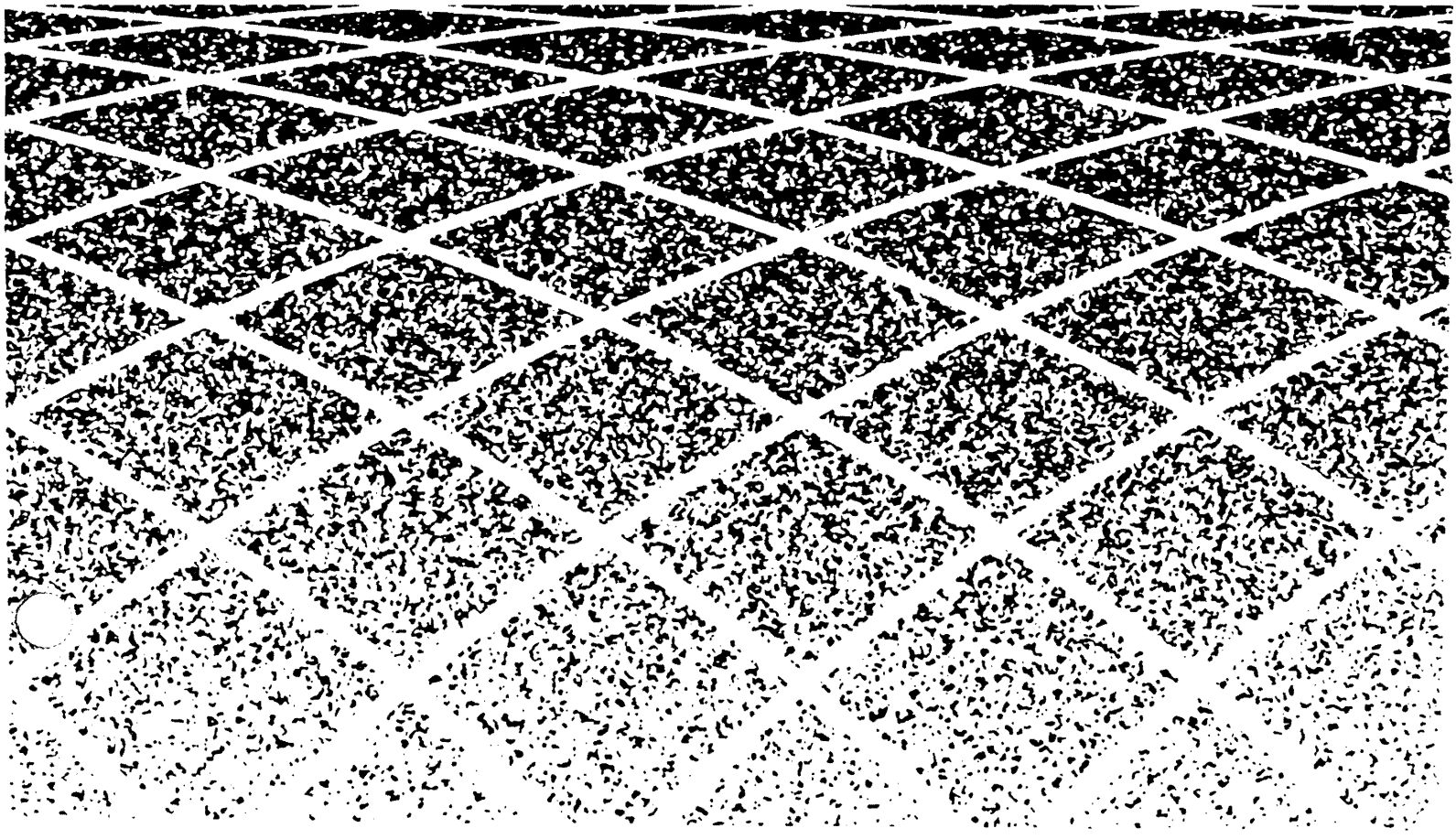
UNIX® System V
AT&T C++ Language System
Release 2.0

Library Manual
Select Code 307-145



NOTICE

The information in this document is subject to change without notice. AT&T assumes no responsibility for any errors that may appear in this document.

Contents

Figures and Tables

Preface

Preface

The *AT&T C++ Language System Library Manual* describes the C++ class libraries provided with Release 2.0 of the AT&T C++ Language System:

- the complex arithmetic library
- the task library
- the iostream library

The manual is part of a set of four documents that are supplied with your C++ Language System. The other documents are:

- the *Release Notes*, which describe the contents of this release, how to install it, and changes to the language
- the *Product Reference Manual*, which provides a complete definition of the C++ language supported by Release 2.0 of the Language System.
- the *Selected Readings*, which contains papers describing aspects of the C++ language

The chapters in this manual cover the following C++ class libraries:

- Chapter 1 describes the complex arithmetic library, which provides a class **complex** that allows you to declare and manipulate complex numbers in C++ programs
- Chapter 2 describes the task library, which allows you to create and control concurrent processes in C++ programs. The last section of Chapter 2 provides porting information for the task library, which is machine dependent.
- Chapter 3 describes the stream library, which allows you to do formatted input and output from C++ programs
- Appendix A contains manual pages for all three class libraries

To make the best use of the *Library Manual*, you must be familiar with the C programming language and the C programming environment under the UNIX® operating system. Refer to Appendix B of the *Release Notes* for further sources of information about these topics.

Acknowledgements

- Chapter 1 is based on the paper, "Complex Arithmetic in C++," by Leonie V. Rose and Bjarne Stroustrup. That paper acknowledges the following contributions:

  Phil Gillis supplied us with the complex functions used for the **cxp** package. Most of the functions presented here are modified versions of those. Stu Feldman provided us with valuable advice and some functions. Doug McIlroy's constructive comments led to a major rewrite. Eric Grosse suggested the FFT function as an example.

- Chapter 2 is based on papers by Bjarne Stroustrup, Jonathan Shopiro, and Stacey Keenan:

  - The first section is taken from "A Set of C++ Classes for Co-routine Style Programming," by Bjarne Stroustrup and Jonathan Shopiro, which was originally published in the Proceedings of the USENIX C++ Workshop, November, 1987. That paper acknowledges the following contributions:

    The task system is in many ways a descendant of A.G. Fraser's set of C functions described in the paper "C Language Routines for Multi-Thread Computation," Bell Telephone Laboratories Internal Memorandum, 1979. M.D. McIlroy acted as "midwife" for many parts of the design.

  - The second section is taken from "Extending the C++ Class System for Real-Time Control," by Jonathan Shopiro. That paper acknowledges the following contributions:

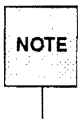
    The original task system was developed by Bjarne Stroustrup, and the original port to the 68000 was done by Rafael Bracho. This work was begun as an attempt to examine concurrency features of Concurrent C and those of the C++ task system by translating a robot control system developed in Concurrent C by Ingemar Cox, whose help is much appreciated. Also useful were conversations with Thomas Cargill and Bart Locanthi.

  - The third section is taken from "A Porting Guide for the C++ Coroutine Library," by Stacey Keenan.

- Chapter 3 is based on the paper "Iostream Examples," by Jerry Schwarz.

1 Complex Arithmetic in C++

Complex Arithmetic in C++



This chapter is taken directly from a paper by Leonie V. Rose and Bjarne Stroustrup.

Abstract

This memo describes a data type **complex** providing the basic facilities for using complex arithmetic in C++. 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: **(0.706107,1.10715)**.

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.

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 pre-processor 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 done 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."

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.[1] 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 **10e-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 **1** 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);
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.

Footnotes

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

2 The Task Library

Introduction

Roadmap for the C++ Task Library Documentation

The three sections of this chapter describe the C++ Language System coroutine or task library.

