C++ Language System Library

Introduction

About This Guide

This manual, the *C++ Language System Library*, introduces the iostream support in the C++ library and describes a data type *complex* that provides the basic facilities for using complex arithmetic in C++.

Audience

This manual assumes that you are familiar with the language as described in *The C++ Programming Language* (see **"Recommended Reading on C++"**) and to have sufficient experience with C++ to recognize many of the problems that the features described in this manual are designed to solve or alleviate.

Document Overview

This guide contains the following chapters:

Chapter 1, "Complex Arithmetic in C++," describes a data type *complex* providing the basic facilities for using complex arithmetic in C++.

Chapter 2, "Iostream Examples," is an introduction to the library and contains many examples.

Index, is a detail index of this manual.

Related Documentation

Refer to the *C++ Translator Release Notes*, that accompanied your C++ installation package, for information specific to your release of the C++ translator. In addition, you should have copies of the following:

- *C++ Programming Guide*, Silicon Graphics, Inc., document number 007−0704−050
- *C++ Language System Product Reference Manual*, Silicon Graphics, Inc., document number 007−1618−020
- *C++ Language System Overview*, Silicon Graphics, Inc., document number 007−1621−020

Recommended Reading on C++

For a precise description of the C++ language, the following is indispensable:

- Ellis, Margaret A., and Stroustrup, Bjarne. *The Annotated C++ Reference Manual* (Reading, Mass.: Addison−Wesley, 1990)

The following books are recommended reading on programming in C++:

- Lippman, Stanley B. *C++ Primer* Second Edition (Reading, Mass.: Addison−Wesley, 1991)
- Stroustrup, Bjarne. *The C++ Programming Language* Second Edition (Addison−Wesley, 1991))

Notation and Syntax Conventions

This guide uses the following notation and syntax conventions:

italic	Indicates arguments in a command line that you must replace with a valid value. In text, it is used to indicate commands, document titles, file names, glossary items, new terms, and variables.

`courier`	Indicates computer output and program listings.
`courier bold`	Indicates user input to the computer and nonprinting keys.
[ ]	Brackets enclose optional command arguments. (Do not enter the brackets.)
. . .	An ellipsis indicates that the preceding optional items can appear more than once in succession.
|	The vertical bar separates items from which you can choose one.

Chapter 1

Complex Arithmetic in C++

This chapter describes a data type *complex* providing the basic facilities for using complex arithmetic in C++.

This chapter contains the following sections:

- **"Introduction"**
- **"Complex Variables and Data Initialization"**
- **"Input and Output"**
- **"Cartesian and Polar Coordinates"**
- **"Arithmetic Operators"**
- **"Mixed Mode Arithmetic"**
- **"Mathematical Functions"**
- **"Efficiency"**
- **"Type complex"**
- **"An FFT Function"**
- **"Errors and Error Handling"**

Note: This chapter is from a paper by Leonie V. Rose and Bjarne Stroustrup.

Introduction

The C++ language does not have a built−in data type for complex numbers, but it does provide language facilities for defining new data types. The type *complex* was designed as a useful demonstration of the power of these facilities.

There are three plausible ways to support complex numbers in a language. First, the type *complex* could be directly supported by the compiler in the same way as the types *int* and *float* are. Alternatively, a preprocessor could be written to translate all use of complex numbers into expressions involving only built−in data types.

A third approach was used to implement type *complex*; it was specified as a user−defined type. This demonstrates that one can achieve the elegance and most of the efficiency of a built in data type without modifying the compiler. It is even much easier to implement than the preprocessor approach, which is likely to provide an inferior user interface.

This facility for complex arithmetic provides the arithmetic operators + , / , * , and − , the assignment operators = , += , −= , *= , and /= , and the comparison operators == and != for complex numbers. Input and output can be performed using the operators >> (get from) and << (put to). The initialization functions and >> accept a Cartesian representation of a *complex*.

The functions *real*() and *imag*() return the real and imaginary part of a *complex*, respectively, and << prints a *complex* as (*real, imaginary*). The internal representation of a *complex*, is, however, inaccessible and in principle unknown to a user.

Polar coordinates can also be used. The function *polar*() creates a *complex* given its polar representation, and *abs*() and *arg*() return the polar magnitude and angle, respectively, of a *complex*. The function *norm*() returns the square of the magnitude of a *complex*. The following complex functions are also provided: *sqrt*(), *exp*(), *log*(), *sin*(), *cos*(), *sinh*(), *cosh*(), *pow*(), and *conj*(). The declaration of *complex* and the declarations of the complex functions can be found under "Type *complex."* A complete program using complex numbers can be found under "**"An FFT Function"** "

The usual arithmetic operators can be used on complex numbers and a library of standard complex mathematical functions is provided. For example:

```
#include <complex.h>
main(){
       complex xx;
       complex yy = complex(1,2.718);
       xx = log(yy/3);
       cout << 1+xx;
}
```

initializes *yy* as a complex number of the form (*real+imag*i*), evaluates the expressions and prints the result: (*0.96476,1.21825*).

The data type *complex* is implemented as a class using the data abstraction facilities in C++. The arithmetic operators +, −, * , and / , the assignment operators = , += , −= , *= , and /= , and the comparison operators == and != are provided for complex numbers. So are the trigonometric and mathematical functions: *sin*(), *cos*(), *cosh*(), *sinh*(), *sqrt*(), *log*(), *exp*(), *conj*(), *arg*(), *abs*(), *norm*(), and *pow* ().

Expressions such as (*xx+1*)**log*(*yy*log*(*3.2*)) that involve a mixture of real and complex numbers are handled correctly. The simplest complex operations, for example, + and += , are implemented without function call overhead.

Complex Variables and Data Initialization

A program using complex arithmetic will contain declarations of *complex* variables. For example:

```
complex zz = complex(3,−5);
```

will declare *zz* to be complex and initialize it with a pair of values. The first value of the pair is taken as the real part of the Cartesian representation of a complex number and the second as the imaginary part. The function *complex*() constructs a complex value given suitable arguments

Such a function is called a constructor. A constructor for a type always has the same name as the type itself.

. It is responsible for initializing *complex* variables, and will convert the arguments to the proper type (*double*). Such initializations may be written more compactly. For example:

```
complex zz(3,−5);
complex c_name(−3.9,7);
complex rpr(SQRT_2, root3);
```

A complex variable can be initialized to a real value by using the constructor with only one argument. For example:

```
complex ra = complex(1);
```

will set up *ra* as a complex variable initialized to (1,0). Alternatively the initialization to a real value can also be written without explicit use of the constructor:

```
complex rb = 123;
```

The integer value will be converted to the equivalent complex value exactly as if the constructor *complex* (123) had been used explicitly. However, no conversion of a *complex* into a *double* is defined, so

```
double dd = complex(1,0);
```

is illegal and will cause a compile−time error.

If there is no initialization in the declaration of a complex variable, then the variable is initialized to (0,0).

For example:

```
complex orig;
```

is equivalent to the declaration:

```
complex orig = complex(0,0);
```

Naturally a complex variable can also be initialized by a complex expression.

For example:

```
complex cx(-0.5000000e+02,0.8660254e+02);
complex cy = cx+log(cx);
```

It is also possible to declare arrays of complex numbers. For example:

```
complex carray[30];
```

sets up an array of 30 complex numbers, all initialized to (0,0). Using the above declarations:

```
complex carr[] = { cx, cy, carray[2], complex(1.1,2.2) };
```

sets up a complex array "*carr*[ ]" of four *complex* elements and initializes it with the members of the list. However, a struct style initialization cannot be used. For example:

```
complex cwrong[] = {1.5, 3.3, 4.2, 4};
```

is illegal, because it makes unwarranted assumptions about the representation of complex numbers.

Input and Output

Simple input and output can be done using the operators >> (get from) and << (put to). They are declared like this using the facility for overloading function operators:

```
ostream& operator<<(ostream&, complex);
istream& operator>>(istream&, complex&);
```

When *zz* is a complex variable *cin>>zz* reads a pair of numbers from the standard input stream *cin* into *zz* . The first number of the pair is interpreted as the real part of the Cartesian representation of a complex number and the second as the imaginary part. The expression *cout<<zz* writes *zz* to the standard output stream *cout*. For example:

```
void copy(istream& from, ostream& to)
{
      complex zz;
      while (from>>zz) to<<zz;
}
```

reads a stream of complex numbers like (3.400000, 5.000000) and writes them like (3.4, 5). The parentheses and comma are mandatory delimiters for input, while white space is optional. A single real number, for example 10*e*–7 or (123), will be interpreted as a *complex* with 0 as the imaginary part by operator >>.

A user who does not like the standard implementation of << and >> can provide alternate versions.

Cartesian and Polar Coordinates

The functions *real*() and *imag*() return the real and imaginary parts of a complex number, respectively. This can, for example, be used to create differently formatted output of a *complex*:

```
complex cc = complex(3.4,5);
cout << real(cc) << "+" << imag(cc) << "*i";
```

will print *3.4+5*i*.

The function *polar*() creates a *complex* given a pair of polar coordinates (magnitude, angle). The functions *arg*() and *abs*() both take a *complex* argument and return the angle and magnitude (modulus), respectively. For example:

```
complex cc = polar(SQRT_2,PI/4);  // also known as complex(1,1)
double magn = abs(cc);            // magn = sqrt(2)
double angl = arg(cc);            // angl = PI/4
cout << "(m=" << magn << ", a=" << angl << ")";
```

If input and output functions for the polar representation of complex numbers are needed they can easily be written by the user.

Arithmetic Operators

The basic arithmetic operators +, − (unary and binary), /, and *, the assignment operators =, +=, −=, *=, and /= , as well as the equality operators == and !=, can be used for complex numbers. The operators have their conventional precedences. For example: *a=b*c+d* for complex variables *a*, *b*, *c*, and *d* is equivalent to *a*=(*b*c*)+*d*. There are no operators for exponentiation and conjugation; instead the functions *pow*() and *conj*() are provided. The operators +=, −=, *=, and /= do not produce a value that can be used in an expression; thus the following examples will cause compile−time errors:

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

Mixed Mode Arithmetic

Mixed mode expressions are handled correctly. Real values will be converted to complex where necessary. For example:

```
complex xx(3.5,4.0);
complex yy = log(yy) + log(3.2);
```

This expression involves a mixture of real values: *log*(3.2), and complex values: *log*(*yy*) and the sum. Another example of mixing real and complex, *xx=1*, is equivalent to *xx=complex*(1) which in turn is equivalent to *xx=complex*(1,0). The interpretation of the expression (*xx+1*)**yy**3.2 is (((*xx+complex*(1))**yy*)**complex*(3.2)).

