

「MT-安全」の libC ライブラリのコンパイルとリンク

マルチスレッド環境で実行するために、libC ライブラリの iostream クラスを使用して作成したアプリケーションについては、ソースコードのコンパイルとリンク時に -mt オプションを指定する必要があります。このオプションを指定すると、プリプロセッサ (前処理部) には -D_REENTRANT が、リンカーには -lthread が渡されます。

注 – libC および libthread とのリンクには、-lthread ではなく -mt を使用してください。-mt オプションを使用すると、ライブラリの正しい順番でのリンクが保証されます。誤って -lthread オプションを使用すると、そのアプリケーションは正しく動かないことがあります。

---

iostream クラスを使用するシングルスレッド用のアプリケーションには特別なコンパイラやリンカーオプションは必要ありません。デフォルトで、コンパイラは libC ライブラリとリンクします。

「MT-安全」の iostream ライブラリの制限

iostream ライブラリのマルチスレッドに対する安全性に制限があるのは、iostream ライブラリで使用しているプログラミング手法の多くが、共有 iostream オブジェクトを使用するマルチスレッド環境では正しく実行できないためです。

エラー状態の検査

「MT-安全」にするには、次の例のように、エラーを起こすような入出力操作を伴う危険領域では、入出力のエラーを調べる必要があります。

コード例 4-1　エラー状態の検査

```
#include <iostream.h>
#include <rlocks.h>
enum iostate { IOok, IOeof, IOfail };

iostate read_number(istream& istr, int& num)
{
    stream_locker sl(istr, stream_locker::lock_now);

    istr >> num;

    if (istr.eof()) return IOeof;
```

コード例 4-1　エラー状態の検査 (続き)

```
    if (istr.fail()) return IOfail;
    return IOok;
}
```

この例では、stream_locker オブジェクトである sl のコンストラクタが istream オブジェクトの istr をロックしています。sl のデストラクタは、read_number の終了時に呼ばれ、istr のロックを解除します。

最後のフォーマットなし入力操作で抽出された文字の取得

「MT-安全」にするには、最も近いフォーマットなし入出力とそれに続く gcount 呼び出しの間に、iostream オブジェクトを排他的に使用するスレッド内で gcount 関数の呼び出しを行う必要があります。次の例を参照してください。

コード例 4-2　gcount の呼び出し

```
#include <iostream.h>
#include <rlocks.h>
void fetch_line(istream& istr, char* line, int& linecount)
{
    stream_locker sl(istr, stream_locker::lock_defer);

    sl.lock(); // ストリーム istr をロック
    istr >> line;
    linecount = istr.gcount();
    sl.unlock(); // istr のロック解除
    ...
}
```

この例では、stream_locker クラスの lock メンバー関数および unlock メンバー関数がプログラム中で相互排他領域を定義しています。

ユーザー定義の入出力操作

「MT-安全」にするには、ユーザー定義型に対して定義された入出力操作で特定の操作順序を持つものは、ロックして危険領域を定義する必要があります。次の例を参照してください。

コード例 4-3　ユーザー定義の入出力操作

```
#include <rlocks.h>
#include <iostream.h>
class mystream: public istream {

    // その他の定義...
    int getRecord(char* name, int& id, float& gpa);
}

#include <rlocks.h>
#include <iostream.h>
int mystream::getRecord(char* name, int& id, float& gpa)
{
    stream_locker sl(this, stream_locker::lock_now);

    *this >> name;
    *this >> id;
    *this >> gpa;
    return this->fail() == 0;
}
```

パフォーマンス

現バージョンの libC ライブラリの「MT-安全」のクラスを使用すると、シングルスレッドのアプリケーションの場合でもパフォーマンスのオーバーヘッドが起こります。libC ライブラリの unsafe_ クラスを使用すると、オーバーヘッドはなくなります。

次の例のように、スコープ決定演算子を使用して unsafe_ 基底クラスのメンバー関数を実行することができます。

```
cout.unsafe_ostream::put('4');
```

```
cin.unsafe_istream::read(buf, len);
```

注 – マルチスレッド対応のアプリケーションでは、unsafe_ クラスの使用は危険です。

次の例のように、unsafe_ クラスを使用する代わりに、cout オブジェクトと cin オブジェクトを「MT-安全ではない」として、通常の操作を使用することもできます。この場合、パフォーマンスはわずかに低下します。

コード例 4-4 「MT-安全」の無効化

```
#include <iostream.h>
//「MT-安全ではない」に設定
cout.set_safe_flag(stream_MT::unsafe_object);
//「MT-安全ではない」に設定
cin.set_safe_flag(stream_MT::unsafe_object);
cout.put('4');
cin.read(buf, len);
```

iostream オブジェクトが「MT-安全」である場合は、相互排他ロック機能が提供されてオブジェクトのメンバー変数が保護されます。このロック機能のため、シングルスレッド環境でのみ実行されるアプリケーションの場合は不要なオーバーヘッドがかかります。次の例のように、iostream オブジェクトの「MT-安全」と「MT-安全ではない」を動的に切り換えて、パフォーマンスを向上させることができます。

コード例 4-5 「MT-安全ではない」の切り換え

```
fs.set_safe_flag(stream_MT::unsafe_object);//「MT-安全ではない」に設定
... さまざまな入出力操作を実行
```

iostream が複数スレッドに共有されない場合、たとえばスレッドが 1 つしかないプログラムや、各 iostream がそれぞれのスレッドに対して非公開なプログラムの場合は、「MT-安全ではない」のストリームを使用しても問題ありません。

また次の例のように、プログラム内で明示的に同期をとらせる場合は、iostream が複数スレッドに共有される環境で「MT-安全ではない」の iostream を使用しても問題ありません。

コード例 4-6 「MT-安全ではない」のオブジェクトを使用して同期をとる方法

```
generic_lock() ;
fs.set_safe_flag(stream_MT::unsafe_object) ;
... さまざまな入出力操作を実行
generic_unlock() ;
```

ここで、generic_lock と generic_unlock の関数は、相互排他、セマフォ、読み取りと書き込みのような単純型を使用した同期機構を提供するものであれば何でもかまいません。

注 – このような目的で使用するには、libC ライブラリで提供されている stream_locker クラスをお勧めします。

詳細は、57 ページの「オブジェクトのロック」を参照してください。

iostream ライブラリのインタフェースの変更

この項では、libC ライブラリを「MT-安全」にするために、iostream ライブラリのインタフェースがどのように変更されたかを説明します。

新しいクラス

次の表は、libC インタフェースに新しく追加されたクラスのリストです。

コード例 4-7　新しいクラス

```
stream_MT
stream_locker
unsafe_ios
unsafe_istream
unsafe_ostream
unsafe_iostream
unsafe_fstreambase
unsafe_strstreambase
```

新しいクラス階層

次の表は、iostream インタフェースに新しく追加されたクラス階層のリストです。

コード例 4-8　新しいクラス階層

```
class streambuf : public stream_MT { ... };
class unsafe_ios { ... };
class ios : virtual public unsafe_ios, public stream_MT  {...};
class unsafe_fstreambase : virtual public unsafe_ios { ... };
```

コード例 4-8　新しいクラス階層 (続き)

```
class fstreambase : virtual public ios, public unsafe_fstreambase
{ ... };
class unsafe_strstreambase : virtual public unsafe_ios { ... };
class strstreambase : virtual public ios, public
unsafe_strstreambase { ... };
class unsafe_istream : virtual public unsafe_ios  { ... };
class unsafe_ostream : virtual public unsafe_ios  { ... };

class istream : virtual public ios, public unsafe_istream { ... };
class ostream : virtual public ios, public unsafe_ostream { ... };
class unsafe_iostream : public unsafe_istream, public
unsafe_ostream { ... };
```

新しい関数

次の表は、iostream インタフェースに新しく追加された関数のリストです。

コード例 4-9　新しい関数

```
class streambuf {
public:
    int sgetc_unlocked();
    void sgetn_unlocked(char *, int);
    int snextc_unlocked();
    int sbumpc_unlocked();
    void stossc_unlocked();
    int in_avail_unlocked();
    int sputbackc_unlocked(char);
    int sputc_unlocked(int);
    int sputn_unlocked(const char *, int);
    int out_waiting_unlocked();
protected:
    char* base_unlocked();
    dhar* ebuf_unlocked();
    int blen_unlocked();
    char* pbase_unlocked();
    char* eback_unlocked();
    char* gptr_unlocked();
    char* egptr_unlocked();
    char* pptr_unlocked();
    void setp_unlocked(char*, char*);
    void setg_unlocked(char*, char*, char*);
    void pbump_unlocked(int);
    void gbump_unlocked(int);
    void setb_unlocked(char*, char*, int);
```

