C++ Library Reference

901 San Antonio Road
Palo Alto, , CA 94303-4900
USA 650 960-1300 fax 650 969-9131

Part No: 805-4957
Revision A, February 1999

Contents

Preface vii

1. Introduction to C++ Libraries 1

Other Libraries 1

The C++ Standard Library 1

Tools.h++ Library 2

Man Pages for the Libraries 2

Sun WorkShop Memory Monitor 2

2. The Complex Arithmetic Library 5

The Complex Library 5

Using Complex Library 6

Type `complex` 6

Constructors of Class `complex` 6

Arithmetic Operators 7

Mathematical Functions 8

Error Handling 10

Input and Output 12

Mixed-Mode Arithmetic 12

Efficiency 13

Complex Man Pages 14

3. The `Iostream` Library 15

Predefined `Iostreams` 15

Basic Structure of `Iostream` Interaction 16

`Iostreams` 17

Output Using `Iostream` 18

Input Using `Iostream` 21

Defining Your Own Extraction Operators 21

Using the `char*` Extractor 22

Reading Any Single Character 22

Binary Input 23

Peeking at Input 23

Extracting Whitespace 23

Handling Input Errors 24

Using `Iostreams` with `stdio` 24

Creating `Iostreams` 25

Dealing with Files Using Class `fstream` 25

Assignment of `Iostreams` 28

Format Control 29

Manipulators 29

Using Plain Manipulators 30

Parameterized Manipulators 31

`Strstreams`: `Iostreams` for Arrays 33

`Stdiobufs`: `Iostreams` for `stdio` files 33

`Streambuf`s 33

Working with `Streambuf`s 33

Using `Streambuf`s 34

`Iostream` Man Pages 35

`Iostream` Terminology 36

4. Using `Iostreams` in a Multithreaded Environment 39

Multithreading 39

Organization of the MT-safe `iostream` Library 40

Public Conversion Routines 41

Compiling and Linking with the MT-safe `libC` Library 42

MT-safe `iostream` Restrictions 43

Performance 44

Interface Changes to the `iostream` Library 46

New Classes 46

New Class Hierarchy 46

New Functions 47

Global and Static Data 48

Sequence Execution 49

Object Locks 49

Class `stream_locker` 50

MT-safe Classes 51

Object Destruction 52

An Example Application 52

A. Associated Man Pages 55

Index 73

Preface

The *C++ Library Reference* describes the C++ libraries, including:

- C++ Standard Library
- Tools.h++ Class Library
- Sun WorkShop Memory Monitor
- Iostream
- Complex

Who Should Use This Book

This is a reference manual intended for programmers with a working knowledge of the C++ language and some understanding of the Solaris™ system and UNIX® commands.

How This Book Is Organized

Chapter 1” gives an overview of the C++ libraries.

Chapter 2” explains the arithmetic operators and mathematical functions in the library.

Chapter 3” discusses the input and output facility used in C++.

Chapter 4” details how to use the `iostream` lilbrary for input and output in a multithreaded environment.

Appendix A" lists the man pages associated with the C++ libraries.

Multiplatform Release

Note - The name of the latest Solaris operating environment release is Solaris 7 but code and path or package path names may use Solaris 2.7 or SunOS 5.7.

The Sun™ WorkShop™ documentation applies to Solaris 2.5.1, Solaris 2.6, and Solaris 7 operating environments on:

- The SPARC™ platform
- The x86 platform, where x86 refers to the Intel implementation of one of the following: Intel 80386, Intel 80486, Pentium, or the equivalent

Note - The term "x86" refers to the Intel 8086 family of microprocessor chips, including the Pentium, Pentium Pro, and Pentium II processors and compatible microprocessor chips made by AMD and Cyrix. In this document, the term "x86" refers to the overall platform architecture. Features described in this book that are particular to a specific platform are differentiated by the terms "SPARC" and "x86" in the text.

Related Books

The following Sun manuals and guides provide additional useful information.

C++ Books

- *C++ User's Guide* provides information on command-line options and how to use the compiler.
- *C++ Programming Guide* discusses issues relating to the use of templates, exception handling, and interfacing with FORTRAN 77.
- *C++ Migration Guide* describes migrations between compiler releases.
- *Standard C++ Library User's Guide* describes how to use the Standard C++ Library.
- *Standard C++ Class Library Reference* provides detail on the Standard C++ Library.

Other Programming Books

- *Tools.h++ User's Guide* provides details on the `Tools.h++` class library.
- *Tools.h++ Class Library Reference* discusses use of the C++ classes for enhancing the efficiency of your programs.
- *C User's Guide* describes compiler options, pragmas, and more.
- *FORTRAN 77 Language Reference Manual* provides a complete language reference.
- *Fortran User's Guide* provides information on command-line options and how to use the compilers.
- *Fortran Programming Guide* discusses issues relating to input/output, libraries, program analysis, debugging, and performance.
- *Fortran Library Reference* gives detail on the language and routines.
- *Numerical Computation Guide* details floating-point computation numerical accuracy issues.

Other Sun WorkShop Books

- *Sun WorkShop Quick Install* provides installation instructions.
- *Sun WorkShop Installation and Licensing Reference* provides supporting installation and licensing information.
- *Using Sun WorkShop* gives information on performing development operations through Sun WorkShop.
- *Debugging a Program With dbx* provides information on using `dbx` commands to debug a program.
- *Analyzing Program Performance With Sun WorkShop* describes the profiling tools; the LoopTool, LoopReport, and LockLint utilities; and use of the Sampling Analyzer to enhance program performance.
- *Sun WorkShop Performance Library Reference Manual* discusses the library of subroutines and functions to perform useful operations in computational linear algebra and Fourier transforms.
- *Sun WorkShop Visual User's Guide* describes how to use Visual to create C++ and Java™ graphical user interfaces.

Solaris Books

- The Solaris *Linker and Libraries Guide* gives information on linking and libraries.
- The Solaris *Programming Utilities Guide* provides information for developers about the special built-in programming tools available in the SunOS™ system.

Ordering Sun Documents

The SunDocsSM program provides more than 250 manuals from Sun Microsystems, Inc. If you live in the United States, Canada, Europe, or Japan, you can purchase documentation sets or individual manuals using this program.

For a list of documents and how to order them, see the catalog section of the SunExpress™ Internet site at `http://www.sun.com/sunexpress`.

Accessing Sun Documents Online

Sun WorkShop documentation is available online from several sources:

- The `docs.sun.com` Web site
- AnswerBook2™ collections
- HTML documents
- Online help and release notes

Using the `docs.sun.com` Web site

The `docs.sun.com` Web site enables you to access Sun technical documentation online. You can browse the `docs.sun.com` archive or search for a specific book title or subject. The URL is `http://docs.sun.com`.

Accessing AnswerBook2 Collections

The Sun WorkShop documentation is also available using AnswerBook2 software. To access the AnswerBook2 collections, your system administrator must have installed the AnswerBook2 documents during the installation process (if the documents are not installed, see your system administrator or Chapter 3 of *Sun WorkShop Quick Install* for installation instructions). For information about accessing AnswerBook2 documents, see Chapter 6 of *Sun WorkShop Quick Install*, Solaris installation documentation, or your system administrator.

Note - To access AnswerBook2 documents, Solaris 2.5.1 users must first download AnswerBook2 documentation server software from a Sun Web page. For more information, see Chapter 6 of *Sun WorkShop Quick Install*.

Accessing HTML Documents

The following Sun Workshop documents are available online only in HTML format:

- Tools.h++ Class Library Reference
- Tools.h++ User's Guide
- *Numerical Computation Guide*
- Standard C++ Library User's Guide
- *Standard C++ Class Library Reference*
- *Sun WorkShop Performance Library Reference Manual*
- *Sun WorkShop Visual User's Guide*
- Sun WorkShop Memory Monitor User's Manual

To access these HTML documents:

1. **Open the following file through your HTML browser:**

   install-directory`/SUNWspro/DOC5.0/lib/locale/C/html/index.html`

   Replace *install-directory* with the name of the directory where your Sun WorkShop software is installed (the default is `/opt`).

   The browser displays an index of the HTML documents for the Sun WorkShop products that are installed.

2. **Open a document in the index by clicking the document's title.**

Accessing Sun WorkShop Online Help and Release Notes

This release of Sun WorkShop includes an online help system as well as online manuals. To find out more see:

- Online Help. A help system containing extensive task-oriented, context-sensitive help. To access the help, choose Help Help Contents. Help menus are available in all Sun WorkShop windows.
- Release Notes. The Release Notes contain general information about Sun WorkShop and specific information about software limitations and bugs. To access the Release Notes, choose Help Release Notes.

What Typographic Changes Mean

The following table describes the typographic changes used in this book.

TABLE P–1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% You have mail.`
`AaBbCc123`	What you type, contrasted with on-screen computer output	`machine_name% su` `Password:`
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type `rm` *filename*.
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in *User's Guide*. These are called *class* options. You *must* be root to do this.

Shell Prompts in Command Examples

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P–2 System Prompts

Shell	Prompt
C shell prompt	`machine_name%`
C shell superuser prompt	`machine_name#`
Bourne shell and Korn shell prompt	`$`
Bourne shell and Korn shell superuser prompt	`#`

CHAPTER 1

Introduction to C++ Libraries

The C++ class libraries are modular components of reusable code. Using class libraries, you can integrate blocks of code that have been previously built and tested.

A C++ library consists of one or more header files and an object library. The header files provide class and other definitions needed to access the library functions. The object library provides compiled functions and data that are linked with your program to produce an executable program.

This manual describes two class libraries provided with the C++ compiler:

- Complex numbers, described in Chapter 2
- `Iostreams`, described in Chapter 3

  For a discussion of building shared and static libraries, and using libraries, see the *C++ User's Guide.*

Other Libraries

In addition to the Complex and Iostreams libraries, this release provides the C++ Standard library, the Tools.h++ library, and the Sun WorkShop Memory Monitor library.

The C++ Standard Library

The C++ Standard Library (libCstd) is based on the RogueWave™ Standard C++ Library, Version 2.0.1. This library is available only for the default mode (`-compat=5`) of the compiler and is not supported with use of the `-compat` or

`-compat=4` options. For details, see the *Standard C++ Library User's Guide*, the *Standard C++ Library User's Guide*, and the *Standard C++ Class Library Reference*.

Tools.h++ Library

Tools.h++ is a C++ foundation class library from RogueWave. Version 7.0.7 of this library is provided with this release. For information on using this library in the default mode of the compiler, see the *C++ Users Guide*.

For further information about this library, see:

- *Tools.h++ User's Guide* (Version 7.0.7)
- *Tools.h++ Class Library Reference* (Version 7.0.7)

Man Pages for the Libraries

The man pages associated with the libraries described in this manual are located in:

install-directory`/SUNWspro/man/man3`

install-directory`/SUNWspro/man/man3cc4`

install-directory`/SUNWspro/man/man3c++`

where the default *install-directory* is `/opt`.

To access these man pages, ensure that your `MANPATH` includes *install-directory*`/SUNWspro/man` (see the *Sun WorkShop Quick Install*).