Mathematical Functions

A library of complex mathematical functions is provided. A complex function typically has a counterpart of the same name in the standard mathematical library. In this case the function name will be overloaded. That is, when called, the function to be invoked will be chosen based on the argument type. For example, *log*(1) will invoke the real *log*(), and *log*(*complex*(1)) will invoke the complex *log*(). In each case the integer is converted to the real value 1.0.

These functions will produce a result for every possible argument. If it is not possible to produce a mathematically acceptable result, the function *complex_error*() will be called and some suitable value returned. In particular, the functions try to avoid actual overflow, calling *complex_error*() with an overflow message instead. The user can supply *complex_error*(). Otherwise a function that simply sets the integer *errno* is used. See "Errors and Error Handling" for details.

```
complex conj(complex);
```

Conj(*zz*) returns the complex conjugate of *zz*.

```
double norm(complex);
```

Norm(*zz*) returns the square of the magnitude of *zz*. It is faster than *abs*(*zz*), but more likely to cause an overflow error. It is intended for comparisons of magnitudes.

```
double   pow(double, double);
complex  pow(double, complex);
complex  pow(complex, int);
complex  pow(complex, double);
complex  pow(complex, complex);
```

Pow(*aa*,*bb*) raises *aa* to the power of *bb*. For example, to calculate (1−*i*)**4:

```
cout << pow( complex(1,-1), 4);
```

The output is (−4,0).

```
double   log(double);
complex  log(complex);
```

Log(*zz*) computes the natural logarithm of *zz*. *log*(0) causes an error, and a huge value is returned.

```
double   exp(double);
complex  exp(complex);
```

Exp(*zz*) computes *e**zz*, *e* being 2.718281828...

```
double   sqrt(double);
complex  sqrt(complex);
```

Sqrt(*zz*) calculates the square root of *zz*.

The trigonometric functions available are:

```
double   sin(double);
complex  sin(complex);
double   cos(double);
complex  cos(complex);
```

Hyperbolic functions are also available:

```
double   sinh(double);
complex  sinh(complex);
double   cosh(double);
complex  cosh(complex);
```

Other trigonometric and hyperbolic functions, for example *tan*() and *tanh*(), can be written by the user using overloaded function names.

Efficiency

C++'s facility for overloading function names allows *complex* to handle overloaded function calls in an efficient manner. If a function name is declared to be overloaded, and that name is invoked in a function call, then the declaration list for that function is scanned in order, and the first occurrence of the appropriate function with matching arguments will be invoked. For example, consider the exponential function:

```
double   exp(double);
complex  exp(complex);
```

When called with a *double* argument the first, and in this case most efficient, *exp*() will be invoked. If a *complex* result is needed, the *double* result is then implicitly converted using the appropriate constructor. For example:

```
complex foo = exp(3.5);
```

is evaluated as

```
complex foo = complex( exp(3.5) );
```

and not

```
complex foo = exp( complex(3.5) );
```

Constructors can also be used explicitly. For example:

```
complex add(complex a1, complex a2)  //silly way of doing a1+a2
{
      return complex( real(a1)+real(a2), imag(a1)+imag(a2);
}
```

Inline functions are used to avoid function call overhead for the simplest operations, for example, *conj*(), + , += , and the constructors (See **"Type complex"**).

Type complex

This is the definition of type *complex*. It can be included as *complex.h*. A *friend* declaration specifies that a function may access the internal representation of a *complex*. The standard header file *stream.h* is included to allow declaration of the stream I/O operators << and >> for complex numbers.

```
#include <stream.h>
#include <errno.h>
#include <math.h>
class complex {
          double  re, im;
public:
      complex() { re=im=0; }
      complex(double r = 0, double i) { re=r; im=i; }
      friend  double  abs(complex);
      friend  double  norm(complex);
      friend  double  arg(complex);
      friend  complex conj(complex);
      friend  complex cos(complex);
      friend  complex cosh(complex);
      friend  complex exp(complex);
      friend  double  imag(complex);
      friend  complex log(complex);
      friend  complex pow(double, complex);
      friend  complex pow(complex, int);
      friend  complex pow(complex, double);
      friend  complex pow(complex, complex);
      friend  complex polar(double, double = 0);
      friend  double  real(complex);
      friend  complex sin(complex);
      friend  complex sinh(complex);
      friend  complex sqrt(complex);
      friend  complex operator+(complex, complex);
      friend  complex operator-(complex);
      friend  complex operator-(complex, complex);
      friend  complex operator*(complex, complex);
      friend  complex operator/(complex, complex);
      friend  int     operator==(complex, complex);
      friend  int     operator!=(complex, complex);
      void operator+=(complex);
      void operator-=(complex);
      void operator*=(complex);
      void operator/=(complex);
};
ostream& operator<<(ostream&, complex);
```

```
ostream& operator<<(ostream&, complex);
istream& operator>>(istream&, complex&);
inline complex operator+(complex a1, complex a2)
{
	return complex(a1.re+a2.re, a1.im+a2.im);
}
inline complex operator-(complex a1,complex a2)
{
	return complex(a1.re-a2.re, a1.im-a2.im);
}
inline complex operator-(complex a)
{
	return complex(-a.re, a.im);
}
inline complex conj(complex a)
{
	return complex(a.re, -a.im);
}
inline int operator==(complex a, complex b)
{
	return (a.re==b.re && a.im==b.im);
}
inline int operator!=(complex a, complex b)
{
	return (a.re!=b.re || a.im!=b.im);
}
inline void complex.operator+=(complex a)
{
	re += a.re;
	im += a.im;
}
inline void complex.operator-=(complex a)
{
	re -= a.re;
	im -= a.im;
}
```

An FFT Function

Transcribed from Fortran as presented in "*FFT as Nested Multiplication, with a Twist*" by Carl de Boor in SIAM Sci. Stat. Comput., Vol 1 No 1, March 1980.

```
#include <complex.h>
void fftstp(complex*, int, int, int, complex*);
const NEXTMX = 12;
int prime[NEXTMX] = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
                        37 };
complex* fft(complex *z1, complex *z2, int n, int inzee)
/*
	Construct the discrete Fourier transform of z1 (or z2)
	in the Cooley-Tukey way, but with a twist.
	z1[before], z2[before].
	inzee==1 means input in z1; inzee==2 means input in z2
*/
{
	int before = n;
	int after = 1;
	int next = 0;
```

```
    int now;
    do {
    int np = prime[next];
    if ( (before/np)*np < before ) {
        if (++next < NEXTMX) continue;
        now = before;
        before = 1;
    }
    else {
        now = np;
        before /= np;
    }
    if (inzee == 1)
        fftstp(z1, after, now, before, z2);
    else
        fftstp(z2, after, now, before, z1);
    inzee = 3 - inzee;
    after *= now;
    } while (1 < before)
    return (inzee==1) ? z1 : z2;
}
void fftstp(complex* zin, int after, int now,
              int before, complex* zout)
/*
    zin(after,before,now)
    zout(after,now,before)
    there is ample scope for optimization
*/
{
    double angle = PI2/(now*after);
    complex omega = complex(cos(angle), -sin(angle));
    complex arg = 1;
    for (int j=0; j<now; j++) {
    for (int ia=0; ia<after; ia++) {
        for (int ib=0; ib<before; ib++) {
             // value = zin(ia,ib,now)
        complex value = zin[ia + ib*after +
                (now-1)*before*after];
        for (int in=now-2; 0<=in; in--) {
         // value = value*arg + zin(ia,ib,in)
            value *= arg;
            value += zin[ia + ib*after + in*before*after];
        }
         // zout(ia,j,ib) = value
        zout[ia + j*after + ib*now*after] = value;
        }
        arg *= omega;
    }
    }
}
```

The main program below calls *fft*() with a sine curve as argument. The complete unedited output is presented on the next page. All but two of the numbers ought to have been zero. The very small numbers shows the roundoff errors. Since C++ floating-point arithmetic is done in double-precision these errors are smaller than the equivalent errors obtained using the published Fortran version.

```
#include <complex.h>
extern complex* fft(complex*, complex*, int, int);
main()
```

```
/*
      test fft() with a sine curve
*/
{
      const n = 26;
      complex* z1 = new complex[n];
      complex* z2 = new complex[n];
      cout << "input: \m";
      for (int i=0; i<n ;i++) {
          z1[i] = sin(i*PI2/n);
          cout << z1[i] << "\m";
      }
      errno = 0;
      complex* zout = fft(z1, z2, n, 1);
      if (errno) cerr << "Cerror " << errno << " occurred\m";
      cout << "output: \m";
      for (int j=0; j<n ;j++) cout << zout[j] << "\m";
}
input:
(0, 0)
(0.239316, 0)
(0.464723, 0)
0.663123, 0)
(0.822984, 0)
(0.935016, 0)
(0.992709, 0)
(0.992709, 0)
(0.935016, 0)
(0.822984, 0)
(0.663123, 0)
(0.464723, 0)
(0.239316, 0)
(4.35984e-17, 0)
(-0.239316, 0)
(-0.464723, 0)
(-0.663123, 0)
(-0.822984, 0)
(-0.935016, 0)
(-0.992709, 0)
(-0.992709, 0)
(-0.935016, 0)
(-0.822984, 0)
(-0.663123, 0)
(-0.464723, 0)
(-0.239316, 0)
output:
(9.56401e-17, 0)
(-3.76665e-16, -13)
(9.39828e-17, 1.11261e-17)
(6.42219e-16, -4.20613e-17)
(7.37279e-17, 2.33319e-16)
(2.85084e-16, 2.87918e-16)
(4.03134e-17, 5.1789e-17)
(2.60865e-16, 6.78794e-17)
(-5.71667e-17, -3.86348e-17)
(2.76315e-16, 2.36902e-17)
(-6.43755e-17, -3.80255e-17)
(1.95031e-16, 9.77858e-17)
(1.49087e-16, -7.57345e-17)
```

```
(3.17224e-16, 1.64294e-17)
(1.49087e-16, 7.57345e-17)
(2.7218e-16, -4.03777e-17)
(-6.43755e-17, 3.80255e-17)
(4.93805e-16, 3.36874e-17)
(-5.71667e-17, 3.86348e-17)
(7.86047e-16, -4.11068e-18)
(4.03134e-17, -5.1789e-17)
(1.60788e-15, -1.06841e-16)
(7.37279e-17, -2.33319e-16)
(5.45186e-15, 2.42719e-16)
(9.39828e-17, -1.11261e-17)
(-1.12013e-14, 13)
```

Errors and Error Handling

These are the declarations used by the error handling:

```
int errno;
int complex_error(int, double);
```

The user can supply *complex_error*(). Otherwise a function that simply sets *errno* is used. The exceptions generated are:

`cosh(zz):`

`C_COSH_RE`	| zz.re | too large. Value with correct angle and huge magnitude returned.
`C_COSH_IM`	| zz.im | too large. Complex(0,0) returned.