- The first section, "A Set of C++ Classes for Co-routine Style Programming," written by Bjarne Stroustrup and revised and updated by Jonathan Shopiro, describes how the task library can be used. Read this section to learn about the basic use of the task library.
- The second section, "Extending the C++ Task System for Real-Time Control," by Jonathan Shopiro, describes new features of the task library to enable tasks to receive UNIX system signals.
- The task system internals for Release 2.0 are described in the third section, "A Porting Guide for the C++ Coroutine Library," by Stacey Keenan. This part tells you about the internals of the task library.
- Man pages for the task library may be found in Appendix A of this manual.

A Set of C++ Classes for Co-routine Style Programming

NOTE

This section is taken directly from a paper by Bjarne Stroustrup and Jonathan E. Shopiro.

Abstract

Some programs are most naturally expressed as a set of relatively independent activities communicating to achieve a common goal. Each activity, here called a *task*, has its own locus of control, a program to execute, and its own private data. Tasks can communicate by explicit sharing of data, by messages, or by data pipes.

This paper describes C++ classes for a range of styles of multi-programming techniques in a single language, single address-space environment. Each task is an instance of a user-defined class derived from class **task**, and the program of the task is the constructor of its class. A task can be suspended and resumed without interfering with its internal state. Class **qhead** and class **qtail** enable a wide range of message passing and data buffering schemes to be implemented simply.

The task system can be used for writing event driven simulations. Tasks execute in a simulated time frame presented by the variable **clock**, and objects of class **timer** provide a convenient and efficient facility for using the clock.

The implementation and use of these concepts rely heavily on the idea of derived classes. Familiarity with the C++ language would be an advantage for the reader.

Introduction

Some programs are most naturally expressed as a set of relatively independent activities communicating to achieve a common goal. Such activities, here called *tasks*, must be able to execute in parallel with each other and communicate through means convenient to the chosen style of task usage.

Facilities for multi-thread computation can be provided in the semantics of a language, as is done in Concurrent Pascal and Mesa or a language without such facilities can be augmented using special run-time support systems and library functions, as has been done for BCPL and C. The use of C classes to implement tasks represents an intermediate approach pioneered by Simula67.

The tools presented here[1] provide the basic facilities for several styles of multi-thread programming in a single language, single address-space environment. The underlying facility is a simple and efficient tasking system with non-preemptive scheduling. That is, a task will only be suspended on its own request, so no "system policy" can be enforced without the cooperation of all tasks. In contrast to pure co-routine systems, however, the task system provides a framework for processor sharing and communication between tasks. The task system is intended for applications, like event driven simulations, where tasks are used to express a quasi-parallel structure for a single program. For this class of applications a concept of simulated time is implemented. A unit of simulated time can represent any amount of real time, and it is possible to compute without consuming simulated time. A few simple random number generating classes and a histogram class for data gathering are also provided. The task system is not intended for handling real parallelism of some underlying real-time system. Consequently, no facilities are provided to map interrupts and other real-time events into the concepts

provided by the task system.

> The current version of the task library has a new degree of extensibility, so that it is now possible to write a class that represents an interrupt or signal that can be waited for.

Implementations of the task system have been used for about eight years on the UNIX system and other operating systems on 3B2, 3B20, VAX, and Motorola 680x0 hardware.

In the following sections the task library will be described in some detail, and examples of its use will be given. The classes used in the task system are presented. This allows a detailed and specific discussion of the concepts involved, but it unfortunately also implies that some concepts cannot be explained in detail where they are first mentioned.

Tasks

The publicly accessible functions and data of class **task** look like this:[2]

```
class task : public sched
{
public:
                    task(char* name=0, int mode=0, int stacksize=0);
                    ~task();
          task*     t_next;
          char*     t_name;
          int       waitvec(object**);
          int       waitlist(object* ...);
          void      wait(object*);
          void      delay(int);
          int       preempt();
          void      sleep(object* t =0);
          void      resultis(int);
          void      cancel(int);
};
```

The base class, **sched**, is responsible for scheduling and for the functionality that is common to tasks and timers (described below). The public part of its declaration is:

```
class sched : public object {
public:
                    sched();
          void      setclock(long);
          long      rdtime();
          int       rdstate();
          int       pending();
          void      cancel(int);
          int       result();
};
```

Class sched is used strictly as a base class: that is, only instances of derived classes are created.

A task is a locus of control, a virtual processor. It too can only be used as a base class. A task executes the program supplied as the constructor of the derived class.[3] The most basic feature of a task is that it can be suspended and later resumed so that several tasks can run in quasi-parallel. Most member functions of class **task** are conditional or unconditional requests for suspension.

A task can be in one of three states:

RUNNING	The task is executing instructions or it will be scheduled to do so without further intervention from other tasks.
IDLE	The task is not in the **RUNNING** state, but it can be transferred to the **RUNNING** state by some suitable action. That is, it is waiting.
TERMINATED	The task has completed its work. It cannot be resumed, but its result can be retrieved.

The function **sched::rdstate()** returns the state.

A simple example of the use of tasks is where one task creates another to run in parallel with itself. Later the creator can obtain the result produced by the "secondary" task. For example, a task which counts the number of spaces in a string could be declared. First a class **Spaces** must be declared.

```
class Spaces : public task
{
public:
                Spaces(char*);
};
```

In the case of class **Spaces** the declaration is trivial. It states that **Spaces** is derived from class **task** so that each object of class **Spaces** becomes an independently scheduled entity. The program for the task is provided by its constructor.

```
Spaces::Spaces(register char* s)
{
        register int    i = 0;
        register char   c;
        while (c = *s++)
                if (c == ' ') i++;
        resultis(i);
}
```

This function counts the spaces in its argument string and returns the result using the class **task** function **resultis()**. A task of class **Spaces** can now be created and used like this:

```
Spaces   ss("a line with four spaces");
// ...
count = ss.result();
```

When an object of class **Spaces** is created, like **ss** here, its constructor becomes a new task that runs in parallel with the task[4] that created it. A task can "return" an integer[5] value using the function **task::resultis(int)**. The task then becomes **TERMINATED** and the value is available for examination by the function **sched::result()**. That is, in this example **ss** will call **resultis()** with the argument 4

which will be returned from **sched::result()** to the parent task. If a task calls **result()** for another task which has not yet completed the calling task will be suspended. After the other task finishes the call to **result()** in the waiting task will return. A task waiting for another to complete is **IDLE**. If a task calls **result()** for itself it will cause a run time error.[6]

A task cannot return a value using the usual function return mechanism; it must use **resultis()**. This function puts the task into the **TERMINATED** state from which it cannot be resumed.

Queues

A *queue* is a type of storage that is organized so that objects are retrieved from it in the order in which they were inserted into it. A queue has a *head* from which data is retrieved and a *tail* where data is inserted. With a little elaboration this basic type of data structure makes an excellent inter-task communication facility.

There is no "class queue" available to a user. Instead, the two classes **qhead** and **qtail** provide the services needed. There is a function **qtail::put()** which adds an object to the *tail* of a queue and a function **qhead::get()** which retrieves an object from the *head* of a queue. This allows explicit separation between the source and the recipient of data. The public part of the declaration of class **qhead** looks like this:

```
class qhead : public object
{
public:
                    qhead(int =WMODE, int =10000);
                    ~qhead();
          object*   get();
          int       putback(object*);
          int       rdcount();
          int       rdmode();
          int       rdmax();
          void      setmode(int);
          void      setmax(int);
          qtail*    tail();
          qhead*    cut();
          void      splice(qtail *);
          int       pending();
          void      print(int, int =0);
};
```

A queue can be created like this:

```
qhead       qh;
```

To obtain a **qtail** for an existing queue execute **tail()** for its head:

```
qtail*      qtp = qh.tail();
```

The queue could now be used as a one way inter-task communication channel by giving its head and tail as arguments to two new tasks, **Producer** and **Consumer**:

```
Producer    pp(qtp);
Consumer    cc(&qh);
```

The producer task **pp** can now **put()** objects to the tail of the queue (denoted by the pointer **qtp**) and the consumer task **cc** can **get()** those objects from its head (denoted by the pointer **&qh**). The function **qtail::put()** takes a pointer to a class **object** as argument, and **qhead::get()** returns such a pointer. Unless the user has specified otherwise a task executing **qhead::get()** will be suspended temporarily if the queue is empty.[7] After another task executes **put()** on the associated queue tail the suspended task will be resumed. Similarly a task executing **qtail::put()** on a full[8] queue will be suspended until some other task removes data from the queue.