To access man pages for the Sun WorkShop Compiler C++ 4.2 libraries, type:

`man -s 3CC4` *library-name*

To access man pages for the Sun WorkShop Compiler C++ 5.0 libraries, type:

`man` *library-name*

Sun WorkShop Memory Monitor

The Sun WorkShop Memory Monitor provides facilities to automatically report and fix memory leaks, memory fragmentation, and premature frees. It has three modes of operation:

- Debugging
- Deployment
- Garbage collection

These modes are dependent on the library you link your application with.

The components of the Sun WorkShop Memory Monitor are:

- `libgc`—the library used in garbage collection and deployment modes
- `libgc_dbg`—the library used in memory debugging mode
- `gcmonitor`—the daemon used in memory debugging mode

For complete documentation on the Sun WorkShop Memory Monitor, see */install-directory*`/SUNWspro/SC5.0/htmldocs/locale/C/gc/index.html`, where the default *install-directory* is `/opt`.

Man pages for the Sun WorkShop Memory Monitor are located in:

install-directory`/SUNWspro/man/man1`

install-directory`/SUNWspro/man/man3`

where the default *install-directory* is `/opt.`

CHAPTER 2

The Complex Arithmetic Library

Complex numbers are numbers made up of a *real* and an *imaginary* part. For example:

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

In the degenerate case, `0 + 3i` is an entirely imaginary number generally written as `3i`, and `5 + 0i` is an entirely real number generally written as `5`. You can represent complex numbers using the `complex` data type.

Note - The Complex Arithmetic library (`libcomplex`) is available only for `-compat=4`, and is not obsolete. For `-compat=5`, this is part of the C++ Standard Library, `libCstd`.

The Complex Library

The complex arithmetic library implements a complex number data type as a new data type and provides:

- Operators
- Mathematical functions (defined for the built-in numerical types)
- Extensions (for iostreams that allow input and output of complex numbers)
- Error handling mechanisms

Complex numbers can also be represented as an *absolute value* (or *magnitude*) and an *argument* (or *angle*). The library provides functions to convert between the real and

imaginary (Cartesian) representation and the magnitude and angle (polar) representation.

The *complex conjugate* of a number has the opposite sign in its imaginary part.

Using Complex Library

To use the complex library, include the header file `complex.h` in your program, and compile and link with the `-library=complex` option.

Type `complex`

The complex arithmetic library defines one class: class `complex`. An object of class `complex` can hold a single complex number. The complex number is constructed of two parts:

- The real part
- The imaginary part

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

The numerical values of each part are held in fields of type `double`. Here is the relevant part of the definition of `complex`:

The value of an object of class `complex` is a pair of `double` values. The first value represents the real part; the second value represents the imaginary part.

Constructors of Class `complex`

There are two constructors for `complex`. Their definitions are:

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

If you declare a complex variable without parameters, the first constructor is used and the variable is initialized, so that both parts are `0`. The following example creates a complex variable whose real and imaginary parts are both `0`:

```
complex aComp;
```

You can give either one or two parameters. In either case, the second constructor is used. When you give only one parameter, it is taken as the value for the real part and the imaginary part is set to `0`. For example:

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

creates a complex variable with the value:

```
4.533 + 0i
```

If you give two values, the first is taken as the value of the real part and the second as the value of the imaginary part. For example:

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

creates a complex variable with the value:

```
8.999 + 2.333i
```

You can also create a complex number using the `polar` function, which is provided in the complex arithmetic library (see “Mathematical Functions ” on page 8). The `polar` function creates a complex value given a pair of polar coordinates, magnitude and angle.

There is no destructor for type `complex`.

Arithmetic Operators

The complex arithmetic library defines all the basic arithmetic operators. Specifically, the following operators work in the usual way and with the usual precedence:

```
+ - / * =
```

The operator `-` has its usual binary and unary meanings.

In addition, you can use the following operators in the usual way:

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

However, these last four operators do not produce values that you can use in expressions. For example, the following does not work:

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

You can also use the following equality operators in their regular meaning:

`== !=`

When you mix real and complex numbers in an arithmetic expression, C++ uses the complex operator function and converts the real values to complex values.

Mathematical Functions

The complex arithmetic library provides a number of mathematical functions. Some are peculiar to complex numbers; the rest are complex-number versions of functions in the standard C mathematical library.

All of these functions produce a result for every possible argument. If a function cannot produce a mathematically acceptable result, it calls `complex_error` and returns some suitable value. In particular, the functions try to avoid actual overflow and call `complex_error` with a message instead. The following tables describe the remainder of the complex arithmetic library functions.

TABLE 2–1 Complex Arithmetic Library Functions

Complex Arithmetic Library Function	Description
`double abs(const complex)`	Returns the magnitude of a complex number.
`double ang(const complex)`	Returns the angle of a complex number.
`complex conj(const complex)`	Returns the complex conjugate of its argument.
`double imag(const complex&)`	Returns the imaginary part of a complex number.
`double norm(const complex)`	Returns the square of the magnitude of its argument. Faster than `abs`, but more likely to cause an overflow. For comparing magnitudes.

TABLE 2–1 Complex Arithmetic Library Functions *(continued)*

Complex Arithmetic Library Function	Description
`complex polar(double mag, double ang=0.0)`	Takes a pair of polar coordinates that represent the magnitude and angle of a complex number and returns the corresponding complex number.
`double real(const complex&)`	Returns the real part of a complex number.

TABLE 2–2 Complex Mathematical and Trigonometric Functions

Complex Arithmetic Library Function	Description
`complex acos(const complex)`	Returns the angle whose cosine is its argument.
`complex asin(const complex)`	Returns the angle whose sine is its argument.
`complex atan(const complex)`	Returns the angle whose tangent is its argument.
`complex cos(const complex)`	Returns the cosine of its argument.
`complex cosh(const complex)`	Returns the hyperbolic cosine of its argument.
`complex exp(const complex)`	Computes `e**x`, where `e` is the base of the natural logarithms, and `x` is the argument given to `exp`.
`complex log(const complex)`	Returns the natural logarithm of its argument.
`complex log10(const complex)`	Returns the common logarithm of its argument.

TABLE 2–2 Complex Mathematical and Trigonometric Functions *(continued)*

Complex Arithmetic Library Function	Description

`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)`
	Takes two arguments: `pow(`*b*, *exp*`)`. It raises *b* to the power of *exp*.

`complex sin(const complex)`	Returns the sine of its argument.
`complex sinh(const complex)`	Returns the hyperbolic sine of its argument.
`complex sqrt(const complex)`	Returns the square root of its argument.
`complex tan(const complex)`	Returns the tangent of its argument.
`complex tanh(const complex)`	Returns the hyperbolic tangent of its argument.

Error Handling

The complex library has these definitions for error handling:

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

The external variable `errno` is the global error state from the C library. `errno` can take on the values listed in the standard header `errno.h` (see the man page `perror`(3)). No function sets `errno` to zero, but many functions set it to other values.

To determine whether a particular operation fails:

1. **Set `errno` to zero before the operation.**

2. **Test the operation.**

The function `complex_error` takes a reference to type `c_exception` and is called by the following complex arithmetic library functions:

- `exp`
- `log`
- `log10`
- `sinh`
- `cosh`

The default version of `complex_error` returns zero. This return of zero means that the default error handling takes place. You can provide your own replacement function `complex_error` that performs other error handling. Error handling is described in the man page `cplxerr`(3C++).

Default error handling is described in the man pages `cplxtrig`(3C++) and `cplxexp`(3C++) It is also summarized in the following table.

TABLE 2–3 Complex Arithmetic Library Functions

Complex Arithmetic Library Function	Default Error Handling Summary
`exp`	If overflow occurs, sets `errno` to `ERANGE` and returns a huge complex number.
`log, log10`	If the argument is zero, sets `errno` to `EDOM` and returns a huge complex number.
`sinh, cosh`	If the imaginary part of the argument causes overflow, returns a complex zero. If the real part causes overflow, returns a huge complex number. In either case, sets `errno` to `ERANGE`.

Input and Output

The complex arithmetic library provides default *extractors* and *inserters* for complex numbers, as shown in the following example:

```
ostream& operator<<(ostream&, const complex&); //inserter
istream& operator>>(istream&, complex&); //extractor
```

See sections “Basic Structure of `Iostream` Interaction” on page 16 and “Output Using `Iostream` ” on page 18 for basic information on extractors and inserters.

For input, the complex extractor `>>` extracts a pair of numbers (surrounded by parentheses and separated by a comma) from the input stream and reads them into a complex object. The first number is taken as the value of the real part; the second as the value of the imaginary part. For example, given the declaration and input statement:

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

and the input `(3.45, 5)`, the value of `x` is equivalent to `3.45 + 5.0i`. The reverse is true for inserters. Given `complex x(3.45, 5)`, `cout<<x` prints `(3.45, 5)`.

The input usually consists of a pair of numbers in parentheses separated by a comma; white space is optional. If you provide a single number, with or without parentheses and white space, the extractor sets the imaginary part of the number to zero. Do not include the symbol `i` in the input text.

The inserter inserts the values of the real and imaginary parts enclosed in parentheses and separated by a comma. It does not include the symbol `i`. The two values are treated as `double`s.

Mixed-Mode Arithmetic

Type `complex` is designed to fit in with the built-in arithmetic types in mixed-mode expressions. Arithmetic types are silently converted to type `complex`, and there are `complex` versions of the arithmetic operators and most mathematical functions. For example:

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

The expression `b+i` is mixed-mode. Integer `i` is converted to type `complex` via the constructor `complex::complex(double,double=0)`, the integer first being converted to type `double`. The result is to be divided by `y`, a `double`, so `y` is also converted to `complex` and the complex divide operation is used. The quotient is thus type `complex`, so the complex sine routine is called, yielding another `complex` result, and so on.

Not all arithmetic operations and conversions are implicit, or even defined, however. For example, complex numbers are not well-ordered, mathematically speaking, and complex numbers can be compared for equality only.

```
complex a, b;
a == b; // OK
a != b; // OK
a <  b; // error: operator < cannot be applied to type complex
a >= b; // error: operator >= cannot be applied to type complex
```

Similarly, there is no automatic conversion from type `complex` to any other type, because the concept is not well-defined. You can specify whether you want the real part, imaginary part, or magnitude, for example.

```
complex a;
double f(double);
f(abs(a)); // OK
f(a);      // error: no match for f(complex)
```

Efficiency

The design of the `complex` class addresses efficiency concerns.

The simplest functions are declared `inline` to eliminate function call overhead.

Several overloaded versions of functions are provided when that makes a difference. For example, the `pow` function has versions that take exponents of type `double` and `int` as well as `complex`, since the computations for the former are much simpler.

The standard C math library header `math.h` is included automatically when you include `complex.h`. The C++ overloading rules then result in efficient evaluation of expressions like this:

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

In this example, the standard math function `sqrt(double)` is called, and the result is converted to type `complex`, rather than converting to type `complex` first and then calling `sqrt(complex)`. This result falls right out of the overload resolution rules, and is precisely the result you want.

Complex Man Pages

The remaining documentation of the complex arithmetic library consists of the man pages listed in the following table.