コード例 4-9　新しい関数 (続き)

```
    int unbuffered_unlocked();
    char *epptr_unlocked();
    void unbuffered_unlocked(int);
    int allocate_unlocked(int);
};

class filebuf : public streambuf {
public:
    int is_open_unlocked();
    filebuf* close_unlocked();
    filebuf* open_unlocked(const char*, int, int=filebuf::openprot);

    filebuf* attach_unlocked(int);
};

class strstreambuf : public streambuf {
public:
    int freeze_unlocked();
    char* str_unlocked();
};

unsafe_ostream& endl(unsafe_ostream&);
unsafe_ostream& ends(unsafe_ostream&);
unsafe_ostream& flush(unsafe_ostream&);
unsafe_istream& ws(unsafe_istream&);
unsafe_ios& dec(unsafe_ios&);
unsafe_ios& hex(unsafe_ios&);
unsafe_ios& oct(unsafe_ios&);

char* dec_r (char* buf, int buflen, long num, int width)
char* hex_r (char* buf, int buflen, long num, int width)
char* oct_r (char* buf, int buflen, long num, int width)
char* chr_r (char* buf, int buflen, long chr, int width)
char* str_r (char* buf, int buflen, const char* format, int width = 0);
char* form_r (char* buf, int buflen, const char* format, ...)
```

大域データと静的データ

マルチスレッド対応のアプリケーションでは、大域データと静的データをスレッド間で安全に共有することができません。各スレッドは独立して実行されますが、大域データと静的データへのアクセスはプロセス内のスレッド間で共有されています。そのような共有オブジェクトをあるスレッドが変更すると、プロセス内のその他の全スレッドがその変更の影響を受け、必要な間同じ状態を保つことが期待できなくなります。C++ では、クラスオブジェクト (クラスのインスタンス) の状態はメンバー変数の値で示されます。したがって、そのクラスオブジェクトが共有されていれば、他のスレッドからも変更されてしまいます。

マルチスレッド対応のアプリケーションで iostream ライブラリを使用し、iostream.h をインクルードしている場合、標準ストリーム (cout、cin、cerr、clog) は、デフォルトで大域的な共有オブジェクトとして定義されます。iostream ライブラリは「MT-安全」なので、iostream オブジェクトのメンバー関数の実行中は、共有オブジェクトの状態は他のスレッドからのアクセスや変更から保護されています。ただし iostream オブジェクトの「MT-安全」が有効なのは、そのオブジェクトの公開メンバー関数の実行中に限られます。次の例を考えてみましょう。

```
int c;
cin.get(c);
```

このコードでスレッド A が get バッファから次の文字を取り出し、スレッド A のバッファーポインタを更新します。ところが、スレッド A の次の命令がやはり get の呼び出しであっても、文字シーケンスからその次の文字が取り出されるという保証は libC ライブラリにおいてはありません。なぜなら、スレッド A による 2 つの get の呼び出しの間に、たとえばスレッド B が get の呼び出しを行う可能性があるからです。

共有オブジェクトとマルチスレッドにおけるこのような問題を解決する方法については、57 ページの「オブジェクトのロック」を参照してください。

順次実行

iostream オブジェクトを使用すると、一連の入出力操作を「MT-安全」にしなければならない状況が頻繁に起こります。次のコードを例に考えてみましょう。

```
cout << " Error message:" << errstring[err_number] << "\n";
```

このコードには、cout ストリームオブジェクトの 3 つのメンバー関数の実行が含まれています。 cout は共有オブジェクトですから、このコードをマルチスレッド環境で正しく実行するには、一連の入出力操作が危険領域として不可分命令的に実行されなければなりません。iostream クラスオブジェクトの一連の入出力操作を不可分命令的に実行するには、なんらかのロックを行う必要があります。

libC ライブラリでは、iostream オブジェクトの操作をロックするための stream_locker クラスが提供されています。stream_locker クラスについては、次の「オブジェクトのロック」を参照してください。

オブジェクトのロック

共有オブジェクトとマルチスレッドで起こる問題の最も簡単な解決方法は、iostream オブジェクトを 1 つのスレッドに対して局所的にして問題自体をなくしてしまうことです。そのためには、次のような方法があります。

- オブジェクトをスレッドのエントリ関数内で局所的に宣言します。
- オブジェクトをスレッド固有データ内で宣言します。スレッド固有データの使用方法については、thr_keycreate(3T) のマニュアルページを参照してください。
- ストリームオブジェクトを特定のスレッド専用にします。オブジェクトのスレッドは規則により非公開になります。

ところが多くの場合 (たとえばデフォルトの共有標準ストリームオブジェクトの場合)、オブジェクトを特定のスレッドの局所オブジェクトとすることが不可能で、他の解決策を探さなければなりません。

iostream クラスのオブジェクトに対する一連の操作を不可分命令的に実行するには、なんらかのロックを使用する必要があります。ロックを行うと、シングルスレッド用のアプリケーションの場合でもいくらかオーバーヘッドが起こります。ロックを使用するか、あるいは、iostream オブジェクトをスレッドの非公開オブジェクトとするかは、アプリケーションで採用しているスレッドモデルによります。

各スレッドが独立の場合と、複数スレッドが共同作業を行う場合

- それぞれの独立したスレッドが自分自身の iostream オブジェクトを使用してデータの入出力を行う場合は、各 iostream オブジェクトをそれぞれのスレッドの非公開オブジェクトとし、ロックを使用する必要はありません。
- 複数スレッドで共同作業を行う場合 (すなわち同一の iostream オブジェクトを共有しなければならない場合) は、共有オブジェクトへのアクセスは同期をとって行う必要があり、一連の操作を不可分命令的に行うためのなんらかのロックを使用する必要があります。

stream_locker クラス

iostream ライブラリでは、iostream オブジェクトに対する一連の操作をロックするための stream_locker クラスが提供されています。したがって、iostream のロックとロック解除を動的に設定することによるオーバーヘッドを最小にすることができます。

stream_locker クラスのオブジェクトを使用すると、ストリームオブジェクトに対する一連の操作を不可分命令的に実行することができます。たとえば、次の例では、ファイル内の位置を指定して、そこからデータを 1 ブロック読み込みます。

コード例 4-10 ロック操作を使用する例

```
#include <fstream.h>
#include <rlocks.h>
void lock_example (fstream& fs)
{
    const int len = 128;
    char buf[len];
    int offset = 48;
    stream_locker s_lock(fs, stream_locker::lock_now);
    . . . . .// ファイルをオープン
    fs.seekg(offset, ios::beg);
    fs.read(buf, len);
}
```

この例では、stream_locker オブジェクトのコンストラクタが相互排他制御域の開始を定義します。相互排他制御域では、一度に 1 つのスレッドしか実行されません。また、関数から戻った後で呼び出されるデストラクタでは、相互排他制御域の終了を定義します。したがって、stream_locker オブジェクトにより、ファイル内の特定の位置のシークと、ファイルからのデータの読み込みとが不可分命令的に実行されます。スレッド A がファイルからデータを読み込む前に、スレッド B がファイル内の位置を変えてしまうことができなくなるためです。

stream_locker オブジェクトのもう 1 つの使用方法として、相互排他制御域を明示的に定義する方法があります。次の例では、入出力操作とそれに続くエラー検査を不可分命令的に実行するため、stream_locker オブジェクトのメンバー関数 lock と unlock を呼び出します。

コード例 4-11 入出力操作とエラー検査を不可分命令的に実行

```
{
    ...
    stream_locker file_lck(openfile_stream,
                           stream_locker::lock_defer);
    ....
    file_lck.lock();  // openfile_stream をロック
    openfile_stream << "Value: " << int_value << "\n";
    if(!openfile_stream) {
    file_error("Output of value failed\n");
    return;
    }
    file_lck.unlock();  // openfile_stream のロックを解除
}
```

stream_locker についての詳細は、stream_locker(3CC4) のマニュアルページを参照してください。

「MT-安全」のクラス

iostream クラスから新たなクラスを派生させて、機能を拡張したり特殊化したりすることができます。派生クラスのオブジェクトをマルチスレッド環境で使用するときは、派生クラスも「MT-安全」にする必要があります。

「MT-安全」のクラスを派生するときには、次の点を考慮してください。

- クラスオブジェクトを「MT-安全」にするには、オブジェクトの内部状態が複数のスレッドから変更されないようにします。そのためには、公開になっているメンバー変数へのアクセス、および相互排他ロックによって保護されているメンバー関数へのアクセスが直列になるようにします。
- 「MT-安全」の基底クラスのメンバー関数に対する呼び出し順を不可分にするには、stream_locker オブジェクトを使用します。
- ロックに伴うオーバーヘッドを避けるには、stream_locker オブジェクトで定義された危険領域内では、streambuf のメンバー関数のうち _unlocked のものを使用します。
- streambuf クラスの公開仮想関数がアプリケーションから直接呼び出される場合は、それらの関数をロックします 。そのような関数としては、xsgetn、underflow、pbackfail、 xsputn、overflow、seekoff、seekpos があります。
- ios オブジェクトのフォーマット状態を拡張するには、ios クラスのメンバー関数 iword と pword を使用します。ただし、複数のスレッドが関数 iword または pword への同じインデックスを共有する場合は問題が起こります。複数のスレッドを「MT-安全」にするには、適切なロックを使用してください。
- char 型よりサイズが大きいメンバー変数の値を返すようなメンバー関数は、ロックする必要があります。

オブジェクトの破棄

複数のスレッドで共有する iostream オブジェクトを削除するときは、すべての副スレッドがそのオブジェクトの使用を終了していることを、主スレッドで確認する必要があります。次の例で、共有オブジェクトを安全に破棄する方法を示します。