The objects transmitted through a queue must be of class **object** or of some class derived from it. Class **object** (described under "The **object** Class") is provided by the task system, and it is up to the programmer to define types of objects suitable for each application.

> In the current version of the task library **qhead** and **qtail** have the form of user extensions, but in the original version they were built in. Since extensibility was limited, the supplied classes had to support a wide range of programming styles. Thus they may seem "feature-rich." The new organization makes it easy to provide new kinds of queues and other forms of task interaction.

A Server Example

As an example of the use of tasks and queues we will define a "server" task that receives requests for service in the form of messages on a queue, handles the requests and returns replies on other queues. One could define a class **Message** as follows:

```
class Message : public object
{
public:
        int     r_operation;
        int     r_arg1;
        int     r_arg2;
        qtail*  r_reply;
};
```

A message, that is an object of class **Message**, describes an operation **r_operation** that is to be performed by the recipient of the message. Arguments for this operation can be passed as **r_arg1** and **r_arg2**, and the result of the operation is to be returned as a message on the queue denoted by **r_reply**.

A server for these messages can be defined as follows:

```
class Server : public task
{
public:
                Server(qhead *);
```

```
};


Server::Server(qhead* in)
{
        for (;;) {
                Message*   req = (Message *) in->get();
                qtail*     reply = req->r_reply;
                int        res = VALUE;
                int        val;
                switch (req->r_operation) {
                case PLUS:
                val = req->r_arg1 + req->r_arg2;
                break;
                case MINUS:
                ...
                default:
                res = ERROR;
                }
                req->r_operation = res;
              req->r_arg1 = val;
              reply->put(req);
        }
}
```

This style of server has proved useful in many contexts. In particular, it is the backbone of many "message-based systems." In this particular example a server, that is an object of class **Server**, and the queue on which it depends can be declared:

```
qtail*    rq = new qtail;
Server*   ser = new Server(rq->head());
```

Other tasks can now send a request to this particular server through **rq**. For example:

```
qhead       rply;
qtail*      rply_to = rply.tail();
Message*    mess = new Message;


mess->r_operation = PLUS;
mess->r_arg1 = 1;
mess->r_arg2 = 2;
mess->r_reply = rply_to;


rq->put(mess);
mess = (Message *) rply.get();
if (mess->r_operation == ERROR) error();
```

More about Queues: Mode and Size

A queue head has a *mode* that controls what happens when **get()** is executed on an empty queue. In EMODE this causes a run time error. In **ZMODE** it will cause **get()** to return the *NULL* pointer instead of a pointer to an object. In **WMODE** a task executing a **get()** on an empty queue will wait on that queue until the queue becomes non-empty. Unless the user specifies the mode explicitly a queue head will be in **WMODE**. The function **qhead::rdmode()** returns the current mode and **qhead::setmode()** can be used to change it.

As mentioned above a queue also has a maximum size. This can be changed using **qhead::setmax()**, and read using **qhead::rdmax()**.

The mode and maximum size for a queue can also be specified when the queue is created. For example:

```
qhead    Q1(ZMODE, 10);
qhead*   QP2 = new qhead(EMODE, 64*BUFSIZE);
```

The public part of the declaration of class **qtail** is similar to that of class **qhead**. The two classes complement each other, and together they provide a representation of the general idea of a queue:

```
class qtail : public object
{
        // ...
public:
                    qtail(int = WMODE, int = 10000);
                    ~qtail();
          int       put(object*);
          int       rdspace();
          int       rdmax();
          int       rdmode();
          qtail*    cut();
          void      splice(qhead*);
          qhead*    head();
          void      setmode(int m);
          void      setmax(int m);
          int       pending();
          void      print(int, int =0);
};
```

A queue tail's mode controls what happens on queue overflow in the same way as a queue head's mode controls what happens on queue underflow. For example, when a task executes **put()** on a full queue where the queue tail is in **WMODE**, then that task will be suspended until the queue is no longer full. The modes of a queue's head and tail need not be the same.

Similarly the maximum number of objects which can be on a queue can be examined by **rdmax()** and changed by **setmax()**. Decreasing the maximum below the current number of objects on the queue is legal. Doing this simply implies that no new objects can be put on the queue until the queue has been drained below the new limit.

Qhead::rdcount() returns the current number of objects in a queue, and **qtail::rdspace()** returns the number of objects which can be inserted into a queue before it becomes full.

Qhead::putback() puts its argument back at the head of the queue, that is

```
qhead     qh(WMODE,10);
object*   oo = qh.get();
qh.putback(oo);
oo = qh.get();
```

will assign the same object to **oo** twice. **Putback()** has proved to be a useful function in many systems in the past, and it also allows a queue head to operate as a stack. When **putback()** is used, the task executing it competes for queue space with tasks using **put()** on the queue's tail. A **putback()** to a full queue causes a run time error in both EMODE and WMODE. In ZMODE it returns NULL.

More about Tasks

When a task is created it can be given three arguments. The first is a character string pointer which is used to initialize the class **task** variable **t_name**. This name can be used to provide more readable output and does not affect the behavior of the task. The string denoted by the pointer will not be copied. The **t_name** is used by the debugging aids and error reporting functions described below. The other two class **task** arguments are tuning parameters and will be described below. If an argument is *NULL* a system default will be used. For example, we could have given each **Server** task a name like this:

```
class Server : public task
{
                    Server(char*, qhead *);
};


void Server::Server(char* name, qhead* in)
: (name)  // argument for Server's base class task
{
          // ...
}


Server my_name_is_fred("fred", qhp);
```

Task::sleep(object* =0) suspends the task unconditionally without specifying what is supposed to cause it to be resumed.

> If an argument is given to **task::sleep(object* =0)** which is a pointer to a pending object, the task will be *remembered* by the object, so that after it is no longer pending, the task will be resumed.

Task::cancel() puts a task into the **TERMINATED** state and sets the return value just like **resultis()**. However, **cancel()** does not invoke the scheduler so that one task can terminate another without losing control itself.

The pointer

```
task*   thistask;
```

denotes the currently active task. If no tasks have been created its value is **0**. It is illegal to assign to **thistask**. The use of **thistask** enables the class **task** functions to be used from external functions without explicit passing of the current task's **this** pointer.

The pointer[9]

```
task*   task_chain;
```

is the start of a chain of all tasks. In the following loop **t** points to every task in turn:

```
task*   t;
for (t=task_chain; t; t=t->t_next) ;
```

It is not possible to have only one task. Therefore, when the first task is created in a program another task is implicitly created. Its name is **main** and its code is the original **main()** function. It can be suspended and resumed like any other task. Please remember that a return from **main()** terminates a C program. If the "main" task should be terminated when there are other tasks which should be left running, then **resultis()** can be used. For example,

```
thistask->resultis(0);
```

can be executed in **main()**. The program will then run on until no more tasks are or can become RUNNING.

