C++ Library Reference

THE NETWORK IS THE COMPUTER™

SunSoft, Inc.
A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043 USA
415 960-1300 fax 415 969-9131

Part No.:802-5661-10
Revision A, December 1996

Please
Recycle

Contents

Preface xi

1. Introduction to C++ Libraries 1

Overview 1

Using Class Libraries 2

Using Standard Libraries 4

Using `libC` with Threads and Signals 4

Statically Linking Standard Libraries 5

Using Shared Libraries 6

Building Shared Libraries 7

Building Static Archives with Templates 8

Building Shared Libraries with Templates 9

Shared Library Behavior 10

2. The Coroutine Library 11

Introduction 11

Using the Coroutine Library 12

Structure of the Coroutine Classes. 12
Objects 12
Tasks 13
Class `task` 15
Waiting States for Tasks 19
Pending Objects 19
System Time 22
Timers 23
Queues. 24
FIFO Queues. 25
Queue Modes 28
Queue Size 29
Cutting and Splicing 29
Scheduling 32
Random Numbers 33
Histograms 34
Real-Time and Interrupts 35
Coroutine Library Limitations 39
3. The Complex Arithmetic Library 41
Introduction 41
The Complex Library. 42
Type `complex` 42
Constructors of Class `complex` 42
Arithmetic Operators 43

Mathematical Functions 44
Error Handling 46
Input and Output 47
Mixed-Mode Arithmetic 48
Efficiency 49
Complex Man Pages 50
4. The `Iostream` Library 51
Predefined `Iostreams` 51
Basic Structure of `Iostream` Interaction 51
`Iostreams` 53
Output Using `Iostreams` 53
Input Using `Iostreams` 57
Defining Your Own Extraction Operators 58
Using the `char*` Extractor 59
Reading Any Single Character 59
Binary Input 60
Peeking at Input 60
Extracting Whitespace 60
Handling Input Errors 61
Using `Iostreams` with `stdio` 62
Creating `Iostreams` 62
Dealing with Files Using Class `fstream` 62
Assignment of `Iostreams` 66
Format Control 67

Manipulators 67
Using Plain Manipulators 68
Parameterized Manipulators 71
`Strstreams`: `Iostreams` for Arrays 72
`Stdiobufs`: `Iostreams` for `stdio` files 72
`Streambufs` 73
Working with `Streambufs` 73
Using `Streambufs` 73
`Iostream` ManPages 75
`Iostream` Terminology 76
5. Using `iostreams` in a Multithreaded Environment 79
Organization of the MT-safe `iostream` Library 80
Public Conversion Routines 82
Compiling and Linking with the MT-safe `libC` Library . . 83
MT-safe `iostream` Restrictions 83
Performance 85
Interface Changes to the `iostream` Library 87
New Classes 87
New Class Hierarchy 88
New Functions 88
Global and Static Data 90
Sequence Execution 91
Object Locks 91
Class `stream_locker` 92

MT-safe Classes 94
Object Destruction 94
An Example Application. 95
A. Coroutine Examples 99
B. Associated Man Pages 103
Index 105

Tables

Table P-1	Notational Conventions	xv
Table 1-1	Command-Line Flags for Standard Libraries	3
Table 2-1	Basic Types in a Coroutine Library	12
Table 2-2	Two Base Classes in a Coroutine Library	12
Table 2-3	Public Members of Class `object`	13
Table 2-4	Class `task` States	15
Table 2-5	Public Parts of Class `task`	17
Table 2-6	Public Parts of Class `timer`	23
Table 2-7	Queue Functions	29
Table 2-8	Random-Number Classes	33
Table 2-9	Class `randint`	33
Table 2-10	Class `urand`	34
Table 2-11	Class `erand`	34
Table 2-12	Class `histogram`	34
Table 3-1	Complex Arithmetic Library Functions	44
Table 3-2	Complex Mathematical and Trigonometric Functions	45

Table 3-3 Complex Arithmetic Library Functions 47
Table 3-4 Man Pages for Type `complex` 50
Table 4-1 `Iostream` Routine Header Files 53
Table 4-2 `Iostream` Predefined Manipulators 67
Table 4-3 `Iostream` Man Pages Overview 75
Table 4-4 `Iostream` Terminology 76
Table 5-1 Core Classes 80
Table 5-2 Reentrant Public Functions 82
Table 5-3 New Classes 87
Table 5-4 New Class Hierarchy 88
Table 5-5 New Functions 88
Table B-1 Man Pages for Complex Library 103
Table B-2 Man Pages for Iostream Library 103
Table B-3 Man Pages for Coroutine Library 104

Preface

Audience

This manual, *C++ Library Reference*, is for programmers who use the C++ programming language.

Purpose of this Manual

This manual gives information on how to use the following C++ libraries:

- Complex
- Coroutine
- `Iostream`

It also lists the manual pages (man pages) for the above libraries and complements the complete C++ documentation set described in the "Documentation" section which follows.

Prerequisite Reading

Although there is no required prerequisite reading for this manual, you should have access to good C++ reference books, such as *The C++ Programming Language* by Bjarne Stroustrup.

You should also have access to the documents described in the following section.

Documentation

C++ Package

The following documentation is included in the C++ package:

Manuals

- *C++ User's Guide*—Describes the use of the compiler. It also contains information on converting source code from previous versions of C++.
- *C++ Library Reference*—Provides a complete definition of this release of C++.
- *Sun WorkShop Installation and Licensing Guide*—Tells you how to install the C++ software and other Sun™ software on the Solaris™ operating environment.
- *Profiling Tools*—Describes some useful utilities to aid you in programming such as `prof`, `gprof`, `lprof`, and `tcov`.
- *Tools.h++ User's Guide*—Introduces you to and tells you how to use the Tools.h++ class library.
- *Tools.h++ Class Library Reference*—Describes a set of C++ classes that can greatly simplify your programming while maintaining the efficiency for which C++ is famous.

Articles

- **"Close as Possible to C, But No Closer"**

  An article by Andrew Koenig and Bjarne Stroustrup.

- **"Object-Oriented Programming"**

  An article by Bjarne Stroustrup.

- **"What Every Computer Scientist Should Know About Floating-Point Arithmetic"**

  A floating-point white paper by David Goldberg included in the `README` directory.

Online Documentation

- **Online Books**

  Certain manuals are available through online documentation viewing tools that take advantage of dynamically linked headings and cross-references. Online documentation enables you to electronically jump from one subject to another and to search for topics by using a word or phrase.

- **Error Messages**

  Error messages give useful information to help you code and debug your program.

- **Manual pages** (man pages)

  Display the man pages with the `man` command. To access a man page type: `man` *name*. Man pages are in:

  `/opt/SUNWspro/man`

Note – Before you use the `man` command, insert this directory at the beginning of your search path. This is usually done in the `.cshrc` file, in a line with `setenv MANPATH=` at the start; or in the `.profile` file, in a line with `export MANPATH=` at the start. For the Bourne shell: `MANPATH=...`at the start, followed by the line `export MANPATH.`

- `README` **file**

  The `README` file gives last-minute information about new software features and bug fixes. To access, type `CC -readme`

- *C++ Migration Guide*

  Helps you migrate your code from C++ 3.0 to the current compiler. This manual (also found in Appendix A of the *C++ User's Guide*) is displayed when you type `CC -migration`.

Solaris

These manuals are available to you online, and are bundled with the operating system documentation:

- *Programming Utilities and Libraries*

The *Programming Utilities and Libraries* manual provides information on the tools that can aid you in programming. These include:

`lex`(1)—Generates programs used in simple lexical analysis of text; solves problems by recognizing different strings of characters.

`yacc`(1)—Imposes structure on computer input and turns it into a C language function that examines the input stream.

`prof`(1)—Produces an execution profile of the modules in a program.

`make`(1S)—Automatically maintains, updates, and regenerates related programs and files.

System V make—Describes a version of `make`(1) that is compatible with older versions of the tool.

`sccs`(1)—Allows control access to shared files and keeps a history of changes made to a project.

`m4`(1)—Processes macro languages.

- *SunOS 5.x Linker and Libraries Manual*
- SunOS 4.x linker and libraries documentation

Commercially Available Books

The following is a partial list of available books on C++.

- *Scientific C++: Building Numerical Libraries,* Guido Buzzi-Ferraris (Addison-Wesley, 1993)
- *A C++ Primer,* 2nd Ed, Stanley B. Lippman (Addison-Wesley, 1989)
- *A Guide to Object-Oriented Programming in C++*, Keith Gorlen (John Wiley & Sons)
- *C++ for C Programmers*, Ira Pohl (Benjamin/Cummings, 2nd Ed, 1994)
- *C++ IOStreams Handbook*, Steve Teale (Addison-Wesley, 1993)
- *The Annotated C++ Reference Manual*, Margaret A. Ellis and Bjarne Stroustrup (Addison-Wesley, 1990)
- *The C++ Programming Language*, 2nd Ed, Bjarne Stroustrup (Addison-Wesley, 1991)

- *Object-Oriented Design with Applications*, 2nd Ed, Grady Booch (Addison-Wesley)
- *Effective C++—50 Ways to Improve Your Programs and Designs*, Scott Meyers
- *Scientific & Engineering C++*, John Barton and Lee Nackman (Addison-Wesley, 1994)

Notational Conventions

The following table describes the notational conventions and symbols used in this manual.

Table P-1 Notational Conventions

Typeface or Symbol	**Meaning**	**Example**
`AaBbCc123`	Command, file, and directory names; on-screen computer output; C++ statements and key words; operating system programs.	Edit your `.login` file. Use `ls -a` to list all files. `system% You have mail.`
`AaBbCc123`	User input, contrasted with on-screen computer output	`system% su` `password:`
AaBbCc123	General arguments, parameters that you replace with appropriate input.	To delete a file, type **`rm`** *filename*.
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be root to do this.