コード例 4-12 共有オブジェクトの破棄

```
#include <fstream.h>
#include <thread.h>
fstream* fp;

void *process_rtn(void*)
{
    // fp を使用する副スレッドの本体...
}
```

コード例 4-12 共有オブジェクトの破棄 (続き)

```
multi_process(const char* filename, int numthreads)
{

  fp = new fstream(filename, ios::in);    // スレッド生成の前に
    // fstream オブジェクトを生成
    // スレッドの生成
    for (int i=0; i<numthreads; i++)
    thr_create(0, STACKSIZE, process_rtn, 0, 0, 0);


    // 全スレッドの終了待ち
    for (int i=0; i<numthreads; i++)
    thr_join(0, 0, 0);

    delete fp;                              // 全スレッドの終了後に
    fp = NULL;                              // fstream オブジェクトを破棄
}
```

アプリケーションの

次の例では、複数のスレッドを使用するアプリケーションの例を示します。この例では、libC ライブラリの iostream オブジェクトを「MT-安全」で使用します。

このアプリケーションでは、スレッドを 255 個まで作成します。各スレッドはそれぞれ異なる入力ファイルから一度に 1 行ずつデータを読み込み、そのデータを出力ファイルに書き込みます。そのとき標準出力ストリーム cout を使用します。出力ファイルは全スレッドで共有するため、どのスレッドから出力されたかを示すタグを付けます。

コード例 4-13 iostream オブジェクトを「MT-安全」で使用

```
// タグ付きのスレッドデータを生成。
// 出力ファイルの形式は <タグ><データの文字列>\n
// <タグ> は符号なしの char 型の整数値を示す。
// このアプリケーションで実行できるスレッドは 255 個まで。
// <データの文字列> は出力可能な任意の文字列を示す
// <タグ> は char 型として記述された整数値なので、
// 出力ファイルの内容を見るには以下のように od を使用する。
//       od -c out.file |more
```

コード例 4-13 iostream オブジェクトを「MT-安全」で使用 (続き)

```
#include <stdlib.h>
#include <stdio.h>
#include <iostream.h>
#include <fstream.h>
#include <thread.h>

struct thread_args {
  char* filename;
  int thread_tag;

};

const int thread_bufsize = 256;

// 各スレッドのエントリルーチン
void* ThreadDuties(void* v) {
// 現スレッドの引数の取得
  thread_args* tt = (thread_args*)v;
  char ibuf[thread_bufsize];
  // スレッドの入力ファイルをオープン
  ifstream instr(tt->filename);
  stream_locker lockout(cout, stream_locker::lock_defer);
  while(1) {
  // 一度に 1 行を読み込み
    instr.getline(ibuf, thread_bufsize - 1, '\n');
    if(instr.eof())
      break;

// 入出力操作を不可分命令的に実行するため、 cout ストリームをロック
    lockout.lock();
  // データにタグを付けて cout に出力
    cout << (unsigned char)tt->thread_tag << ibuf << "\n";
    lockout.unlock();
  }
  return 0;
}

main(int argc, char** argv) {
  // argv: 1+ 各スレッドのファイル名リスト
   if(argc < 2) {
     cout << "usage: " << argv[0] << " <files..>\n";
     exit(1);
   }
  int num_threads = argc - 1;
  int total_tags = 0;
```

コード例 4-13 iostream オブジェクトを「MT-安全」で使用 (続き)

```
// スレッド ID の配列
  thread_t created_threads[thread_bufsize];
// スレッドのエントリルーチンへの引数の配列
  thread_args thr_args[thread_bufsize];
  int i;
  for( i = 0; i < num_threads; i++) {
    thr_args[i].filename = argv[1 + i];
// スレッドへのタグの割り当て — タグの値は 255 以下
    thr_args[i].thread_tag = total_tags++;

// スレッドの生成
    thr_create(0, 0, ThreadDuties, &thr_args[i],
           THR_SUSPENDED, &created_threads[i]);
  }

  for(i = 0; i < num_threads; i++) {
    thr_continue(created_threads[i]);
  }
  for(i = 0; i < num_threads; i++) {
    thr_join(created_threads[i], 0, 0);
  }
  return 0;
}
```

第5章

C++ 標準ライブラリ

デフォルトのモード (標準モード) でコンパイルする場合、コンパイラは C++ 標準で指定されている完全なライブラリにアクセスします。標準ライブラリには、一般に STL (Standard Template Library) として知られているものに加えて、次のようなものが含まれています。

- 文字列クラス
- 数値クラス
- 標準バージョンのストリーム入出力クラス
- 基本的なメモリー割り当て
- 例外クラス
- 実行時型情報

STL という用語は正式には定義されていません。しかし、コンテナ、反復子、およびアルゴリズムを含むと通常は理解されています。標準ライブラリヘッダーの次のサブセットは STL であると考えられています。

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

C++ 標準ライブラリ (libCstd) は RogueWave™ 標準 C++ ライブラリのバージョン2に基づいています。このライブラリはコンパイラのデフォルトのモード (-compat=5) だけで使用でき、互換モード (-compat または -compat=4) ではサポートされていません。

コンパイラに付属されているバージョンではなく、独自のバージョンの C++ 標準ライブラリを使用する必要がある場合、-library=no%Cstd オプションを指定します。ただし、コンパイラ付属の標準ライブラリを別のものに置き換えることは危険で、よい結果が保証されるとは限りません。詳細は、『C++ ユーザーズガイド』のライブラリの使い方に関する章を参照してください。

標準ライブラリについての詳細は、『標準 C++ ライブラリ・ユーザーズガイド』と『Standard C++ Class Library Reference』(英語) を参照してください。これらの文書にアクセスする方法については、「はじめに」の「関連マニュアル」を参照してください。C++ 標準ライブラリに関する文書については、「はじめに」の「市販の書籍」を参照してください。

C++ 標準ライブラリのヘッダーファイル

表 5-1 に、すべての標準ライブラリのヘッダーと簡単な説明を示します。

表 5-1 C++ 標準ライブラリのヘッダーファイル

ヘッダーファイル	説明
<algorithm>	コンテナに適用できる標準のアルゴリズム
<bitset>	固定サイズのビットシーケンス
<complex>	複素数を表す数値型
<deque>	両端での追加と削除をサポートするシーケンス
<exception>	事前定義されている例外クラス
<fstream>	ファイルへのストリーム入出力
<functional>	関数オブジェクト
<iomanip>	iostream マニピュレータ
<ios>	iostream 基底クラス
<iosfwd>	iostream クラスの前方宣言
<iostream>	基本的なストリーム入出力機能

表 5-1　C++ 標準ライブラリのヘッダーファイル

ヘッダーファイル	説明
<istream>	入力ストリーム
<iterator>	シーケンスを反復するためのクラス
<limits>	数値型の属性
<list>	順位付けされたシーケンス
<locale>	国際化のサポート
<map>	キーと値の組み合わせを扱う連想コンテナ
<memory>	特別なメモリーアロケータ
<new>	基本的なメモリー割り当てと解放
<numeric>	汎用数値演算
<ostream>	出力ストリーム
<queue>	先頭での追加と終端での削除をサポートするシーケンス
<set>	一意なキーを扱う連想コンテナ
<sstream>	ソースまたはシンクとしてメモリー内文字列を使用するストリーム入出力
<stack>	始端での追加と削除をサポートするシーケンス
<stdexcept>	追加の標準例外クラス
<streambuf>	iostream 用のバッファクラス
<string>	文字のシーケンス
<typeinfo>	実行時型識別情報
<utility>	比較演算子
<valarray>	数値プログラミングに便利な値配列
<vector>	ランダムアクセスをサポートするシーケンス

5.2 C++ 標準ライブラリのマニュアルページ

表 5-2 に、標準ライブラリの各構成要素に関連するマニュアルページを示します。

表 5-2 C++ 標準ライブラリ用のマニュアルページ

マニュアルページ	概要
Algorithms	コンテナとシーケンスに各種処理を行うための汎用アルゴリズム
Associative_Containers	特定の順序で並んだコンテナ
Bidirectional_Iterators	読み書きの両方が可能で、順方向、逆方向にコンテナをたどることができる反復子
Containers	標準テンプレートライブラリ (STL) コレクション
Forward_Iterators	読み書きの両方が可能な順方向反復子
Function_Objects	operator() が定義済みのオブジェクト
Heap_Operations	make_heap、pop_heap、push_heap、sort_heap を参照
Input_Iterators	読み取り専用の順方向反復子
Insert_Iterators	反復子がコンテナ内の要素を上書きせずにコンテナに挿入することを可能にする、反復子アダプタ
Iterators	コレクションをたどったり、変更したりするためのポインタ汎用化機能
Negators	述語関数オブジェクトの意味を逆にするための関数アダプタと関数オブジェクト
Operators	C++ 標準テンプレートライブラリ出力用の演算子
_Iterators	書き込み専用の順方向反復子
Predicates	ブール値 (真偽) または整数値を返す関数または関数オブジェクト
Random_Access_Iterators	コンテナの読み取りと書き込みをして、コンテナにランダムアクセスすることを可能にする反復子
Sequences	一群のシーケンスをまとめたコンテナ