It is illegal for a task to return. Always call **resultis()** instead of **return**, and never just "drop out of the bottom" of a task. Unless a task contains an infinite loop so that it will never terminate place a call of **resultis()** at the end of its body.

The task system does not provide a garbage collector. It is left to the programmer to ensure that pointers to deallocated store are not used.

Waiting

Functions like **sched::result()**, **qhead::get()**, and **qtail::put()** each provide a way of waiting for one single specific event to happen. More general facilities are sometimes needed.

When an object must be waited for, we say it is *pending*. For example,

- A queue head whose associated queue is empty is pending because if a task calls **get()** for it, the task must wait until some other task puts some data in the queue,
- Similarly, a queue tail whose queue is full is pending because a **put()** must wait, and
- A task that has not terminated is pending because its result is not available.

Each class derived from object may have its own definition of the virtual **pending()** function. An object may have several operations that could suspend the calling task, but it can have only one definition of **pending()**. Therefore (for example) it is not possible to combine a queue head and a queue tail into a single object, because the former is pending when its queue is empty, and the latter when its queue is full. New kinds of objects, with new kinds of interaction can be added to the task library, with the fundamental requirement being a definition of **pending()** for the new datatype.

Task::wait(object*) provides a way of waiting on an arbitrary object. If the argument points to a pending object, the calling task will be suspended until the object is no longer pending. If the argument is not pending the caller will not be suspended at all. For example, if **taskp** is a pointer to a task then

```
wait(taskp);
```

will suspend the task executing it until the task denoted by **taskp** finishes.

Each class derived from class **object** which is ever going to be "waited on" must have rules specifying under which conditions a task executing a **wait()** for it will be resumed. The rules for class **task**, **qhead**, and **qtail** have been stated.

The conditions for wakeup are reflected in state changes in the objects, and are not just transitory unrecorded signals. For example, if a task executes a **wait()** for a non-empty **qhead** it will immediately continue, that is the condition for returning from a **wait()** for a **qhead** is that the queue is non-empty, not a brief state change from empty to non-empty. Rules of this type simplify programming considerably by eliminating race conditions.

When the state of an object changes from pending to not pending, **object::alert()** must be called for the object. This function changes the state of all tasks "remembered" by the object from IDLE to RUNNING and puts them on the scheduler's **run_chain**. Thus all such operations should be member functions of the object's class or a related class. For example, in **qtail::put()**, if the queue was empty, a call to **alert()** is made for the associated queue head. If it was possible to put an object on a queue without calling a member function, then there would be no guarantee that **alert()** would be called.

The functions **task::waitvec()** and **task::waitlist()** suspend a task waiting for one of a list of objects, for example to wait for messages to arrive on one of a number of queue heads. **Waitlist(object* ...)**[10] takes a list of object pointers terminated by a zero as argument; for example:

```
qhead*   q1;
qhead*   q2;
// ...
short    who = waitlist(q1, q2, 0);
```

will suspend the task executing it until either **q1** or **q2** is non-empty. If either is non-empty when **waitlist()** is called the task will continue immediately.

The value returned is the position in the list of the object that caused the return from the wait, that is if **q2** caused the task to resume the value **1** will be assigned to **who**. Positions are numbered starting from *0*. **Waitlist()** can take any number of arguments. The degenerate example

```
waitlist(0);
```

causes unconditional suspension of the task executing it without any guarantee of later resumption. It is equivalent to **sleep()** and **wait(0)**.

Please note that one should not assume that because **waitlist()** returns a particular value indicating one object as the cause of resumption none of the other objects are "ready." The value returned by waitlist() only indicates what is known to have happened, and it does not exclude other independent possibilities.

> However if **waitlist()** indicates a particular object, that object is guaranteed to be "ready," because **waitlist()** does not return until the object is no longer pending.

Because every class in the task system allows non-blocking examination of the conditions which might lead to suspension using the three wait functions, the value returned by **waitlist()** can always be ignored. The information it conveys can always be obtained by direct inquiry. In many cases, however, the value returned can be trusted and used to write simpler, more efficient programs.

Waitvec(), a variation of **waitlist()**. takes the address of a vector holding a list of object pointers. For example:

```
object*   vec[] = { q1, q2, 0 };
short     who = waitvec(vec);
```

is equivalent to the previous example.

System Time and Timers

The **long** variable **clock** measures simulated time. It is initialized to zero. It is illegal to assign to clock.

Task::delay(int) suspends a task for a specified time. That is,

```
long   t = clock;
delay(n);
actual_delay = clock-t;
```

will assign the value **n** to **actual_delay**. **Delay()** is useful for representing service delays in simulations. While a task is delayed in this way its state is still **RUNNING**, but it will not be affected by the actions of other tasks except if **cancel()** or **preempt()** is used on it. **Delay(n)** makes an **IDLE** task RUNNING so that it will start executing at time **clock+n**.

Task::**preempt()** makes a **RUNNING** task **IDLE** and returns the number of time units left of its delay. Applying **preempt()** to an **IDLE** or **TERMINATED** task causes a run time error. This function is useful when tasks are used to represent processes in a system with preemptive scheduling and delay times are used to represent the time used by executing processes. The value returned by **preempt()** allows the preempted task to be re-started with a new delay time which is a function of the delay time at the time of preemption. For example:

```
int   time_left = other_task->preempt();
// ...
other_task->delay(time_left+10);
```

A timer provides a facility for implementing time-outs and other time dependent phenomena.

Class timer has this declaration:

```
class timer : public sched {
public:
                timer(int);
                ~timer();
        void    reset(int);
        void    print(int, int =0);
};
```

A timer is quite similar to a task with a constructor consisting of the single statement

```
delay(d);
```

that is, when a timer is created it simply waits for the number of time units given to it as its argument, and then wakes up any tasks waiting for it.

A timer's state can be either **RUNNING** or **TERMINATED**. This state can be inspected by using **sched::rdstate()**.

A common use of timers is to wait for a task and a timer. For example, one can wait for the completion of a task handling a simulated input operation and also on a timer. The timer ensures that the waiting task will eventually be resumed even if the input operation is never completed:[11]

```
timer*   tt = new timer(15);
short    res = waitlist(io_ptr,tt,0);
switch (res) {
case 0:
        /* normal completion of i/o */
        ...
        break;
case 1:
        /* time out occurred */
        ...
        break;
default:
        error(IMPOSSIBLE);
}
```

Sched::result and sched::cancel() have the same use and effects on timers as on tasks. Since there is no timer::resultis(), the value returned by sched::result() is undefined for a timer unless cancel() was used.

Timer::reset() re-sets the timer delay to the value of its argument. This makes repeated use of timers possible. A timer can be **reset()** even when it is **TERMINATED.**

A unit of simulated time can be used to represent any unit of real time. Only **delay()** causes the clock to advance.

More About Queues: Cutting and Splicing

One of the most convenient and powerful ways of using tasks involves tasks defined to do a transformation on a data stream. Such a task is called a filter. It reads its input from one queue and writes its output onto another queue. Tasks at the "other ends" of these queues tend to view these queues plus the filter as one entity. The data source simply sees an output queue that is being emptied at some rate, and the task at the receiving end sees an input queue being filled. In other words, a task sees only its input and output queues and cares little about the "internal organization" of the programs that operate on the other ends of those queues.

For example, one task could produce a stream of lines of characters, that is objects of class **Line**, and another expect an input stream consisting of words, that is objects of class **Word**. A filter that handles the conversion could be defined and used like this:

```
class Line_to_word : task
{
public:
                Line_to_word(qhead*, qtail*);
        Word*   next_word(Line*);
};


Line_to_word::Line_to_word(qhead* in_q, qtail* out_q)
{
        Line*   l;
        Word*   w;
        for(;;) {
                l = (Line *) in_q->get();
                while(w = next_word(l)) out_q->put((object *)w);
        }
}
```

```
qhead*          line_q = new qhead(WMODE,10);
qtail*          word_q = new qtail(WMODE,50);
Producer*       prod = new Producer(line_q->tail());
Consumer*       cons = new Consumer(word_q->head());
Line_to_word*   filt = new Line_to_word(line_q, word_q);
```

In this way the filter **filt** is programmed into the path between **cons** and **prod** using two queues to separate filt's input from its output.