Code samples are included in boxes and may display the following:

%	C shell prompt	`demo%`
$	Bourne shell prompt	`demo$`
#	Superuser prompt, either shell	`demo#`
[ ]	Square brackets contain arguments that can be optional or required.	`-d[y|n]`

Table P-1 Notational Conventions *(Continued)*

Typeface or Symbol	Meaning	Example
|	The "pipe" or "bar" symbol separates arguments, only *one* of which may be used at one time.	`-d[y|n]`
,	The comma separates arguments, *one or more* of which may be used at one time.	`-xinline=[`*f1,..,fn*`]`
:	The colon, like the comma, is sometimes used to separate arguments.	`-R`*dir*`[:`*dir*`]`
...	The ellipsis indicates omission in a series.	`-xinline=[`*f1,..,fn*`]`
%	The percent sign indicates the word following it has a special meaning.	`-ftrap=%all`
<>	In ASCII files, such as the `README` file, angle brackets contain a variable that must be replaced by an appropriate value.	`-xtemp=<dir>`

Introduction to C++ Libraries 1

Overview

Class libraries are modular components of reusable code. Using class libraries 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 three class libraries provided with the C++ compiler:

- Coroutines (tasks), described in Chapter 2, “The Coroutine Library”
- Complex numbers, described in Chapter 3, “The Complex Arithmetic Library”
- `Iostreams`, described in Chapter 4, “The Iostream Library”

1

Using Class Libraries

Generally, two steps are involved in using a class library. First, include the appropriate header in your source code. Second, link your program with the object library. Code Example 1-1 is an example of how to use the `iostream` class library. First, the source code, which we assume is in a file called `prog.cc`:

Code Example 1-1 Using the `iostream` Class Library

```
// file prog.cc
#include <iostream.h>

main()
{
    cout << "Hello, world!\n";
    return 0;
}
```

This simple example includes the basic header for the `iostream` classes, `iostream.h`. It then makes use of the predefined output stream `cout`, and the overloaded `operator<<` (often pronounced “insert”) to accomplish output.

Linking the final program requires nothing extra when using `iostreams`. The object code for the library is included in `libC.a`, which is always linked with your program. The command that compiles and links `prog.cc` into an executable program called `prog` is:

```
demo% CC prog.cc -o prog
```

The complex number and coroutine libraries have their own separate object libraries, and require that the appropriate library be linked explicitly. Code Example 1-2 is an example that uses complex numbers. It creates a complex number having the value `1+i`, then prints it out using `iostreams`:

Code Example 1-2 Using the Complex Library

```
// file prog2.cc
#include <iostream.h>
#include <complex.h>

main()
{
    complex OnePlusI(1.0, 1.0);
    cout << OnePlusI << "\n";
    return 0;
}
```

When you link this program, you link the `iostream` library automatically, but you need to link the complex number library explicitly.

```
demo% CC prog2.cc -o prog2 -library=complex
```

The `-l` flag causes the `CC` driver to find the complex library in its standard place and link it into the program. See the manual page `CC`(1) for more information about this flag.

Note – The `-l` flag appears at the end of the command line.

Alternatively, use the `–library` plus the command-line flags for the supplied libraries listed in Table 1-1:

Table 1-1 Command-Line Flags for Standard Libraries

Library	flag
`iostream`	none needed
`complex`	`-library=complex`

Table 1-1 Command-Line Flags for Standard Libraries *(Continued)*

Library	flag
tasking	-ltask
Tools.h++ v6	-library=rwtool6
Tools.h++ v7	-library=rwtool7

Using Standard Libraries

Under normal circumstances, you need not do anything special to compile a program that calls routines in a standard library. However, the standard library header file must be included at the beginning of your program using a format like:

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

The standard directory location for the system header files is:

```
/usr/include
```

The standard location for C++ header files is:

`/opt/SUNWspro/SC4.2/include/CC`

If the header files you want to use are in a different directory from the standard location, you can specify the location on the `CC` command line. For example, if the header files are in `/usr/libraries/include`, you can specify that location in the following command:

```
demo% CC -I/usr/libraries/include myprog.cc
```

Using `libC` *with Threads and Signals*

The `libC` library is multi-thread safe (see Chapter 5), but is not `async` safe. This means that in a multi-threaded application, functions available in `libC` should not be used in signal handlers. Doing so could result in a deadlock situation.

It is not safe to use the following in a signal handler in a multi-threaded application:

- iostreams
- `new` and `delete`
- exceptions

Statically Linking Standard Libraries

The `CC` driver links in several libraries by default, including `libc` and `libm`, by passing `-l` options to `ld`. The options are:

`-lC, -lC_mtstubs, -lm, -lw, -lcx, and -lc`

These options link shared versions of the libraries `libC`, `libw`, `libm`, and `libc`. If you want some of these libraries to be linked statically, you can use `-nolib` described in the *C++ User's Guide*. With the `-nolib` option, the driver does not pass any `-l` options to `ld`; you must pass these options yourself. The following example shows how you would link statically with `libC`, and dynamically with `libw, libm,` and `libc` on Solaris 2.x:

```
demo% CC test.c -nolib -Bstatic -lC -lC_mtstubs -Bdynamic -lm
-lw -lcx -lc
```

The order of the `-l` options is important. The `-lC`, --lm, `-lw,` and `–lcx` options appear before `-lc`. `-nolib` suppresses all `-l` options that are passed to `ld`. Some `CC` options link to other libraries. These library links are also suppressed by `-nolib`.

Note – The `-lcx` option does not exist on Intel or PowerPC.

For example, using the `-mt` option causes the `CC` driver to pass `-lthread` to `ld` in addition to passing `-lC`, `-lm`, `-lw`, `–lcx`, and `-lc`. If you use both `–mt` and `–nolib`, the `CC` driver does not pass any `-l` options to `ld`. For further information on `-nolib`, see the *C++ User's Guide*. For further information on `ld`, see the *Linker and Libraries Guide.*

You may also use the `–library` and `–staticlib` flags to link statically. This alternative is much easier than the one described above. The previous example, for instance could be performed as:

```
% CC test.c -staticlib=libC
```

Using Shared Libraries

The following shared libraries are included:

`libC.so.5, libcomplex.so.5, librwtool.so.2`

The occurrence of each shared object is recorded in the resulting `a.out` file; this information is used by `ld.so` to perform dynamic link editing at runtime. Because the work of incorporating the library code into an address space is deferred, the runtime behavior of the program using shared library is sensitive to an environment change, that is, moving a library from one directory to another. For example, if your program is linked with `libcomplex.so.5` in `/opt/SUNWspro/SC4.2/lib` on Solaris 2.x, and the `libcomplex.so.5` library is later moved into `/opt2/SUNWspro/SC4.2/lib`, the following message is displayed when you run the binary code:

```
ld.so: libcomplex.so.5: not found
```

You can still run the old binary code without recompiling it by setting the environment variable `LD_LIBRARY_PATH` to the new library directory.

In a C shell:

```
demo% setenv LD_LIBRARY_PATH \
/opt2/SUNWspro/SC4.2/lib:{$LD_LIBRARY_PATH}
```

In a Bourne shell:

```
demo$ LD_LIBRARY_PATH=/opt2/SUNWspro/SC4.2/lib:{$LD_LIBRARY_PATH}
demo$ export LD_LIBRARY_PATH
```

The `LD_LIBRARY_PATH` has a list of directories, usually separated by colons. After you type `a.out`, the dynamic loader searches the directories in `LD_LIBRARY_PATH` before the default directories.

To see which libraries are linked dynamically in your executable, use the `ldd` command, as follows:

```
% ldd a.out
```

This step should rarely be necessary, because the shared libraries are seldom moved.

For further information on using shared libraries, please see the Solaris 2.x *Linker and Libraries Guide*, and the SunOS 4.x linker documentation.

Building Shared Libraries

In the following example, `lsrc1.cc` and `lsrc2.cc` are C++ modules that contain library functions. `sa1.cc` and `sa2.cc` are modules that contain exported library objects that must be initialized.

Because of the nature of C++ and the automatic generation of some object files, such as templates, always use the `CC` command to build libraries to ensure that these object files are correctly added to your library.

The C++ compiler does not initialize global variables if they are defined in a shared library. For initializers and exceptions to work, you must use the `CC -G` command to build a shared library.

When shared libraries are opened with `dlopen`, `RTLD_GLOBAL` must be used for exceptions to work.