表 5-2　C++ 標準ライブラリ用のマニュアルページ (続き)

マニュアルページ	概要
Stream_Iterators	汎用アルゴリズムをストリームに直接に使用することを可能にする、ostream と istream 用の反復子機能を含む
__distance_type	反復子が使用する距離のタイプを決定する (廃止予定)
__iterator_category	反復子が属するカテゴリを決定する (廃止予定)
__reverse_bi_iterator	コレクションを逆方向にたどる反復子
accumulate	1 つの範囲内のすべての要素の累積値を求める
adjacent_difference	1 つの範囲内の隣り合う 2 つの要素の差のシーケンスを出力する
adjacent_find	シーケンスから、等しい値を持つ最初の 2 つの要素を検出する
advance	特定の距離で、順方向または逆方向 (使用可能な場合) に反復子を移動する
allocator	標準ライブラリコンテナ内の記憶管理用のデフォルトの割り当てオブジェクト
auto_ptr	単純でスマートなポインタクラス
back_insert_iterator	コレクションの末尾への項目の挿入に使用する挿入反復子
back_inserter	コレクションの末尾への項目の挿入に使用する挿入反復子
basic_filebuf	入力または出力シーケンスをファイルに関連付けるクラス
basic_fstream	1 つのファイル記述子に関連付けられた、複数の指定ファイルまたはその他デバイスに対する読み書きをサポートする
basic_ifstream	1 つのファイル記述子に関連付けられた、複数の指定ファイルまたはその他デバイスに対する読み書きをサポートする
basic_ios	すべてのストリームが共通に必要とする関数を含む基底クラス

表 5-2　C++ 標準ライブラリ用のマニュアルページ (続き)

マニュアルページ	概要
basic_iostream	ストリームバッファが制御する文字シーケンスの書式設定と解釈をサポートする
basic_istream	ストリームバッファが制御するシーケンスからの入力の読み取りと解釈をサポートする
basic_istringstream	メモリー上の配列からの basic_string<chartT,traits,Allocator> クラスのオブジェクトの読み取りをサポートする
basic_ofstream	1 つのファイル記述子に関連付けられた、複数の指定ファイルまたはその他デバイスに対する書き込みをサポートする
basic_ostream	ストリームバッファが制御するシーケンスに対する出力の書式設定と書き込みをサポートする
basic_ostringstream	basic_string<chartT,traits,Allocator> クラスのオブジェクトの書き込みをサポートする
basic_streambuf	各種のストリームバッファーを派生させて、文字シーケンスを制御しやすいようにする抽象基底クラス
basic_string	文字に似た要素シーケンスを処理するためのテンプレート化されたクラス
basic_stringbuf	入力または出力シーケンスを任意の文字シーケンスに関連付ける
basic_stringstream	メモリー上の配列に対する basic_string<chartT,traits,Allocator> クラスのオブジェクトの書き込みと読み取りをサポートする
binary_function	2 項関数オブジェクトを作成するための基底クラス
binary_negate	2 項判定子の結果の補数を返す関数オブジェクト
binary_search	コンテナ上の値について 2 等分検索を行う
bind1st	関数オブジェクトに値を結合するためのテンプレート化されたユーティリティ
bind2nd	関数オブジェクトに値を結合するためのテンプレート化されたユーティリティ

表 5-2　C++ 標準ライブラリ用のマニュアルページ (続き)

マニュアルページ	概要
binder1st	関数オブジェクトに値を結合するための テンプレート化されたユーティリティ
binder2nd	関数オブジェクトに値を結合するための テンプレート化されたユーティリティ
bitset	固定長のビットシーケンスを格納、操作するため のテンプレートクラスと関数
cerr	<cstdio> で宣言されたオブジェクトの stderr に 関連付けられたバッファリングなしストリーム バッファに対する出力を制御する
char_traits	basic_string コンテナと iostream クラス用の 型と演算を持つ特性 (traits) クラス
cin	<cstdio> で宣言されたオブジェクトの stderr に 関連付けられたストリームバッファからの入力を 制御する
clog	<cstdio> で宣言されたオブジェクトの stderr に 関連付けられたストリームバッファに対する出力 を制御する
codecvt	コード変換ファセット
codecvt_byname	指定ロケールに基づいたコードセット変換分類機 能を含むファセット
collate	文字列照合、比較、ハッシュファセット
collate_byname	文字列照合、比較、ハッシュファセット
compare	真または偽を返す 2 項関数または関数 オブジェクト
complex	C++ 複素数ライブラリ
copy	ある範囲の要素をコピーする
copy_backward	ある範囲の要素をコピーする
count	指定条件を満たすコンテナ内の要素の個数を カウントする
count_if	指定条件を満たすコンテナ内の要素の個数を カウントする

表 5-2　C++ 標準ライブラリ用のマニュアルページ (続き)

マニュアルページ	概要
cout	<cstdio> で宣言されたオブジェクトの stdout に関連付けられたストリームバッファに対する出力を制御する
ctype	文字分類機能を取り込むファセット
ctype_byname	指定ロケールに基づいた文字分類機能を含むファセット
deque	ランダムアクセス反復子と、先頭および末尾の両方での効率的な挿入と削除をサポートするシーケンス
distance	2 つの反復子間の距離を求める
divides	1 つ目の引数を 2 つ目の引数で除算した結果を返す
equal	2 つのある範囲が等しいかどうか比較する
equal_range	並べ替えの順序を崩さずに値を挿入できる最大の二次範囲をコレクションから検出する
equal_to	1 つ目と 2 つ目の引数が等しい場合に真を返す 2 項関数オブジェクト
exception	倫理エラーと実行時エラーをサポートするクラス
facets	複数種類のロケール機能をカプセル化するために使用するクラス群
filebuf	入力または出力シーケンスをファイルに関連付けるクラス
fill	指定された値である範囲を初期化する
fill_n	指定された値である範囲を初期化する
find	シーケンスから値に一致するものを検出する
find_end	シーケンスからサブシーケンスに最後に一致するものを検出する
find_first_of	シーケンスから、別のシーケンスの任意の値に一致するものを検出する
find_if	シーケンスから指定された判定子を満たす値に一致するものを検出する

表 5-2　C++ 標準ライブラリ用のマニュアルページ (続き)

マニュアルページ	概要
for_each	ある範囲のすべての要素に関数を適用する
fpos	iostream クラスの位置情報を保持する
front_insert_iterator	コレクションの先頭に項目を挿入するための挿入反復子
front_inserter	コレクションの先頭に項目を挿入するための挿入反復子
fstream	1 つのファイル記述子に関連付けられた、複数の指定ファイルまたはその他デバイスに対する読み書きをサポートする
generate	値生成クラスによって生成された値でコンテナを初期化する
generate_n	値生成クラスによって生成された値でコンテナを初期化する
get_temporary_buffer	メモリーを処理するためのポインタベースのプリミティブ
greater	1 つ目の引数が 2 つ目の引数より大きい場合に真を返す 2 項関数オブジェクト
greater_equal	1 つ目の引数が 2 つ目の引数より大きいか等しい場合に真を返す 2 項関数オブジェクト
gslice	配列から汎用化されたスライスを表現するために使用される数値配列クラス
gslice_array	valarray から BLAS に似たスライスを表現するために使用される数値配列クラス
has_facet	ロケールに指定ファセットがあるかどうかを判定するための関数テンプレート
ifstream	1 つのファイル記述子に関連付けられた、複数の指定ファイルまたはその他デバイスからの読み取りサポートする
includes	ソートされたシーケンスに対する基本演算セット
indirect_array	valarray から選択された要素の表現に使用される数値配列クラス

表 5-2　C++ 標準ライブラリ用のマニュアルページ (続き)

マニュアルページ	概要
inner_product	2 つの範囲 A および B の内積 (A × B) を求める
inplace_merge	ソートされた 2 つのシーケンスを 1 つにマージする
insert_iterator	コレクションを上書きせずにコレクションに項目を挿入するときに使用する挿入反復子
inserter	コレクションを上書きせずにコレクションに項目を挿入するときに使用する挿入反復子
ios	すべてのストリームが必要とする共通の関数を含む基底クラス
ios_base	メンバーの型を定義して、そのメンバーから継承するクラスのデータを保持する
iosfwd	入出力ライブラリテンプレートクラスを宣言し、そのクラスを wide および tiny 型文字専用にする
isalnum	文字が英字または数字のどちらであるかを判定する
isalpha	文字が英字であるかどうかを判定する
iscntrl	文字が制御文字であるかどうかを判定する
isdigit	文字が 10 進数であるかどうかを判定する
isgraph	文字が図形文字であるかどうかを判定する
islower	文字が英小文字であるかどうかを判定する
isprint	文字が印刷可能かどうかを判定する
ispunct	文字が区切り文字であるかどうかを判定する
isspace	文字が空白文字であるかどうかを判定する
istream	ストリームバッファーが制御する文字シーケンスからの入力の読み取りと解釈をサポートする
istream_iterator	istream に対する反復子機能を持つストリーム反復子
istreambuf_iterator	作成元のストリームバッファーから連続する文字を読み取る