TABLE 2–4 Man Pages for Type `complex`

Man Page	Overview
`cplx.intro(3C++)`	General introduction to the complex arithmetic library
`cartpol(3C++)`	Cartesian and polar functions
`cplxerr(3C++)`	Error-handling functions
`cplxexp(3C++)`	Exponential, log, and square root functions
`cplxops(3C++)`	Arithmetic operator functions
`cplxtrig(3C++)`	Trigonometric functions

CHAPTER 3

The `Iostream` Library

C++, like C, has no built-in input or output statements. Instead, I/O facilities are provided by a library. The standard C++ I/O library is the `iostream` library.

This chapter provides an introduction to the `iostream` library and examples of its use. It does not provide a complete description of the `iostream` library. See the `iostream` library man pages for more details.

Note - This library is part of `libC` for `-compat=4`. For standard mode, we recommend using the `iostream` library from the standarc C++ library. There is a separate library for `-compat=5` (`libiostream`).

This library is for compatibility with existing sources for `-compat=5`. The new iostreams library, conforming to the ISO standard, is available in the C++ standard library, `libCstd`. If you have sources that use the old iostreams, and if you want to compile your sources with `-compat=5` (default), use the `iostream` library as follows:

`CC` *filename* `-library=iostream`

or

`CC -compat=5` *filename* `-library=iostream`

Make sure that your source file includes `iostream.h`.

Predefined `Iostreams`

There are four predefined `iostreams`:

- `cin`, connected to standard input

- `cout`, connected to standard output
- `cerr`, connected to standard error
- `clog`, connected to standard error

The predefined `iostreams` are fully buffered, except for `cerr`. See “Output Using `Iostream` ” on page 18 and “Input Using `Iostream` ” on page 21.

Basic Structure of `Iostream` Interaction

By including the `iostream` library, a program can use any number of input or output streams. Each stream has some source or sink, which may be one of the following:

- Standard input
- Standard output
- Standard error
- A file
- An array of characters

A stream can be restricted to input or output, or a single stream can allow both input and output. The `iostream` library implements these streams using two processing layers.

- The lower layer implements sequences, which are simply streams of characters. These sequences are implemented by the `streambuf` class, or by classes derived from it.
- The upper layer performs formatting operations on sequences. These formatting operations are implemented by the `istream` and `ostream` classes, which have as a member an object of a type derived from class `streambuf`. An additional class, `iostream`, is for streams on which both input and output can be performed.

Standard input, output, and error are handled by special class objects derived from class `istream` or `ostream`.

The `ifstream`, `ofstream`, and `fstream` classes, which are derived from `istream`, `ostream`, and `iostream` respectively, handle input and output with files.

The `istrstream`, `ostrstream`, and `strstream` classes, which are derived from `istream`, `ostream`, and `iostream` respectively, handle input and output to and from arrays of characters.

When you open an input or output stream, you create an object of one of these types, and associate the `streambuf` member of the stream with a device or file. You generally do this association through the stream constructor, so you don’t work with the `streambuf` directly. The `iostream` library predefines stream objects for the

standard input, standard output, and error output, so you don't have to create your own objects for those streams.

You use operators or `iostream` member functions to insert data into a stream (output) or extract data from a stream (input), and to control the format of data that you insert or extract.

When you want to insert and extract a new data type—one of your classes—you generally overload the insertion and extraction operators.

Iostreams

To use `iostream` routines, you must include the header files for the part of the library you need. The header files are described in the following table.

TABLE 3–1 `Iostream` Routine Header Files

Header File	Description
`iostream.h`	Declares basic features of `iostream` library.
`fstream.h`	Declares `iostreams` and `streambufs` specialized to files. Includes `iostream.h`.
`strstream.h`	Declares `iostreams` and `streambufs` specialized to character arrays. Includes `iostream.h`.
`iomanip.h`	Declares manipulators: values you insert into or extract from `iostreams` to have different effects. Includes `iostream.h`.
`stdiostream.h`	(obsolete) Declares `iostreams` and `streambufs` specialized to use `stdio FILEs`. Includes `iostream.h`.
`stream.h`	(obsolete) Includes `iostream.h`, `fstream.h`, `iomanip.h`, and `stdiostream.h`. For compatibility with old-style streams from C++ version 1.2.

You usually don't need all of these header files in your program. Include only the ones that contain the declarations you need. The `iostream` library is part of `libC`, and is linked automatically by the `CC` driver.

Output Using `Iostream`

Output using `iostream` usually relies on the overloaded left-shift operator (`<<`) which, in the context of `iostream`, is called the insertion operator. To output a value to standard output, you insert the value in the predefined output stream `cout`. For example, given a value `someValue`, you send it to standard output with a statement like:

```
cout << someValue;
```

The insertion operator is overloaded for all built-in types, and the value represented by `someValue` is converted to its proper output representation. If, for example, `someValue` is a `float` value, the `<<` operator converts the value to the proper sequence of digits with a decimal point. Where it inserts `float` values on the output stream, `<<` is called the float inserter. In general, given a type `X`, `<<` is called the `X` inserter. The format of output and how you can control it is discussed in the `ios`(3C++) man page.

The `iostream` library does not support user-defined types. If you define types that you want to output in your own way, you must define an inserter (that is, overload the `<<` operator) to handle them correctly.

The `<<` operator can be applied repetitively. To insert two values on `cout`, you can use a statement like the one in the following example:

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

The output from the above example will show no space between the two values. So you may want to write the code this way:

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

The << operator has the precedence of the left shift operator (its built-in meaning). As with other operators, you can always use parentheses to specify the order of action. It is often a good idea to use parentheses to avoid problems of precedence. Of the following four statements, the first two are equivalent, but the last two are not.

```
cout << a+b;     // + has higher precedence than <<
cout << (a+b);
cout << (a&y);   // << has precedence higher than &
cout << a&y;   // probably an error:  (cout << a) & y
```

Defining Your Own Insertion Operator

The following example defines a `string` class:

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


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

public:
 // (functions not relevant here)

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

The insertion and extraction operators must in this case be defined as friends because the data part of the `string` class is `private`.

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

Here is the definition of `operator<<` overloaded for use with `string`s.

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

`operator<<` takes `ostream&` (that is, a reference to an `ostream`) as its first argument and returns the same `ostream`, making it possible to combine insertions in one statement.

Handling Output Errors

Generally, you don't have to check for errors when you overload `operator<<` because the `iostream` library is arranged to propagate errors.

When an error occurs, the `iostream` where it occurred enters an error state. Bits in the `iostream`'s state are set according to the general category of the error. The inserters defined in `iostream` ignore attempts to insert data into any stream that is in an error state, so such attempts do not change the `iostream`'s state.

In general, the recommended way to handle errors is to periodically check the state of the output stream in some central place. If there is an error, you should handle it in some way. This chapter assumes that you define a function `error`, which takes a

string and aborts the program. `error` is not a predefined function. See “Handling Input Errors ” on page 24 for an example of an `error` function. You can examine the state of an `iostream` with the operator `!`, which returns a nonzero value if the `iostream` is in an error state. For example:

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

There is another way to test for errors. The `ios` class defines `operator void *()`, so it returns a NULL pointer when there is an error. You can use a statement like:

```
if (cout << x) return ; // return if successful
```

You can also use the function `good`, a member of `ios`:

```
if ( cout.good() ) return ; // return if successful
```

The error bits are declared in the `enum`:

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

For details on the error functions, see the `iostream` man pages.

Flushing

As with most I/O libraries, `iostream` often accumulates output and sends it on in larger and generally more efficient chunks. If you want to flush the buffer, you simply insert the special value `flush`. For example:

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

`flush` is an example of a kind of object known as a *manipulator*, which is a value that can be inserted into an `iostream` to have some effect other than causing output of its value. It is really a function that takes an `ostream&` or `istream&` argument and returns its argument after performing some actions on it (see “Manipulators ” on page 29).

Binary Output

To obtain output in the raw binary form of a value, use the member function `write` as shown in the following example. This example shows the output in the raw binary form of `x`.

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

The previous example violates type discipline by converting `&x` to `char*`. Doing so is normally harmless, but if the type of `x` is a class with pointers, virtual member functions, or one that requires nontrivial constructor actions, the value written by the above example cannot be read back in properly.

Input Using `Iostream`

Input using `iostream` is similar to output. You use the extraction operator `>>` and you can string together extractions the way you can with insertions. For example:

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

This statement gets two values from standard input. As with other overloaded operators, the extractors used depend on the types of `a` and `b` (and two different extractors are used if `a` and `b` have different types). The format of input and how you can control it is discussed in some detail in the `ios`(3C++) man page. In general, leading whitespace characters (spaces, newlines, tabs, form-feeds, and so on) are ignored.

Defining Your Own Extraction Operators

When you want input for a new type, you overload the extraction operator for it, just as you overload the insertion operator for output.

Class `string` defines its extraction operator in the following code example:

CODE EXAMPLE 3–1 `string` Extraction Operator

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

The `get` function reads characters from the input stream `istr` and stores them in `holder` until `maxline-1` characters have been read, or a new line is encountered, or EOF, whichever happens first. The data in `holder` is then null-terminated. Finally, the characters in `holder` are copied into the target string.

By convention, an extractor converts characters from its first argument (in this case, `istream& istr`), stores them in its second argument, which is always a reference, and returns its first argument. The second argument must be a reference because an extractor is meant to store the input value in its second argument.

Using the `char*` Extractor

This predefined extractor is mentioned here because it can cause problems. Use it like this:

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

This extractor skips leading whitespace and extracts characters and copies them to `x` until it reaches another whitespace character. It then completes the string with a terminating null (0) character. Be careful, because input can overflow the given array.

You must also be sure the pointer points to allocated storage. For example, here is a common error:

```
char * p; / / not initialized cin >> p;
```

There is no telling where the input data will be stored, and it may cause your program to abort.

Reading Any Single Character

In addition to using the `char` extractor, you can get a single character with either form of the `get` member function. For example:

```
char c;
cin.get(c); // leaves c unchanged if input fails

int b;
b = cin.get(); // sets b to EOF if input fails
```

Note - Unlike the other extractors, the `char` extractor does not skip leading whitespace.

Here is a way to skip only blanks, stopping on a tab, newline, or any other character:

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

Binary Input

If you need to read binary values (such as those written with the member function `write`), you can use the `read` member function. The following example shows how to input the raw binary form of `x` using the `read` member function, and is the inverse of the earlier example that uses `write`.

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

Peeking at Input

You can use the `peek` member function to look at the next character in the stream without extracting it. For example:

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

Extracting Whitespace

By default, the `iostream` extractors skip leading whitespace. You can turn off the *skip* flag to prevent this from happening. The following example turns off whitespace skipping from `cin`, then turns it back on:

```
cin.unsetf(ios::skipws); // turn off whitespace skipping
. . .
cin.setf(ios::skipws); // turn it on again
```

You can use the `iostream` manipulator `ws` to remove leading whitespace from the `iostream`, whether or not skipping is enabled. The following example shows how to remove the leading whitespace from `iostream` `istr`:

```
istr >> ws;
```

Handling Input Errors

By convention, an extractor whose first argument has a nonzero error state should not extract anything from the input stream and should not clear any error bits. An extractor that fails should set at least one error bit.