To build a C++ shared library `libfoo.so.1`, type:

```
% CC -G -pic -o libfoo.so.1 lsrc1.cc lsrc2.cc
```

To assign a name to a shared library for versioning purposes, type:

```
% CC -G -pic -o libfoo.so.1 lsrc1.cc lsrc2.cc -h libfoo.so.1
```

1

Building Static Archives with Templates

The mechanism of using templates to build static archives is identical to that of building an executable. The driver `CC` is used in place of `ar`. `CC` automatically invokes `tdb_link`, which handles the preprocessing of object files that may contain templates or references to templates. Without `tdb_link`, referenced templates may not be included in the archives as required. For example:

Code Example 1-3 Array Class

array.h	`#ifndef _ARRAY_H_`
	`#define _ARRAY_H_`
	`const int ArraySize = 20;`
	`template <class Type> class Array {`
	`  private:`
	`  Type* data;`
	`  int  size;`
	`  public:`
	`  Array(int sz=ArraySize);`
	`  int GetSize();`
	`};`
	
	`#endif // _ARRAY_H_`

array .cc	`#include "array.h"`
	
	`template <class Type> Array<Type>::Array(int sz)`
	`{`
	`  size = sz;`
	`  data = new Type[size];`
	`}`
	
	`template <class Type> int Array<Type>::GetSize()`
	`{`
	`  return size;`
	`}`

Code Example 1-4 Array Class

foo.cc

```
#include "array.h"

int foo()
{
    Array<int> IntArray;
    int size = IntArray.GetSize();
    return size;
}
```

When the above program is compiled with `CC`, three object (`.o`) files are created; for example `foo.o`, `constructor.o`, and `GetSize.o`. The two template object files, `Array_constructor.o` and `GetSize.o`, are placed in the template repository. If `ar` is used to build an archive, the three files must be manually included in the command line to resolve the template references. You may not be able to accomplish this in a normal programming environment since `make` may not know which template files are actually created and referenced. The solution is to use the `-xar` option, such as:

```
% CC -c foo.cc # Compile main file, templates are created
% CC -xar -o foo.a foo.o # "Link" the files, placing them in an
archive
```

The `-xar` flag causes `CC` to create an archive. The `-o` directive is required to name the newly created library. `tdb_link` examines the object files on the command line, cross-references the object files with those known to the template database, and adds those templates required by the user's object files (along with the main object files themselves) to the archive. Using the `-xar` flag is only for creating or updating an existing archive, not for maintaining the archive. It is equivalent to specifying `ar -cr.`

Building Shared Libraries with Templates

Shared libraries are built in the same way as static libraries, except for one difference. Instead of specifying `-xar` on the command line, use `-G` instead. When `tdb_link` is invoked via `CC`, a shared library is created instead of a static archive. All object files on the command line should have been compiled with `-pic`.

To create a shared library using the above source files:

```
%CC -G -pic -c foo.cc # Compile main file, templates are created
%CC -G -o foo.so foo.o # "Link" the files, placing them in a
shared library
```

Shared Library Behavior

All static constructors and destructors are called from the `.init` and `.fini` sections respectively. All static constructors in a shared library linked to an application are called *before* `main()` is executed. This behavior is different from that on Solaris 1.x, where only the static constructors from library modules used by the application are called.

The Coroutine Library 2

Introduction

A *coroutine* program is made up of routines that run in parallel with other routines instead of carrying out their actions and terminating like ordinary functions. These special routines, called coroutines or *tasks*, can communicate with one another and can give rise to other coroutines. The coroutine library provides a set of classes that enable you to write programs in this style.

Tasks do not actually execute concurrently. A single task continues to execute until it suspends or terminates itself; usually, another task then resumes execution. You can use the coroutine library to simulate concurrent execution, and use simulated time to make the execution of the coroutines actually appear parallel.

Note – If your application is compiled with `-O4 or O5`, you will not be able to use the coroutine library. You must use an `-O3 or` lower level of optimization in order to use the coroutine library.

Note – As of the time of this printing, the coroutine library will not be supported beyond the current version.

2

Using the Coroutine Library

To use the task library, include the header file `task.h` in your program, and link with the `-ltask` option.

Structure of the Coroutine Classes

The coroutine library provides basic types. These types are described in Table 2-1.

Table 2-1 Basic Types in a Coroutine Library

Type	Description
Task	A coroutine is created as an instance of any class immediately derived from class `task`. The body of the coroutine is the constructor of the derived class.
Queue	A data structure that makes ordered collections of objects. Classes `qhead`, `qtail`.
Timer	A class that implements time-outs and other time-dependent functions. Class `timer`.
Histogram	A data structure provided to help gather data.
Interrupt handler	A class that represents external events. Class `Interrupt_handler`.

In addition, two important base classes are described in Table 2-2.

Table 2-2 Two Base Classes in a Coroutine Library

Base Class	Definition
Class `object`	Provides the root of class hierarchy.
Class `sched`	Provides the basic definition for an object that knows about time. Used as a base class for classes `timer` and `task`, and implements task scheduling.

Objects

The coroutine library defines class `object` as a base class for other classes in the library. For example, messages passed between tasks are instances of classes derived from class `object`. You can derive your own special-purpose classes from class `object`.

The public members of interest in class `object` are described in Table 2-3:

Table 2-3 Public Members of Class `object`

Class member	Description
`enum objtype`	`OBJECT`, `TIMER`, `TASK`, `QHEAD`, `QTAIL`, `INTHANDLER`
`objtype o_type();`	(virtual) Returns the type of the current object.
`int pending();`	(virtual) Returns non-zero (true) if not ready.
`void print(int how, int =0);`	(virtual) Primarily for debugging. Prints out all state information for a task and its base classes. Parameter `how` takes any combination of `CHAIN` (data on all tasks in chain) and `VERBOSE` (extra information) bits. The second parameter affects indentation of printed information and is for internal use.
`void alert();`	Makes remembered tasks eligible for execution.
`void forget(task*);`	Forgets a previously remembered task.
`void remember(task*);`	Remembers a task for `alert`.
`static task* this_task();`	Returns the currently running task.

Tasks

Tasks are the basic features of coroutine-style programming. A task runs until it explicitly allows another task to run. When one task suspends or terminates itself, the task system chooses the next task on the list of ready-to-run tasks and runs it.

A task can give up control of the processor by suspending or terminating itself, but nothing can force it to do so. The currently active task is always in control. No task can preempt another task.

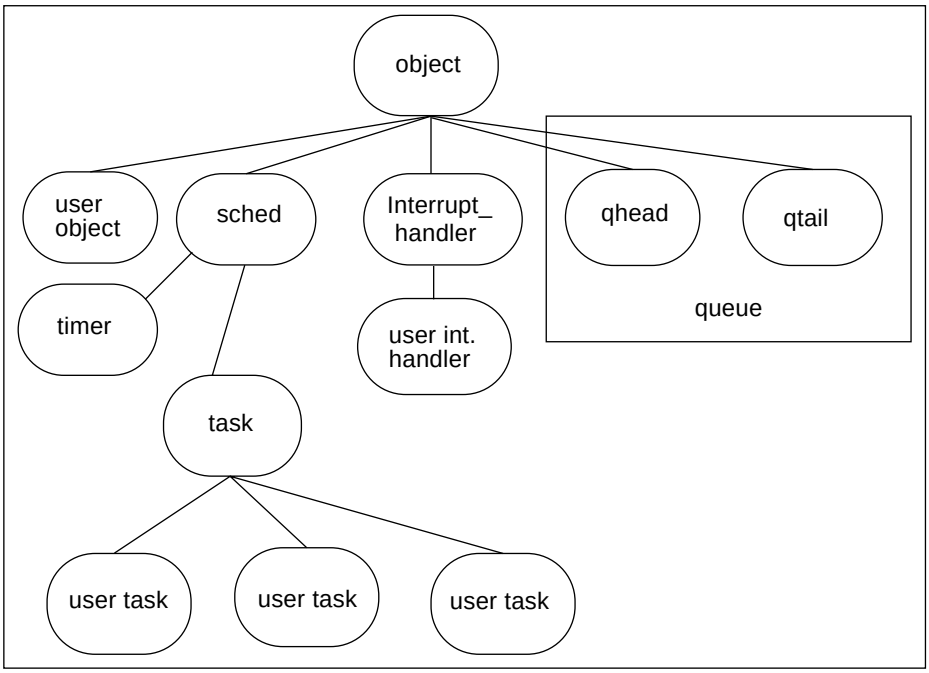
When a task suspends, the task system saves the state of the task so the task can get its environment back when it resumes. This behavior generally means saving the stack and hardware registers. The task system then restores the environment of another task and that task resumes execution.

A task system is like the operating system: each task is a process that carries on its individual action and communicates with other processes. There are important differences, however:

- A task system is a single operating system process. The task system relies on the operating system for I/O, memory management, and other functions that every real operating system must perform.
- Every task in a task system shares the same address space. Processes under the operating system have their own address spaces. Sharing address space has an advantage in that tasks can share information simply by passing pointers, but a disadvantage in that a badly behaved task can interfere with other tasks.
- A task system can support hundreds or thousands of times as many concurrent tasks as an operating system can support processes. Simulations written with the task library often have thousands of tasks.

Figure 2-1 shows the organization of the classes in the coroutine library.

Figure 2-1 Coroutine Library Structure.



Class `task`

A task is an object of a class derived from class `task`. The action of a task is contained in the constructor of the task's class. Before returning, the constructor of a task terminates it by a call to `resultis`. A task is always in one of three states, as shown in Table 2-4.

Table 2-4 Class `task` States

State	Definition
`RUNNING`	Executing instructions or on scheduler's ready-to-run list.
`IDLE`	Waiting for something to happen before returning to running state (suspended).
`TERMINATED`	Completely finished running. Cannot return to a running or idle state. Another task can still access its result if it has not been destroyed.