`exp(zz):`

`C_EXP_RE_POS`	zz.im too small. Value with correct angle and huge magnitude returned.
`C_EXP_RE_NEG`	zz.re too small. Complex(0,0) returned.
`C_EXP_IM`	| zz.im | too large. Complex(0,0) returned.

`log(zz):`

`C_LOG_0`	zz==0. Value with a large real part and zero imaginary part returned.

`sinh(zz):`

`C_SINH_RE`	| zz.re | too large. Value with correct angle and huge magnitude returned.
`C_SINH_IM`	| zz.im | too large. Complex(0,0) returned.

Chapter 2

Iostream Examples

This chapter, containing many examples, is an introduction to the library. Overloading and other C++ features are used to provide an interface that combines flexibility and type checking. Predefined and user defined operations are easily mixed.

This chapter is not a complete description of the *iostream* library. Some classes and members are not described at all. Some are used without complete descriptions. The reader is referred to the *iostream* man pages for more details.

This chapter contains the following sections:

- **"Introduction"**
- **"Output"**
- **"Input"**
- **"Creating Streams"**
- **"Format Control"**
- **"Manipulators"**
- **"The Sequence Abstraction"**
- **"Deriving New streambuf Classes"**
- **"Extending Streams"**
- **"Comparison of Iostreams, Streams, and Stdio"**

- **"Converting from Streams to Iostreams"**

Note: This chapter is from a paper by Jerry Schwarz.

Introduction

C and C++ share the property that they do not contain any special input or output statements. Instead, I/O is implemented using ordinary mechanisms and standard libraries. In C this is the *stdio* library. In C++ (since Release 2.0 of the AT&T C++ Language System) it is the *iostream* library. Because C++ is an extension of C it is possible for a C++ program to use *stdio*. Using *stdio* may be the easiest way for a C programmer to get started with C++, but using *stdio* is not a good style for C++ I/O. Its main drawbacks are its type insecurity and the inability to extend it consistently for user defined classes.

This chapter consists mainly of examples of the use of parts of the *iostream* library. It assumes a reasonable familiarity with C++, including such extensions to C as references, operator overloading, and the like. An attempt has been made to create examples that not only illustrate features of the *iostream* library, but represent good programming style. A programmer who is new to C++ may copy the examples "cookbook style," but cannot be said to have mastered C++ until he or she understands the examples.

Some of the examples are moderately complicated and demonstrate advanced features of the *iostream* library. These are included so that the chapter will continue to be useful as an aid even after the programmer has written a few programs using iostreams. The author is annoyed by "tutorials" that show how to do simple things that he could figure out himself, but are silent about the harder, more sophisticated kinds of code that he frequently wants to write.

The declarations for the *iostream* library exist in several header files. To use any part of it, a program should include *iostream.h*. Other header files may be needed for other operations. These are mentioned below, but the *#include* lines are never put in the examples.

The *iostream* library is divided into two levels. The low−level (based on the *streambuf* class) is responsible for producing and consuming characters. This level is an independent abstraction and may be used without the upper level. This is appropriate when the program is moving characters around without much (or any) formatting operations. The *streambuf* class supports alternate sources and sinks of characters.

The upper level is responsible for formatting. The *iostream* library supports formatted I/O in C++. There are three significant classes. *istream* and *ostream* are responsible for input and output formatting, respectively. They are both derived from class *ios*, which contains members relating to error conditions and the interface to the low−level. A third class, *iostream*, is derived (multiple inheritance) from both *istream* and *ostream*. It plays only a minor role in the library. A "stream class" is any class derived from *istream* or *ostream*.

The topics covered in this chapter are:

- Output—predefined output conversions, ways to deal with errors, and ways to adapt the library for output of user classes.
- Input—predefined input conversions, and ways to adapt the library for input of user classes.
- Constructing specialized streams—file I/O and incore operations.
- Format Control—An *ios* contains some format state variables. This section describes how they are manipulated by user code and interpreted by the predefined operations.
- Manipulators—A powerful method for customizing operations.
- Streambufs—How to use the low−level interface.
- Deriving *streambuf* Classes—Methods for creating specialized classes that specialize *streambuf* to deal with alternate producers and consumers of characters.
- Extending Streams—Deriving classes from *istream* and *ostream*, adding state variables, and initialization issues.
- Comparison of I/O libraries.
- Compatibility—Converting a program that uses the old stream library to use the new library.

Output

Suppose we want to print the variable *x*. The main mechanism for doing output in the *iostream* library is the *insertion* operator <<. This operator is usually called left shift (because that is its built–in meaning for integers) but in the context of iostreams it is called insertion.

```
cout << x ;
```

cout is a predefined *ostream* and if *x* has a numeric type (other than *char* or *unsigned char*) the insertion operator will convert *x* to a sequence of digits and punctuation, and send this sequence to standard output. There are different operations depending on the type of *x*, and the mechanism used to select the operator is ordinary overload resolution. The insertion operator for type *t* is called the "*t* inserter."

If we have two values we might do:

```
cout << x << y ;
```

which will output *x* and *y*, but without any separation between them. To annotate the output we might do:

```
cout << "x=" << x
       << ",y=" << y
       << ",sum=" << (x + y) << "\n" ;
```

This will not only print the values of *x*, *y*, and their sum, but labels as well. It uses the string (*char**) inserter, which copies zero terminated strings to an *ostream*.

Notice the parentheses around the sum. These are not needed because the precedence of *+* is higher than that of *<<*. But, when using *<<* as insertion, it is easy to forget that C++ is giving it a precedence appropriate to shift. Getting in the habit of always putting in parentheses is a good way to avoid nasty surprises such as having *cout<<x&y* output *x* rather than *x&y*.

The output might look like:

```
x=23,y=159,sum=182
```

A pointer (*void**) inserter is also defined.

```
int x = 99 ;
cout << &x ;
```

It prints the pointer in hex.

A *char* inserter is defined:

```
char a = 'a' ;
cout << a << '\n' ;
```

This prints *a* and newline.

User Defined Insertion Operators

What if we want to insert a value of class type?

Inserters can be declared for classes and values of class type and used with exactly the same syntax as inserters for the primitive types. That is, assuming the proper declarations and definitions, the examples from the previous section can be used when *x* or are variables with class types.

The simplest kinds of examples are provided by a *struct* that contains a few values.

```
struct Pair { int x ; int y ; } ;
```

We want to insert such values into an *ostream*, so we define:

```
ostream& operator<<(ostream& o, Pair p) {
           return o << p.x << " " << p.y ;
           }
```

This operator inserts two integral values (separated by a space) contained in *p* into *o*, and then returns a reference to *o*.

The pattern of taking an *ostream&* as its first argument and returning the same *ostream* is what makes it possible for insertions to be strung together conveniently.

As a slightly more elaborate example, consider the following class, which is assumed to implement a variable size vector:

```
class Vec {
private:
            ...
public:
                    Vec();
          int     size() ;
          void    resize(int) ;
          float&  operator[](int) ;
          ...
} ;
```

We imagine that *Vec* has a current *size*, which may be modified by *resize*, and that access to individual (float) elements of the vector is supplied by the subscript operator. We want to insert *Vec* values into an *ostream*, so we declare:

```
ostream& operator<< (ostream& o, const Vec& v) ;
```

The definition of this operator is given below. Using *Vec&* rather than *Vec* as the type of the second argument avoids some unnecessary copying, which in this case might be expensive. Of course, using *Vec** would have a similar advantage in terms of performance, but would obscure the fact that it is the value of the *Vec* itself that is being output, and not the pointer.

The definition might be:

```
ostream& operator<< (ostream& o, const Vec& v)
{
           o << "[" ;        // prefix
           for ( int x = 0 ; x < v.size() ; ++x )
                     // use comma as separator
                     if ( x!=0 ) o << ',' ;
                     o << v[x] ;
                     }
           return o << "]" ; // suffix
           }
```

This will output the list as a comma separated list of numbers surrounded by brackets. The code takes care to get the empty list right and to avoid a trailing comma.

Propagating Errors

None of the examples so far has checked for errors. Omitting such checks would be bad style, except that the *iostream* library is arranged so that errors are propagated.

Streams have an error state. When an error occurs bits are set in the state according to the general category of the error. By convention, inserters ignore attempts to insert things into an *ostream* with error bits set, and such attempts do not change the stream's state. The error bits are declared in an *enum*, which is declared inside the declaration of class *ios*.

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

ios::goodbit is not really a "bit." It is zero and indicates the absence of any bit.

In the definitions of the *Pair* and *Vec* inserters, if an error occurs some wasted computation may be done as the code does insertions that have no effect. But eventually the error will be properly propagated to the caller.

It is a good idea to check the output stream in some central place. For example:

```
if (!cout) error("aborting because of output error") ;
```

The state of *cout* is examined with *operator!*, which will have a non−zero value if the state indicates an error has occurred. This and other examples in this chapter assume that *error*() is a function to be called when an error is discovered, and that it does not return. But *error*() is not part of the *iostream* library.

An *ostream* can also appear in a "boolean" position and be tested.

```
if ( cout << x ) return ;
... ; // error handling
```

The magic here is that *ios* contains a definition for *operator void** that returns a non−null value when the error state is non−zero.

An explicit member function also exists:

```
if ( ... , cout.good() ) return ;
... ;              // error handling
```

The reader is referred to the man pages for other member functions that examine the error state.

Flushing

In many circumstances the *iostream* library accumulates characters so that it can send them to the ultimate output consumer in larger (presumably more efficient) chunks. This is a problem mainly in interactive programs where the user may need to see the output before entering input. It can also be a problem during debugging when the programmer may need to see how far the program has gotten before dumping core. The easiest way to make sure that everything inserted into an *ostream* has been sent to the ultimate consumer is to insert a special value, *flush*. For example:

```
cout << "Please enter date:" << flush ;
```

Inserting *flush* into an *ostream* forces all characters that have been previously inserted to be sent to the ultimate consumer of the *ostream*. *flush* is an example of a kind of object known as a *manipulator,* a value that may be inserted into an *ostream* to have some effect. It is really a function that takes an *ostream&* argument and returns its argument after performing some actions on it.

Another useful way to cause flushing is the *endl* manipulator, which inserts a newline and then flushes.

```
cout << "x=" << x << endl ;
```

Binary Output

Sometimes a program needs to output binary data or a single character.

```
int c='A' ;
cout.put(c) ;
cout << (char)c;
```

The last two lines are equivalent. Each inserts a single character (*A*) into *cout*.

If we want to output a larger object in its binary form a loop using *put* would be possible, but a more efficient method is to use the *write* member. For example:

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

will output the raw binary form of *x*.

The reader should notice that the above example violates C++ type discipline by converting *&x* to *char**. Sometimes this is harmless, but if the type of *x* is a class with virtual member functions, or one that requires non−trivial constructor actions, the value written by the above cannot be read back in properly.