As with output errors, you should check the error state periodically and take some action, such as aborting, when you find a nonzero state. The `!` operator tests the error state of an `iostream`. For example, the following code produces an input error if you type alphabetic characters for input:

```
#include <unistd.h>
#include <iostream.h>
void error (const char* message) {
  cerr << message << "\n" ;
  exit(1);
}
int 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;
}
```

Class `ios` has member functions that you can use for error handling. See the man pages for details.

Using `Iostreams` with `stdio`

You can use `stdio` with C++ programs, but problems can occur when you mix `iostreams` and `stdio` in the same standard stream within a program. For example, if you write to both `stdout` and `cout`, independent buffering occurs and produces unexpected results. The problem is worse if you input from both `stdin` and `cin`, since independent buffering may turn the input into trash.

To eliminate this problem with standard input, standard output and standard error, use the following instruction before performing any input or output. It connects all the predefined `iostreams` with the corresponding predefined `stdio FILE`s.

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

Such a connection is not the default because there is a significant performance penalty when the predefined streams are made unbuffered as part of the connection. You can use both `stdio` and `iostreams` in the same program applied to different files. That is, you can write to `stdout` using `stdio` routines and write to other files attached to `iostreams`. You can open `stdio FILE`s for input and also read from `cin` so long as you don't also try to read from `stdin`.

Creating `Iostreams`

To read or write a stream other than the predefined `iostreams`, you need to create your own `iostream`. In general, that means creating objects of types defined in the `iostream` library. This section discusses the various types available.

Dealing with Files Using Class `fstream`

Dealing with files is similar to dealing with standard input and standard output; classes `ifstream`, `ofstream`, and `fstream` are derived from classes `istream`, `ostream`, and `iostream`, respectively. As derived classes, they inherit the insertion and extraction operations (along with the other member functions) and also have members and constructors for use with files.

Include the file `fstream.h` to use any of the `fstreams`. Use an `ifstream` when you only want to perform input, an `ofstream` for output only, and an `fstream` for a stream on which you want to perform both input and output. Use the name of the file as the constructor argument.

For example, copy the file `thisFile` to the file `thatFile` as in the following example:

```
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);
```

This code:

- Creates an `ifstream` object called `fromFile` with a default mode of `ios::in` and connects it to `thisFile`. It opens `thisFile`.
- Checks the error state of the new `ifstream` object and, if it is in a failed state, calls the `error` function, which must be defined elsewhere in the program.
- Creates an `ofstream` object called `toFile` with a default mode of `ios::out` and connects it to `thatFile`.
- Checks the error state of `toFile` as above.
- Creates a `char` variable to hold the data while it is passed.
- Copies the contents of `fromFile` to `toFile` one character at a time.

Note - It is, of course, undesirable to copy a file this way, one character at a time. This code is provided just as an example of using `fstreams`. You should instead insert the `streambuf` associated with the input stream into the output stream. See "`Streambuf`s " on page 33, and the man page `sbufpub(3C++)`.

Open Mode

The mode is constructed by `or`-ing bits from the enumerated type `open_mode`, which is a public type of class `ios` and has the definition:

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

Note - The `binary` flag is not needed on Unix, but is provided for compatibility with systems that do need it. Portable code should use the `binary` flag when opening binary files.

You can open a file for both input and output. For example, the following code opens file `someName` for both input and output, attaching it to the `fstream` variable `inoutFile`.

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

Declaring an `fstream` Without Specifying a File

You can declare an `fstream` without specifying a file and open the file later. For example, the following creates the `ofstream` `toFile` for writing.

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

Opening and Closing Files

You can close the `fstream` and then open it with another file. For example, to process a list of files provided on the command line:

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

Opening a File Using a File Descriptor

If you know a file descriptor, such as the integer `1` for standard output, you can open it like this:

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

When you open a file by providing its name to one of the `fstream` constructors or by using the `open` function, the file is automatically closed when the `fstream` is destroyed (by a `delete` or when it goes out of scope). When you `attach` a file to an `fstream`, it is not automatically closed.

Repositioning within a File

You can alter the reading and writing position in a file. Several tools are supplied for this purpose.

- `streampos` is a type that can record a position in an `iostream`.
- `tellg` (`tellp`) is an `istream` (`ostream`) member function that reports the file position. Since `istream` and `ostream` are the parent classes of `fstream`, `tellg` and `tellp` can also be invoked as a member function of the `fstream` class.

- `seekg (seekp)` is an `istream (ostream)` member function that finds a given position.
- The `seek_dir enum` specifies relative positions for use with `seek`.

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

For example, given an `fstream aFile`:

```
streampos original = aFile.tellp();  //save current position
aFile.seekp(0, ios::end); //reposition to end of file
aFile << x;               //write a value to file
aFile.seekp(original);    //return to original position
```

`seekg (seekp)` can take one or two parameters. When it has two parameters, the first is a position relative to the position indicated by the `seek_dir` value given as the second parameter. For example:

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

moves to 10 bytes from the end while

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

moves to 10 bytes forward from the current position.

Note - Arbitrary seeks on text streams is not portable, but you can always return to a previously saved `streampos` value.

Assignment of `Iostreams`

`Iostreams` does not allow assignment of one stream to another.

The problem with copying a stream object is that there are then two versions of the state information, such as a pointer to the current write position within an output file, which can be changed independently. As a result, problems could occur.

Format Control

Format control is discussed in detail in the in the man page `ios`(3C++).

Manipulators

Manipulators are values that you can insert into or extract from `iostreams` to have special effects.

Parameterized manipulators are manipulators that take one or more parameters.

Because manipulators are ordinary identifiers, and therefore use up possible names, `iostream` doesn't define them for every possible function. A number of manipulators are discussed with member functions in other parts of this chapter.

There are 13 predefined manipulators, as described in Table 3–2. When using that table, assume the following:

- `i` has type `long`.
- `n` has type `int`.
- `c` has type `char`.
- `istr` is an input stream.
- `ostr` is an output stream.

TABLE 3–2 `Iostream` Predefined Manipulators

Predefined Manipulator	Description
`ostr << dec, istr >> dec`	Makes the integer conversion base 10.
`ostr << endl`	Inserts a newline character (`'\n"`) and invokes `ostream::flush()`.
`ostr << ends`	Inserts a null (0) character. Useful when dealing with `strstreams`.
`ostr << flush`	Invokes `ostream::flush()`.
`ostr << hex, istr >> hex`	Makes the integer conversion base 16.
`ostr << oct, istr >> oct`	Make the integer conversion base 8.

TABLE 3–2 `Iostream` Predefined Manipulators *(continued)*

Predefined Manipulator	Description
7 `istr >> ws`	Extracts whitespace characters (skips whitespace) until a non-whitespace character is found (which is left in `istr`).
8 `ostr << setbase(n), istr >> setbase(n)`	Sets the conversion base to `n` (0, 8, 10, 16 only).
9 `ostr << setw(n), istr >> setw(n)`	Invokes `ios::width(n)`. Sets the field width to `n`.
10 `ostr << resetiosflags(i), istr >> resetiosflags(i)`	Clears the flags bitvector according to the bits set in `i`.
11 `ostr << setiosflags(i), istr >> setiosflags(i)`	Sets the flags bitvector according to the bits set in `i`.
12 `ostr << setfill(c), istr >> setfill(c)`	Sets the fill character (for padding a field) to `c`.
13 `ostr << setprecision(n), istr >> setprecision(n)`	Sets the floating-point precision to `n` digits.

To use predefined manipulators, you must include the file `iomanip.h` in your program.

You can define your own manipulators. There are two basic types of manipulator:

- Plain manipulator—Takes an `istream&`, `ostream&`, or `ios&` argument, operates on the stream, and then returns its argument.
- Parameterized manipulator—Takes an `istream&`, `ostream&`, or `ios&` argument, one additional argument (the parameter), operates on the stream, and then returns its stream argument.

Using Plain Manipulators

A plain manipulator is a function that:

- Takes a reference to a stream
- Operates on it in some way
- Returns its argument

The shift operators taking (a pointer to) such a function are predefined for `iostreams`, so the function can be put in a sequence of input or output operators.

The shift operator calls the function rather than trying to read or write a value. An example of a `tab` manipulator that inserts a `tab` in an `ostream` is:

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

This is an elaborate way to achieve the following:

```
const char tab = "\t";
...
cout << x << tab << y;
```

Here is another example, which cannot be accomplished with a simple constant. Suppose you want to turn whitespace skipping on and off for an input stream. You can use separate calls to `ios::setf` and `ios::unsetf` to turn the `skipws` flag on and off, or you could define two manipulators:

```
#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;
}
```

Parameterized Manipulators

One of the parameterized manipulators that is included in `iomanip.h` is `setfill`. `setfill` sets the character that is used to fill out field widths. It is implemented as shown in the following example::

```
//file setfill.cc
#include<iostream.h>
#include<iomanip.h>

//the private manipulator
```

```
static ios& sfill(ios& i, int f) {
         i.fill(f);
         return i;
}
//the public applicator
smanip_int setfill(int f) {
       return smanip_int(sfill, f);
}
```

A parameterized manipulator is implemented in two parts:

- The *manipulator*. It takes an extra parameter. In the previous code example, it takes an extra `int` parameter. You cannot place this manipulator function in a sequence of input or output operations, since there is no shift operator defined for it. Instead, you must use an auxiliary function, the applicator.
- The *applicator*. It calls the manipulator. The applicator is a global function, and you make a prototype for it available in a header file. Usually the manipulator is a static function in the file containing the source code for the applicator. The manipulator is called only by the applicator, and if you make it static, you keep its name out of the global address space.

Several classes are defined in the header file `iomanip.h`. Each class holds the address of a manipulator function and the value of one parameter. The `iomanip` classes are described in the man page `manip`(3C++). The previous example uses the `smanip_int` class, which works with an `ios`. Because it works with an `ios`, it also works with an `istream` and an `ostream`. The previous example also uses a second parameter of type `int`.

The applicator creates and returns a class object. In the previous code example the class object is an `smanip_int`, and it contains the manipulator and the `int` argument to the applicator. The `iomanip.h` header file defines the shift operators for this class. When the applicator function `setfill` appears in a sequence of input or output operations, the applicator function is called, and it returns a class. The shift operator acts on the class to call the manipulator function with its parameter value, which is stored in the class.

In the following example, the manipulator `print_hex`:

- Puts the output stream into the hex mode.
- Inserts a `long` value into the stream.
- Restores the conversion mode of the stream.

The class `omanip_long` is used because this code example is for output only, and it operates on a `long` rather than an `int`:

```
#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);
   }
```

Strstreams: Iostreams for Arrays

See the strstream(3C++) man page.

Stdiobufs: Iostreams for stdio files

See the stdiobuf(3C++) man page.

Streambufs

Iostreams are the formatting part of a two-part (input or output) system. The other part of the system is made up of streambufs, which deal in input or output of unformatted streams of characters.

You usually use streambufs through iostreams, so you don't have to worry about the details of streambufs. You can use streambufs directly if you choose to, for example, if you need to improve efficiency or to get around the error handling or formatting built into iostreams.

Working with Streambufs

A streambuf consists of a stream or sequence of characters and one or two pointers into that sequence. Each pointer points between two characters. (Pointers cannot actually point between characters, but it is helpful to think of them that way.) There are two kinds of streambuf pointers:

- A *put* pointer, which points just before the position where the next character will be stored

- A *get* pointer, which points just before the next character to be fetched

A `streambuf` can have one or both of these pointers.