This is a fairly static use of a filter. Often one would like to insert a filter into an existing data path. For example, a macro-based text formatting program could be organized as a sequence of filters — each doing its small part of the common task. First some filters re-arrange the input into a form suitable for the formatter proper, then the "input independent" formatter does its job producing output of a standard form, and last some output filters adjust this output to a form suitable for physical output. The task **filt** is an example of such a filter. In this scenario it would be useful to have each macro defined as a filter which the formatter proper inserts just in front of itself when the macro expansion is needed and which removes itself when it is not needed any more. Assuming that data streams are represented by queues, this can be achieved by using the class **qhead** functions **cut()** and **splice()**.

When the task **formatter** recognizes a call to the macro "foo" it creates a new task of class **Macro** to handle a macro of type FOO and diverts its own input through it. This is done by first "cutting" the input queue to create a place to insert the new filter, and then creating the filter giving it the new **qhead** and **qtail** as arguments:

```
qhead*   newhead = input_queue->cut();
qtail*   newtail = input_queue->tail();
Macro*   f = new Macro(FOO,newhead,newtail);
```

Qhead::cut() splits the queue to which it is applied into two. **Newhead**, the pointer returned from **cut()**, denotes the **qhead** for the original queue and has the same **mode** as the original **qhead**. The original **qhead** is now attached to a new empty queue with the same **max** as the original.

Puts to the original **qtail** will therefore place objects on the filter's input queue, and gets from the original **qhead** will retrieve objects from the filter's output queue.

The result of these operations has been to insert a filter with an input and an output queue into a queue without changing the appearance of that queue to anyone using it, and without halting the flow of objects through that queue. In our example the macro expansion filter **foo** will **get()** the input which would otherwise have gone to the formatter, interpret it as macro arguments, and output the expanded input as its output.

The filter can be removed again by splicing its input and output queues together with **qhead::splice()**:

```
newhead->splice(newtail);
```

Splice() deletes the **qhead** to which it is applied, the **qtail** given to it as an argument, and the queue denoted by that **qtail**. If the **splice()** operation causes an empty queue to become non-empty or a full queue to become non-full all tasks waiting for such a state change are resumed.

Deleting the filter completes the cleanup:

```
delete f;
```

Typically a filter would remove itself when its task was completed, because the task that inserted it would not be programmed to be aware of the presence of the filter it inserted. The sequence of operations which enables a task to remove itself without a trace is:

```
        cancel(any_value);
        delete this;
```

This will work because **cancel()** does not imply immediate suspension, only a guarantee that the task cannot be resumed.

Qtail::cut() and **qtail::splice()** are similar to **qhead**, but they operate on the other end of the queue.

Encapsulation

Passing information between tasks through queues can lead to rather tedious, repetitive (and therefore error prone) packing and unpacking of information into messages. Simple encapsulation techniques can be used to relieve the programmer of this. For example, by adding a constructor to the class **Message** the server example could be re-written thus:

```
class Message : object
{
public:
        int     r_operation;
        int     r_arg1;
        int     r_arg2;
        qtail*  r_reply;
                Message(int op, int a1, int a2, qtail* rp) :
                r_operation(op), r_arg1(a1),
                r_arg2(a2), r_reply(rp) {}
};
```

```
        Message*   mess;
        rq->put(new Message(PLUS, 1, 2, rply_to));
        mess = (Message *) rply.get();
        if (mess->r_operation == ERROR) error();
```

Furthermore, because the message queues obviously are meant to hold only **Message** objects a specific message queue could be defined and used:

```
class Mqhead : qhead
{
public:
        Message*    get() { return (Message *) qhead::get(); };
};


class Mqtail : qtail
{
public:
        int    put(Message* m) { return qtail::put(m); };
};
```

The use of **Mqtail::put()** ensures that only class **Message** objects are put on the queue, and no type cast is needed when class **Message** objects are taken from the queue using **Mqhead.get()**. For example:

```
mess = rply->get();
```

Because the body of **Mqtail::put()** is present in the class **Mqtail**, declaration calls of **Mqtail::put()** will be expanded inline. This ensures that using a **Mqtail** is no less efficient than using a **qtail** directly. In many cases some error handling can also be handled by the derived **put()** and **get()** functions.

An alternative solution is to provide the server class with functions which handle the packing:

```
class Server : task
{
        qtail*    inp;
public:
                  Server(char*, qhead*);
        int       plus(int, int, Mqtail *);
        int       minus(int, int, Mqtail *);
};


int Server::plus(int arg1, int arg2, Mqtail * rqt)
{
        Message*    mess;
        int         x;
        inp->put(new Message(PLUS,arg1,arg2,rqt));
        mess = rqt->head()->get();
        x = mess->r_operation;
        delete mess;
        return x;
}
```

so now the server task can be requested to perform services like this:

```
Mqtail   qq;
Server   ss("plus_and_minus", 0, 0);
int      two = ss.plus(1, 1, &qq);
int      ten = ss.minus(12, 2, &qq);
```

For large programs this style of inter-task communication promises not only increased clarity, but also increased efficiency. The message queue interaction may, where necessary, be transparently replaced by a specially tailored inter-task communication facility.

> These techniques are now widely applied in C++ programming, but when this paper was first written, they were new to C.

Histograms and Random Numbers

To ease data gathering class histogram is provided.

```
struct histogram
// "nbin" bins covering the range [l:r] uniformly
// nbin*binsize == r-l
{
        int     l, r;
        int     binsize;
        int     nbin;
        int*    h;
        long    sum;
        long    sqsum;
                histogram(int=16, int=0, int=16);
        void    add(int);
        void    print();
};
```

A histogram consists of **nbin** bins **h[0], ... h[nbin-1]** covering a range **[l:r]** of integers. The function **add()** adds one to the correct bin for its integer argument. The sum of the integers added is maintained in **sum**, and the sum of their squares is maintained in **sqsum**. If an argument to **add()** is outside the range **[l:r]** the range is adapted by either decreasing **l** or increasing **r**. The number of bins remains constant so the size of the range covered by a bin is doubled each time the size of the range [l:r] is. The **print()** function prints out the numbers of entries for each non-empty bin.

In most simulations some form of random number generation is needed. The generators provided here are intended to help the developer of a simulation to get started and to provide a paradigm for generators of more suitable distributions.

```
class randint
// uniform distribution in the interval [0,MAXINT_AS_FLOAT]
{
        long    randx;
public:
                randint(long s = 0);
        void    seed(long s);
        int     draw();
        float   fdraw();
};
```

The following program shows the use of class **randint**. The **ints** returned by **randint::draw()** are uniformly distributed in the interval **[0:largest_positive_int]**. The **floats** returned by **randint::fdraw()** are uniformly distributed in the interval **[0:1]**.

```
main()
{
        randint    ir;
        register   i;
        for (i=0; i<100; i++)
                printf("i=%d f=%f ", ir.draw(), ir.fdraw());
}
```

Each object of class **randint** provides an independent sequence of random numbers. **Randint::seed()** can be used to reinitialize a generator. The **draw()** function calls the underlying C library **rand(3)**. Using class **randint**, generators for other distributions are easily programmed. Note that **erand::draw()** calls **log()** from the math library, so a program using it must be loaded with **-lm**.

```
class urand : public randint
// uniform distribution in the interval [low,high]
{
public:
        int   low, high;
              urand(int l, int h) { low=l; high=h; }
        int   draw() { return int(low + (high-low) *
                 (0+randint::draw()/MAXINT_AS_FLOAT)); }
};


class erand : public randint
// exponential distribution random number generator
{
public:
        int   mean;
              erand(int m) { mean=m; };
        int   draw();
};
```

Implementation Details

The following sections contain many implementation-dependent details. The implementation described is the UNIX version. Implementation-dependent information is unfortunately often necessary to allow tuning and ease debugging.