表 5-2　C++ 標準ライブラリ用のマニュアルページ (続き)

マニュアルページ	概要
istringstream	メモリー上の配列からの basic_string<chartT,traits,Allocator> クラスのオブジェクトの読み取りをサポートする
istrstream	メモリー上の配列から文字を読み取る
isupper	文字が英大文字であるかどうかを判定する
isxdigit	文字が 16 進数であるかどうかを判定する
iter_swap	2 つの位置の値を交換する
iterator	基底反復子クラス
iterator_traits	反復子に関する基本的な情報を返す
less	1 つ目の引数が 2 つ目の引数より小さい場合に真を返す 2 項関数オブジェクト
less_equal	1 つ目の引数が 2 つ目の引数より小さいか、等しい場合に真を返す 2 項関数オブジェクト
lexicographical_compare	2 つの範囲を辞書式に比較する
limits	numeric_limits セクションを参照
list	双方向反復子をサポートするシーケンス
locale	多相性を持つ複数のファセットからなる地域対応化クラス
logical_and	1 つ目の 2 つ目の引数が等しい場合に真を返す場合に 2 項関数オブジェクト
logical_not	引数が偽の場合に真を返す単項関数オブジェクト
logical_or	引数のいずれかが真の場合に真を返す 2 項関数オブジェクト
lower_bound	ソートされたコンテナ内の最初に有効な要素位置を求める
make_heap	ヒープを作成する
map	一意のキーを使用してキー以外の値にアクセスする連想コンテナ
mask_array	valarray の選別ビューを提供する数値配列クラス
max	2 つの値の大きい方の値を検出して返す

表 5-2　C++ 標準ライブラリ用のマニュアルページ (続き)

マニュアルページ	概要
max_element	1 つの範囲内の最大値を検出する
mem_fun	大域関数の代わりとしてポインタをメンバー関数に適合させる関数オブジェクト
mem_fun1	大域関数の代わりとしてポインタをメンバー関数に適合させる関数オブジェクト
mem_fun_ref	大域関数の代わりとしてポインタをメンバー関数に適合させる関数オブジェクト
mem_fun_ref1	大域関数の代わりとしてポインタをメンバー関数に適合させる関数オブジェクト
merge	ソートされた 2 つのシーケンスをマージして、3 つ目のシーケンスを作成する
messages	メッセージ伝達ファセット
messages_byname	メッセージ伝達ファセット
min	2 つの値の小さい方の値を検出して返す
min_element	1 つの範囲内の最小値を検出する
minus	1 つ目の引数から 2 つ目の引数を減算した結果を返す
mismatch	2 つのシーケンスの要素を比較して、互いに値が一致しない最初の 2 つの要素を返す
modulus	1 つ目の引数を 2 つ目の引数で除算することによって得られた余りを返す
money_get	入力に対する通貨書式設定ファセット
money_put	出力に対する通貨書式設定ファセット
moneypunct	通貨句読文字ファセット
moneypunct_byname	通貨句読文字ファセット
multimap	キーを使用してコンテナキーでない値にアクセスするための連想コンテナ
multiplies	1 つ目と 2 つ目の引数を乗算した結果を返す 2 項関数オブジェクト
multiset	格納済みのキー値に高速アクセスするための連想コンテナ

表 5-2　C++ 標準ライブラリ用のマニュアルページ (続き)

マニュアルページ	概要
negate	引数の否定値を返す単項関数オブジェクト
next_permutation	並べ替え関数に基づいてシーケンスの内容を連続的に入れ替えたものを生成する
not1	単項述語関数オブジェクトの意味を逆にするための関数アダプタ
not2	単項述語関数オブジェクトの意味を逆にするための関数アダプタ
not_equal_to	1 つ目の引数が 2 つ目の引数と等しくない場合に真を返す 2 項関数オブジェクト
nth_element	コレクションを再編して、ソートで *n* 番目の要素より後になった全要素をその要素より前に、*n* 番目の要素より前の全要素をその要素より後ろにくるようにする
num_get	入力に対する書式設定ファセット
num_put	出力に対する書式設定ファセット
numeric_limits	スカラー型に関する情報を表すためのクラス
numpunct	数値句読文字ファセット
numpunct_byname	数値句読文字ファセット
ofstream	1 つのファイル記述子に関連付けられた、複数の指定ファイルまたはその他デバイスへの書き込みをサポートする
ostream	ストリームバッファーが制御するシーケンスに対する出力の書式設定と書き込みをサポートする
ostream_iterator	ostream と istream に反復子を使用可能にするストリーム反復子
ostreambuf_iterator	作成元のストリームバッファーに連続する文字を書き込む
ostringstream	basic_string<chartT,traits,Allocator> クラスのオブジェクトの書き込みをサポートする
ostrstream	メモリー上の配列に書き込みを行う
pair	異種の値の組み合わせ用テンプレート

表 5-2　C++ 標準ライブラリ用のマニュアルページ (続き)

マニュアルページ	概要
partial_sort	エンティティのコレクションをソートするためのテンプレート化されたアルゴリズム
partial_sort_copy	エンティティのコレクションをソートするためのテンプレート化されたアルゴリズム
partial_sum	ある範囲の値の連続した部分小計を求める
partition	指定述語を満たす全エンティティを、満たさない全エンティティの前に書き込む
permutation	並べ替え関数に基づいてシーケンスの内容を連続的に入れ替えたものを生成する
plus	1 つ目と 2 つ目の引数を加算した結果を返す 2 項関数オブジェクト
pointer_to_binary_function	binary_function の代わりとしてポインタを 2 項関数に適用する関数オブジェクト
pointer_to_unary_function	unary_function の代わりとしてポインタを関数に適用する関数オブジェクトクラス
pop_heap	ヒープの外に最大要素を移動する
prev_permutation	並べ替え関数に基づいてシーケンスの内容を連続的に入れ替えたものを生成する
priority_queue	優先順位付きの待ち行列のように振る舞うコンテナアダプタ
ptr_fun	関数の代わりとしてポインタを関数に適用するときに多重定義される関数
push_heap	ヒープに新しい要素を書き込む
queue	先入れ先出しの待ち行列のように振る舞うコンテナアダプタ
random_shuffle	コレクションの要素を無作為にシャッフルする
raw_storage_iterator	反復子ベースのアルゴリズムが初期化されていないメモリーに結果を書き込めるようにする
remove	目的の要素をコンテナの先頭に移動し、目的の要素シーケンスの終了位置を表す反復子を返す
remove_copy	目的の要素をコンテナの先頭に移動し、目的の要素シーケンスの終了位置を表す反復子を返す

表 5-2　C++ 標準ライブラリ用のマニュアルページ (続き)

マニュアルページ	概要
remove_copy_if	目的の要素をコンテナの先頭に移動し、目的の要素シーケンスの終了位置を表す反復子を返す
remove_if	目的の要素をコンテナの先頭に移動し、目的の要素シーケンスの終了位置を表す反復子を返す
replace	コレクション内の要素の値を置換する
replace_copy	コレクション内の要素の値を置換して、置換後のシーケンスを結果に移動する
replace_copy_if	コレクション内の要素の値を置換して、置換後のシーケンスを結果に移動する
replace_if	コレクション内の要素の値を置換する
return_temporary_buffer	メモリーを処理するためのポインタベースのプリミティブ
reverse	コレクション内の要素を逆順にする
reverse_copy	コレクション内の要素を逆順にしながら、その結果を新しいコレクションにコピーする
reverse_iterator	コレクションを逆方向にたどる反復子
rotate	先頭から中央直前の要素までのセグメントと中央から末尾までの要素のセグメントを交換する
rotate_copy	先頭から中央直前の要素までのセグメントと中央から末尾までの要素のセグメントを交換する
search	値シーケンスから、要素単位で指定範囲の値に等しいサブシーケンスを検出する
search_n	値シーケンスから、要素単位で指定範囲の値に等しいサブシーケンスを検出する
set	一意のキーを扱う連想コンテナ
set_difference	ソートされた差を作成する基本的な集合演算
set_intersection	ソートされた積集合を作成する基本的な集合演算
set_symmetric_difference	ソートされた対称差を作成する基本的な集合演算
set_union	ソートされた和集合を作成する基本的な集合演算

表 5-2　C++ 標準ライブラリ用のマニュアルページ (続き)

マニュアルページ	概要
slice	配列の BLAS に似たスライスを表す数値配列クラス
slice_array	valarray の BLAS に似たスライスを表す数値配列クラス
smanip	パラメータ化されたマニピュレータを実装するときに使用する補助クラス
smanip_fill	パラメータ化されたマニピュレータを実装するときに使用する補助クラス
sort	エンティティのコレクションをソートするためのテンプレート化されたアルゴリズム
sort_heap	ヒープをソートされたコレクションに変換する
stable_partition	各グループ内の要素の相対的な順序を保持しながら、指定判定子を満たす全エンティティを満たさない全エンティティの前に書き込む
stable_sort	エンティティのコレクションをソートするためのテンプレート化されたアルゴリズム
stack	先入れ先出しのスタックのように振る舞うコンテナアダプタ
streambuf	各種のストリームバッファーを派生させて、文字シーケンスを制御しやすいようにする抽象基底クラス
string	basic_string<char,char_trais<char>, allocator<char>> 用の型定義
stringbuf	入力または出力シーケンスを任意の文字シーケンスに関連付ける
stringstream	メモリー上の配列に対する basic_string<chartT,traits,Allocator> クラスのオブジェクトの書き込みおよび読み取りをサポートする
strstream	メモリー上の配列に対する読み取りと書き込みを行う
strstreambuf	入力または出力シーケンスを、要素が任意の値を格納する超小型の文字配列に関連付ける