Position of Pointers

The positions of the pointers and the contents of the sequences can be manipulated in various ways. Whether or not both pointers move when manipulated depends on the kind of `streambuf` used. Generally, with queue-like `streambuf`s, the get and put pointers move independently; with file-like `streambuf`s the get and put pointers always move together. A `strstream` is an example of a queue-like stream; an `fstream` is an example of a file-like stream.

Using `Streambuf`s

You never create an actual `streambuf` object, but only objects of classes derived from class `streambuf`. Examples are `filebuf` and `strstreambuf`, which are described in man pages `filebuf`(3c++) and `ssbuf`(3), respectively. Advanced users may want to derive their own classes from `streambuf` to provide an interface to a special device or to provide other than basic buffering. Man pages `sbufpub`(3C++) and `sbufprot`(3C++) discuss how to do this.

Apart from creating your own special kind of `streambuf`, you may want to access the `streambuf` associated with an `iostream` to access the public member functions, as described in the man pages referenced above. In addition, each `iostream` has a defined inserter and extractor which takes a `streambuf` pointer. When a `streambuf` is inserted or extracted, the entire stream is copied.

Here is another way to do the file copy discussed earlier, with the error checking omitted for clarity:

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

We open the input and output files as before. Every `iostream` class has a member function `rdbuf` that returns a pointer to the `streambuf` object associated with it. In the case of an `fstream`, the `streambuf` object is type `filebuf`. The entire file associated with `fromFile` is copied (inserted into) the file associated with `toFile`. The last line could also be written like this:

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

The source file is then extracted into the destination. The two methods are entirely equivalent.

Iostream Man Pages

A number of C++ man pages give details of the iostream library. The following table gives an overview of what is in each man page.

TABLE 3–3 Iostream Man Pages Overview

Man Page	Overview
filebuf	Details the public interface for the class filebuf, which is derived from streambuf and is specialized for use with files. See the sbufpub(3C++) and sbufprot(3C++) man pages for details of features inherited from class streambuf. Use the filebuf class through class fstream.
fstream	Details specialized member functions of classes ifstream, ofstream, and fstream, which are specialized versions of istream, ostream, and iostream for use with files.
ios	Details parts of class ios, which functions as a base class for iostreams. It contains state data common to all streams.
ios.intro	Gives an introduction to and overview of iostreams.
istream	Details the following: • Member functions for class istream, which supports interpretation of characters fetched from a streambuf • Input formatting • Positioning functions described as part of class ostream. • Some related functions • Related manipulators
manip	Describes the input and output manipulators defined in the iostream library.
ostream	Details the following: • Member functions for class ostream, which supports interpretation of characters written to a streambuf • Output formatting • Positioning functions described as part of class ostream • Some related functions • Related manipulators

TABLE 3–3 `Iostream` Man Pages Overview *(continued)*

Man Page	Overview
`sbufprot`	Describes the interface needed by programmers who are coding a class derived from class `streambuf`. Also refer to the `sbufpub`(3C++) man page because some public functions are not discussed in the `sbufprot(3C++)` man page.
`sbufpub`	Details the public interface of class `streambuf`, in particular, the public member functions of `streambuf`. This man page contains the information needed to manipulate a `streambuf`-type object directly, or to find out about functions that classes derived from `streambuf` inherit from it. If you want to derive a class from `streambuf`, also see the `sbufprot`(3C++) man page.
`ssbuf`	Details the specialized public interface of class `strstreambuf`, which is derived from `streambuf` and specialized for dealing with arrays of characters. See the `sbufpub`(3C++) man page for details of features inherited from class `streambuf`.
`stdiobuf`	Contains a minimal description of class `stdiobuf`, which is derived from `streambuf` and specialized for dealing with `stdio` `FILE`s. See the `sbufpub`(3C++) man page for details of features inherited from class `streambuf`.
`strstream`	Details the specialized member functions of `strstream`s, which are implemented by a set of classes derived from the `iostream` classes and specialized for dealing with arrays of characters.

`Iostream` Terminology

The `iostream` library descriptions often use terms similar to terms from general programming, but with specialized meanings. The following table defines these terms as they are used in discussing the `iostream` library.

TABLE 3–4 `Iostream` Terminology

`Iostream` **Term**	**Definition**
Buffer	A word with two meanings, one specific to the `iostream` package and one more generally applied to input and output. When referring specifically to the `iostream` library, a buffer is an object of the type defined by the class `streambuf`. A buffer, generally, is a block of memory used to make efficient transfer of characters for input of output. With buffered I/O, the actual transfer of characters is delayed until the buffer is full or forcibly flushed. An unbuffered buffer refers to a `streambuf` where there is no buffer in the general sense defined above. This chapter avoids use of the term buffer to refer to `streambuf`s. However, the man pages and other C++ documentation do use the term buffer to mean `streambuf`s.
Extraction	The process of taking input from an `iostream`.
`Fstream`	An input or output stream specialized for use with files. Refers specifically to a class derived from class `iostream` when printed in `courier` font.
Insertion	The process of sending output into an `iostream`.
`Iostream`	Generally, an input or output stream.
`Iostream` library	The library implemented by the include files `iostream.h`, `fstream.h`, `strstream.h`, `iomanip.h`, and `stdiostream.h`. Because `iostream` is an object-oriented library, you should extend it. So, some of what you can do with the `iostream` library is not implemented.
Stream	An `iostream`, `fstream`, `strstream`, or user-defined stream in general.
`Streambuf`	A buffer that contains a sequence of characters with a put or get pointer, or both. When printed in `courier` font, it means the particular class. Otherwise, it refers generally to any object of class `streambuf` or a class derived from `streambuf`. Any stream object contains an object, or a pointer to an object, of a type derived from `streambuf`.
`Strstream`	An `iostream` specialized for use with character arrays. It refers to the specific class when printed in `courier` font.

CHAPTER 4

Using `Iostreams` in a Multithreaded Environment

This chapter describes how to use the `iostream` classes of the `libC` and `libiostream` libraries for input-output (I/O) in a multithreaded environment. It also provides examples of how to extend functionality of the library by deriving from the `iostream` classes. This chapter is *not* a guide for writing multithreaded code in C++, however.

The discussion here applies only to the old iostreams (`libC` and `libiostream`) and does not apply to `libCstd`, the new iostream that is part of the C++ Standard Library.

Multithreading

Multi-threading (MT) is a powerful facility that can speed up applications on multiprocessors; it can also simplify the structuring of applications on both multiprocessors and uniprocessors. The `iostream` library has been modified to allow its interfaces to be used by applications in a multithreaded environment by programs that utilize the multithreading capabilities when running Solaris version 2.2, 2.3, 2.4, 2.5, 2.5.1, 2.6, or 7. Applications that utilize the single-threaded capabilities of previous versions of the library are not affected by the behavior of the modified `iostream` interfaces.

A library is defined to be MT-safe if it works correctly in an environment with threads. Generally, this “correctness” means that all of its public functions are reentrant. The `iostream` library provides protection against multiple threads that attempt to modify the state of objects (that is, instances of a C++ class) shared by

more than one thread. However, the scope of MT-safety for an `iostream` object is confined to the period in which the object's public member function is executing.

Caution - An application is *not* automatically guaranteed to be MT-safe because it uses MT-safe objects from the `libC` library. An application is defined to be MT-safe only when it executes as expected in a multithreaded environment.

Organization of the MT-safe `iostream` Library

The organization of the MT-safe `iostream` library is slightly different from other versions of the `iostream` library. The exported interface of the library refers to the `public` and `protected` member functions of the `iostream` classes and the set of base classes available, and is consistent with other versions; however, the class hierarchy is different. See "Interface Changes to the `iostream` Library" on page 46 for details.

The original core classes have been renamed with the prefix `unsafe_`. Table 4–1 lists the classes that are the core of the `iostream` package.

TABLE 4–1 Core Classes

Class	Description
stream_MT	The base class for MT-safe classes.
streambuf	The base class for buffers.
unsafe_ios	A class that contains state variables that are common to the various stream classes; for example, error and formatting state.
unsafe_istream	A class that supports formatted and unformatted conversion from sequences of characters retrieved from the `streambuf`s.
unsafe_ostream	A class that supports formatted and unformatted conversion to sequences of characters stored into the `streambuf`s.
unsafe_iostream	A class that combines `unsafe_istream` and `unsafe_ostream` classes for bi-directional operations.

Each MT-safe class is derived from the base class `stream_MT`. Each MT-safe class, except `streambuf`, is also derived from the existing `unsafe_` base class. Here are some examples:

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

The class `stream_MT` provides the mutual exclusion (mutex) locks required to make each `iostream` class MT-safe; it also provides a facility that dynamically enables and disables the locks so that the MT-safe property can be dynamically changed. The basic functionality for I/O conversion and buffer management are organized into the `unsafe_` classes; the MT-safe additions to the library are confined to the derived classes. The MT-safe version of each class contains the same protected and public member functions as the `unsafe_ base` class. Each member function in the MT-safe version class acts as a wrapper that locks the object, calls the same function in the `unsafe_ base` class, and unlocks the object.

Note - The class `streambuf` is *not* derived from an unsafe class. The public and protected member functions of class `streambuf` are reentrant by locking. Unlocked versions, suffixed with `_unlocked`, are also provided.

Public Conversion Routines

A set of reentrant public functions that are MT-safe have been added to the `iostream` interface. A user-specified buffer is an additional argument to each function. These functions are described as follows.

TABLE 4–2 Reentrant Public Functions

Function	Description
char *oct_r (char *buf, int buflen, long num, int width)	Returns a pointer to the ASCII string that represents the number in octal. A width of nonzero is assumed to be the field width for formatting. The returned value is not guaranteed to point to the beginning of the user-provided buffer.
char *hex_r (char *buf, int buflen, long num, int width)	Returns a pointer to the ASCII string that represents the number in hexadecimal. A width of nonzero is assumed to be the field width for formatting. The returned value is not guaranteed to point to the beginning of the user-provided buffer.

TABLE 4–2 Reentrant Public Functions *(continued)*

Function	Description
char *dec_r (char *buf, int buflen, long num, int width)	Returns a pointer to the ASCII string that represents the number in decimal. A width of nonzero is assumed to be the field width for formatting. The returned value is not guaranteed to point to the beginning of the user-provided buffer.
char *chr_r (char *buf, int buflen, long num, int width)	Returns a pointer to the ASCII string that contains character `chr`. If the width is nonzero, the string contains `width` blanks followed by `chr`. The returned value is not guaranteed to point to the beginning of the user-provided buffer.
char *form_r (char *buf, int buflen, long num, int width)	Returns a pointer of the string formatted by `sprintf`, using the format string `format` and any remaining arguments. The buffer must have sufficient space to contain the formatted string.

Caution - The public conversion routines of the `iostream` library (`oct`, `hex`, `dec`, `chr`, and `form`) that are present to ensure compatibility with an earlier version of `libC` are *not* MT-safe.