Input

Iostream input is similar to output. It uses *extraction* (>>) operators that can be strung together. For example:

```
cin >> x >> y ;
```

inputs two values from the predefined *istream cin*, which is by default the standard input. The extractor used will be appropriate for the types. The lexical details of numbers are discussed below under "Format Control." Whitespace characters (spaces, newlines, tabs, form−feeds) will be ignored before *x* and between *x* and *y*. For most types (including all the numeric ones), at least one whitespace character is required between *x* and *y* to mark where *x* ends.

There is a *char* extractor. For example:

```
char c ;
cin >> c ;
```

skips whitespace, extracts the next visible character from the *istream* and stores it in *c*. ("Non−whitespace" is too ugly a phrase for extensive use. This chapter uses "visible" instead. Strictly speaking this terminology is incorrect. For example, it classifies control characters as visible. But the term is reasonably euphonious and reasonably clear.)

Sometimes it is desirable to extract the next character unconditionally. For example:

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

The next character is extracted and stored in *c*, whether or not it is whitespace.

User Defined Extraction Operators

Creating extractors for classes is similar to creating inserters. The *Pair* extractor could be defined thus:

```
istream& operator>>(istream& i, Pair& pair)
{
            return i >> pair.x >> pair.y ;
            }
```

By convention, an extractor converts characters from its first (*istream&*) argument, stores the result in its second (reference) argument, and returns its first argument. Making the second argument a reference is essential because the purpose of an extractor is to store a new value in the second argument.

A subtle point is the propagation of errors by extractors. By convention, an extractor whose first argument has a non−zero error state will not extract any more characters from the *istream* and will not clear bits in the error state, but it is allowed to set previously unset error bits. Further, an extractor that fails for some reason must set at least one error bit. The code in the *Pair* extractor does nothing explicitly to respect these conventions, but because the only way it modifies *i* is with extractors that honor the conventions, the conventions will be respected.

Conventions also apply to the meaning of the individual error bits. In particular *ios::failbit* indicates that some problem was encountered while getting characters from the ultimate producer, while *ios::badbit* means that the characters read from the stream did not conform to the expectation of the extractor. For example, suppose that the components of a *Pair* are supposed to be non−zero.

The above definition might become:

```
istream& operator>>(istream& i, Pair& pair)
{
            i >> pair.x >> pair.y ;
            if ( !i ) return i ;
            if ( pair.x == 0 || pair.y == 0 ) {
                    i.clear(ios::badbit|i->rdstate()) ;
                    }
            return i ;
            }
```

This uses the (misleadingly named) *clear*() member function to set the error state to indicate that the extractor found incorrect data. ORing *ios::badbit* with *i–>rdstate*() (the current state) preserves any bits that may previously have been set.

The *Pair* extractor has been defined so that it can input values that were output by the *Pair* inserter. Maintaining this symmetry is an important general principle that is worth some effort.

The next example is the *Vec* extractor, which will require an opening [followed by a sequence of numbers, followed by a]. Recall that the *Vec* inserter uses, as a separator and does not insert any whitespace between numbers. The extractor must accept such input. It will also accept slightly more general formats. In particular it allows extra whitespace, and it allows any visible character to be used as a separator. It also deals properly with a variety of special conditions such as errors in the input format.

```
istream& operator>>(iostream& i, Vec& v)
{
           int n = 0;              // number of elements
           char delim ;
           v.resize(n) ;
           // verify opening prefix
           i >> delim ;
           if ( delim != '[' ) ;
                    i.putback(delim);
                    i.clear(ios::badbit|i.rdstate()) ;
                    return i ;
                    }
           if ( i.flags() & ios::skipws ) i >> ws ;
           if ( i.peek() == ']' ) return i ;
           // loop
           while ( i && delim != ']' ) {
                    v.resize(++n) ;
                    i >> v[n-1] >> delim ;
                    }
           return i ;
           }
```

The steps this code performs are:

- Turn *v* into an empty vector. This is done by the first *resize* operation.
- Verify that the next character in the istream is *[*.

  If the next character is not [(or if the state of the iostream already has error bits set), mark the state of *i* as bad, put *delim* back in *e* (where it may later be extracted again), and return. Putting *delim* back in the stream is not essential but it is consistent with the behavior of the predefined extractors.
- Optionally skip some whitespace.

  Whether to skip is controlled by the *ios::skipws* flag set in a collection of bits known as *i* 's format flags. This bit also controls skipping of whitespace in the predefined extractors. If it is set, whitespace was skipped before extracting the character stored into *delim*.
- If the next character is], the input represents an empty vector and since *v* has already been resized the extractor can just return.

  The next character is examined using the *peek*() member function. This returns the next character that would be extracted but leaves it in the stream.
- The code now loops, extracting numbers and delimiters until either the closing] is found or an input error occurs. An explicit check of the state of *i* is required to prevent an infinite loop should an error occur in extracting *vec[n–1]* or *delim*.

char* Extractor

A useful extractor, but one that must be used with caution, takes a *char** second argument. For example,

```
char p[100];
cin >> p;
```

skips whitespace on *cin*, extracts visible characters from *cin* and copies them into *p* until another whitespace character is encountered. Finally it stores a terminating null (0) character. The *char** extractor must be used with caution because if there are too many visible characters in the *istream*, the array will overflow.

The above example is more carefully written as:

```
char p[100] ;
cin.width(sizeof(p)) ;
cin >> p ;
```

There are very few circumstances (perhaps there are none at all) in which it is appropriate to use the *char** extractor without setting the "width" of the *istream*.

To make specifying a width more convenient, the *setw* manipulator (declared in *iomanip.h*) may be used. The above example is equivalent to:

```
char p[100] ;
cin >> setw(sizeof(p)) >> p ;
```

Binary Input

The *char* extractor skips whitespace. Programs frequently need to read the next character whether or not it is whitespace. This can be done with the *get*() member function.

For example,

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

get() returns the *istream* and a common idiom is:

```
char c ;
while ( cin.get(c) ) {
            ...
            }
```

Programs also occasionally need to read binary values (e.g., those written with *write*()) and this can be done with the *read*() member function.

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

This does the inverse of the earlier *write* example (namely, it inputs the raw binary form of *x*).

If a program is doing a lot of character binary input, it may be more efficient to use the lower level part of the *iostream* library (*streambuf* classes) directly rather than through streams.

Creating Streams

The examples so far have used the predefined streams, *cin* and *cout*. For some programs, reading from standard input and writing to standard output suffices. But other programs need to create streams with alternate sources and sinks for characters. This section discusses the various kinds of streams that are available in the *iostream* library.

Files

The classes *ofstream* and *ifstream* are derived from *ostream* and *istream* and inherit the insertion and extraction operations respectively. In addition they contain members and constructors that deal with files. The examples in this section assume that the header file *fstream.h* has been included.

If the program wants to read or write a particular file it can do so by declaring an *ifstream* or *ofstream* respectively. For example,

```
ifstream  source("from") ;
```

```
if ( !source ) error("unable to open 'from' for input");
ofstream  target("to") ;
if ( !target ) error("unable to open 'to' for output");
char c ;
while ( target && source.get(c) ) target.put(c) ;
```

copies the file *from* to the file *to*. If the *ifstream*() or *ofstream*() constructor is unable to open a file in the requested mode it indicates this in the error state of the stream.

In some circumstances a program may wish to declare a file stream without specifying a file. This may be done and the filename supplied later. For example:

```
ifstream file ;
... ;
file.open(argv[1]) ;
```

It is even possible to reuse the same variable by closing it between calls to *open*().

For example:

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

In some circumstances the program may already have a file descriptor (such as the integer 0 for standard input) and want to use a file stream. For example,

```
ifstream infile ;
if ( strcmp(argv[1],"-") ) infile.open(argv[1],input) ;
else                       infile.attach(0) ;
```

opens *infile* to read a file named by *argv[1]*, unless the name is –. In that case it will connect *infile* with the standard input (file descriptor 0). A subtle point is that closing a file stream (either explicitly or implicitly in the destructor) will close the underlying file descriptor if it was opened with a filename, but not if it was supplied with *attach*.

Sometimes the program wants to modify the way in which the file is opened or used. For example, in some cases it is desirable that writes append to the end of a file rather than rewriting the previous values. The file stream constructors take a second argument that allows such variations to be specified. For example,

```
ofstream outfile("out",ios::app|ios::nocreate) ;
```

declares *outfile* and attempts to attach it to a file named *out*. Because *ios::app* is specified all writes will append to the file. Because *ios::nocreate* is specified the file will not be created. That is, the open will fail (indicated in *outfile*'s error status) if the file does not previously exist. The *enum open_mode* is declared in *ios*.

```
class ios {
      enum open_mode { in, out, app, ate, nocreate, noreplace } ;
} ;
```

These modes are each individual bits and may be ORed together. Their detailed meanings are described in the man pages.

Sometimes it is desirable to use the same file for both input and output. *fstream* is an *iostream* (a class derived via multiple inheritance from both *istream* and *ostream*). The type *streampos* is used for positions in an iostream. For example,

```
fstream tmp("tmp",ios::in|ios::out) ;
...
streampos p = tmp.tellp() ;// tellp() returns current position
```

```
tmp << x ;
...
tmp.seekg(p) ;              // seekg() repositions iostream
tmp >> x ;
```

saves the position of the file in *p*, writes *x* to it, and later returns to the same position to restore the value of *x*.

A variant of *seekg*() takes a streamoff (integral value) and a *seek_dir* to specify relative positioning. For example,

```
tmp.seekg(-10,ios::end) ;
```

positions the file 10 bytes from the end, and

```
tmp.seekg(10,ios::cur) ;
```

moves the file forward 10 bytes.

Incore Formatting

Despite its name, the *iostream* library may be used in situations that do not involve input or output. In particular, it can be used for "incore formatting" operations in arrays of characters. These operations are supported by the classes *istrstream* and *ostrstream*, which are derived from *istream* and *ostream* respectively. The examples of this section assume that the header file *strstream.h* has been included.

For example, to interpret the contents of the string *argv[1]* as an integer value, the code might look like:

```
int i ;
istrstream(argv[1]) >> i ;
```

The argument of the *istrstream*() constructor is a *char* pointer. In this example, there is no need for a named *strstream*. An anonymous constructor is more direct.

The inverse operation, taking a value and converting it to characters that are stored into an array, is also possible. For example,

```
char s[32] ;
ostrstream(s,sizeof(s)) << x << ends ;
```

will store the character representation of *x* in *s* with a terminating null character supplied by the *ends* (*endstring*) manipulator. The *iostream* library requires that a size be supplied to the constructor and nothing is ever stored outside the bounds of the supplied array. In this case, an "output error" will occur if an attempt is made to insert more than 32 characters.