表 5-2　C++ 標準ライブラリ用のマニュアルページ (続き)

マニュアルページ	概要
swap	値を交換する
swap_ranges	ある位置の値の範囲を別の位置の値と交換する
time_get	入力に対する時刻書式設定ファセット
time_get_byname	指定ロケールに基づいた、入力に対する時刻書式設定ファセット
time_put	入力に対する時刻書式設定ファセット
time_put_byname	指定ロケールに基づいた、入力に対する時刻書式設定ファセット
tolower	文字を小文字に変換する
toupper	文字を大文字に変換する
transform	コレクション内の値の範囲に演算を適用し、結果を格納する
unary_function	単項関数オブジェクトを作成するための基底クラス
unary_negate	単項述語の結果の補数を返す関数オブジェクト
uninitialized_copy	構造構文を使用してある範囲の値を別の位置にコピーするアルゴリズム
uninitialized_fill	コレクション内の値の設定に構造構文アルゴリズムを使用するアルゴリズム
uninitialized_fill_n	コレクション内の値の設定に構造構文アルゴリズムを使用するアルゴリズム
unique	1 つの範囲の値から連続する重複値を削除し、得られた一意の値を結果に書き込む
unique_copy	1 つの範囲の値から連続する重複値を削除し、得られた一意の値を結果に書き込む
upper_bound	ソートされたコンテナ内の最後に有効な値位置を求める
use_facet	ファセットの取得に使用するテンプレート関数
valarray	数値演算用に最適化された配列クラス
vector	ランダムアクセス反復子をサポートするシーケンス

表 5-2　C++ 標準ライブラリ用のマニュアルページ (続き)

マニュアルページ	概要
wcerr	<cstdio> で宣言されたオブジェクトの stderr に関連付けられたバッファリングなしストリームバッファーに対する出力を制御する
wcin	<cstdio> で宣言されたオブジェクトの stdin に関連付けられたストリームバッファーからの入力を制御する
wclog	<cstdio> で宣言されたオブジェクトの stderr に関連付けられたストリームバッファーに対する出力を制御する
wcout	<cstdio> で宣言されたオブジェクトの stderr に関連付けられたストリームバッファーに対する出力を制御する
wfilebuf	入力または出力シーケンスをファイルに関連付ける
wfstream	ファイル記述子に関連付けられた指定ファイルまたはその他デバイスに対する読み取りと書き込みをサポートする
wifstream	ファイル記述子に関連付けられた指定ファイルまたはその他デバイスからの読み取りをサポートする
wios	すべてのストリームが共通に必要とする関数を取り込む基底クラス
wistream	ストリームバッファーが制御するシーケンスからの入力の読み取りと解釈をサポートする
wistringstream	メモリー上の配列からの basic_string<chartT,traits,Allocator> クラスのオブジェクトの読み取りをサポートする
wofstream	メモリー上の配列への basic_string<chartT,traits,Allocator> クラスのオブジェクトの書き込みをサポートする
wostream	ストリームバッファーが制御するシーケンスに対する出力の書式設定と書き込みをサポートする

表 5-2　C++ 標準ライブラリ用のマニュアルページ (続き)

マニュアルページ	概要
wostringstream	basic_string<chartT,traits,Allocator> クラスのオブジェクトの書き込みをサポートする
wstreambuf	各種のストリームバッファーを派生させて、文字シーケンスを制御しやすいようにする抽象基底クラス
wstring	basic_string<wchar_t,char_trais <wchar_t>,allocator<wchar_t>> 用の型定義
wstringbuf	入力または出力シーケンスを任意の文字シーケンスに関連付ける

索引

記号

<< 挿入演算子
  complex, 11
  iostream::, 18, 20
! 演算子、ios::, 21
>> 抽出子演算子
  complex, 11

A

abs、complex::, 8
accumulate のマニュアルページ, 69
acos、complex::, 9
adjacent_difference のマニュアルページ, 69
adjacent_find のマニュアルページ, 69
advance のマニュアルページ, 69
Algorithms のマニュアルページ, 68
<algorithm>、ヘッダーファイル, 66
allocator のマニュアルページ, 69
ang、complex::, 8
asin、complex::, 9
Associative_Containers のマニュアルページ, 68
atan、complex::, 9
auto_ptr のマニュアルページ, 69

B

back_inserter のマニュアルページ, 69
back_insert_iterator のマニュアルページ, 69
badbit、ios::, 21
basic_filebuf のマニュアルページ, 69
basic_fstream のマニュアルページ, 69
basic_ifstream のマニュアルページ, 69
basic_iostream のマニュアルページ, 70
basic_ios のマニュアルページ, 69
basic_istream のマニュアルページ, 70
basic_istringstream のマニュアルページ, 70
basic_ofstream のマニュアルページ, 70
basic_ostream のマニュアルページ, 70
basic_ostringstream のマニュアルページ, 70
basic_streambuf のマニュアルページ, 70
basic_stringbuf のマニュアルページ, 70
basic_stringstream のマニュアルページ, 70
basic_string のマニュアルページ, 70
Bidirectional_Iterators のマニュアルページ, 68
binary_function のマニュアルページ, 70
binary_negate のマニュアルページ, 70
binary_search のマニュアルページ, 70
bind1st のマニュアルページ, 70

bind2nd のマニュアルページ, 70
binder1st のマニュアルページ, 71
binder2nd のマニュアルページ, 71
bitset のマニュアルページ, 71
<bitset>、ヘッダーファイル, 66

C

C++ 標準ライブラリ
  構成要素, 65 ~ 83
cartpol、complex のマニュアルページ, 14
cerr
  iostream::, 16, 56
  マニュアルページ, 71
c_exception、定義, 10
char*、抽出子, 23 ~ 24
char_traits のマニュアルページ, 71
chr、iostream::, 48
chr_r、iostream::, 48
cin
  iostream::, 16, 56
  マニュアルページ, 71
clog
  iostream::, 56
  事前定義 iostreams, 16
  マニュアルページ, 71
codecvt_byname のマニュアルページ, 71
codecvt のマニュアルページ, 71
collate_byname のマニュアルページ, 71
collate のマニュアルページ, 71
compare のマニュアルページ, 71
complex
  エラー処理, 9 ~ 11
  演算子, 7
  型 complex, 6
  効率, 13
  混合モード, 12 ~ 13
  コンストラクタ, 6 ~ 7
  三角関数, 9
  数学関数, 8 ~ 9
  入出力, 11
  複素数ライブラリ, 5 ~ 6
  マニュアルページ, 14, 71
complex()、コンストラクタ, 6 ~ 7
complex.h、complex ヘッダーファイル, 6
complex_error
  定義, 10
  メッセージ, 8
<complex>、ヘッダーファイル, 66
conj、complex::, 8
Containers のマニュアルページ, 68
copy_backward のマニュアルページ, 71
copy のマニュアルページ, 71
cos、complex::, 9
cosh、complex::, 9, 11
count_if のマニュアルページ, 71
count のマニュアルページ, 71
cout
  iostream::, 16, 19, 56
  マニュアルページ, 72
cplx.intro、complex のマニュアルページ, 14
cplxerr、complex のマニュアルページ, 14
cplxexp、complex のマニュアルページ, 14
cplxops、complex のマニュアルページ, 14
ctype_byname のマニュアルページ, 72
ctype のマニュアルページ, 72

D

dec
  iostream::, 48
dec_r、iostream::, 48
deque のマニュアルページ, 72
<deque>、ヘッダーファイル, 66
_distance_type のマニュアルページ, 69
distance のマニュアルページ, 72
divides のマニュアルページ, 72
double、complex 値, 6

E

EDOM、errno 設定, 11
eofbit、ios::, 21
equal_range のマニュアルページ, 72
equal_to のマニュアルページ, 72
equal のマニュアルページ, 72
ERANGE、errno 設定, 11
errno、定義, 10 ~ 11
error、iostream::, 21
exception のマニュアルページ, 72
<exception>、ヘッダーファイル, 66
exp、complex::, 9 ~ 11

F

facets のマニュアルページ, 72
failbit、ios::, 21
filebuf
  streambuf::, 55
  マニュアルページ, 72
fill_n のマニュアルページ, 72
fill のマニュアルページ, 72
find_end のマニュアルページ, 72
find_first_of のマニュアルページ, 72
find_if のマニュアルページ, 72
find のマニュアルページ, 72
float 型挿入子、iostream 出力, 18
for_each のマニュアルページ, 73
form、iostream::, 48
form_r、iostream::, 48
Forward_Iterators のマニュアルページ, 68
fpos のマニュアルページ, 73
front_inserter のマニュアルページ, 73
front_insert_iterator のマニュアルページ, 73
fstream
  iostream::, 17
  マニュアルページ, 73
fstream.h
  iostream ヘッダーファイル, 18
<fstream>、ヘッダーファイル, 66
<functional>、ヘッダーファイル, 66
Function_Objects のマニュアルページ, 68