This example shows a portion of the public interface of class `task`. Part of it is inherited from class `sched`:

```
class task : public sched {
public:
    enum modetype { DEDICATED, SHARED };

protected:
    task(char* =0, modetype =DEFAULT_MODE, int =SIZE);

public:
    ~task();

     task* t_next;
      unsigned char* t_name;

      void      wait(object*);
      int       waitlist(object*...);
      int       waitvec(object**);

      void      delay(int);
      int       preempt();
      void      sleep(object* =0);

      void      resultis(int);
      void      cancel(int);
      void      print(int, int =0);
      // Flags for first parameter of print
      #define CHAIN ...
      #define VERBOSE ...

      //These are inherited from class sched

      enum statetype {IDLE, RUNNING, TERMINATED };
      statetype rdstate();
       long rdtime();
      int result();
};
```

Parts of a Task

Table 2-5 describes the public part of class task:

Table 2-5 Public Parts of Class `task`

Class `task` **Public Part**	**Description**
`enum modetype`	The task stack may be `DEDICATED` or `SHARED`. The default mode is dedicated (see coroutine library man pages).
`task(char* name=0, mode typemode=DEDICATED, int stacksize=3000)`	Constructor for class `task`. Protected to prevent creation of objects type `task`. You must derive your own class from `task`.
`~task()`	Destructor for class `task`. Takes care of cleanup.
`task* t_next`	Points to the next task in the task list.
`unsigned char* t_name`	Optional name of a task. You can give each task object a name, which is printed as a debugging aid.
`wait(), waitlist(), waitvec(), delay(), preempt(), sleep()`	Functions that deal with suspending a task (see “Waiting States for Tasks” on page 19).
`void resultis(int)`	Sets the return value of task and terminates it. Use this function instead of `return` from a task. (You *cannot* use `return`.) `resultis` also invokes the task scheduling mechanism.
`void cancel(int)`	Like `resultis`, sets return value of task and terminates it. Does not invoke task scheduling, so is a useful way to terminate another task without interrupting the current task.
`enum statetype`	The states that a task may be in: `IDLE`, `RUNNING`, `TERMINATED`.
`statetype rdstate()`	Returns the state of a task.
`long rdtime()`	Returns the current time. A simulated time is kept, which provides the illusion of passing time and simultaneous task execution.
`int result()`	Returns the result value of another task. That value is provided by `resultis` or `cancel`. A task cannot call `result` for itself. If the queried task has not terminated, calling task is suspended until queried task terminates and thus has a result to return.

A Simple Task Example

A simple example of a task is one where the function `main` creates two tasks, one of which needs to get information from the other. Appendix A, “Coroutine Examples” shows this example.

In the following example, one task gets a string from the user while the second counts the number of `'0'` characters in the string.

The task classes are `getLine` and `countZero`.

Code Example 2-1 Classes

```
class getLine : public task {
public:
    getLine();
};

class countZero : public task {
public:
    countZero(getLine*);
};
```

The implementation of the constructor for `getLine` is simple.
Code Example 2-2 assumes type `int` and `char*` are the same size and can be freely cast back and forth. This may not be the case with other C++ implementations:

Code Example 2-2 `getline` Constructor

```
getLine::getLine()
{
    char* tmpbuf = new char[512];
    cout << "Enter string: ";
    cin >> tmpbuf;
    resultis((int)tmpbuf);
}
```

Code Example 2-3 shows the constructor for `countZero`.

Code Example 2-3 `countZero` Constructor

```
countZero::countZero(getLine *g)
{
    char *s, c;
    int i = 0;
    s = (char*)g->result();
    while( c = *s++)
        if( c == '0' )
            i++;
    resultis(i);
}
```

The main program looks like this:

Code Example 2-4 Zero-char Counter Main Program

```
// Simple zero-char counter program
int main()
{
    getLine g;
    countZero c(&g);
    cout << "Count result = "
         << c.result() << "\n";
    thistask->resultis(0);
    return 0;
}
```

Waiting States for Tasks

When a task waits for some other task to take some action or produce some information, it becomes `IDLE`. Later, when the condition that led to its suspension becomes satisfied, the task again becomes `RUNNING`.

A `RUNNING` task state does not necessarily mean the task is executing. The task may be on the ready-to-run list, which means that it will execute eventually.

Pending Objects

An object is said to be *pending* if it is waiting for some event. For example, an empty queue head is pending, since nothing can be removed until an item is appended.

A task can call the `pending()` member function for another object to find out if it is pending. When an object is no longer pending, it calls `alert` to notify other objects that are waiting for it that it is no longer pending.

Calling `result` *to Wait for Information*

`result` is a task member function that a task can call on another task. It returns a single `int` value. For example:

```
// within someTask()
secondClass secondObject();
int i = secondObject.result();
```

If `secondObject` has not terminated when `someTask` calls member function `result`, `someTask` is suspended (becomes `IDLE`) until `secondObject` does terminate. At that point, `someTask` resumes (becomes `RUNNING`) with the result from `secondObject` available.

Suspending when Dealing with Queues

When you try to get a message from an empty queue (see “FIFO Queues” on page 25) or try to put a message in a full queue, the queue function suspends your task if the mode of the queue is WMODE. When the condition passes, your task becomes RUNNING again.

Putting Your Task to Sleep

You can put a task to sleep until a pending task is no longer pending. If the task you want to wait for is not a pending task and you use `sleep`, the calling task suspends itself indefinitely. If you want to wait for a task that may be nonpending, and have your task continue execution, use `wait`. You can put a task to sleep by calling:

```
void sleep(object* t = 0);
```

The calling task goes to sleep until the object pointed to by the parameter is no longer pending. If the task is not pending when you execute this call, the calling task goes to sleep indefinitely. If you give a null pointer—as in `sleep(0)`—your task goes to sleep indefinitely.

Waiting for an Object

You can make a task can wait for another task to become ready (nonpending) by using the `wait` task member function. Make a task wait by calling:

```
void wait(object* ob);
```

The calling task waits until the object pointed to by the parameter is no longer pending. If the task is not pending when you execute this call, or the object pointer is null, the calling task is not suspended.

Waiting for a List of Tasks

Tasks have two member functions that make them wait for any one of a list of pending objects to become no longer pending. The two functions are:

```
int waitlist(object*, ...);
int waitvec(object**);
```

You give `waitlist` a null-terminated list of objects to wait for. These objects can be queues, tasks, or other objects as shown in the following example,

```
qhead* firstQ;
qtail* secondQ;
taskType* aTask;
. . .
int which = waitlist(firstQ, secondQ, aTask, (object*)0);
```

If all of the items are pending, the calling task is suspended (becomes `IDLE`). When any one of the items in the list becomes ready (no longer pending), `waitlist` returns and the calling task resumes its `RUNNING` state. If one of the items is ready when it is called, `waitlist` returns immediately. The return value of `waitlist` is the position in the list of a ready task, counting from 0. There may be more than one ready task, in which case one is arbitrarily identified as the task that caused `waitlist` to return.

`waitvec` works exactly like `waitlist`, except that it takes a null-terminated vector (array) of objects. The following example is equivalent to the example using `waitlist`:

```
object* vec[] = {firstQ, secondQ, aTask, 0};
int which = waitvec(vec);
```

Waiting for a Predetermined Time

You can set a specific timed delay. With this kind of delay, the task remains in a RUNNING state, thus simulating the passage of time. (See “System Time” on page 22.) For example:

```
// ... do something
delay(6); // wait
// ... do s
ome more
```

In this example, after the call to `delay` has returned, six units of simulated time will have passed. Other tasks may or may not have run in the meantime, depending on their own scheduling requests.

System Time

The task system maintains a simulated time, which need not be (and usually is not) related to real time. The static member function `sched::get_clock` returns the current simulated time, which is by default initialized to zero. The static member function `sched::setclock` can be used to initialize the system clock to a starting time, but cannot be called once the time has advanced.

Function `task::delay` is the only way for a `task` to cause the system time to advance. The current task is set to run again when the specified number of time units have passed. The scheduler checks the scheduled runtime for the next task on the task list and advances the system time to that value. Eventually, the task requesting the delay reaches the front of the task list, and simulated time will have advanced by the requested amount.

A task can also create a `timer`, an object which exists for a predetermined amount of simulated time and which can be waited on, as described in the next section.

Timers

A `timer` behaves like a mini-task whose only function is to execute a delay; it has no result. Like any object, it can be waited on. One difference from a `task` is that a `timer` can be reset; it does not have to be destroyed first and reconstructed. Table 2-6 describes the public parts of class `timer`:

Table 2-6 Public Parts of Class `timer`

Public Part of Class `timer`	**Description**
`timer(int delay)`	Constructor for a timer of specified lifetime.
`~timer()`	Destructor that takes care of cleanup.
`void reset(int delay)`	Sets a new delay value for a `timer`, so that it can be reused.

One use for a timer is for implementing a time-out. Suppose you want to wait for some task, `get_input`, but for not more than five time units. You can use a `timer` like this:

Code Example 2-5 Using the `timer` Class

```
input_task *get_input = new input_task(...);
timer timeout(5); // expires in 5 units
switch( waitlist(get_input, &timeout, 0) ) {
  case 0: // input completed
          timeout.reset(0); // cancel the timer
          ... // do something with input
          break;
  case 1: // timer expired
          ... // do something without input
          break;
  default: // impossible!
          ...
}
// timer can be reset and used again if desired
```

2

Queues

In “A Simple Task Example” on page 17, the two tasks act like ordinary functions: the first one completes its action before the second one begins execution. This is because information is passed using the `resultis` and `result` functions; the information is not passed until the task has terminated.

A more concurrent way to write these tasks is to give them a different way of passing information and let each routine loop indefinitely. For example, you could write `countZero` as shown in this example:

```
countZero::countZero(qhead *lineQ, qtail *countQ)
{
    char c;
    lineHolder *inmessage;
    while( 1 ) {
        inmessage = (lineHolder*)lineQ->get();
        char *s = inmessage->line;
        int i = 0;
        while( c = *s++ )
            if( c == '0' )
                i++;
        numZero *num = new numZero(i);
        countQ->put(num);
    }
    resultis(1); // never gets here
}
```

Appendix A, "Coroutine Examples", Code Example A-2 gives the full text of a program written this way.

Queues provide such intertask communication. A queue is a data structure made up of a series of linked objects. Queues can hold only descendants of type `object`. You may use a queue as a first-in, first-out (FIFO) queue, or as a first-in, last-out queue (stack), by appropriate selection of access functions.