Compiling and Linking with the MT-safe `libC` Library

When you build an application that uses the `iostream` classes of the `libC` library to run in a multithreaded environment, compile and link the source code of the application using the `-mt` option. This option passes `-D_REENTRANT` to the preprocessor and `-lthread` to the linker.

Note - Use `-mt` (rather than `-lthread`) to link with `libC` and `libthread`. This option ensures proper linking order of the libraries. Using `-lthread` improperly could cause your application to work incorrectly.

Single-threaded applications that use `iostream` classes do not require special compiler or linker options. By default, the compiler links with the `libC` library.

MT-safe `iostream` Restrictions

The restricted definition of MT-safety for the `iostream` library means that a number of programming idioms used with `iostream` are unsafe in a multithreaded environment using shared `iostream` objects.

Checking Error State

To be MT-safe, error checking must occur in a critical region with the I/O operation that causes the error. The following example illustrates how to check for errors:

CODE EXAMPLE 4–1 Checking Error State

```
#include <iostream.h>
enumm 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;
 if (istr.fail()) return IOfail;
 return IOok;
}
```

In this example, the constructor of the `stream_locker` object `sl` locks the `istream` object `istr`. The destructor of `sl`, called at the termination of `read_number`, unlocks `istr`.

Obtaining Characters Extracted by Last Unformatted Input Operation

To be MT-safe, the `gcount` function must be called within a thread that has exclusive use of the `istream` object for the period that includes the execution of the last input operation and `gcount` call. The following example shows a call to `gcount`:

CODE EXAMPLE 4–2 Calling `gcount`

```
#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(); // lock the stream istr
 istr >> line;
 linecount = istr.gcount();
 sl.unlock(); // unlock istr
 ...
```

```
}
```

In this example, the `lock` and `unlock` member functions of class `stream_locker` define a mutual exclusion region in the program.

User-Defined I/O Operations

To be MT-safe, I/O operations defined for a user-defined type that involve a specific ordering of separate operations must be locked to define a critical region. The following example shows a user-defined I/O operation:

CODE EXAMPLE 4–3 User-Defined I/O Operations

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

 // other definitions...
 int getRecord(char* name, int& id, float& gpa);
};

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;
}
```

Performance

Using the MT-safe classes in this version of the `libC` library results in some amount of performance overhead, even in a single-threaded application; however, if you use the `unsafe_` classes of `libC`, this overhead can be avoided.

The scope resolution operator can be used to execute member functions of the base `unsafe_` classes; for example:

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


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

Note - The `unsafe_` classes cannot be safely used in multithreaded applications.

Instead of using `unsafe_` classes, you can make the `cout` and `cin` objects `unsafe` and then use the normal operations. A slight performance deterioration results. The following example shows how to use `unsafe cout` and `cin`:

CODE EXAMPLE 4–4 Disabling mt-safety

```
#include <iostream.h>
cout.set_safe_flag(stream_MT::unsafe_object); //disable mt-safety
cin.set_safe_flag(stream_MT::unsafe_object); //disable mt-safety
cout.put('4');
cin.read(buf, len);
```

When an `iostream` object is MT-safe, mutex locking is provided to protect the object"s member variables. This locking adds unnecessary overhead to an application that only executes in a single-threaded environment. To improve performance, you can dynamically switch an `iostream` object to and from MT-safety. The following example makes an `iostream` object MT-unsafe:

CODE EXAMPLE 4–5 Switching to MT-unsafe

```
fs.set_safe_flag(stream_MT::unsafe_object);// disable MT-safety
 .... do various i/o operations
```

You can safely use an MT-unsafe stream in code where an iostream is *not* shared by threads; for example, in a program that has only one thread, or in a program where each iostream is private to a thread.

If you explicitly insert synchronization into the program, you can also safely use MT-unsafe iostreams in an environment where an iostream is shared by threads. The following example illustrates the technique:

CODE EXAMPLE 4–6 Using Synchronization with MT-unsafe Objects

```
generic_lock() ;
fs.set_safe_flag(stream_MT::unsafe_object) ;
... do various i/o operations
generic_unlock() ;
```

where the `generic_lock` and `generic_unlock` functions can be any synchronization mechanism that uses such primitives as mutex, semaphores, or reader / writer locks.

Note - The `stream_locker` class provided by the `libC` library is the preferred mechanism for this purpose.

See “Object Locks” on page 49 for more information.

Interface Changes to the `iostream` Library

This section describes the interface changes made to the `iostream` library to make it MT-safe.

New Classes

The following table lists the new classes added to the `libC` interfaces.

CODE EXAMPLE 4–7 New Classes

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

New Class Hierarchy

The following table lists the new class hierarchy added to the `iostream` interfaces.

CODE EXAMPLE 4–8 New Class Hierarchy

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

New Functions

The following table lists the new functions added to the `iostream` interfaces.

CODE EXAMPLE 4–9 New Functions

```
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();
  char* 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);
  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, ...)
```

Global and Static Data

Global and static data in a multithreaded application are not safely shared among threads. Although threads execute independently, they share access to global and static objects within the process. If one thread modifies such a shared object, all the other threads within the process observe the change, making it difficult to maintain state over time. In C++, class objects (instances of a class) maintain state by the values in their member variables. If a class object is shared, it is vulnerable to changes made by other threads.

When a multithreaded application uses the `iostream` library and includes `iostream.h`, the standard streams—`cout`, `cin`, `cerr`, and `clog`— are, by default, defined as global shared objects. Since the `iostream` library is MT-safe, it protects the state of its shared objects from access or change by another thread while a member function of an `iostream` object is executing. However, the scope of MT-safety for an object is confined to the period in which the object's public member function is executing. For example,

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

gets the next character in the `get` buffer and updates the buffer pointer in *ThreadA*. However, if the next instruction in *ThreadA* is another `get` call, the `libC` library does not guarantee to return the next character in the sequence. It is not guaranteed because, for example, *ThreadB* may have also executed the `get` call in the intervening period between the two `get` calls made in *ThreadA*.

See "Object Locks" on page 49" for strategies for dealing with the problems of shared objects and multithreading.

Sequence Execution

Frequently, when `iostream` objects are used, a sequence of I/O operations must be MT-safe. For example, the code:

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

involves the execution of three member functions of the `cout` stream object. Since `cout` is a shared object, the sequence must be executed atomically as a critical section to work correctly in a multithreaded environment. To perform a sequence of operations on an `iostream` class object atomically, you must use some form of locking.

The `libC` library now provides the `stream_locker` class for locking operations on an `iostream` object. See “Object Locks” on page 49” for information about the `stream_locker` class.

Object Locks

The simplest strategy for dealing with the problems of shared objects and multithreading is to avoid the issue by ensuring that `iostream` objects are local to a thread. For example,

- Declare objects locally within a thread’s entry function.
- Declare objects in thread-specific data. (For information on how to use thread specific data, see the `thr_keycreate`(3T) man page.)
- Dedicate a stream object to a particular thread. The object thread is `private` by convention.

However, in many cases, such as default shared standard stream objects, it is not possible to make the objects local to a thread, and an alternative strategy is required.

To perform a sequence of operations on an `iostream` class object atomically, you must use some form of locking. Locking adds some overhead even to a single-threaded application. The decision whether to add locking or make `iostream` objects private to a thread depends on the thread model chosen for the application: Are the threads to be independent or cooperating?

- If each independent thread is to produce or consume data using its own `iostream` object, the `iostream` objects are private to their respective threads and locking is not required.

- If the threads are to cooperate (that is, they are to share the same `iostream` object), then access to the shared object must be synchronized and some form of locking must be used to make sequential operations atomic.

Class `stream_locker`

The `iostream` library provides the `stream_locker` class for locking a series of operations on an `iostream` object. You can, therefore, minimize the performance overhead incurred by dynamically enabling or disabling locking in `iostream` objects.

Objects of class `stream_locker` can be used to make a sequence of operations on a stream object atomic. For example, the code shown in the example below seeks to find a position in a file and reads the next block of data.

CODE EXAMPLE 4–10 Example of Using Locking Operations

```
#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);
 . . . . .// open file
 fs.seekg(offset, ios::beg);
 fs.read(buf, len);
}
```

In this example, the constructor for the `stream_locker` object defines the beginning of a mutual exclusion region in which only one thread can execute at a time. The destructor, called after the return from the function, defines the end of the mutual exclusion region. The `stream_locker` object ensures that both the seek to a particular offset in a file and the read from the file are performed together, atomically, and that *ThreadB* cannot change the file offset before the original *ThreadA* reads the file.

An alternative way to use a `stream_locker` object is to explicitly define the mutual exclusion region. In the following example, to make the I/O operation and subsequent error checking atomic, `lock` and `unlock` member function calls of a `stream_locker` object are used.

CODE EXAMPLE 4–11 Making I/O Operation and Error Checking Atomic

```
{
 ...
 stream_locker file_lck(openfile_stream,
                         stream_locker::lock_defer);
 ....
```

```
 file_lck.lock();  // lock openfile_stream
 openfile_stream << "Value: " << int_value << "\n";
 if(!openfile_stream) {
   file_error("Output of value failed\n");
   return;
 }
 file_lck.unlock();  // unlock openfile_stream
}
```

For more information, see the `stream_locker`(3C++) man page.

MT-safe Classes

You can extend or specialize the functionality of the `iostream` classes by deriving new classes. If objects instantiated from the derived classes will be used in a multithreaded environment, the classes must be MT-safe.

Considerations when deriving MT-safe classes include:

- Making a class object MT-safe by protecting the internal state of the object from multiple-thread modification. To do this, serialize access to member variables in public and protected member functions with mutex locks.
- Making a sequence of calls to member functions of an MT-safe base class atomic, using a `stream_locker` object.
- Avoiding locking overhead by using the `_unlocked` member functions of `streambuf` within critical regions defined by `stream_locker` objects.
- Locking the public virtual functions of class `streambuf` in case the functions are called directly by an application. These functions are: `xsgetn`, `underflow`, `pbackfail`, `xsputn`, `overflow`, `seekoff`, and `seekpos`.
- Extending the formatting state of an `ios` object by using the member functions `iword` and `pword` in class `ios`. However, a problem can occur if more than one thread is sharing the same index to an `iword` or `pword` function. To make the threads MT-safe, use an appropriate locking scheme.
- Locking member functions that return the value of a member variable greater in size than a `char`.

Object Destruction

Before an `iostream` object that is shared by several threads is deleted, the main thread must verify that the subthreads are finished with the shared object. The following example shows how to safely destroy a shared object.

CODE EXAMPLE 4–12 Destroying a Shared Object

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

void *process_rtn(void*)
{
 // body of sub-threads which uses fp...
}

multi_process(const char* filename, int numthreads)
{
 fp = new fstream(filename, ios::in); // create fstream object
                                      // before creating threads.
 // create threads
 for (int i=0; i<numthreads; i++)
   thr_create(0, STACKSIZE, process_rtn, 0, 0, 0);

  ...
 // wait for threads to finish
 for (int i=0; i<numthreads; i++)
   thr_join(0, 0, 0);

 delete fp;                          // delete fstream object after
 fp = NULL;                         // all threads have completed.
}
```

An Example Application

The following code provides an example of a multiply-threaded application that uses `iostream` objects from the `libC` library in an MT-safe way.