In case it is inconvenient to preallocate enough space for the string, a program can use an *ostrstream*() constructor without any arguments. For example, suppose we want to read the entire contents of a file into memory.

```
ifstream in("infile") ;
// strstream with dynamic allocation
strstream incore ;
char c ;
while ( incore && in.get(c) ) incore.put(c) ;
// str returns pointer to allocated space
char* contents = incore.str() ;
...
// once str is called space belongs to caller
delete contents ;
```

The file *infile* is read and its contents inserted into *incore*. Space will be allocated using the ordinary C++ allocation (*operator new*) mechanism, and automatically increased as more characters are inserted. *incore.str*() returns a pointer to the currently allocated space and also "freezes" the *strstream* so that no more characters can be inserted. Until *incore* is frozen, it is the responsibility of the *strstream*() destructor to free any space that might have been allocated. But after the call to *str*(), the space becomes the caller's responsibility.

Predefined Streams

There are four predefined streams, *cin*, *cout*, *cerr*, and *clog*. The first three are connected to standard input, standard output, and standard error respectively. *clog* is also connected to standard error but, unlike *cerr*, *clog* is buffered. That is, characters are accumulated and written to standard error in chunks. *cout* is also buffered.

Frequently programs want to use either standard input and output or some external file depending on their command line arguments. One way is to use the predefined streams and assign to them. Assignment of streams is not possible in general but the predefined streams have special types which allow it. The reader is referred to the man pages for a discussion of the semantics of assignment. A more flexible style is to use a pointer or reference to a stream:

```
istream* in = &cin ;
...
if ( infile ) in = new ifstream(infile) ;
...
*in << x ;
```

Problems can occur when mixing code that uses iostreams with code that uses *stdio*. There is no connection between the predefined iostreams and the *stdio* standard *FILE* s except that they use the same file descriptors. It is possible to eliminate this problem by calling

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

which will connect the predefined iostreams with the corresponding *stdio* *FILE* s. Such connection is not the default because there is a significant performance penalty when the predefined files are made unbuffered as part of the connection.

Format Control

The default treatment of scalar types is that integral values (except *char* and *unsigned char*) are inserted in decimal, pointers (except *char** and *unsigned char**) in hex, floats and doubles with 6 digits of precision and all without leading or trailing padding. *char* and *unsigned char* values are just inserted as single characters. *char** and *unsigned char** values are treated as pointers to strings (null terminated sequences of characters). The default treatment for extraction of integer types is decimal numbers with leading whitespace permitted. An optional sign (*+* or *−*) is permitted, but without whitespace between it and the digits. Extraction is terminated by a non−digit character. Extraction for floating point types is similar except that the lexical possibilities for floating point numbers are an optional sign followed (without intervening whitespace) by a number according to C++ lexical rules.

For many purposes these defaults are adequate. When they are not, the program can do more formatting itself, or it can use the format control features of the *iostream* library. The examples in this section use these features.

Associated with each iostream is a collection of "format state variables" that control the details of conversions. The most important of these is a *long int* value that is interpreted as a collection of bits. These bits are declared as:

```
enum  { skipws=01,            // skip whitespace on input
          left=02,  right=04, internal=010,
                                // padding location
          dec=020, oct=040, hex=0100,
                                // conversion base
          showbase=0200, showpoint=0400, uppercase=01000,
          showpos=02000,
                                // modifiers
          scientific=04000, fixed=010000
                                // floating point notation
          } ;
```

These may be examined and set individually or collectively. For example, the *ios::skipws* controls whether leading whitespace is skipped by extractors.

```
char c ;
cin.setf(0,ios::skipws) ;           // turn off skipping
cin >> c ;
cin.setf(ios::skipws,ios::skipws) ; // turn it back on
```

The second argument of *setf* indicates which bits should be set. The first indicates what values they should be set to.

Manipulators are declared (in *iomanip.h*) that will have an equivalent effect. The above is equivalent to:

```
cin >> resetiosflags(ios::skipws)
      >> c
      >> setiosflags(ios::skipws) ;
```

resetiosflags resets (makes zero) the indicated bits and *setiosflags* sets (makes them 1) the indicated bits.

Commonly, we want to save the flags (or other state variables) and restore their value later. Consider:

```
long f = cin.flags() ;
cin.setf(ios::skipws,ios::skipws) ;
cin >> c ;
cin.flags(f) ;
```

The variant of *flags* without an argument returns the current value. The variant with an argument stores the argument into the *flags* state variable. This code does the same extraction as the previous code, but instead of arbitrarily leaving *cin* with skipping on it restores skipping to its previous status.

The pattern of member functions is repeated for other state variables. That is, if *svar* is some state variable, and *s* is a stream, then *s.svar*() returns the current value of the state variable and *s.svar(x)* stores the value *x* into the state variable.

Field Widths

The default behavior of the inserters is to insert only as many characters as is necessary to represent the value, but frequently programs want to have fixed size fields.

```
cout.width(5) ;
cout << x ;
```

will output extra space characters preceding the digits to bring the total number of inserted characters to five. If the value of *x* will not fit in five characters, enough characters will be inserted to express its value. The numeric inserters never truncate. The *width* state variable might be regarded as an implicit parameter of extractors because it is reset to 0 (which induces the default behavior) whenever it is used.

```
cout.width(5) ;
cout << x << " " << y ;
```

will output *x* in at least five characters, but will use only as many characters as necessary in outputting the separating space and *y*.

The value of the *width* state variable is honored by the inserters of the *iostream* library, but user defined inserters are responsible for interpreting it themselves. For example, the *Pair* inserter defined previously does nothing special with *width* and so if it is non−zero when the inserter is called the width will apply to the first *int* inserted, and not the second. If the inserter wants to honor *width* its definition might look like:

```
ostream& operator<<(ostream& o, Pair p) {
           int w = o.width() ;
           o.width(w/2) ;
           o << p.x << " " ;
           o.width(w/2-((w+1)&1)) ;
```

```
            o << p.y ;
            return o ;
            }
```

This inserts each number in half the requested width.

It is slightly awkward to mix calls to the *width*() member function with insertion operations. The manipulator *setw*() may be used. An alternative definition of the *Pair* inserter might be:

```
iostream& operator<<(iostream& ios, Pair p) {
            int w = ios.width() ;
            return ios << setw(w/2) << pair.x << " "
                     << setw(w/2+(w+1)&1)) << pair.y ;
            }
Pair
```

width is always interpreted as a minimum number of characters. There is no direct way to specify a maximum number of characters. In cases where a program wants to insert exactly a certain number of characters, it must do the work itself. For example,

```
if ( strlen(s) > w ) cout.write(s,w) ;
else                 cout << setw(w) << s ;
```

will always insert exactly *w* characters.

width is generally ignored by extractors, which tend to rely on the contents of the iostream to detect the end of a field. There is, however, an important exception. The *char** extractor interprets a non−zero width to be the size of the array. For example,

```
char a[16] ;
cin >> setw(sizeof(a)) >> a ;
if ( !isspace(cin.peek() ) error("String too long") ;
```

protects the program in case there are sixteen or more visible characters. As a further measure of protection, the extractor stores a trailing null in the last byte of the array when it stops because there are too many visible characters. This means that the number of characters extracted (not counting leading whitespace) will be at most one less than the specified width.

Flags control whether padding (when it occurs) causes the field to be left or right justified. The *fill* state variable (whose initial value is a space) supplies the character to be inserted.

```
cout.fill(*) ;
cout.setf(ios::left,ios::adjustfield) ;
cout << setw(5) << 13 << "," ;
cout.fill(#) ;      // set state variable
cout.setf(ios::right,ios::adjustfield) ;
cout << setw(5) << 14 << "\n" ;
```

results in a line of output that looks like:

```
13***,###14
```

Conversion Base

Integers are normally inserted and extracted in decimal notation, but this is controlled by flag bits. If none of *ios::dec*, *ios::hex*, or *ios::oct* are set the insertion is done in decimal but extractions are interpreted according to the C++ lexical conventions for integral constants. If *ios::showbase* is set then insertions will convert to an external form that can be read according to these conventions.

For example,

```
int x = 64;
cout << dec << x << " "
       << hex << x << " "
       << oct << x << endl ;
```

```
cout.setf(ios::showbase,ios::showbase) ;
cout << dec << x << " "
       << hex << x << " "
       << oct << x << endl ;
```

will result in the lines:

```
64 40 100
64 0x40 0100
```

setf() with only one argument turns the specified bits on, but doesn't turn any bits off.

Reading the lines shown above could be done by:

```
cin >> dec >> x
      >> hex >> x
      >> oct >> x
      >> resetiosflags(ios::basefield)
      >> x >> x >> x ;
```

The value stored in *x* will be 64 for each extraction. The *resetiosflags*() manipulator turns off the specified bits in the flags.

Miscellaneous Formatting

As a precaution against looping, zero width fields are considered a bad format by the extractors. So if the next character is whitespace and *ios::skipws* is not set, the arithmetic extractors will set an error bit.

The number of significant digits inserted by the floating point (*double*) inserter is controlled by the *precision* state variable. The details of the conversion are further controlled by certain flags. The reader is referred to the man page for more details.

It is good practice to flush *ostream* s appropriately. The *flush* and *endl* manipulators make it relatively easy to do so. Yet, there are circumstances in which some automatic flushing is appropriate. This is supported by the *ostream** valued state variable *tie*. If *i.tie* is non−null and an *istream* needs more characters, the *ostream* pointed at by *tie* is flushed. Initially *cin* is tied in this fashion to *cout* so that attempts to get more characters from standard input result in flushing standard output. This seems to handle most interactive programs reasonably well without imposing a large performance penalty on non−interactive programs and without creating different behavior when programs are connected to pipes rather than directly to a terminal. (Programs that won't work when their input or output is connected to a pipe are one of the author's pet peeves.) The overheads implied by tying are relatively small when compared with "big" extractors (such as the arithmetic ones) but may be large when single character operations are being performed. For this reason it is sometimes a good idea to break the tie by setting the state variable to 0. For example:

```
char c ;
// break the tie to improve performance of get.
cin.tie(0) ;
while ( cin.get(c) ) cout.put(c) ;
```

Manipulators

A manipulator is a value that can be inserted into or extracted from a stream to cause some special side effect. That is, some side effect besides inserting a representation of its value, or extracting characters and converting them to a value. A parameterized manipulator is a function (or a member of a class with an *operator*()) that returns a manipulator. Previous sections contain examples of the use of manipulators and parameterized manipulators. This section contains examples illustrating how to define manipulators. The predefined manipulators and macros discussed in this section are declared in the header file *iomanip.h*.

A (plain) manipulator is a function that takes an *istream&* or *ostream&* argument, operates on it in some way, and returns it. A (pointer to a) function of this type may be extracted from or inserted into a stream, respectively.