G

gcInitialize のマニュアルページ, 4
gcmonitor
  Sun WorkShop Memory Monitor デーモン, 3
  マニュアルページ, 4
gcount、istream::, 50
generate_n のマニュアルページ, 73
generate のマニュアルページ, 73
get、char 抽出子, 24
get_temporary_buffer のマニュアルページ, 73
goodbit、ios::, 21
good、ios::, 21
greater_equal のマニュアルページ, 73
greater のマニュアルページ, 73
gslice_array のマニュアルページ, 73
gslice のマニュアルページ, 73

H

hardfail、ios::, 21
has_facet のマニュアルページ, 73
Heap_Operations のマニュアルページ, 68
hex
  iostream::, 48
hex_r、iostream::, 48
HTML 形式のマニュアルページへのアクセス, 2

I

ifstream
  iostream::, 17
  マニュアルページ, 73
imag、complex::, 9
includes のマニュアルページ, 73

indirect_array のマニュアルページ, 73
inner_product のマニュアルページ, 74
inplace_merge のマニュアルページ, 74
Input_Iterators のマニュアルページ, 68
inserter のマニュアルページ, 74
Insert_Iterators のマニュアルページ, 68
insert_iterator のマニュアルページ, 74
iomanip.h、iostream ヘッダーファイル, 18
<iomanip>、ヘッダーファイル, 66
ios_base のマニュアルページ, 74
iosfwd のマニュアルページ, 74
<iosfwd>、ヘッダーファイル, 66
io_state、ios::, 21
iostream
  iostream::, 16
  MT-安全インタフェースの変更, 53
  MT-安全の制限, 49
  新しいインタフェース関数, 54 ~ 56
  新しいクラス階層, 53 ~ 54
  エラービット, 21
  クラスの拡張, 59
  構造, 16 ~ 17
  コンストラクタ, 17
  事前定義, 16
  出力, 18
  出力エラー, 20 ~ 21
  使用方法, 17
  単一スレッドアプリケーション, 49
  入力, 22
  フラッシュ, 22
  ヘッダファイル, 17
  マニュアルページ, 15
  ライブラリ公開変換ルーチン, 48
iostream.h、iostream ヘッダーファイル, 18, 56
iostream ライブラリ, 15
<iostream>、ヘッダーファイル, 66
ios のマニュアルページ, 74
<ios>、ヘッダーファイル, 66
isalnum のマニュアルページ, 74
isalpha のマニュアルページ, 74
iscntrl のマニュアルページ, 74
isdigit のマニュアルページ, 74
isgraph のマニュアルページ, 74
islower のマニュアルページ, 74
isprint のマニュアルページ, 74
ispunct のマニュアルページ, 74
isspace のマニュアルページ, 74
istream
  iostream::, 16
  マニュアルページ, 74
istreambuf_iterator のマニュアルページ, 74
istream_iterator のマニュアルページ, 74
<istream>、ヘッダーファイル, 67
istringstream のマニュアルページ, 75
istrstream
  iostream::, 17
  マニュアルページ, 75
isupper のマニュアルページ, 75
isxdigit のマニュアルページ, 75
_iterator_category のマニュアルページ, 69
Iterators のマニュアルページ, 68
iterator_traits のマニュアルページ, 75
iterator のマニュアルページ, 75
<iterator>、ヘッダーファイル, 67
iter_swap のマニュアルページ, 75
iword、ios::, 60

L

less_equal のマニュアルページ, 75
less のマニュアルページ, 75
lexicographical_compare のマニュアルページ, 75
libC
  新しいクラス, 53
  コアクラス, 46
  互換モード, 15, 18
  コンパイルとリンク、MT-安全, 49
  マルチスレッド環境ライブラリ, 45

libcomplex、互換モード, 5
libCstd、標準モード, 5
libgc_dbg、Sun WorkShop Memory Monitor ライブラリ, 3
libgc、Sun WorkShop Memory Monitor ライブラリ, 3
libiostream
  標準モード, 15, 18
  マルチスレッド環境ライブラリ, 45
limits のマニュアルページ, 75
<limits>、ヘッダーファイル, 67
list のマニュアルページ, 75
<list>、ヘッダーファイル, 67
locale のマニュアルページ, 75
<locale>、ヘッダーファイル, 67
log10、complex::, 9 ~ 11
log、complex::, 9 ~ 11
logical_and のマニュアルページ, 75
logical_not のマニュアルページ, 75
logical_or のマニュアルページ, 75
lower_bound のマニュアルページ, 75
-lthread、アプリケーションのリンク, 49

M

make_heap のマニュアルページ, 75
map のマニュアルページ, 75
<map>、ヘッダーファイル, 67
mask_array のマニュアルページ, 75
math.h、complex ヘッダーファイル, 13
max_element のマニュアルページ, 76
max のマニュアルページ, 75
mem_fun1 のマニュアルページ, 76
mem_fun_ref1 のマニュアルページ, 76
mem_fun_ref のマニュアルページ, 76
mem_fun のマニュアルページ, 76
Memory Monitor の構成要素, 3
<memory>、ヘッダーファイル, 67
merge のマニュアルページ, 76
messages_byname のマニュアルページ, 76
messages のマニュアルページ, 76
min_element のマニュアルページ, 76
minus のマニュアルページ, 76
min のマニュアルページ, 76
mismatch のマニュアルページ, 76
modulus のマニュアルページ, 76
money_get のマニュアルページ, 76
moneypunct_byname のマニュアルページ, 76
moneypunct のマニュアルページ, 76
money_put のマニュアルページ, 76
-mt、iostream::, 49
MT-安全
  アプリケーション, 45
  オブジェクト, 45
  クラス、派生での考慮点, 59
  公開関数, 47
  パフォーマンスのオーバーヘッド, 51 ~ 53
  ライブラリ, 45
multimap のマニュアルページ, 76
multiplies のマニュアルページ, 76
multiset のマニュアルページ, 76

N

negate のマニュアルページ, 77
Negators のマニュアルページ, 68
<new>、ヘッダーファイル, 67
next_permutation のマニュアルページ, 77
norm、complex::, 9
not1 のマニュアルページ, 77
not2 のマニュアルページ, 77
not_equal_to のマニュアルページ, 77
nth_element のマニュアルページ, 77
numeric_limits のマニュアルページ, 77
<numeric>、ヘッダーファイル, 67
num_get のマニュアルページ, 77
numpunct_byname のマニュアルページ, 77
numpunct のマニュアルページ, 77

num_put のマニュアルページ, 77

O

oct
  iostream::, 48
oct_r、iostream::, 48
ofstream
  iostream::, 17
  マニュアルページ, 77
Operators のマニュアルページ, 68
ostream
  iostream::, 16
  マニュアルページ, 77
ostreambuf_iterator のマニュアルページ, 77
ostream_iterator のマニュアルページ, 77
<ostream>、ヘッダーファイル, 67
ostringstream のマニュアルページ, 77
ostrstream
  iostream::, 17
  マニュアルページ, 77
Output_Iterators のマニュアルページ, 68
overflow、streambuf::, 60

P

pair のマニュアルページ, 77
partial_sort_copy のマニュアルページ, 78
partial_sort のマニュアルページ, 78
partial_sum のマニュアルページ, 78
partition のマニュアルページ, 78
pbackfail、streambuf::, 60
peek、istream::, 25
permutation のマニュアルページ, 78
plus のマニュアルページ, 78
pointer_to_binary_function のマニュアルページ, 78
pointer_to_unary_function のマニュアルページ, 78
polar、complex::, 7, 9
pop_heap のマニュアルページ, 78
pow、complex::, 9
Predicates のマニュアルページ, 68
prev_permutation のマニュアルページ, 78
priority_queue のマニュアルページ, 78
private、オブジェクトスレッド, 57
ptr_fun のマニュアルページ, 78
push_heap のマニュアルページ, 78
pword、ios::, 60

Q

queue のマニュアルページ, 78
<queue>、ヘッダーファイル, 67

R

Random_Access_Iterators のマニュアルページ, 68
random_shuffle のマニュアルページ, 78
raw_storage_iterator のマニュアルページ, 78
read、istream::, 25
real、complex::, 9
remove_copy_if のマニュアルページ, 79
remove_copy のマニュアルページ, 78
remove_if のマニュアルページ, 79
remove のマニュアルページ, 78
replace_copy_if のマニュアルページ, 79
replace_copy のマニュアルページ, 79
replace_if のマニュアルページ, 79
replace のマニュアルページ, 79
return_temporary_buffer のマニュアルページ, 79
_reverse_bi_iterator のマニュアルページ, 69
reverse_copy のマニュアルページ, 79
reverse_iterator のマニュアルページ, 79

reverse のマニュアルページ, 79
RogueWave
  Tools.h++ ライブラリ, 3
rotate_copy のマニュアルページ, 79
rotate のマニュアルページ, 79