Task Stack Allocation

The two arguments **mode** and **stacksize** allow the user to guide the system's handling of the task. Their exact interpretation is implementation dependent. Users who are not interested in implementation details and/or want a more portable program should set them both to zero. The system will then choose (hopefully reasonable) implementation-dependent default values.

The **stacksize** argument indicates the maximum amount of stack storage that the task is allowed to use. Using more is an error. It will be expressed in a unit of store (typically machine words) suitable for stack allocation on the host system.

The mode provides additional information. The value **SHARED** indicates that the stack space should be taken from the stack space of the parent task, that is the task which created the new task. Where **SHARED** stacks are used the active part of the stack is copied to a save area when a task is suspended, and copied back when it is resumed. *Since* **SHARED** *stack locations are not dedicated to a single task pointers to local variables should not be passed to other tasks.* The time needed to suspend and resume a task with **SHARED** stack is approximately proportional to the amount of stack space actually used at the time of suspension.

If, on the other hand, the mode is **DEDICATED** then a new and separate stack area is allocated, and no copying of stack space will occur.

Scheduling

Functions of a system class, known as the scheduler, are invoked as the result of any function of class task which causes the suspension of a running task, and may be invoked by any function from the standard classes described here. The scheduler selects the next task to run. When the scheduler finds no more tasks to run, and there are no **interrupt_handlers**, it examines the pointer variable **exit_fct**, and if this is non-zero the scheduler will call the function denoted by it.

Whenever **clock** is advanced the scheduler examines the pointer variable **clock_task**. If this denotes a task, then that task will be resumed before any other task. The **clock_task** must be **IDLE** when resumed by the scheduler. The class **task** function **sleep()** is useful to ensure this.

Debugging and Tuning Aids

The task system has been designed under the assumption that a typical use of tasks may involve hundreds of tasks and need tuning to achieve an acceptable time-space tradeoff. The task of debugging such a system can safely be assumed to be non-trivial.

Classes were used in the implementation of the task system largely because they allow the scope of data and functions to be explicitly restricted to the object to which they belong. This allows better type checking of a multi-threaded program than could be achieved by a function-based implementation. The classes which constitute the task system were designed to allow quite strong type checking of programs using them.

A number of run time errors are detected by the task system. For example it is illegal to **delete** a queue on which a task is waiting. When such a run time error is detected the task system function **object::task_error** is called with the number of the error and the **this** pointer of the object which caused the error as arguments. A list of run time errors appears under "Run-Time Errors." Task_error() will in turn examine the pointer **error_fct**, and if this is non-zero call the function denoted by it with a copy of its own arguments. Otherwise **task_error()** will call the system function **exit()** with the error number as argument.

When returning from **task_error()** after executing an **error_fct** which returned rather than using **exit()** the task system will re-try the operation which caused the error (provided that **error_fct** could have affected the condition which caused the error). For example, a **put()** to a **qhead** will be re-tried because the user's **error_fct** might have either caused the **get()** function to be used on the queue, or used **chmax()** to allow more objects to be inserted into that queue.

> This error handling mechanism is primarily designed for debugging and it is expected that user error functions will print some appropriate error message and exit.

Beware of infinite loops.

All task system classes have a function **print()** which can be used to print the contents of their objects on **stdout**. A **print()** function takes an **int** argument indicating the amount of information to be printed. **Print(0)** gives the minimum amount of information, **print(VERBOSE)** rather more, and **print(CHAIN)** will call **print()** for objects on lists associated with the object with its own arguments. The **print()** argument constants can be combined by the **or** operator. For example

```
thistask->print(VERBOSE);
run_chain->print(VERBOSE|CHAIN);
```

will verbosely describe every non-**TERMINATED** timer and every **RUNNING** task. For tasks information about the run time stack is printed by **print(STACK)**. If the variable **_hwm** is set to a non-zero value, **print(STACK)** will also give an estimate of the maximum amount of stack space ever used by the task, the stack's "high water mark." For tasks that share a stack, the high water mark printed will be the high water mark of the most greedy task. For example, information describing stack usage for all tasks can be printed by:

```
task_chain->print(STACK|CHAIN);
```

The output of the **print()** functions is implementation-dependent and hopefully self-explanatory.

Overheads and Performance

The store used for representing a class object in addition to the user specified data is:

object	3 words
timer	5 words
task	18 words + stacksize
queue	15 words (including the qhead and the qtail)

The times needed to execute some of the task system functions are (very) approximately:

C procedure call + return	1 unit
task suspend + resume	9 units (using result())
put	2 units
get	2 units
wait, waitvec, or waitlist	3 units

The last four actions can all cause a task to be suspended. When this happens add 6 units of time.

> For timing results relative to UNIX process switching, see "Extending the C++ Task System for Real-Time Control."

The task system uses about 15K bytes of store for program and data, but much of this is redundant virtual function tables that will be eliminated in a future version of the C++ compiler.

The object Class

The task system as described above is implemented using a lower level of abstraction based on the direct use of the class **object**. Class **object** can also be used as a base for other (user defined) abstractions, but beware, it is an implementation tool that is not intended to be used directly.

Class **object** is a base class for all classes in the task system and also the most basic facility for inter-task communication. The declaration of class **object** looks like this:

```
class object
{
friend sched;
friend task;
    olink*        o_link;
public:
    object*       o_next;
    virtual int   o_type();
                  object() { o_link=0; o_next=0; }
                  ~object();
    void          remember(task* t) { o_link = new olink(t,o_link); }
    void          forget(task*);     // remove all occurrences of task from chain
    void          alert();           // prepare IDLE tasks for scheduling
    virtual int   pending();         // TRUE if this object should be waited for
    virtual void  print(int, int =0);// first arg VERBOSE, CHAIN, or STACK
};
```

The task system implements objects of type TASK, QHEAD, QTAIL, and TIMER.

> Virtual functions make it unnecessary to ever test the type of an object. The virtual function **o_type()** is never called.

A task can be added to the set of tasks "remembered" by an object by executing **object::remember()** and a task can be removed from this set by executing **object::forget()**. Executing **object::alert()** has the effect of transferring all IDLE tasks remembered by the object to the **run_chain** and the RUNNING state.

> The virtual function **object::pending()** provides the "glue" that allows new kinds of objects and new communication protocols to be added to the task system. The object may have any kind of operation that may cause the invoking task to wait, but it must implement its own version of **pending()** to tell whether the operation would cause a wait.

A task can be "remembered" by several objects or several times by the same object without any ill effects. **Forget()** will insure that its argument is not "remembered" any more, and it causes no bad effects when used for an object that does not "remember" its argument task. No record is kept of how many **alert()** operations have been executed on an object. **Alert()** does not cause an object to **forget()** tasks. Executing a **remember()** does not suspend a task. Applying **alert()** to an object that does not remember any tasks is legal, but has no effect. Caveat emptor!

The functions **object::remember()**, **object::forget()**, **object::pending()**, and **object::alert()** provide a simple, efficient, but unstructured and therefore error-prone communication mechanism.

The declarations for the task system classes can be found in **/usr/include/CC/task.h** on systems where it is implemented.