Many examples of manipulators (such as *flush* or *endl*) have already appeared in this paper. For example, a manipulator to insert a tab can be defined:

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

This seems over elaborate. Why not simply define tab as a character or string? One possible reason has to do with the namespace. There can be only one (global) variable in a C++ program named *tab* but because of overloading there can be many functions with that name.

Another common use of manipulators is to shorten the long names and sequences of operations required by the *iostream* library. For example,

```
ostream& fld(ostream& o) {
            o.setf(ios::showbase,ios::showbase) ;
            o.setf(ios::oct, ios::basefield) ;
            o.width(10) ;
            return o ;
            }
...
cout << fld << x ;
```

It is common for the function that manipulates a stream to need an auxiliary argument. *setw*() is an example of such a parameterized manipulator. To use parameterized manipulators the program must include *iomanip.h*. For example, we might want to supply the value to be printed to *fld* in the above.

```
ostream& fld(ostream& o, int n ) {
            long f = flags(ios::showbase|ios::oct) ;
            o << setw(10) << n ;
            flags(f) ;            // restore original flags
            return o ;
            }
OMANIP(int) fld(int n) {
            return OMANIP(int)(fld,n) ;
            }
...
cout << fld(23) ;
```

OMANIP is a macro and *OMANIP*(*int*) expands to the name of a class declared in *iomanip.h*. An *OMANIP* (*int*) insertion operator is also declared in *iomanip.h* and is used in the example. Note that *fld* in the above is overloaded; it is both the function that manipulates the stream and a function that returns an *OMANIP (int)*.

If we need parameterized manipulators for parameter types other than *int* and *long* (which are declared in *iomanip.h*), they must be declared. For example, suppose we want to read numbers that may have a suffix.

```
typedef long& Longref ;
IOMANIPdeclare(Longref) ;
    // Declares IMANIP(Longref), OMANIP(Longref), IOMANI(Longref)
    //          IAPP(Longref), OAPP(Longref), IOAPP(Longref)
istream& in_k(istream& i, long& n)
{
            // Extract an integer.
            // If suffix is present multiply by 1024
            i >> n ;
            if ( i.peek() == 'k' ) {
                    i.ignore(1) ;
```

```
                  n *= 1024 ;
                  }
         return i ;
         }
IAPP(Intref) in_k = in_k ;
         // IAPP(Intref) is the type of an Intref applicator
         // in_k on right is function, on left variable
...
long n ;
cin >> in_k(n) ;
```

The *IOMANIPdeclare*(*T*) declares manipulators (and applicators) for type *T*. Because of the way the macro *IOMANIPdeclare* expands, the argument must be an identifier. In this case a *typedef* is required to create manipulators for *long&*. An applicator is something that behaves like a function returning a manipulator. That is, it is a class with an *operator*() member.

Sometimes we want a manipulator with more than one parameter. One way to achieve this effect is to define a manipulator on a class. For example, a manipulator that can be used to repeat a string might look like:

```
cout << repeat("ab",3) << endl ;
```

to result in a line containing "ababab." A possible definition of repeat would be

```
struct Repeatpair {
         const char* s ;
         int n ;
} ;
IOMANIPdeclare(Repeatpair) ;
static ostream& repeat(ostream& o, Repeatpair p) {
         // insert p.s into o, p.n times
         for ( int n = p.n ; n > 0 ; --n ) o << p.s ;
         return o ;
         }
OMANIP(Repeatpair) repeat(const char* s, int n) {
         Repeatpair p ;
         p.s=s ; p.n=n ;
         return OMANIP(Repeatpair)(repeat,p) ;
         }
```

Manipulators are a powerful and flexible method of extending the default inserters and extractors.

The Sequence Abstraction

The *iostream* library is built in two layers: The formatting layer discussed in previous sections, and a sequence layer based on the class *streambuf*. The formatting layer is responsible for converting between sequences of characters and various types of values and for high−level manipulations of the streams. The sequencing layer is responsible for producing and consuming those sequences of characters. The most common way of using *streambufs* is with a stream. But *streambuf* is an independent class and may be used directly.

Abstractly, a *streambuf* represents a sequence of characters and two pointers into that sequence, a *get* and a *put* pointer. These pointers should be thought of as pointing at the locations either before or after characters in the sequence, rather than at specific characters. The sequences and pointers may be manipulated in a variety of ways, with the two fundamental ones being fetching the character after the *get* pointer, and storing a character in the position after the *put* pointer. Storing either replaces any previous character at that location or, if the *put* pointer was at the end of the sequence, extends the sequence. Other manipulations may move the pointers in various ways.

For the examples of this section, we assume that there are two *streambuf*s, pointed at by *in* and *out*. Methods for constructing *streambuf*s appear later, but it is easy enough to get at the *streambuf* associated with a stream via *rdbuf*(). So we assume that *in* and *out* have been initialized with

```
streambuf* in = cin.rdbuf() ;
streambuf* out = cout.rdbuf() ;
```

An *istream* or *ostream* retains no information about the state of the associated *streambuf*. For example a program may alternate between extracting characters from *in* and *cin*.

The simplest operations are getting and putting characters. A simple loop to copy characters from one *streambuf* to another would be:

```
int c ;
while (( c = in->sbumpc()) != EOF ) {
          if ( out->sputc(c) == EOF ) error("output error") ;
          }
```

sbumpc() fetches the character after the *get* pointer and advances the *get* pointer over the fetched character. *sputc*() stores a character into the sequence and moves the *put* pointer past it. Both functions report errors by returning *EOF*, which is why *c* must be declared an *int* rather than a *char*. *EOF*s returned while fetching tend to mean that the *streambuf* has run out of characters from the ultimate producer. *EOF* s returned when storing tend to signal real errors. Because, unlike *iostreams*, *streambuf*s do not contain any error state, it is possible that a store or fetch might fail one time and succeed the next time it is tried.

The *streambuf* class contains several different member functions for manipulating the *get* pointer. The following loop represents a common idiom:

```
int c = in->sgetc() ;
while ( c!=EOF && !isspace(c) ) {
       c = in->snextc() ;
       }
```

It scans the *streambuf* looking for a whitespace character (i.e., one for which *isspace* is non−zero). It stops when it finds that character leaving it available for extraction. This is because *sgetc*()*f1 and snextc*() do not behave the way many programmers expect. *sgetc*() returns the character after the *get* pointer, but does not move the pointer. *snextc*() moves the *get* pointer and then returns the character that follows the new location. As usual both these functions return *EOF* to signal an error.

The copy loop moved characters one at a time. It is possible to do larger chunks, as in:

```
static const int Bufsize = 1024 ;
char buf[Bufsize] ;
int p, g ;
do {
       g = sgetn(buf, Bufsize) ;
       p = sputn(buf, g) ;
       if ( p!=g ) error("output error");
       } while ( g>0 ) ;
```

sgetn(*b*,*n*) attempts to fetch *n* characters from the sequence into the array starting at *b*. Similarly *sputn*(*b,n*) tries to store the *n* characters starting at *b* into the sequence. Both move the pointer (get or put respectively) over the characters they have processed and return the number transferred. For *sgetn*() this will be less than the number requested when the end of sequence is reached. When *sputn*() returns less than the number requested, it indicates an error of some sort.

Buffering Exposed

As the name suggests *streambuf*s may implement the sequence abstraction by buffering between the source and sink of characters. This results in an unfortunate pun. The word "buffer" is frequently used informally to designate a *streambuf*, but it is also used to describe the chunking of characters. Thus, the oxymoron "unbuffered buffer" refers to a *streambuf* in which characters are passed to the ultimate consumer as soon as they are stored, and obtained from the ultimate producer whenever they are retrieved.

In light of the buffering provided by *streambuf*s, the reader will not be surprised to discover that arrays of characters are used in the implementation. The *streambuf* class contains some member functions that make the presence of such arrays visible to the program. With some effort, they might be used to "break the abstraction," but the intended purpose is to deal with the delays implicit in buffering.

The earlier example using *sgetn*() and *sputn*() to copy from *in* to *out* waits until *Bufsize* characters become available (or the end of the sequence is reached) before passing any to *out*. If the source of characters has delays (e.g., it is a person typing at a terminal) and we want the characters to be passed on as soon as they become available; the program might use operations on single characters instead, or it might use an adaptive method such as:

```
static const int Bufsize = 1024 ;
char buf[Bufsize] ;
int p, g ;
do {
       in->sgetc() ;     // force a character in buffer
       g = in->in_avail() ;
       if ( g > Bufsize ) g = Bufsize ;
       g = in->sgetn(buf,g) ;
       p = out->sputn(buf,g) ;
       out->sync() ;
       if ( p!=g ) error("output error");
       } while ( g > 0 )
```

in_avail returns the number of characters immediately available in the array. Calling *sgetc*() first forces there to be at least one such character (unless the *get* pointer is at the end of the sequence). Recall that *sgetc*() returns the next character, but doesn't move the *get* pointer. The code calls *sync*() after it has put characters into *out*, thus causing these characters to be sent to the ultimate consumer.

In some circumstances, such as when *streambuf*s are being used for interprocess messages, the chunks in which characters are produced and consumed may have significance. The above preserves these chunks provided they are less than *Bufsize* and they fit into the arrays of *in* and *out*. To ensure that this latter condition is met, the code should provide large enough arrays explicitly with:

```
char ibuf[Bufsize+8], obuf[Bufsize+8] ;
in->setbuf(ibuf,sizeof(ibuf)) ;
out->setbuf(obuf,sizeof(obuf)) ;
```

The calls to *setbuf*() should be done before any fetches or stores are done. The arrays are eight larger than required by the largest chunk to allow for various overheads. Of course, this code behaves properly only when *in* delivers the characters in the appropriate chunks.

Using streambufs in Streams

The positions of the *put* pointer after operations that store characters and position of the *get* pointer after operations that fetch characters are well defined by the sequence abstraction. But the location of the *get* pointer after stores, and the location of the *put* pointer after fetches is not. Most specializations of *streambuf* (i.e., classes derived from it) follow one of two patterns. Either the class is queue−like, which means that the *put* pointer and the *get* pointer are independent and moving one has no effect on the other. Or the class is file−like, which means that when one pointer moves the other is adjusted to point to the same place. So a file−like class behaves as if there were only one pointer. Other possibilities are logically possible, but do not seem to be as useful.

A queue−like *streambuf*, may be shared between two streams. For example:

```
strstreambuf b ;
ostream ins(&b) ;
istream extr(&b) ;
while ( ... ) {
       ins << x ; ... ;
       extr >> x ;  ... ;
       }
```

This example explicitly uses the *strstreambuf* class (declared in *strstream.h*) which is also used (implicitly) by the *istrstream* and *ostrstream* classes. The *istream*() and *ostream*() constructors require a *streambuf* argument. They use that *streambuf* as a producer or consumer of characters. The characters inserted into *ins* may later be extracted from *extr*. If an attempt is ever made to extract more characters than have been inserted, the extraction will fail. If more characters are later inserted, *extr*'s error state can be cleared and the extraction retried.