FIFO Queues

A FIFO queue is made of two objects: a `qhead` and a `qtail`. You create a queue by creating a `qhead` object for it. You then create a tail by calling the member function of `qhead`:

```
qtail* qhead::tail();
```

You can place objects on the queue with the member function of `qtail`. The return value is `1` if the action is successful:

```
int qtail::put(object*)
```

then take objects from the queue with the member function of `qhead`:

```
object* qhead::get()
```

You can also put an object back at the head of the queue with a `qhead` member function. Thus you treat a queue head like a stack:

```
int qhead::putback(object*)
```

A problem with the `putback` function is that if you try to use it on a full queue, you produce a runtime error in queue mode `WMODE` as well as `EMODE`. See “Queue Modes” on page 28 for an explanation of these modes.

To expand the task sample program so it uses queues, you must first create classes for objects that hold the information you want to pass.

Code Example 2-6 Zero-counter Program Using FIFO Queue

FIFO.h

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

class getLine : public task {
public:
    getLine(qhead*, qtail*);
};

class countZero : public task {
public:
    countZero(qhead*, qtail*);
};

class lineHolder : public object {
public:
    char *line;
    lineHolder(char* s) : line(s) { }
};

class numZero : public object {
public:
    int zero;
    numZero(int count) : zero(count) { }
};
```

Now, you can rewrite the main function as shown below:

Code Example 2-7 Zero-counter Main Program

main.cc

```
// Zero-counter program using queues
#include <task.h>
#include "FIFO.h"
#include "countZero.h"
#include "getLine.h"
int main()
{
    qhead *stringQhead = new qhead;
    qtail *stringQtail = stringQhead->tail();
    qhead *countQhead = new qhead;
    qtail *countQtail = countQhead->tail();

    countZero counter(stringQhead, countQtail);
    getLine g(countQhead, stringQtail);
    thistask->resultis(0);
    return 0;
}
```

Code Example 2-8 is the implementation for `countZero`:

Code Example 2-8 `countZero` Constructor

countZero.h

```
countZero::countZero(qhead *lineQ, qtail *countQ)
{
    char c;
    lineHolder *inmessage;
    while( 1 ) {
        inmessage = (lineHolder*)lineQ->get();
        char *s = inmessage->line;
        int i = 0;
        while( c = *s++ )
            if( c == '0' )
                i++;
        numZero *num = new numZero(i);
        countQ->put(num);
    }
    resultis(1); // never gets here
}
```

In this version, `countZero` is created first in the main program, after establishing queues for communication. When `countZero` tries to get a message from the queue there is none. `countZero` suspends, because this is the default waiting-type queue. At that point, the main program creates the line getter. Code Example 2-9 is the implementation of `getline`:

Code Example 2-9 `getLine` Constructor

getLine.h	

```
getLine::getLine(qhead* countQ, qtail* lineQ)
{
    numZero *qdata;
    while( 1 ) {
        cout << "Enter a string, ^C to end session: ";
        char tmpbuf[512];
        cin >> tmpbuf;
        lineQ->put(new lineHolder(tmpbuf));
        qdata = (numZero*) countQ->get();
        cout << "Count of zeroes = " << qdata->zero << "\n";
    };
    resultis(1); // never gets here
}
```

When this routine begins execution, it first gets a line from standard input, places that on the line queue, and then asks the count queue for the count. That action makes it suspend itself until the zero counter places its message on the queue.

As a real program, this example has a number of glaring problems. For one thing, there is no clear way to terminate it; it will loop indefinitely. For another, it continually creates objects without destroying them as it loops. Those details were left out for simplicity.

Queue Modes

Three queue modes govern what happens when a task asks for a message from an empty queue or tries to put a message into a full queue:

1. `WMODE`—The calling task is suspended until condition of queue changes (default).
2. `ZMODE`—The queue returns a null pointer.
3. `EMODE`—A run-time error is produced.

Each `qhead` and `qtail` has its own mode; the head and tail for a queue can have different modes.

You can find out the current mode using the head and tail member function:

```
qmodetype rdmode();
```

and set the mode using the head and tail member function:

```
void setmode(qmodetype m);
```

Queue Size

By default, a queue is limited to 10,000 objects, although space for that number of objects is not actually allocated. Table 2-7 describes queue functions related to queue size.

Table 2-7 Queue Functions

Function	Description
`int rdmax()`	Maximum number of objects allowed in queue.
`void setmax(int)`	Sets new maximum number of objects allowed. You can set the maximum to a number less than the number currently in the queue. In that case, the queue is considered full until the number falls to the new maximum.
`int rdcount()`	Number of objects in queue.
`int rdspace()`	Number of additional objects which can be inserted in queue.

Cutting and Splicing

Since a queue is made up of a separate head and tail, you can cut and splice queues. The main use for this feature is to insert a *filter*, a special task which outputs a transformed version of its input. By cutting an existing queue and splicing in a filter, you can perform transformations without changing or affecting any existing code using the original queue.

Suppose you have a `Generator` task which creates lines of text, perhaps prose, poetry, or computer program source text. You also have a `Printer` task which displays this text on some device. The two tasks communicate by means of a FIFO queue called `Buffer`. `Generator` just writes text into the `Buffer` queue, one line at a time, until it is done. `Printer` just picks up lines from `Buffer` and displays them. You would like to do some formatting on the lines, such as justifying, indenting, splitting and merging lines. By cutting `Buffer` in two and splicing in a filter task called `Format` you can do this without modifying or even recompiling the `Generator` or `Printer` tasks.

First look at Code Example 2-10, where the `Generator` and `Printer` communicate via the buffer:

Code Example 2-10 `Buffer` Class

```
#include <task.h>
class Generator : public task {
public:
    Generator(qtail *target);
    ...
};

class Printer : public task {
public:
    Printer(qhead *source);
    ...
};

int main() {
    ...

    // buffer up to 100 lines, using Wait mode
    qhead *Buffer = new qhead(WMODE, 100);

    // generator writes to the tail of the buffer
    Generator *gen = new Generator(Buffer->qtail());

    // printer reads from the head of the buffer
    Printer *prt = new Printer(Buffer);

    ...
};
```

You can now cut the `Buffer` queue, and insert our filter between the head and the tail. You need a declaration for the filter `Format`, and you splice it into the cut `Buffer` queue:

Code Example 2-11 Cutting and Splicing a Queue

```
#include <task.h>
class Format : public task {
public:
    Format(qhead *source, qtail *target);
    ...
};

Format::Format(qhead *source, qtail *target)
{
    ...
};

int main ()
{
    qhead *Buffer = new qhead(WMODE, 100);
    ...
    // insert formatter into Buffer
    qhead *formhead = Buffer->cut();
    qtail *formtail = Buffer->tail();
    Format form(formhead, formtail);
    ...

    // finished with formatting, restore original Buffer
    formhead->splice(formtail);
    return 1;
}
```

You can do this cutting and splicing anytime, inserting and removing filters as needed. As explained in the manual page `queue`(3C++), `Generator` continues to write to the same `qtail` as before, but there is a new `qhead` associated with it, `formhead`. Similarly, `Printer` continues to extract from the same `qhead` as before, but it is attached to a new `qtail`, `formtail`. The formatter, `form`, reads from the old `qhead`, and writes to the `qtail` of the queue that Printer reads.

When you have finished with this filter, you can use the `splice` function to restore the original queue. `splice` deletes the extra `qhead` and `qtail` which are created by `cut`.

Scheduling

Scheduling is a cooperative effort among all tasks. Although you don't work directly with scheduling, you may need to know what it can and cannot do. Scheduling does the following:

- Maintains the run chain. The run chain is the list of tasks having state RUNNING and therefore ready to run. This is the main activity of scheduling.
- Maintains the simulated time. The time is set to the scheduled time of the next task which is run.
- Executes between tasks. It consists of what must be done after a task has given up execution and before the next task on the run chain continues execution.
- When a task changes its state from IDLE to RUNNING, scheduling adds it to the run chain.
- When a task gives up execution but does not change its state (still has the state RUNNING), scheduling puts it on the run chain according to the next simulated time it is scheduled to run. Tasks run in a round-robin fashion.
- If the run chain is empty but there are active interrupt handlers (see "Real-Time and Interrupts" on page 35), the entire task system becomes dormant until an interrupt occurs.
- If the run chain is empty and there are no interrupt handlers, scheduling exits because no task can become RUNNING. An error is reported if any tasks have not terminated.

Scheduling cannot preempt a task, and vice versa. The currently running task stops execution by explicitly invoking a `wait`, `sleep`, or `resultis` function, or by calling on a pending task.

Random Numbers

Simulations commonly need random numbers for time delays, arrival times or rates, and for other purposes. The coroutine library provides several simple random (actually *pseudo-random*) number generators which are useful for most purposes. They are all based on the C library `rand` function. If you need better quality pseudo-random numbers, you can use these classes as a model for your own versions.

Three classes of random-number generators are provided, as shown in Table 2-8.

Table 2-8 Random-Number Classes

Class	Description
`randint`	Uniformly-distributed random numbers, int or floating-point.
`urand`	Uniformly-distributed random ints in a given range.
`erand`	Exponentially-distributed ints about a given mean.

Class `randint` has the members described in Table 2-9.

Table 2-9 Class `randint`

Member	Description
`randint(long seed=0)`	Constructor, providing initial seed for function `rand`.
`int draw()`	Returns uniformly-distributed `int`s in the range `0` to `INT_MAX`.
`float fdraw()`	Returns uniformly-distributed `float`s in the range `0.0` to `1.0`.
`double ddraw()`	Returns uniformly-distributed `double`s in the range `0.0` to `1.0`.
`void seed(long)`	Sets a new seed and reinitializes the generator.

Class `urand` has the members described in Table 2-10.

Table 2-10 Class `urand`

Member	Description
`urand(int low, int high)`	Constructor, providing lower and upper bounds of the range.
`int draw()`	Returns uniformly-distributed `int`s in the range `low` through `high`.

Class `erand` has the members described in Table 2-11.

Table 2-11 Class `erand`

Member	Description
`erand(int mean);`	Constructor, providing the mean value.
`int draw()`	Returns exponentially-distributed `int`s about the mean.

Histograms

The coroutine library provides class `histogram` for data gathering. A histogram consists of a set of bins, each containing a count of the number of items within the range of the bin. When you construct an object of class `histogram`, you specify the initial range and number of bins. If values outside the current range must be counted, the range is automatically extended by doubling the range of each individual bin. The number of bins cannot be changed. The `add` function increments the count of the bin associated with the given value. The `print` function displays the contents of the histogram in the form of a table. Other data is maintained, as described in Table 2-12.

Table 2-12 Class `histogram`

Member	Description
`histogram(int nbins=16, int l=0, int r=16)`	Constructor; sets the number of bins and initial range
`void add(int value)`	Increment count of the histogram bin for `value`.
`void print()`	Prints the current histogram.
`int l, r`	Denotes the left and right boundaries of the current range.

Table 2-12 Class `histogram` *(Continued)*

Member	Description
`int binsize`	Denotes the current size of each bin.
`int nbin`	Denotes the number of bins (fixed).
`int* h`	Denotes the pointer to storage for bins.
`long sum`	Denotes the sum of all bins.
`long sqsum`	Denotes the sum of squares of all bins.

Real-Time and Interrupts

As noted in “System Time” on page 22, the coroutine library normally runs independently of realtime, and uses only a simulated passage of time in arbitrary units. A class which handles interrupts is available to allow real-time response to external events. You can define an interrupt handler for any UNIX signal using class `Interrupt_handler`:

```
class Interrupt_handler : public object {
public:
    virtual int pending();     // False once after each interrupt
     Interrupt_handler(int sig); // Create handler for signal sig
     ~Interrupt_handler();

private:
     virtual void interrupt(); // the interrupt handler function
     int signo;                // signal number
     int gotint;        // got an interrupt but alert not done
    Interrupt_handler *prev;  // previous handler for this signal
};
```

When the signal occurs, the virtual member function `interrupt` gets control, interrupting whatever task is currently running. When `interrupt` returns, the original task resumes where it left off. This seems to violate the non-preemptive nature of the task system, but function `interrupt` is not a task. For this reason, function `interrupt` should just establish whatever data is necessary for a normal task to process. The base-class version of `interrupt` does nothing but return. You derive your own handler class from `Interrupt_handler` to do whatever you need.

The next time the scheduler is invoked, a special predefined task called the interrupt alerter is run ahead of other waiting tasks. Its job is to alert all handlers whose signals have occurred since the last time it ran. Any tasks that were waiting on an `Interrupt_handler` are thus alerted, and become ready to run. This is how you schedule a task `T` to run when an interrupt occurs:

1. Create an object `IH` derived from `Interrupt_handler` for the signal.
2. Write the `interrupt` member function for it.
3. Have task `T` wait on handler `IH`.

Code Example 2-12 uses the keyboard interrupt, normally the Delete key, as a signal to process data kept in a queue. A discussion follows the sample program.

Code Example 2-12 Handling Interrupts

```
#include <task.h>
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>

static char **gargv;    // next command-line argument
static char **oargv;    // first command-line argument
static int gcount = 0;  // number of args gotten

int get_data() {        // return the next command-line argument
    ++gcount;
    if( *gargv == 0 )   // recycle if not enough
        gargv = oargv;
    return atoi(*(gargv++));
}

// KB interrupt handler
class KBhandler : public Interrupt_handler {
    void interrupt();
    int *simQ, *simQ_end, *simQ_h, *simQ_t; // simulated queue
public:
   int getNext(int&);         // get the next item from the queue
    KBhandler(int size = 5);
    ~KBhandler() { delete [] simQ; }
};
```

Code Example 2-12 Handling Interrupts *(Continued)*

```
KBhandler::KBhandler(int size) : Interrupt_handler(SIGINT) {
    // set up simulated queue
    simQ_t = simQ_h = simQ = new int[size];
    simQ_end = &simQ[size];
}

void KBhandler::interrupt() {
    // put the next command-line arg into the simulated queue
    int *p = simQ_t;
    *p = get_data();
    if( ++p == simQ_end) p = simQ;
    if( p != simQ_h)
        simQ_t = p;
    else {
        puts("interrupt queue overflow");
        task_error(0, 0);
    }
}

int KBhandler::getNext(int& val) {
    int *p = simQ_h;
    if( p == simQ_t )
        return 0;    // queue empty
    val = *p;
    if( ++p == simQ_end ) p = simQ;
    simQ_h = p;
    return 1;        // data available
}

// our user task which will wait for interrupts
class KBprinter : public task {
    KBhandler *handler;
public:
    KBprinter();
};
```

Code Example 2-12 Handling Interrupts *(Continued)*

```
KBprinter::KBprinter() :
        task("KBprinter"),
        handler(new KBhandler) {
    while( gcount < 5 ) {    // first 5 values only
        wait(handler);       // wait for KB interrupt
        int i;
        while( handler->getNext(i) ) // get any available values
            printf("found %d\n", i);
    }
    resultis(0);             // task is finished
}

int main(int argc, char **argv) {
    if( argc < 2 ) {
        printf("Usage: %s <list of integers>\n", argv[0]);
        return 1;             // error exit
    }
    oargv = gargv = &argv[1]; // make command-line data available
    KBprinter theTask;        // print data on each KB interrupt
    theTask.result();         // wait for theTask to finish
    thistask->resultis(0);    // terminate main task
    return 0;
}
```

The previous sample program is a simulation of a simulation. Imagine that you have a queue of data to be processed, and that an external interrupt (UNIX signal) should trigger a round of processing. In the example, you expect a list of integer values on the program command line, and you use these to simulate a source of integer data. Function `get_data` returns the next command-line integer, cycling back to the beginning if there are not enough of them.

Class `KBhandler`, derived from `Interrupt_handler`, provides the handling of the keyboard interrupt `SIGINT` (usually the Delete key). Rather than work with an actual `queue` for this example, the constructor sets up a simulated queue as an array of integers. Whenever a keyboard interrupt occurs, member function `interrupt` gets the next piece of input data and puts it in the queue. Member function `getNext` retrieves the value at the front of the queue, if any, and returns a status value indicating whether the data is available.

Class `KBprinter` is the task which waits for a keyboard interrupt and prints all available data. Its constructor sets up a `KBhandler` and waits on it. When a keyboard interrupt occurs, any tasks waiting on the handler are alerted

automatically. In this case, `KBprinter` is the waiting task. It resumes execution, prints anything in the queue, then returns to waiting. As a simple way to stop this example, we terminate after getting five integers.

The main program creates a `KBprinter` task, then waits for it to finish by calling `result` on it.

Note – The main program is an anonymous task, and should terminate by calling `resultis` on itself.

Coroutine Library Limitations

The coroutine library is *flat* because a class derived from `task` may not have derived classes. Only *one level* of derivation is allowed. This is the way the library was designed and reflects the way the tasks are manipulated on the stack. The enhancement of allowing multiple levels would require a rewrite of the design. For example, the following is *not* allowed:

```
class base : public task { ... };
class task1 : public base { ... }; // compiles, but will not work
class task2 : public base { ... }; // compiles, but will not work
```

If you must have certain sets of tasks share a hierarchy, you may adopt a multiple inheritance scheme. For example, you could define a class with shared information. Each task would have class `task` as its first immediate base class and the shared-data class as another immediate base class:

```
class base { ... }; // shared portion
class task1 : public task, public base { ... }; // OK
class task2 : public task, public base { ... }; // OK
```

A pointer to `task` does not allow access to anything in the `base` portion of `task1` or `task2` with the multiple inheritance approach.

2

The Complex Arithmetic Library 3

Introduction

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.

The complex arithmetic library implements a complex number data type as a new data type. It provides all operators and many mathematical functions defined for the built-in numerical types, and also provides extensions for iostreams that allow input and output of complex numbers. The complex arithmetic library provides error handling.

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.

3

The Complex Library

To use the complex library, include the header file `complex.h` in your program, and link with the `-lcomplex` or `-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 and the imaginary part. The numerical values of each part are held in fields of type `double`. Here is the relevant part of the definition of `complex`:

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

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 44). 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. Table 3-1 and Table 3-2 describe the remainder of the complex arithmetic library functions.

Table 3-1 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.

Table 3-1 Complex Arithmetic Library Functions *(Continued)*

Complex Arithmetic Library Function	**Description**
`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.
`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 3-2 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.

`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.

3

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, set `errno` to zero before the operation, then test it afterward.

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` which 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 Table 3-3

Table 3-3 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 51 and “Output Using Iostreams” on page 53 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 coverted 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 which 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.

3

Complex Man Pages

The man pages listed in Table 3-4 comprise the remainder of the documentation of the complex arithmetic library.

Table 3-4 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

The `Iostream` *Library* 4

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

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 Iostreams” on page 53 and “Input Using Iostreams” on page 57.

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 Table 4-1.

Table 4-1 `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 these header files in your program. Include only the ones that contain the declarations you need. The `iostreams` library is part of `libC`, and is linked automatically by the `CC` driver.

Output Using `Iostreams`

Output using `iostreams` 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

Code Example 4-1 defines a `string` class:

Code Example 4-1 `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`.

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

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

`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.

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

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 61 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 67).

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 `Iostreams`

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 Code Example 4-2:

Code Example 4-2 `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 1n `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, Code Example 4-3 produces an input error if you type alphabetic characters for input:

Code Example 4-3 Handling Extraction Errors

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

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

4

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 Code Example 4-4:

Code Example 4-4 Copying Files with Streams

```
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 "Streambufs" on page 73, 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 which 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 seeking 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 manual 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 4-2. 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 4-2 `Iostream` Predefined Manipulators

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

Table 4-2 `Iostream` Predefined Manipulators *(Continued)*

	Predefined Manipulator	Description
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.