Run Time Errors

When an error is detected at run time, **task_error()** is called. This function will examine **error_fct** and if this variable denotes a function, that function will be called with the error number and **this** as arguments, otherwise the error number will be given as an argument to **print_error()** which will print an error message on **stderr** and terminate the program.

E_OLINK	Attempt to delete an object which remembers a task.
E_ONEXT	Attempt to delete an object which is still on some chain.
E_GETEMPTY	Attempt to get from an empty queue in E_MODE.
E_PUTOBJ	Attempt to put an object already on some queue.
E_PUTFULL	Attempt to put to a full queue in E_MODE.
E_BACKOBJ	Attempt to putback an object already on some queue.
E_BACKFULL	Attempt to putback to a full queue in E_MODE.
E_SETCLOCK	Clock was non-zero when setclock() was called.
E_CLOCKIDLE	The clock_task was not IDLE when the clock was advanced.
E_RESTERM	Attempt to resume a TERMINATED task.
E_RESRUN	Attempt to resume a RUNNING task.
E_NEGTIME	Negative argument to delay().
E_RESOBJ	Attempt to resume task or timer already on some queue.
E_HISTO	Bad arguments for histogram constructor.
E_STACK	Task run time stack overflow.
E_STORE	No more free store — new() failed.
E_TASKMODE	Illegal mode argument for task constructor.
E_TASKDEL	Attempt to delete a non-TERMINATED task.
E_TASKPRE	Attempt to preempt a non-RUNNING task.
E_TIMERDEL	Attempt to delete a non-TERMINATED timer.
E_SCHTIME	Scheduler run chain is corrupted: bad time.
E_SCHOBJ	Sched object used directly instead of as a base class.
E_QDEL	Attempt to delete a non-empty queue.
E_RESULT	A task attempted to obtain its own result().
E_WAIT	A task attempted to wait() for itself to TERMINATE.
E_FUNCS	Internal error — cannot determine the call frame layout.

E_FRAMES	Internal error — cannot determine frame size.
E_REGMASK	Internal error — unexpected register mask.
E_FUDGE_SIZE	Internal error — fudged frame too big.
E_NO_HNDLR	No handler for the generated signal.
E_BADSIG	Attempt to use a signal number that is out of range.
E_LOSTHNDLR	Signal handler not on chain.

A Program using Tasks

```
#include <task.h>

/* trivial test example:
        make a set of tasks which pass an object round between themselves
        use printf to indicate progress
        WARNING: this program sets up an infinite loop
*/

class pc : task
{
                pc(char*, qtail*, qhead *);
};

pc::pc(char* n, qtail* t, qhead * h) : (n,0,0)
{
        printf("new pc(%s,%d,%d)\n",n,t,h);

        while (1) {
                object*   p = h->get();
                printf("task %s\n",n);
                t->put(p);
        }
}

main()
{
        qhead*   hh = new qhead;
        qtail*   t = hh->tail();
        qhead*   h;
        short    i;

        printf("main\n");

        for (i=0; i<20; i++) {
                char*   n = new char[2]; /* make a one letter task name */
                n[0] = 'a'+i;
                n[1] = 0;

                h = new qhead;
                new pc(n,t,h);
                printf("main()'s loop\n");
                t = h->tail();
        }
```

```
        new pc("first pc",t,hh);
        printf("main: here we go\n");
        t->put(new object);
        printf("main: exit\n");
        thistask->resultis(0);
}
```

Extending the C++ Task System for Real-Time Control

This section is taken from a paper by Jonathan E. Shopiro.

Abstract

The task system for coroutine programming was one of the first libraries written in C++ and it has served admirably in several applications. It is small, efficient, and easy to use. As part of a robot control project, it was extended to support real-time control. The new task library is more robust, more easily extendible, and more portable than the original. It is upward compatible, so that programs written for the old task library can still be used. This section documents the new features and the internal structure of the revised system, and is intended for users of the task library and for authors of other coroutine systems.

Overview

The C++ task library is a coroutine[12] support system for C++. A task is an object with an associated coroutine. The task library includes a scheduler that enables each task to execute just when it has work to do, and to wait when necessary for whatever is needed.

Programming with tasks is particularly appropriate for simulations, real-time process control, and other applications which are naturally represented as sets of concurrent activities. A task can represent a simple part of a complex system, and when the task gains control, it can process its current input data, perhaps creating other data that will be processed by other tasks. It can then relinquish control, waiting for more input or an external event.

In a program using the C++ task system, all tasks share the same address space so that pointers can be passed between tasks, and it is easy to share common data structures. Also, the scheduler is non-preemptive, so that each task runs until it explicitly gives up the single processor, and only then does the scheduler choose a new task to run. This eliminates the need for locks on shared data (which would be required if preemptive scheduling or multiple processors were used) and allows task-switching to be accomplished with low overhead, but requires the programmer to be careful that no task monopolizes the processor.

The rest of this section is an overview of control flow in the task system along with a brief note on task system performance. The section "Real-Time Extensions" describes the interrupt handler class and how it can be used to provide real-time response to external events. Familiarity with C++ is assumed.

The Structure of the Task System

Control in the task system is based on a concept of operations which may succeed immediately or be blocked, and *objects*[13] which may be *ready* or *pending* (not ready). When a task executes a blocking operation on an object that is ready, the operation succeeds immediately and the task continues running, but if the object is pending, the task waits. Control then returns to the scheduler, which chooses the next task from the *run chain*, a list that contains all the tasks that are ready to run (not waiting or terminated). For example, a queue head is ready when the associated queue has data, and **get** (which extracts an item from the queue) is a blocking operation for queue heads. Similarly, **put** is a blocking

operation for queue tails, which are ready unless the associated queue is full.

Each different kind of object can have its own way of determining whether it is ready or not, which makes it easy to add new capabilities to the system. On the other hand, each kind of object can have only one criterion for readiness (although it may have several blocking operations), so it is not possible for one object to act as both a queue head and a queue tail, for example.

Each object contains a list (the *remember chain*) of the tasks that are waiting for it. When any operation changes the state of a pending object so that it becomes ready, those tasks are moved to the run chain; this is called an *alert*. Thus the cycle is: a task runs until it blocks; it is saved on the remember chain of one or more pending objects; some other task or an interrupt alerts the object; the original task is moved to the run chain; eventually the task runs again.

Task System Performance

The fundamental operations of the task system are task creation and task switching. In order to make a meaningful evaluation of their performance, equivalent programs using tasks and UNIX Operating System processes were written. These programs are given under "Example Programs." Each of the first pair of programs (**tcreate.c** and **ucreate.c**) repeatedly creates new trivial tasks (processes) and waits for them to terminate. Each of the second pair of programs (**tswitch.c** and **uswitch.c**) creates a single child task (process) and repeatedly exchanges control with it through a pair of semaphores (see under "Semaphores") in the task version, and through UNIX signals in the process version. The programs were run on a SUN 3/280 under 4.2BSD, using the free store allocator (**malloc.c**) from Ninth Edition UNIX, which is much faster than the one supplied with 4.2BSD. The results were that **tcreate.c** was 37 times faster than **ucreate.c**, and **tswitch.c** was 10 times faster than **uswitch.c**.

It is important to note that the task system and the UNIX Operating System are not equivalent and that the results of these performance measurements do not imply that the task system is 23.5 times better than UNIX. Among the significant differences between tasks and processes are the following.

- A set of tasks runs as a single UNIX process. The task system relies on the UNIX Operating System for I/O, memory management, etc.
- Tasks share an address space, while processes have separate address spaces. This means that tasks can share data by simply passing pointers, while processes must go through one of several much more complex and expensive procedures to share data. By the same token, tasks can interfere with each other as easily as they can cooperate, while errant processes usually kill only themselves.
- The task system can support two or three orders of magnitude more concurrent tasks (especially with the **SHARED** option; see "Task Switching") than the UNIX Operating System can support processes. It is not uncommon for a simulation to require thousands of tasks.