Because of the dynamic allocation performed by *strstreambuf*s the queue is unbounded, but there is a serious drawback. Space is not reclaimed until *b* is destroyed.

Deriving New streambuf Classes

The *streambuf* class is intended to serve as a base class. Although it contains members to manipulate the sequences, it does not contain any mechanism for producing or consuming characters. These must be provided by a derived class. The *iostream* library contains several such derived *streambuf* classes, but a program may define new ones.

The members of a class that are intended for use by derived classes are *protected*, and the data structure as seen by a derived class is said to be the protected interface of the *streambuf* class. This abstraction exposes the details of the array management that is implicit in the buffering provided by *streambuf*s. It consists of two parts. The first part is member functions of *streambuf* that permit access to and manipulation of the arrays and pointers used to implement the sequence abstraction. The second part is virtual members of *streambuf* that must be supplied by the derived class.

The principle example of this section will be the implementation of *fctbuf*, whose declaration looks like:

```
typedef int (*action)(char* b, int n, open_mode m) ;
class fctbuf : public streambuf {
public:
       fctbuf(action f,open_mode m) ;
private: ...
} ;
```

When called with *m=ios::out*, an *action*() function processes the *n* characters starting at *b*. When called with *m=ios::in*, it stores *n* characters starting at *b*. It returns non−zero to indicate success and zero to indicate failure.

The declaration of *fctbuf* looks like:

```
class fctbuf : public streambuf {
public:                           // constructor
           fctbuf(action a, open_mode m) ;
private:                          // data members
       action     fct ;
       open_mode
                  mode ;
       char       small[1] ;
protected:                        // virtuals
       int     overflow(int) ;
       int     underflow() ;
       streambuf*
               setbuf(char*,int,int) ;
       int     sync() ;
} ;
```

The constructor just initializes the data elements. The action function *a* will be called only in modes compatible with *m*.

```
fctbuf::fctbuf(action a, open_mode m)
           : fct(a), mode(m) { }
```

The virtual functions define details that make *fctbuf*() behave properly. The *streambuf* protected interface is organized around three areas (*char* arrays), the holding area, the get area, and the put area.

Characters are stored into the put area and fetched from the get area.

As characters are stored in the put area, it shrinks until there is no more space available. If an attempt is made to store a character when the put area has no space, a new area must be established. Before that can be done the old characters must be consumed. Both these tasks are the responsibility of the *overflow*() function. Similarly, the get area is shrunk by fetches and is eventually empty. If more characters are needed the *underflow*() function must create a new get area. Both *overflow*() and *underflow*() will use the holding area to initialize the put or get area (respectively).

setbuf

The virtual function *setbuf* is called by user code to offer an array for use as a holding area. It can also be used to turn off buffering.

```
streambuf* fctbuf::setbuf(char* b, int len)
{
       if ( base() ) return 0 ;
       if ( b!=0 && len > sizeof(small) ) {
                 // set up holding area
                 setb(b,b+len) ;
                 }
       else {
                 // Use a one character array to achieve
                 // "unbuffered" actions.
                 setb(small,small+sizeof(small)) ;
                 }
       setp(0,0) ;              // put area
       setg(0,0,0) ;            // get area
       return this ;
       }
```

The actions of this function are:

- *base*() points to the first character of the holding area. If a holding area has already been set up (*base* non−zero) a new one cannot be established and *setbuf*() returns a null pointer as an error indication.
- If an array is supplied and is sufficiently large, *setb*() is called to set up the pointers to the holding area. Its first argument becomes *base*, the first *char* of the holding area, and its second becomes *ebuf*, the *char* after the last. Otherwise the *fctbuf* will become unbuffered. This is noted by setting up a one character holding area.
- Finally the pointers related to the put area are set to 0 by *setp*() and the pointers related to the get area are set to 0 by *setg*().

overflow

The virtual function *overflow*() is called to send some characters to the consumer, and establish the put area. Usually (but not always) when it is called, the put area has no space remaining.

```
int fctbuf::overflow(int c) {
       // check that output is allowed
       if ( !(mode&ios::out) ) return EOF ;
       // Make sure there is a holding area
       if (  allocate()==EOF ) return EOF ;
       // Verify that there are no characters in
       // get area.
       if ( gptr() && gptr() < egptr() ) return EOF ;
       // Reset get area
       setg(0,0,0) ;
       // Make sure there is a put area
       if ( !pptr() ) setp(base(),base()) ;
       // Determine how many characters have been
```

```
        // inserted but not consumed.
        int w = pptr()-pbase() ;
        // If c is not EOF it is a character that must
        // also be consumed.
        if ( c != EOF ) {
                 // We always leave space
                 *pptr() = c ;
                 ++w ;
                 }
        // consume characters.
        int ok = (*fct)(pbase(), w, ios::out) ;
        if ( ok ) {
                 // Set up put area. Be sure that there
                 // is space at end for one extra character.
                 setp(base(),ebuf()-1) ;
                 return zapeof(c) ;
                 }
        else {
                 // Indicate error.
                 setp(0,0) ;
                 return EOF ;
                 }
        }
```

Some explanations of this code:

- It first tests for various error conditions, such as trying to do insertion when there are characters that have been produced but not extracted. This is a problem because the code only uses one area to hold characters for insertion and extraction. It would also be possible to ignore this condition and just throw away the characters or a more elaborate version of *fctbuf* might use separate areas for insertion and extraction.
- *allocate*() is a part of the *streambuf* protected interface. If no reserve area has previously been specified it allocates heap space.
- *pbase* is the value of *pptr* established by the last call to *setp*(). As characters are stored, *pptr* is moved so that it always points to the first unused character. Thus the characters between *pbase* and *pptr* have been stored and not consumed. They are now sent to the consumer.
- The value returned by the consumer is checked to verify that it has been able to consume all the characters that were passed to it. If not, there is no put area and *EOF* is returned.
- When all has gone well the put area is established by *setp*() whose first argument becomes *pptr* (pointing to the first *char* of the put area) and whose second becomes *epptr* (pointing to the *char* after the last *char* of the put area). In this case when no errors have occurred the whole holding area minus the last character is used as a put area. The last character will usually be filled in by the character supplied to the next call to *overflow*().
- Finally, if all has gone well, *c* is returned unless it is *EOF*. If *c* is *EOF* something else must be returned because *EOF* is returned to signal an error. The macro *zapeof*() deals with this contingency.

underflow

The underflow function is called when characters are needed for fetching and none are available in the get area. Its general outline is similar to *overflow*(), but it deals with the get area rather than the put area.

```
int fctbuf::underflow() {
        // Check that input is allowed
        if ( !(mode&ios::in) ) return EOF ;
        // Make sure there is a holding area.
        if (allocate()==EOF) return EOF ;
        // If there are characters waiting for output
        // send them ;
        if ( pptr() && pptr() > pbase() ) overflow(EOF) ;
        // Reset put area
        setp(0,0) ;
```

```
        // Setup get area ;
        if ( blen() > 1 ) setg(base(),base()+1,ebuf()) ;
        else              setg(base(),base(),ebuf()) ;
        // Produce characters
        int ok = (*fct)(base(),blen(),ios::in) ;
        if ( ok ) {
                return zapeof(*base()) ;
                }
        else {
                setg(0,0,0) ;
                return EOF ;
                }
        }
```

Some explanations:

- *EOF* is returned immediately if we aren't supposed to do input or if a holding area cannot be allocated.
- *allocate*() is called to make sure that there is a holding area.
- *setg*() is used to establish the get area where *fct* will be asked to store characters. Its first argument sets up a pointer, *eback*, that marks the limit to which *putback* can move *gptr*. The second argument becomes *gptr*, and the last becomes *egptr*, pointing at the char after the last *char* containing values stored by the producer.
- *blen*() returns the size of the holding area. It may be as small as 1.
- If the action function indicated success *underflow*() returns the first character. It is left in the get area and may be extracted again. *zapeof*() is used to make sure that the returned result is not *EOF*. If *zapeof*() were omitted this might occur on a machine in which *char* s are signed and *EOF* is −1.

sync

The virtual function *sync*() is called to maintain synchronization between the various areas and the producer or consumer. It is also called by the *streambuf*() destructor.

```
int fctbuf::sync()
{
        if ( gptr() && egptr() > gptr() ) {
                // no way to return characters to producer
                return EOF ;
                }
        if ( pptr() && pptr() > pbase() ) {
                // Flush waiting output
                return overflow(EOF) ;
                }
        // nothing to do
        return 0 ;
        }
```

The virtual functions defined above implement a correct *streambuf* class. A possible refinement would be to provide implementations of the virtual *xsputn*() and *xsgetn*() functions. These functions are called when chunks of characters are being inserted and extracted respectively. Their default actions are to copy the data into the buffer. If they were defined in the *fctbuf* class they could call the functions directly and avoid the extra copy.

Extending Streams

There are two kinds of reasons to extend the basic stream classes. The first is to specialize to a particular kind of *streambuf* and the second is to add some new state variables.

Specializing istream or ostream

When the *iostream* library is specialized for a new source or sink of characters the natural pattern is this: First derive a class from *streambuf*, such as *fctbuf* in the previous section. Then derive classes from whichever of *istream*, *ostream*, or *iostream* is appropriate. For example, suppose we want to do this with the *fctbuf* class defined in the previous section. The streams might get the definitions:

```
class fctbase : virtual public ios {
public:
                     fctbase(action a, open_mode m)
                             : buf(a,m) { init(&buf) ; }
private:
           fctbuf     buf ;
} ;
class ifctstream : public fctbase, public istream {
public:
                     ifctbase(action a)
                              : fctbase(a, ios::in) { }
} ;
class ofctstream : public fctbase, public ostream {
public:
                     ofctbase(action a)
                              : fctbase(a, ios::out) { }
} ;
class iofctstream : public fctbase, public iostream {
public:
                     iofctstream(action a open_mode m)
                              : fctbase(a, m) { }
} ;
```

Derivations from *ios* are virtual so that when the class hierarchy joins (as it does in *iofctstream*) there will be only one copy of the error state information. Because the derivation from *ios* is virtual an argument cannot be supplied to its constructor. The *streambuf* is supplied via *ios::init*(), which is a protected member of *ios* intended precisely for this purpose.