The example application creates up to 255 threads. Each thread reads a different input file, one line at a time, and outputs the line to an output file, using the standard output stream, `cout`. The output file, which is shared by all threads, is tagged with a value that indicates which thread performed the output operation.

CODE EXAMPLE 4–13 Using `Iostream` Objects in a MT-safe Way

```
// create tagged thread data
// the output file is of the form:
//   <tag><string of data>\n
// where tag is an integer value in a unsigned char.
// Allows up to 255 threads to be run in this application
// <string of data> is any printable characters
// Because tag is an integer value written
as char,
// you need to use od to look at the output
file, suggest:
//   od -c out.file |more

#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;

// entry routine for each thread
void* ThreadDuties(void* v) {
// obtain arguments for this thread
  thread_args* tt = (thread_args*)v;
  char ibuf[thread_bufsize];
  // open thread input file
  ifstream instr(tt->filename);
  stream_locker lockout(cout, stream_locker::lock_defer);
  while(1) {
  // read a line at a time
    instr.getline(ibuf, thread_bufsize - 1, '\n');
    if(instr.eof())
      break;
  // lock cout stream so the i/o operation is atomic
    lockout.lock();
  // tag line and send to cout
    cout << (unsigned char)tt->thread_tag << ibuf << "\n";
    lockout.unlock();
  }
  return 0;
}

int main(int argc, char** argv) {
  // argv: 1+ list of filenames per thread
   if(argc < 2) {
     cout << ''usage: " << argv[0] << " <files..>\n";
     exit(1);
   }
  int num_threads = argc - 1;
  int total_tags = 0;

// array of thread_ids
  thread_t created_threads[thread_bufsize];
```

```
// array of arguments to thread entry routine
  thread_args thr_args[thread_bufsize];
  int i;
  for( i = 0; i < num_threads; i++) {
    thr_args[i].filename = argv[1 + i];
// assign a tag to a thread - a value less than 256
    thr_args[i].thread_tag = total_tags++;
// create threads
    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;
}
```

APPENDIX **A**

Associated Man Pages

The manual pages associated with the libraries described in this manual are listed in the following tables.

TABLE A–1 Man Pages for Complex Library

Man Page	Overview
`cplx.intro`	General introduction to the complex arithmetic library
`cartpol`	Cartesian and polar functions
`cplxerr`	Error-handling functions
`cplxexp`	Exponential, log, and square root functions
`cplxops`	Arithmetic operator functions
`cplxtrig`	Trigonometric functions

TABLE A–2 Man Pages for Iostream Library

Man Page	Overview
`ios.intro`	Gives an introduction to and overview of iostreams
`filebuf`	Details the public interface for the class `filebuf`, which is specialized for use with files.
`fstream`	Details specialized member functions of classes `ifstream`, `ofstream`, and `fstream`, which are specialized for use with files.
`ios`	Details parts of class `ios`, which functions as a base class for iostreams
`istream`	Describes class `istream`, which supports interpretation of characters fetched from a streambuf
`manip`	Describes the input and output manipulators defined in the iostream library
`ostream`	Describes class `ostream`, which supports interpretation of characters written to a streambuf
`sbufprot`	Describes the protected interface of class `streambuf`
`sbufpub`	Details the public interface of class `streambuf`
`stdiobuf`	Describes class `stdiobuf`, which is specialized for dealing with `stdio FILE`s
`strstream`	Details the specialized member functions of strstreams, which are specialized for dealing with arrays of characters
`ssbuf`	Details the specialized public interface of class `strstreambuf`, which is specialized for dealing with arrays of characters.
`stream_MT`	Describes the base class that provides dynamic changing of iostream class objects to and from MT safety
`stream_locker`	Describes the class used for application-level locking of iostream class objects

TABLE A–3 Man Pages for Sun WorkShop Memory Monitor

Man Page	Overview
`gcmonitor`	Web interface for Sun WorkShop Memory Monitor
`gcFixPrematureFrees`	Enable and disable fixing of premature frees by the Sun WorkShop Memory Monitor
`gcInitialize`	Configure Sun WorkShop Memory Monitor at startup

TABLE A–4 Man Pages for C++ Standard Library

Man Page	Overview
`Algorithms`	Generic algorithms for performing various operations on containers and sequences
`Associative_Containers`	Ordered containers
`Bidirectional_Iterators`	An iterator that can both read and write and can traverse a container in both directions
`Containers.`	A standard template library (STL) collection
`Forward_Iterators`	A forward-moving iterator that can both read and write
`Function_Objects`	Object with an operator() defined
`Heap_Operations`	See entries for make_heap, pop_heap, push_heap and sort_heap
`Input_Iterators`	A read-only, forward moving iterator
`Insert_Iterators`	An iterator adaptor that allows an iterator to insert into a container rather than overwrite elements in the container
`Iterators`	Pointer generalizations for traversal and modification of collections
`Negators`	Function adaptors and function objects used to reverse the sense of predicate function objects

TABLE A–4 Man Pages for C++ Standard Library *(continued)*

Man Page	Overview
Operators	Operators for the C++ Standard Template Library Output
Output_Iterators	A write-only, forward moving iterator
Predicates	A function or a function object that returns a boolean (true/false) value or an integer value
Random_Access_Iterators	An iterator that reads, writes, and allows random access to a container
Sequences	A container that organizes a set of sequences
Stream_Iterators	Includes iterator capabilities for ostreams and istreams that allow generic algorithms to be used directly on streams
__distance_type	Determines the type of distance used by an iterator - obsolete
__iterator_category	Determines the category to which an iterator belongs - obsolete
__reverse_bi_iterator	An iterator that traverses a collection backwards
accumulate	Accumulates all elements within a range into a single value
adjacent_difference	Outputs a sequence of the differences between each adjacent pair of elements in a range
adjacent_find	Find the first adjacent pair of elements in a sequence that are equivalent
advance	Moves an iterator forward or backward (if available) by a certain distance
allocator	The default allocator object for storage management in Standard Library containers
auto_ptr	A simple, smart pointer class
back_insert_iterator	An insert iterator used to insert items at the end of a collection

TABLE A–4 Man Pages for C++ Standard Library *(continued)*

Man Page	Overview
back_inserter	An insert iterator used to insert items at the end of a collection
basic_filebuf	Class that associates the input or output sequence with a file
basic_fstream	Supports reading and writing of named files or devices associated with a file descriptor
basic_ifstream	Supports reading from named files or other devices associated with a file descriptor
basic_ios	A base class that includes the common functions required by all streams
basic_iostream	Assists in formatting and interpreting sequences of characters controlled by a stream buffer.
basic_istream	Assists in reading and interpreting input from sequences controlled by a stream buffer
basic_istringstream	Supports reading objects of class basic_string<charT,traits,Allocator> from an array in memory
basic_ofstream	Supports writing into named files or other devices associated with a file descriptor
basic_ostream	Assists in formatting and writing output to sequences controlled by a stream buffer
basic_ostringstream	Supports writing objects of class basic_string<charT,traits,Allocator>
basic_streambuf	Abstract base class for deriving various stream buffers to facilitate control of character sequences
basic_string	A templatized class for handling sequences of character-like entities
basic_stringbuf	Associates the input or output sequence with a sequence of arbitrary characters
basic_stringstream	Supports writing and reading objects of class basic_string<charT,traits,Alocator> to/from an array in memory

TABLE A–4 Man Pages for C++ Standard Library *(continued)*

Man Page	Overview
binary_function	Base class for creating binary function objects
binary_negate	A function object that returns the complement of the result of its binary predicate
binary_search	Performs a binary search for a value on a container
bind1st	Templatized utilities to bind values to function objects.
bind2nd	Templatized utilities to bind values to function objects.
binder1st	Templatized utilities to bind values to function objects
binder2nd	Templatized utilities to bind values to function objects
bitset	A template class and related functions for storing and manipulating fixed-size sequences of bits
cerr	Controls output to an unbuffered stream buffer associated with the object stderr declared in <cstdio>
char_traits	A traits class with types and operations for the basic_string container and iostream classes
cin	Controls input from a stream buffer associated with the object stdin declared in <cstdio>
clog	Controls output to a stream buffer associated with the object stderr declared in <cstdio>
codecvt	A code conversion facet
codecvt_byname	A facet that includes code set conversion classification facilities based on the named locales
collate	A string collation, comparison, and hashing facet
collate_byname	A string collation, comparison, and hashing facet
compare	A binary function or a function object that returns true or false
complex	C++ complex number library
copy	Copies a range of elements

TABLE A–4 Man Pages for C++ Standard Library *(continued)*

Man Page	Overview
copy_backward	Copies a range of elements
count	Count the number of elements in a container that satisfy a given condition
count_if	Count the number of elements in a container that satisfy a given condition
cout	Controls output to a stream buffer associated with the object stdout declared in <cstdio>
ctype	A facet that includes character classification facilities
ctype_byname	A facet that includes character classification facilities based on the named locales
deque	A sequence that supports random access iterators and efficient insertion / deletion at both beginning and end
distance	Computes the distance between two iterators
divides	Returns the result of dividing its first argument by its second
equal	Compares two ranges for equality
equal_range	Finds the largest subrange in a collection into which a given value can be inserted without violating the ordering of the collection
equal_to	A binary function object that returns true if its first argument equals its second
exception	A class that supports logic and runtime errors
facets	A family of classes used to encapsulate categories of locale functionality
filebuf	Class that associates the input or output sequence with a file
fill	Initializes a range with a given value
fill_n	Initializes a range with a given value
find	Finds an occurrence of value in a sequence

TABLE A–4 Man Pages for C++ Standard Library *(continued)*

Man Page	Overview
find_end	Finds the last occurrence of a sub-sequence in a sequence
find_first_of	Finds the first occurrence of any value from one sequence in another sequence
find_if	Finds an occurrence of a value in a sequence that satisfies a specified predicate
for_each	Applies a function to each element in a range
fpos	Maintains position information for the iostream classes
front_insert_iterator	An insert iterator used to insert items at the beginning of a collection
front_inserter	An insert iterator used to insert items at the beginning of a collection
fstream	Supports reading and writing of named files or devices associated with a file descriptor
generate	Initialize a container with values produced by a value-generator class
generate_n	Initialize a container with values produced by a value-generator class
get_temporary_buffer	Pointer based primitive for handling memory
greater	A binary function object that returns true if its first argument is greater than its second
greater_equal	A binary function object that returns true if its first argument is greater than or equal to its second
gslice	A numeric array class used to represent a generalized slice from an array
gslice_array	A numeric array class used to represent a BLAS-like slice from a valarray
has_facet	A function template used to determine if a locale has a given facet

TABLE A–4 Man Pages for C++ Standard Library *(continued)*

Man Page	Overview
ifstream	Supports reading from named files or other devices associated with a file descriptor
includes	A basic set of operation for sorted sequences
indirect_array	A numeric array class used to represent elements selected from a valarray
inner_product	Computes the inner product A X B of two ranges A and B
inplace_merge	Merges two sorted sequences into one
insert_iterator	An insert iterator used to insert items into a collection rather than overwrite the collection
inserter	An insert iterator used to insert items into a collection rather than overwrite the collection
ios	A base class that includes the common functions required by all streams
ios_base	Defines member types and maintains data for classes that inherit from it
iosfwd	Declares the input/output library template classes and specializes them for wide and tiny characters
isalnum	Determines if a character is alphabetic or numeric
isalpha	Determines if a character is alphabetic
iscntrl	Determines if a character is a control character
isdigit	Determines if a character is a decimal digit
isgraph	Determines if a character is a graphic character
islower	Determines whether a character is lower case
isprint	Determines if a character is printable
ispunct	Determines if a character is punctuation
isspace	Determines if a character is a space