S

search_n のマニュアルページ, 79
search のマニュアルページ, 79
seekoff、streambuf::, 60
seekpos、streambuf::, 60
Sequences のマニュアルページ, 68
set_difference のマニュアルページ, 79
set_intersection のマニュアルページ, 79
set_symmetric_difference のマニュアルページ, 79
set_union のマニュアルページ, 79
set のマニュアルページ, 79
<set>、ヘッダーファイル, 67
sin、complex::, 9
sinh、complex::, 9 ~ 11
skip フラグ、iostream, 25
slice_array のマニュアルページ, 80
slice のマニュアルページ, 80
smanip_fill のマニュアルページ, 80
snamip のマニュアルページ, 80
Solaris バージョン、サポートされる, xi
sort_heap のマニュアルページ, 80
sort のマニュアルページ, 80
sqrt、complex::, 9
<sstream>、ヘッダーファイル, 67
stable_partition のマニュアルページ, 80
stable_sort のマニュアルページ, 80
stack のマニュアルページ, 80
<stack>、ヘッダーファイル, 67
Standard C++ Class Library Reference, 66
Standard Template Library (STL)、構成要素, 65
<stdexcept>、ヘッダーファイル, 67
stdiostream.h、iostream ヘッダーファイル, 18
STL (Standard Template Library)、構成要素, 65
stream.h、iostream ヘッダーファイル, 18
streambuf
  iostream::, 16, 46
  新しい関数, 54 ~ 55
  公開仮想関数, 60
  マニュアルページ, 80
  ロック, 47
<streambuf>、ヘッダーファイル, 67
Stream_Iterators のマニュアルページ, 69
stream_locker
  iostream::, 53, 57 ~ 59
  同期、MT-安全オブジェクト, 53
  マニュアルページ, 59
stream_MT、iostream::, 47, 53
string
  iostream::, 20
  マニュアルページ, 80
stringbuf のマニュアルページ, 80
stringstream のマニュアルページ, 80
<string>、ヘッダーファイル, 67
strstream
  iostream::, 17
  マニュアルページ, 80
strstream.h、iostream ヘッダーファイル, 18
strstreambuf
  streambuf::, 55
  マニュアルページ, 80
swap_ranges のマニュアルページ, 81
swap のマニュアルページ, 81

T

tan、complex::, 9
tanh、complex::, 9
thr_keycreate のマニュアルページ, 57
time_get_byname のマニュアルページ, 81
time_get のマニュアルページ, 81

time_put_byname のマニュアルページ, 81
time_put のマニュアルページ, 81
tolower のマニュアルページ, 81
Tools.h++ ライブラリ、RogueWave, 3
toupper のマニュアルページ, 81
transform のマニュアルページ, 81
<typeinfo>、ヘッダーファイル, 67

U

unary_function のマニュアルページ, 81
unary_negate のマニュアルページ, 81
underflow、streambuf::, 60
uninitialized_copy のマニュアルページ, 81
uninitialized_fill_n のマニュアルページ, 81
uninitialized_fill のマニュアルページ, 81
unique_copy のマニュアルページ, 81
unique のマニュアルページ, 81
unsafe_fstream、iostream::, 53
unsafe_ios、iostream::, 46, 53
unsafe_iostream、iostream::, 47, 53
unsafe_istream、istream::, 47, 53
unsafe_ostream、ostream::, 47, 53
unsafe_strstreambase、iostream::, 53
upper_bound のマニュアルページ, 81
use_facet のマニュアルページ, 81
<utility>、ヘッダーファイル, 67

V

valarray のマニュアルページ, 81
<valarray>、ヘッダーファイル, 67
vector のマニュアルページ, 81
<vector>、ヘッダーファイル, 67
void *()、ios::, 21

W

wcerr のマニュアルページ, 82
wcin のマニュアルページ, 82
wclog のマニュアルページ, 82
wcout のマニュアルページ, 82
wfilebuf のマニュアルページ, 82
wifstream のマニュアルページ, 82
wios のマニュアルページ, 82
wistream のマニュアルページ, 82
wistringstream のマニュアルページ, 82
wofstream のマニュアルページ, 82
wostream のマニュアルページ, 82
wostringstream のマニュアルページ, 83
write、ostream::, 22
ws、iostream マニピュレータ, 26
wstreambuf のマニュアルページ, 83
wstream のマニュアルページ, 82
wstringbuf のマニュアルページ, 83
wstring のマニュアルページ, 83

X

xsgetn、streambuf::, 60
xsputn、streambuf::, 60
X 挿入子、iostream, 18

あ

値
  double, 6
  flush, 22
  挿入、cout, 19
  マニピュレータ, 18
新しい機能, xx
アプリケーション
  MT-安全, 45
  MT-安全 iostream オブジェクト, 61 ~ 63
  リンク, 49

い

インタフェース、新しい iostream 関数, 54 ~ 56

え

エラー検査、MT-安全, 49
エラー処理
  complex, 9 ~ 11
エラービット, 21
エラーメッセージ、complex_error, 8
演算子
  complex, 11
  iostream, 18 ~ 22
  基本演算子, 7
  スコープ決定, 51
演算ライブラリ、複素数, 5 ~ 14

お

オーバーヘッド、MT-安全クラスのパフォーマンス, 51 ~ 53
オブジェクト
  stream_locker, 60
  共有オブジェクトの破棄, 60
  共有の使用方法, 57
  大域共有, 56
オブジェクトスレッド、private, 57
オブジェクトライブラリ、関数, 1

か

階層、新しい iostream クラス, 53 ~ 54
角度、複素数, 6
数、複素数, 5 ~ 8
環境
  マルチスレッド, 45 ~ 63
関数
  MT-安全な公開, 47
  streambuf 公開仮想, 60

き

規則、多重定義, 13
共役複素数, 6
共有オブジェクト、処理の方法, 57
極座標、複素数, 6

く

空白
  抽出子, 25
  読み飛ばし, 25

こ

コアクラス、LibC, 46
公開関数、MT-安全, 47
互換モード
  libC, 15, 18
  libcomplex, 5
混合モード、複素数演算ライブラリ, 12 ~ 13
コンストラクタ
  complex, 6 ~ 7
  iostream, 17
コンパイル、MT-安全 libC ライブラリ, 49

さ

先読み、入力の, 25
三角関数、複素数演算ライブラリ, 9

し

実数、複素数, 5, 8
出力, 15
  cout, 19
  string, 20
  エラー処理, 20
  バイナリ, 22
  文字列, 20
順次実行、MT-安全の入出力操作, 57

す

数学関数、複素数演算ライブラリ, 8 ~ 9
スコープ決定演算子、unsafe_ クラス, 51

せ

制限、MT-安全 iostream, 49
静的データ、マルチスレッドアプリケーションでの, 56
絶対値、複素数, 6

そ

相互排他制御域の定義, 59
相互排他ロック、MT-安全の クラス, 52, 60
操作、順次実行, 57
挿入
  演算子, 18 ~ 20

た

大域共有オブジェクト、デフォルト, 56
大域データ、マルチスレッドアプリケーションでの, 56
多重定義、規則 9, 13

ち

抽出
  演算子, 23
抽出子
  char*, 23 ~ 24
  空白, 25
  ユーザー定義の iostream, 23

て

データ型、複素数, 5 ~ 6
テンプレート、Standard Template Library (STL), 65

に

入出力、complex, 11, 15
入力
  iostream, 22
  先読み, 25
  バイナリ, 25

は

バイナリ入力、読み取り, 25
破棄、共有オブジェクト, 60
パフォーマンス、MT-安全クラスのオーバーヘッド, 51 ~ 53

ひ

左シフト演算子
  complex, 11
  iostream, 18
標準 C++ ライブラリ・ユーザーズガイド, 66
標準 iostream クラス, 15
標準エラー、iostreams, 16
標準出力、iostreams, 16
標準ストリーム、iostream.h, 56
標準入力、iostreams, 16
標準モード
  iostream, 15
  libCstd, 5
  libiostream, 15, 18

へ

ヘッダーファイル
  complex, 6, 13
  iostream, 18, 56
  関数, 1
  標準ライブラリ, 66
変更、iostream ライブラリ, 45

ま

マニュアルページ
  C++ 標準ライブラリ, 68 ~ 83
  complex, 14
  iostream, 15
  アクセス, 1 ~ 4
マルチスレッド、環境, 45 ~ 63

み

右シフト演算子
  complex, 11

ゆ

ユーザー定義型
  MT-安全, 51
ユーザー定義の型
  iostream, 19
優先順位の問題, 19

ら

ライブラリ
  C++ のクラス, 1
  C++ 標準, 65 ~ 83
  libC, 45
  libiostream マルチスレッド環境, 45
  Sun Workshop Memory Monitor, 3
  Tools.h++, 3
  従来型の iostream, 15 ~ 43
  複素数演算, 5 ~ 14

り

リファレンス
  C++, 66
  Tools.h++, 3
リンク
  MT-安全 libC, 49
  -mt オプション, 49

る

ルーチン、iostream 公開変換, 48

ろ

ロック
  streambuf, 47
  「stream_locker」も参照
  オブジェクト, 57 ~ 59
  相互排他, 52, 60