Extending State Variables

In many circumstances we would like to add state variables to streams. For example, suppose we are printing trees and would like to have an indentation level associated with an *ostream*.

```
int xdent = ios::xalloc() ;
           // generate a unique index
ostream& indent(ostream& o) {
           // manipulator that inserts newlines and
           // appropriate number of tabs
           o << '\n' ;
           int count = o.iword(xdent) ;
           while ( count-- > 0 ) o << ' ' ;
           return o ;
           }
ostream& redent(ostream& o, int n) {
           // parameterized manipulator that modifies
           // indentation level
           o.iword(xdent) += n ;
           }
OAPP(int) redent = redent ;
```

o.iword(xdent) is a reference to the *xdent*th integer state variable. Each call to *ios::xalloc* returns a different index. The index may then be used to access a word associated with the stream. The reason for calling *ios::xalloc* to get an index rather than just picking an arbitrary one is that it allows combining code that uses the indentation level with code that may have extended the formatting state variables for some other purpose.

A subtle problem occurs in the above example because *xdent* is initialized by a function call. What if *indent*() or *redent*() were called before *xdent* was initialized? Can that happen? Yes it can. It can happen if *indent*() or *redent*() is called from inside a constructor that is itself called to initialize some variable with program extent. Problems with order of initialization when doing I/O in constructors are common. The solution relies on "tricks" to force initialization order. In this case we would put into the header file containing the declarations of *indent*() and *redent*():

```
static class Indent_init {
           static int count ;
public:
                    Indent_init() ;
                    ~Indent_init() ;
} indent_init ;
```

Each file that includes this header file will have a local variable *indent_init* that has to be initialized. Because this variable is declared in the header its initialization will occur early. The definition of the constructor and destructor looks like:

```
static Iostream_init* io ;
Indent_init::Indent_init()
{
       // count keeps track of the difference between how
       // many constructor and destructor calls there are
       if ( count++ > 0 ) return ;
       // This code is executed only the first time
       io = new Iostream_init ;
       xdent = ios::xalloc() ;
       }
Indent_init::~Indent_init()
{
       if ( --count ) > 0 ) return ;
       // This code will be executed the last time
       delete io ;
       }
```

The *iostream* library uses this idea itself. The constructor for *Iostream_init* causes the *iostream* library to be initialized the first time it is called. It also keeps track of how many times the constructor is called and will do finalization operations on various data structures the last time it is called. It is therefore important that any values of type *Iostream_init* that are constructed by a program are eventually deleted. This is the purpose of having an *Indent_init* destructor; even though there are no finalization operations associated with indentation, it must delete *io*.

Comparison of Iostreams, Streams, and Stdio

The *stdio* library served C programmers well for many years. However, it has several deficiencies:

- The use of functions, like *printf*(), that accept variable numbers and types of arguments mean that type checking is subverted at an important point in many programs.
- There is no mechanism for extending it to user defined classes. The only way to add new format specifiers to *printf*() is to reimplement it.
- The mechanism is closely tied to file I/O. *sprintf*() explicitly extends it to incore operations, but there is no general method for creating alternate sources and sinks of data.

After *stdio*, the next stage of development was the stream library. Its most significant innovation was the introduction of insertion and extraction operations. The first two problems with *stdio* were elegantly solved. It was in use by C++ programmers for several years. But the stream library had problems of its own:

- The mechanism for creating sources and sinks of characters (*streambuf* class) was not documented or designed for extension.
- The full range of UNIX file operations was not supported. In particular there were no repositioning

operations (*seek*s).

- There was only limited control over formatting. Programs frequently reverted to *printf*() like functions to specify alternative formats for numbers. A fixed size area was allocated for converting values to strings and then outputting the strings. Although it was not a problem in practice, in theory this buffer was subject to overflows.

The *iostream* library presented in this chapter has resolved these problems. It is relatively new, and whether significant problems will emerge in the future is not yet known. Some apparent deficiencies are:

- There is no way to determine if a producer has characters available, and no way to select input from one of multiple sources. This is, of course, also a deficiency of *stdio* and streams.
- There is no way to process data in the buffers without copying them out. This extra copying step can be expensive when simple operations (e.g., scanning for a specific character) are being performed.
- Some formatting operations tend to be wordier than the equivalent *stdio* operations. This is compensated for by the ability to define manipulators and inserters.

Converting from Streams to Iostreams

The *iostream* library is mostly upward compatible with the older stream library, but there are a few places where differences may affect programs. This section discusses those differences.

The major conceptual difference is that in the *iostream* library, *streams* and *streambuf*s are regarded solely as abstract classes. The old stream classes provided certain specialized behaviors, specifically incore formatting and file I/O. In the *iostream* library these are supported solely through derived classes.

The old stream library declared everything in the header file *stream.h*. The *iostream* library uses *iostream.h* and some other headers. A *stream.h* is supplied that includes *iostream.h* and other headers required for compatibility. stream.h also defines a variety of items whose names are different in the *iostream* and *stream* libraries.

streambuf Internals

The internals of the *streambuf* class in the stream library were all public. Any program that relies on these internals will break because they are different (and private) in the *iostream* library.

How to derive new *streambuf* classes was not documented in the stream library. But it is such a natural idea to do so that many programs do it. Converting these programs to the *iostream* library may require changes in the derived *overflow*() and *underflow*() functions. The functionality of these functions in the *iostream* library is essentially the same as in the stream library. But because the internals of *streambuf* have changed, some code changes will probably be required. In particular the code will have to use the (protected) *streambuf* member functions *setb*(), *setg*(), and *setp*() instead of directly manipulating the pointers.

Incore Formatting

In the stream library the use of arrays of characters as sources or sinks was supported as the default behavior of *streambuf*. Although some attempt to preserve the default behavior is made by the *iostream* library these uses of a *streambuf* are considered obsolete. The support of incore operations is specifically the responsibility of the *strstreambuf* declared in *strstream.h*. *streambuf*s created for this purpose can usually be replaced directly by *strstreambuf*s that have equivalent behavior. The stream usage:

```
char* buf[10] ;
streambuf b(buf,10) ;
```

is equivalent to the iostream:

```
char* buf[10] ;
strstreambuf b(buf,10) ;
```

and the old method for initializing a *streambuf* for extraction:

```
char* buf[10] ;
```

```
streambuf b ;
b.setbuf(buf,10,buf+5) ;
```

is equivalent to the iostream method:

```
char* buf[10] ;
strstreambuf b(buf,10,buf+5) ;
```

Frequently these uses of *streambuf* do not appear explicitly in the program but are the consequence of using certain constructors of *istream* and *ostream*. These constructors are supplied in the *iostream* library, but are considered obsolete. The equivalent forms using *strstream* should be used.

The old method of storing a formatted value into an array:

```
char* buf[10] ;
ostream out(10,b) ;
```

is replaced by:

```
char* buf[10] ;
ostrstream out(b,10) ;
```

Note that the order of the arguments is reversed. The new order creates more consistency between various uses of strstreams.

The old method of extracting a formatted value from an array:

```
char* buf[10] ;
istream in(10,b) ;
```

is replaced by

```
char* buf[10] ;
istrstream in(b,10) ;
```

The old *istream*() constructor allowed an optional extra argument to specify skipping of whitespace. In the *iostream* library this is part of a greatly expanded collection of state variables and so an extra argument is not provided for the *istrstream*() constructor. However, the obsolete form of *istream*() constructor continues to accept these optional arguments.

Filebuf

Both libraries contain a *filebuf* class for using streams to do I/O. It is declared in *fstream.h* in the *iostream* library. The stream library had constructors that implied the use of *filebuf*s. In the *iostream* library these constructors are replaced by constructors of certain derived classes. The old usage:

```
int fd ;
istream in(fd) ;       // file descriptor
ostream out(fd) ;       // file descriptor
```

is replaced by:

```
int fd ;
ifstream in(fd) ;       // file descriptor
ofstream out(fd) ;      // file descriptor
```

The optional extra arguments of the stream constructors (for specifying whitespace skipping and "tying") are not supported. The equivalent functionality is supported by format state variables.

Interactions with stdio

The libraries differ significantly in the way they interact with *stdio*. The old stream header *stream.h* included *stdio.h* and some stream data structures could contain a pointer to a *stdio FILE*. In the *iostream* library specialized streams and *streambuf*s (declared in *stdiostream.h*) are provided to make the connection.

The old usage:

```
FILE* stdiofile ;
filebuf fb(stdiofile) ;
istream in(stdiofile) ;
ostream out(stdiofile) ;
constructor, obsolete form
constructor, obsolete form
constructor, obsolete form
```

is replaced by:

```
FILE* stdiofile ;
stdiobuf fb(stdiofile) ;
stdiostream in(stdiofile) ;
stdiostream out(stdiofile) ;
```

In the old library the predefined streams *cin*, *cout*, and *cerr* were directly connected to the *stdio* *FILE* s *stdin*, *stdout*, and *stderr*. I/O was mixed character by character. Further, these streams were unbuffered in the sense that insertion and extraction was done by doing character by character puts and gets on the corresponding *stdio* *FILE* s. In the *iostream* library the predefined streams are attached directly to file descriptors rather than to the *stdio* streams. This means that for output the characters are mixed only as flushes are done and the input buffer of one is not visible to the other.

In practice the biggest problems seem to come from attempts to mix code that uses *stdout* with code that uses *cout*. The best solution is to cause flushes to be inserted whenever the program switches from one library to the other. An alternative is to use:

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

This causes the predefined streams to be connected to the corresponding *stdio* files in an unbuffered mode. The major drawback of this solution is the large overheads associated with insertion of characters in this mode. Typically insertion into *cout* is slowed by a factor of 4 after a call of *sync_with_stdio*().

The old stream library contained some "stringifying" functions that were called with various arguments and returned a string. These are declared in *stream.h* and available primarily for compatibility. The only such formatting function that seems to provide a significant functionality that is not easily available in the *iostream* library is *form*(), which allows *printf*() like formatting. In fact, *form*() is just a wrapper for calls to *sprintf*(). The programmer can easily write manipulators and inserters that do the same thing.

Assignment

In the old library it was possible to assign one stream to another. This is possible in the *iostream* library only if the left hand side is declared to be an assignable class. A general assignment cannot be allowed because of the interactions of derived classes. What, for example, should be the effect of assigning an *ifstream* to an *istrstream*? Most programs that use this feature can be converted by using a reference or pointer to a stream. The old usage:

```
ostream out ;
out = cout ;
out << x ;
```

can be replaced by:

```
ostream* out ;
out = cout ;
out << x ;
```

or:

```
ostream_with_assign out ;
out = &cout ;
*out << x ;
```

char Insertion Operator

The stream library did not contain an insertion operator for *char*. So inserting a *char* was taken as inserting an integer value, and it was converted to decimal. This omission was due to problems with overload resolution in earlier versions of the C++ Language System. Any old code such as:

```
char c ;
cout << c ;
```

may be replaced by:

```
char c ;
cout << (int)c ;
```