TABLE A–4 Man Pages for C++ Standard Library *(continued)*

Man Page	Overview
`istream`	Assists in reading and interpreting input from sequences controlled by a stream buffer
`istream_iterator`	A stream iterator that has iterator capabilities for istreams
`istreambuf_iterator`	Reads successive characters from the stream buffer for which it was constructed
`istringstream`	Supports reading objects of class `basic_string<charT,traits,Alocator>` from an array in memory
`istrstream`	Reads characters from an array in memory
`isupper`	Determines whether a character is upper case
`isxdigit`	Determines whether a character is a hexadecimal digit
`iter_swap`	Exchanges values in two locations
`iterator`	A base iterator class
`iterator_traits`	Returns basic information about an iterator
`less`	A binary function object that returns true if tis first argument is less than its second
`less_equal`	A binary function object that returns true if its first argument is less than or equal to its second
`lexicographical_compare`	Compares two ranges lexicographically
`limits`	Refer to the numeric_limits section of this reference guide
`list`	A sequence that supports bidirectional iterators
`locale`	A localization class containing a polymorphic set of facets
`logical_and`	A binary function object that returns true if both of its arguments are true
`logical_not`	A unary function object that returns true if its argument is false

TABLE A–4 Man Pages for C++ Standard Library *(continued)*

Man Page	Overview
logical_or	A binary function object that returns true if either of its arguments are true
lower_bound	Determines the first valid position for an element in a sorted container
make_heap	Creates a heap
map	An associative container with access to non-key values using unique keys
mask_array	A numeric array class that gives a masked view of a valarray
max	Finds and returns the maximum of a pair of values
max_element	Finds the maximum value in a range
mem_fun	Function objects that adapt a pointer to a member function, to take the place of a global function
mem_fun1	Function objects that adapt a pointer to a member function, to take the place of a global function
mem_fun_ref	Function objects that adapt a pointer to a member function, to take the place of a global function
mem_fun_ref1	Function objects that adapt a pointer to a member function, to take the place of a global function
merge	Merges two sorted sequences into a third sequence
messages	Messaging facets
messages_byname	Messaging facets
min	Finds and returns the minimum of a pair of values
min_element	Finds the minimum value in a range
minus	Returns the result of subtracting its second argument from its first
mismatch	Compares elements from two sequences and returns the first two elements that don"t match each other

TABLE A–4 Man Pages for C++ Standard Library *(continued)*

Man Page	Overview
`modulus`	Returns the remainder obtained by dividing the first argument by the second argument
`money_get`	Monetary formatting facet for input
`money_put`	Monetary formatting facet for output
`moneypunct`	Monetary punctuation facets
`moneypunct_byname`	Monetary punctuation facets
`multimap`	An associative container that gives access to non-key values using keys
`multiplies`	A binary function object that returns the result of multiplying its first and second arguments
`multiset`	An associative container that allows fast access to stored key values
`negate`	Unary function object that returns the negation of its argument
`next_permutation`	Generates successive permutations of a sequence based on an ordering function
`not1`	A function adaptor used to reverse the sense of a unary predicate function object
`not2`	A function adaptor used to reverse the sense of a binary predicate function object
`not_equal_to`	A binary function object that returns true if its first argument is not equal to its second
`nth_element`	Rearranges a collection so that all elements lower in sorted order than the nth element come before it and all elements higher in sorter order than the nth element come after it
`num_get`	A numeric formatting facet for input
`num_put`	A numeric formatting facet for output
`numeric_limits`	A class for representing information about scalar types

TABLE A–4 Man Pages for C++ Standard Library *(continued)*

Man Page	Overview
numpunct	A numeric punctuation facet
numpunct_byname	A numeric punctuation facet
ofstream	Supports writing into named files or other devices associated with a file descriptor
ostream	Assists in formatting and writing output to sequences controlled by a stream buffer
ostream_iterator	Stream iterators allow for use of iterators with ostreams and istreams
ostreambuf_iterator	Writes successive characters onto the stream buffer object from which it was constructed
ostringstream	Supports writing objects of class basic_string<charT,traits,Allocator>
ostrstream	Writes to an array in memory
pair	A template for heterogeneous pairs of values
partial_sort	Templatized algorithm for sorting collections of entities
partial_sort_copy	Templatized algorithm for sorting collections of entities
partial_sum	Calculates successive partial sums of a range of values
partition	Places all of the entities that satisfy the given predicate before all of the entities that do not
permutation	Generates successive permutations of a sequence based on an ordering function
plus	A binary function object that returns the result of adding its first and second arguments
pointer_to_binary_function	A function object that adapts a pointer to a binary function, to take the place of a binary_function
pointer_to_unary_function	A function object class that adapts a pointer to a function, to take the place of a unary_function
pop_heap	Moves the largest element off the heap

TABLE A–4 Man Pages for C++ Standard Library *(continued)*

Man Page	Overview
prev_permutation	Generates successive permutations of a sequence based on an ordering function
priority_queue	A container adapter that behaves like a priority queue
ptr_fun	A function that is overloaded to adapt a pointer to a function, to take the place of a function
push_heap	Places a new element into a heap
queue	A container adaptor that behaves like a queue (first in, first out)
random_shuffle	Randomly shuffles elements of a collection
raw_storage_iterator	Enables iterator-based algorithms to store results into uninitialized memory
remove	Moves desired elements to the front of a container, and returns an iterator that describes where the sequence of desired elements ends
remove_copy	Moves desired elements to the front of a container, and returns an iterator that describes where the sequence of desired elements ends
remove_copy_if	Moves desired elements to the front of a container, and returns an iterator that describes where the sequence of desired elements ends
remove_if	Moves desired elements to the front of a container, and returns an iterator that describes where the sequence of desired elements ends
replace	Substitutes elements in a collection with new values
replace_copy	Substitutes elements in a collection with new values, and moves the revised sequence into result
replace_copy_if	Substitutes elements in a collection with new values, and moves the revised sequence into result
replace_if	Substitutes elements in a collection with new values
return_temporary_buffer	A pointer-based primitive for handling memory

TABLE A–4 Man Pages for C++ Standard Library *(continued)*

Man Page	Overview
reverse	Reverses the order of elements in a collection
reverse_copy	Reverses the order of elements in a collection while copying them to a new collection
reverse_iterator	An iterator that traverses a collection backwards
rotate	Swaps the segment that contains elements from first through middle-1 with the segment that contains the elements from middle through last
rotate_copy	Swaps the segment that contains elements from first through middle-1 with the segment that contains the elements from middle through last
search	Finds a sub-sequence within a sequence of values that is element-wise equal to the values in an indicated range
search_n	Finds a sub-sequence within a sequence of values that is element-wise equal to the values in an indicated range
set	An associative container that supports unique keys
set_difference	A basic set operation for constructing a sorted difference
set_intersection	A basic set operation for constructing a sorted intersection
set_symmetric_difference	A basic set operation for constructing a sorted symmetric difference
set_union	A basic set operation for constructing a sorted union
slice	A numeric array class for representing a BLAS-like slice from an array
slice_array	A numeric array class for representing a BLAS-like slice from a valarray
smanip	Helper classes used to implement parameterized manipulators

TABLE A–4 Man Pages for C++ Standard Library *(continued)*

Man Page	Overview
smanip_fill	Helper classes used to implement parameterized manipulators
sort	A templatized algorithm for sorting collections of entities
sort_heap	Converts a heap into a sorted collection
stable_partition	Places all of the entities that satisfy the given predicate before all of the entities that do not, while maintaining the relative order of elements in each group
stable_sort	A templatized algorithm for sorting collections of entities
stack	A container adapter that behaves like a stack (last in, first out)
streambuf	Abstract base class for deriving various stream buffers to facilitate control of character sequences
string	A typedef for basic_string<char, char_traits<char>, allocator<char>>
stringbuf	Associates the input or output sequence with a sequence of arbitrary characters
stringstream	Supports writing and reading objects of class basic_string<charT,traits,Alocator> to/from an array in memory
strstream	Reads and writes to an array in memory
strstreambuf	Associates either the input sequence or the output sequence with a tiny character array whose elements store arbitrary values
swap	Exchanges values
swap_ranges	Exchanges a range of values in one location with those in anothe
time_get	A time formatting facet for input
time_get_byname	A time formatting facet for input, based on the named locales

TABLE A–4 Man Pages for C++ Standard Library *(continued)*

Man Page	Overview
time_put	A time formatting facet for output
time_put_byname	A time formatting facet for output, based on the named locales
tolower	Converts a character to lower case.
toupper	Converts a character to upper case
transform	Applies an operation to a range of values in a collection and stores the result
unary_function	A base class for creating unary function objects
unary_negate	A function object that returns the complement of the result of its unary predicate
uninitialized_copy	An algorithm that uses construct to copy values from one range to another location
uninitialized_fill	An algorithm that uses the construct algorithm for setting values in a collection
uninitialized_fill_n	An algorithm that uses the construct algorithm for setting values in a collection
unique	Removes consecutive duplicates from a range of values and places the resulting unique values into the result
unique_copy	Removes consecutive duplicates from a range of values and places the resulting unique values into the result
upper_bound	Determines the last valid position for a value in a sorted container
use_facet	A template function used to obtain a facet
valarray	An optimized array class for numeric operations
vector	A sequence that supports random access iterators
wcerr	Controls output to an unbuffered stream buffer associated with the object stderr declared in <cstdio>
wcin	Controls input from a stream buffer associated with the object stdin declared in <cstdio>

TABLE A–4 Man Pages for C++ Standard Library *(continued)*

Man Page	Overview
`wclog`	Controls output to a stream buffer associated with the object stderr declared in <cstdio>
`wcout`	Controls output to a stream buffer associated with the object stdout declared in <cstdio>
`wfilebuf`	Class that associates the input or output sequence with a file
`wfstream`	Supports reading and writing of named files or devices associated with a file descriptor
`wifstream`	Supports reading from named files or other devices associated with a file descriptor
`wios`	A base class that includes the common functions required by all streams
`wistream`	Assists in reading and interpreting input from sequences controlled by a stream buffer
`wistringstream`	Supports reading objects of class `basic_string<charT,traits,Allocator>` from an array in memory
`wofstream`	Supports writing into named files or other devices associated with a file descriptor
`wostream`	Assists in formatting and writing output to sequences controlled by a stream buffer
`wostringstream`	Supports writing objects of class `basic_string<charT,traits,Allocator>`
`wstreambuf`	Abstract base class for deriving various stream buffers to facilitate control of character sequences
`wstring`	A typedef for `basic_string<wchar_t, char_traits<wchar_t>, allocator<wchar_t> >`
`wstringbuf`	Associates the input or output sequence with a sequence of arbitrary characters

Index