The following are some examples.

Using Plain Manipulators

A plain manipulator is a function that:

1. Takes a reference to a stream
2. Operates on it in some way
3. 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 we want to turn whitespace skipping on and off for an input stream. We can use separate calls to `ios::setf` and `ios::unsetf` to turn the `skipws` flag on and off, or we could define two manipulators, as shown in Code Example 4-5:

Code Example 4-5 Toggle Whitespace Skipping

```
#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 Code Example 4-6:

Code Example 4-6 Parameterized Manipulators

```
//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 Code Example 4-7, the manipulator `print_hex`:

1. Puts the output stream into the hex mode.
2. Inserts a `long` value into the stream.
3. 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`:

Code Example 4-7 Manipulator `print_hex`

```
#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.

Streambuf*s*

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

You usually use `streambuf`s through `iostreams`, so you don't have to worry about the details of `streambuf`s. You can use `streambuf`s 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 Streambuf*s*

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`(3C++), 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` which 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` *ManPages*

A number of C++ man pages give details of the `iostream` library. Table 4-3 gives an overview of what is in each man page.

Table 4-3 `Iostream` Man Pages Overview

Man Page	Overview
`ios.intro`	Gives an introduction to and overview of `iostreams`.
`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.
`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
`sbufprot`	Describes the interface needed by programmers who are coding a class derived from class `streambuf`. Also refer to the `sbufpub` man page because some public functions are not discussed in the `sbufprot` 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` man page.

Table 4-3 `Iostream` Man Pages Overview *(Continued)*

Man Page	Overview
`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.
`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`.

`Iostream` *Terminology*

The `iostream` library descriptions often use terms similar to terms from general programming, but with specialized meanings. Table 4-4 defines these terms as they are used in discussing the `iostream` library.

Table 4-4 `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 forceably 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.

Table 4-4 `Iostream` Terminology *(Continued)*

`Iostream` **Term**	**Definition**
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.

4

Using `iostreams` *in a Multithreaded Environment* 5

This chapter describes how to use the `iostream` classes of the `libC` library 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.

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 and higher. Applications that utilize the single-threaded capabilities of previous versions of the library are not affected by the behavior of the modified `iostream` interfaces.

Note – The MT-safe version of the iostream library is *not* backwards compatible. You must be running Solaris version 2.2 or higher; it does *not* work on previous versions of Solaris (that is, 2.1 and prior).

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" for details.

The original core classes have been renamed with the prefix `unsafe_`. Table 5-1 lists the classes that now comprise the core of the `iostream` package.

Table 5-1 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 in Table 5-2.

Table 5-2 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.
`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. Code Example 5-1illustrates how to check for errors:

Code Example 5-1 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. Code Example 5-2 shows a call to `gcount`:

Code Example 5-2 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. Code Example 5-3 shows a user-defined I/O operation:

Code Example 5-3 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. Code Example 5-4 shows how to use `unsafe cout` and `cin`:

Code Example 5-4 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. Code Example 5-5 makes an `iostream` object MT-unsafe:

Code Example 5-5 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. Code Example 5-6 illustrates the technique:

Code Example 5-6 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 91 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

Table 5-3 lists the new classes added to the `libC` interfaces.

Table 5-3 New Classes

```
stream_MT
stream_locker
unsafe_ios
unsafe_istream
unsafe_ostream
```

Table 5-3 New Classes *(Continued)*

```
unsafe_iostream
unsafe_fstreambase
unsafe_strstreambase
```

New Class Hierarchy

Table 5-4 lists the new class hierarchy added to the `iostream` interfaces.

Table 5-4 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

Table 5-5 lists the new functions added to the `iostream` interfaces.

Table 5-5 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();
```

Table 5-5 New Functions *(Continued)*

```
        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&);
```

Table 5-5 New Functions *(Continued)*

```
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 `iostream` 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” 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 91” 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 Code Example 5-7 seeks to find a position in a file and reads the next block of data.

Code Example 5-7 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
```

Code Example 5-7 Example of Using Locking Operations *(Continued)*

```
#include <fstream.h>
#include <rlocks.h>

void lock_example (fstream& fs)
    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 Code Example 5-8, 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 5-8 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`(3) man page.

5

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. Code Example 5-9 shows how to safely destroy a shared object.

Code Example 5-9 Destroying a Shared Object

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

void *process_rtn(void*)
```

*Code Example 5-9 (Continued)*Destroying a Shared Object *(Continued)*

```
{
    // 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

Code Example 5-10 is 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 5-10 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
```

Code Example 5-10 Using `Iostream` Objects in a MT-safe Way *(Continued)*

```
// <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;
}

main(int argc, char** argv) {
  // argv: 1+ list of filenames per thread
```

Code Example 5-10 Using `Iostream` Objects in a MT-safe Way *(Continued)*

```
   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;
}
```

5

Coroutine Examples A

This appendix shows the full text of the two sample programs discussed in Chapter 2, “The Coroutine Library”.

Code Example A-1 counts the number of `'0'` characters in a string using cooperating tasks.

Code Example A-1 Zero-Counter Program

```
// Simple zero-char counter program
#include <task.h>
#include <iostream.h>

class getLine : public task {
public:
    getLine();
};

getLine::getLine()
{
    char* tmpbuf = new char[512];
    cout << “Enter string: “;
    cin >> tmpbuf;
    resultis((int)tmpbuf);
}


class countZero : public task {
public:
    countZero(getLine*);
```

Code Example A-1 Zero-Counter Program *(Continued)*

```
// Simple zero-char counter program
};

countZero::countZero(getLine *g)
{
    char *s, c;
    int i = 0;
    s = (char*)g->result();
    while( c = *s++)
        if( c == '0' )
            i++;
    resultis(i);
}

int main()
{
    getLine g;
    countZero c(&g);
    cout << "Count result = "
         << c.result() << "\n";
    thistask->resultis(0);
    return 0;
}
```

Code Example A-2 does the same, but illustrates the use of queues. The program loops forever, requiring a keyboard interrupt to terminate. It also never deletes the objects it allocates with new, and is thus not realistic.

Code Example A-2 Zero-Counter Using Queues

```
// Zero-counter program using queues
#include <task.h>
#include <iostream.h>

class getLine : public task {
public:
    getLine(qhead*, qtail*);
};

class countZero : public task {
public:
    countZero(qhead*, qtail*);
};
```

Code Example A-2 Zero-Counter Using Queues *(Continued)*

```
// Zero-counter program using queues
class lineHolder : public object {
public:
    char *line;
    lineHolder(char* s) : line(s) { }
};

class numZero : public object {
public:
    int zero;
    numZero(int count) : zero(count) { }
};

getLine::getLine(qhead* countQ, qtail* lineQ)
{
    numZero *qdata;
    while( 1 ) {
        cout << “Enter a string, ^C to end session: “;
        char tmpbuf[512];
        cin >> tmpbuf;
        lineQ->put(new lineHolder(tmpbuf));
        qdata = (numZero*) countQ->get();
        cout << “Count of zeroes = “ << qdata->zero << “\n”;
    };
    resultis(1); // never gets here
}

countZero::countZero(qhead *lineQ, qtail *countQ)
{
    char c;
    lineHolder *inmessage;
    while( 1 ) {
        inmessage = (lineHolder*)lineQ->get();
        char *s = inmessage->line;
        int i = 0;
        while( c = *s++ )
            if( c == ‘0’ )
                i++;
        numZero *num = new numZero(i);
        countQ->put(num);
    }
    resultis(1); // never gets here
}
```

Code Example A-2 Zero-Counter Using Queues *(Continued)*

```
// Zero-counter program using queues
int main()
{
    qhead *stringQhead = new qhead;
    qtail *stringQtail = stringQhead->tail();
    qhead *countQhead = new qhead;
    qtail *countQtail = countQhead->tail();

    countZero counter(stringQhead, countQtail);
    getLine g(countQhead, stringQtail);
    thistask->resultis(0);
    return 0;
}
```

Associated Man Pages B

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

Table B-1 Man Pages for Complex Library

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.

Table B-2 Man Pages for Iostream Library

Man Page	**Overview**
`ios.intro(3C++)`	Gives an introduction to and overview of iostreams.
`filebuf(3C++)`	Details the public interface for the class `filebuf`, which is specialized for use with files.
`fstream(3C++)`	Details specialized member functions of classes `ifstream`, `ofstream`, and `fstream`, which are specialized for use with files.
`ios(3C++)`	Details parts of class `ios`, which functions as a base class for iostreams.
`istream(3C++)`	Describes class `istream`, which supports interpretation of characters fetched from a streambuf.
`manip(3C++)`	Describes the input and output manipulators defined in the iostream library.

Table B-2 Man Pages for Iostream Library *(Continued)*

Man Page	**Overview**
`ostream(3C++)`	Describes class `ostream`, which supports interpretation of characters written to a streambuf.
`sbufprot(3C++)`	Describes the protected interface of class `streambuf`.
`sbufpub(3C++)`	Details the public interface of class `streambuf`.
`stdiobuf(3C++)`	Describes class `stdiobuf`, which is specialized for dealing with `stdio FILE`s.
`strstream(3C++)`	Details the specialized member functions of strstreams, which are specialized for dealing with arrays of characters.
`ssbuf(3C++)`	Details the specialized public interface of class `strstreambuf`, which is specialized for dealing with arrays of characters.
stream_MT`(3C++)`	Describes the base class that provides dynamic changing of iostream class objects to and from MT safety.
stream_locker`(3C++)`	Describes the class used for application-level locking of iostream class objects.

Table B-3 Man Pages for Coroutine Library

Man Page	Overview
`task.intro(3C++)`	General introduction to the coroutine library.
`task(3C++)`	Description of all the coroutine classes.
`tasksim(3C++)`	Description of the histogram and random number classes.
`queue(3C++)`	Description of the queue classes.
`interrupt(3C++)`	Description of the extensions for real-time programming.

Index

A

absolute value, complex number, 41
angle, complex number, 41
applications, MT-safe, 80
archives, static, 8
argument, complex number, 41

B

binary input, 60
binary output, 57
buffer, 76
buffer, flushing output, 57

C

C++ documentation, xii
`c_exception`, 46
`cerr`, 51, 90
`char` extractor, 59
`char* extractor`, 59
characters, reading, 59
`chr`, 82
`chr_r`, 82
`cin`, 51, 90
class `c_exception`, 46
class `complex`, 42
  arithmetic, 43
  constructors, 42
  efficiency, 49
  error handling, 46
  input/output, 47
  mathematical functions, 44
  mixed-mode arithmetic, 48
class `fstream`, 52, 62
class `histogram`, 34
class `ifstream`, 52, 62
class `Interrupt_handler`, 35
class `iostream`, 52
class `istream`, 52
class `istrstream`, 52
class `object`, 12
class `ofstream`, 52, 62
class `omanip_long`, 72
class `ostream`, 52
class `ostrstream`, 52
class `qtail`, 25
class `sched`, 12
class `smanip_int`, 71
class `stdiobuf`, 72
class `streambuf`, 73
class `strstream`, 52
class `task`, 13, 15

class timer, 23
class urand, 34
clog, 51, 90
compiler options
  lcomplex, 3
  lrwtool, 4
  ltask, 4
  xar, 9
complex conjugate, 41
complex numbers, 41
complex, *See* class complex
complex::
  abs, 44
  acos, 45
  arg, 44
  asin, 45
  atan, 45
  complex, 42
  conj, 44
  cos, 45
  cosh, 45, 47
  exp, 45, 47
  imag, 44
  log, 45, 47
  log10, 45, 47
  norm, 45
  polar, 45
  pow, 45
  real, 45
  sin, 45
  sinh, 45, 47
  sqrt, 45
  tan, 45
  tanh, 45
complex_error, 46
copying files, 74
core classes, LibC, 80
coroutine
  See also task
coroutine class structure, 12
coroutine library, 12
cout, 51, 90

D

dec, 82
dec (manipulator), 67
dec_r, 82
defining a mutual exclusion region, 93
deriving MT-safe classes, considerations, 94

E

EDOM, 47
EMODE, 28
endl (manipulator), 67
ends (manipulator), 67
erand::
  constructor, 34
  draw, 34
ERANGE, 47
errno, 46
Error, xiii
error checking, 83
error state, iostreams, 56
extend functionality of iostream classes, 94
extraction, 76
extraction operators, 57
extractor
  char, 59
  char*, 59
  whitespace, 60

F

FIFO queues, 25
file descriptors, using, 65
files
  copying, 74
  opening and closing, 65
  repositioning, 65
  using fstreams with, 62
filter, 29
flush (manipulator), 57, 67

form, 82
form_r, 82
format control with iostreams, 67
fstream, 76
fstream.h, 53, 63
fstream::
    attach, 65
    close, 65
    open, 65

G

gcount function, 84
get pointer, 73
global data, in a multithreaded application, 90
global shared objects, default, 90

H

hex, 82
hex (manipulator), 67
hex_r, 82
histogram::
    add, 34
    binsize, 35
    constructor, 34
    h, 35
    l, 34
    nbin, 35
    print, 34
    r, 34
    sqsum, 35
    sum, 35
histograms, 34

I

I/O library, 51
input, 51, 57
    binary, 60
    peeking, 60
input errors, handling, 61
insertion, 76
insertion operator, 53
    defining, 55
interrupt alerter, 36
Interrupt_handler::
    ~Interrupt_handler, 35
    Interrupt_handler, 35
    pending, 35
interrupts, 35
iomanip classes, 71
iomanip.h, 53, 68
ios::
    app, 64
    ate, 64
    badbit, 56
    beg, 66
    cur, 66
    end, 66
    eofbit, 56
    failbit, 56
    good, 56
    goodbit, 56
    hardfail, 56
    in, 64
    io_state, 56
    nocreate, 64
    noreplace, 64
    open_mode, 64
    operator !, 56, 61
    operator void *, 56
    out, 64
    seek_dir, 66
    sync_with_stdio, 62
    trunc, 64
iostream, 76
    library, 51
    manual pages, 75
iostream.h, 53
iostreams
    copying, 66
    creating, 62
    error bits, 56
    errors, 55, 61
    flushing, 57
    formats, 67

header files, 53
input, 57
manipulators, 67
output to, 53
predefined, 51
stdio, 62, 72
structure, 51
terminology, 76
using, 53
istream::
operator >>, 57
peek, 60
read, 60
seekg, 65
tellg, 65

L

left-shift operator, 53
libC core classes, 80
libC library, 79, 82
library, 1
class, 1
object, 1
shared, 6
standard, 4
looking ahead at input, 60

M

magnitude, complex number, 41
man, xiii
manipulators, 67
plain, 68
predefined, 67
Manual, xiii
MT-safe, 79
MT-safe applications, 80
MT-safe classes
considerations for deriving, 94
MT-safe objects, 80
multithreaded environment, 79
mutex locks, 94

O

object, MT-safe, 80
object::
alert, 13
forget, 13
INTHANDLER, 13
o_type, 13
OBJECT, 13
objtype, 13
pending, 13
print, 13
QHEAD, 13
QTAIL, 13
remember, 13
TASK, 13
this_task, 13
TIMER, 13
oct, 82
oct (manipulator), 67
oct_r, 82
Online, xiii
operations, performing a sequence of, 91
ostream::
operator <<, 53, 55
seekp, 65
tellp, 65
write, 57
output, 51
output buffer flushing, 57
output, binary, 57
output, handling errors, 55
overflow function, 94

P

parameterized manipulators, 67, 71
pbackfail function, 94
peeking at input, 60
pending objects, 19
pending tasks, 19
plain manipulators, 68
polar, complex number, 41
prerequsite reading, xii

public functions, MT-safe, 82
public virtual functions, of
streambuf, 94
Purpose, xi
purpose of guide, xi
put pointer, 73

Q

qhead::
cut, 32
get, 25
putback, 25
qtail, 25
rdcount, 29
rdmax, 29
rdmode, 29
rdspace, 29
setmax, 29
setmode, 29
splice, 32
qmodetype, 29
qtail::
cut, 32
put, 25
rdcount, 29
rdmax, 29
rdmode, 29
rdspace, 29
setmax, 29
setmode, 29
splice, 32
queue
cutting and splicing, 29
EMODE, 28
size, 29
WMODE, 28
ZMODE, 28
queues, 24
FIFO, 25
modes, 28
queues, and suspension, 20

R

randint::
ddraw, 33
draw, 33
fdraw, 33
randint, 33
seed, 33
random numbers, 33
README, xiii
real time, 35
repositioning within a file, 65
resetiosflags (manipulator), 68
right-shift operator, 57
run chain, 32

S

sched::
get_clock, 22
IDLE, 15
rdstate, 16
rdtime, 16
result, 16
RUNNING, 15
setclock, 22
statetype, 16
TERMINATED, 15
scheduling, 32
scope resolution operator, 85
seekoff function, 94
seekpos, 94
setbase (manipulator), 67
setfill (manipulator), 68
setioflags (manipulator), 68
setprecision (manipulator), 68
setw (manipulator), 68
shared objects, strategies for dealing
with, 91
single characters, reading, 59
skip flag, 60
skipping whitespace, 60
sleeping tasks, 20

standard error, 51
standard input, 51
standard output, 51
standard streams, 90
static data, in a multithreaded application, 90
stdio and iostreams, 72
stdio, with iostreams, 62
stdiostream.h, 53
stream, 77
stream.h, 53, 90
stream_locker, 92
stream_locker class, 91
stream_locker object, 94
stream_MT, 81
streambuf, 77, 81
  file-like, 73
  get pointer, 73
  put pointer, 73
  queue-like, 73
streambufs, using, 73
streampos, 65
strstream, 77
strstream.h, 53

T

task
  example, 17
  interrupts, 35
  parts of, 17
  preemption, 32
  real time, 35
  run chain, 32
  states, 15, 19
  time, 22
  wait states, 19
  waiting, 21, 22
task::
  ~task, 17
  cancel, 17
  constructor, 17
  DEDICATED, 17
  delay, 17, 22
  IDLE, 17
  modetype, 17
  preempt, 17
  rdstate, 17
  rdtime, 17
  result, 17, 20
  resultis, 17
  RUNNING, 17
  SHARED, 17
  sleep, 17, 20
  statetype, 17
  t_name, 17
  t_next, 17
  TERMINATED, 17
  wait, 17, 21
  waitlist, 17, 21
  waitvec, 17, 21
tasks
  pending, 19
  scheduling, 32
  sleeping, 20
  waiting, 21
timer::
  constructor, 23
  destructor, 23
  reset, 23
timers, 23

U

underflow function, 94
unsafe_ios, 80
unsafe_iostream, 80
unsafe_istream, 80
unsafe_ostream, 80
urand::
  constructor, 34
  draw, 34
user-defined types, 85

W

waiting for an object, 21
whitespace, 60

whitespace, skipping, 60
WMODE, 28
ws (manipulator), 61, 67

X

xsgetn, 94
xsputn function, 94

Z

ZMODE, 28