The memory required for the task system is about 14,000 bytes for code and data, plus about 70 bytes per task, plus stack storage for each task. By default each task has its own stack buffer with a default size of 3000 bytes, but tasks can share a stack buffer and then storage is required only for the active stack of each task (typically 50 to 100 bytes). This option is very useful for applications with thousands of tasks. Queues occupy 60 bytes (including both head and tail) plus the size of whatever is stored on the queue. Lists of tasks are maintained in various places, for example the run chain and remember chains; each occurrence of a task on a list adds 8 bytes to the total memory requirement.

Real-Time Extensions

The application that motivated this work on the task system was a control system for two robots operating in the same workspace. The most important requirement of this application that was not fulfilled by the original task system was the need for tasks to wait for external events. For example, after a motion command was sent to a robot, the task that sent the command needed to wait for the interrupt that was generated by the robot hardware when the command was complete or had failed. A related requirement of some real time systems is to respond to external events in a timely manner, for example to retrieve data from an unbuffered external device. Also, in the original task system, the scheduler would exit when the run chain was empty. This is inappropriate in a system that is intended to respond to external events because some task might become runnable after an interrupt.

Hardware interrupts are handled differently by different machines and operating systems, so the interface to the task system must also vary. For didactic reasons, the version described here is for the UNIX Operating System using signals as interrupts, but it should be clear how to adapt it to other environments.

In the task system events that can be waited for are represented by instances of class **object** or derived classes. When the function **object::alert()** is called, the tasks that were waiting for that object are made runnable. A natural solution to the problem of waiting for external events was to define a new kind of object to represent external events, and when such an event occurs, to call **object::alert()** for the appropriate object. These objects are called interrupt handlers.

```
class Interrupt_handler : public object {
        int id;                    // signal or interrupt number
        int got_interrupt;         // an interrupt has been received but not alerted
        Interrupt_handler *old;    // previous handler for this signal
        virtual void interrupt() {}// runs at real time
public:
        int pending();             // FALSE once after interrupt
        Interrupt_handler(int sig_num);
        ~Interrupt_handler();
};
```

After an interrupt handler is created, a task can wait for it, exactly as for any other object. When the interrupt occurs, the handler's **interrupt()** function will be executed immediately, or rather, as soon as the operating system can route the interrupt to the process. When the interrupt function returns, control will resume at the point where the current task was interrupted.

At the next entry to the scheduler, when the currently running task blocks, a special task, the *interrupt alerter*, will be scheduled. This task alerts the handler (and any other handlers that have received interrupts since it was last scheduled). Thus the waiting task becomes runnable. As long as any interrupt handler exists, the scheduler will wait for an interrupt, rather than exiting when the run chain is empty. The **pending** function for an interrupt handler always returns *TRUE* except the first time it is called after an interrupt occurs.

Interrupt_handler::interrupt() is a null function, but since it is virtual, the programmer can specify the action to be taken at interrupt time by simply defining an **interrupt()** function in a class derived from **Interrupt_handler**. An example is given under "Interrupts." In this way real-time response can be obtained without resorting to a preemptive, priority-based scheduler which would be more complex and less efficient, and would require locking of shared data structures.

Avoiding Interference

Whenever shared data structures are manipulated by concurrent processes, there is the potential for interference, where one process is in the middle of modifying a data structure and another process accesses it and finds it in an invalid state. Segments of code that access shared data structures are called *critical regions*.[14] If more than one process can be in a critical region at one time, there is a potential for interference.

Interference is easy to avoid in the task system, because of the non-preemptive nature of the scheduler. There are only two ways in which interference can arise: a task switch occurring within a critical region, or an interrupt routine that accesses shared data.

It is almost always possible to write code so that no operation that could cause a task to block is inside a critical region. The style of programming where coroutines share information by sending messages to each other in the form of objects on queues typically leads to programs where there are no shared data structures or critical regions at all. Even if coroutines must share access to a data structure and alternately modify it, no problems will arise if the routines that do the modification refrain from operations that could cause the task to block. A properly modular program will generally satisfy this requirement without any extra effort.

Semaphores

If, for some unusual reason, it is necessary to put an operation that could cause the task to block in a critical region, then the affected data structure should be protected by a semaphore, which will allow only one task at a time to access the object. The following example code outlines this technique.

```
class My_data {
        Semaphore sema;
        // user data
public:
        void    lock() { sema.wait(); }
        void    unlock() { sema.signal(); }
                My_data() : sema(1) { ... }  // see note
};
```

Each critical region must begin with a call to **My_data::lock()** for the object to be accessed, and end with a call to **My_data::unlock()**. This will ensure that no interference occurs, even if the operations in the critical region cause the task to block.[15]

The implementation of semaphores using the task system is easy.

```
class Semaphore : public object {
        int     sigs;  // the number of excess signals
public:
                Semaphore(int i =0) { sigs = i; }
        int     pending() { return sigs <= 0; }
        void    wait();
        void    signal() { if (sigs++ == 0) alert(); }
};


void
Semaphore::wait()
{
        for (;;) {
                if (--sigs >= 0)
                return;
                sigs++;
                thistask->sleep(this);
        }
}
```

Semaphores are useful tools for building other kinds of synchronization besides mutual exclusion. For example, whenever one task wants to wait for an operation to be completed by another task, it can wait on a semaphore.

Interrupts

The other case where interference can occur is a little more complex. The **interrupt()** routine of an **Interrupt_handler** can be executed at any time, and it would be contrary to the reason for its existence to lock it out. The mechanism that alerts the handler after the interrupt has occurred is carefully designed to be safe from interference, and sometimes the alert is all that is necessary for an application. If it is necessary to gather data from an external device immediately after an interrupt occurs, but the interrupts do not come in rapid succession (for example, the next interrupt won't occur until after the device is reset), the interrupt routine can save the data and the task that is waiting for the interrupt can process the data before resetting the device. In this case even though the data is shared, the interrupt routine cannot access the data at the same time as the task.

Sometimes, however, it is necessary to handle interrupts that can come in rapid succession, with a requirement to gather data at each interrupt, so that several interrupts may occur before the task that will process the data can be scheduled, and more interrupts may occur even while the task is running. This problem is best handled by establishing a queue of the interrupt data records. Then the only shared data between the interrupt handler and the task processing the data can be the queue head and tail pointers, which can be atomically updated. In the following toy example, the interrupt routine records the value returned by an arbitrary function, **get_data()**, each time the signal **SIGINT** is sent. A waiting task is then scheduled and prints all accumulated data.

```
class Delete_handler : public Interrupt_handler {
        void    interrupt();
        int*    localq;                 // data buffer beginning
        int*    localq_end;             // data buffer end
        int*    localq_h;               // queue head
        int*    localq_t;               // queue tail
public:
        int     getX(int&);             // the next item, if any
                Delete_handler(unsigned local_q_size =5);
                ~Delete_handler() { delete [localq_end - localq] localq; }
};
```

The delete handler (so called because **SIGINT** is normally sent when the user presses the delete key) is an interrupt handler that maintains a local queue of data. Its interrupt function will put data on the local queue, using **localq_t**, the queue tail pointer, and its **getX()** function is used by a task to retrieve the data.

```
Delete_handler::Delete_handler(unsigned local_q_size)
: (SIGINT)          // base class constructor arg
{
        localq_t = localq_h = localq = new int[local_q_size];
        localq_end = &localq[local_q_size];
}
```

The constructor initializes the local queue. The size of the local queue determines how many interrupts can be awaiting processing.

```
void
Delete_handler::interrupt()
{
        register int*   p = localq_t;
        *p = get_data();
        if (++p == localq_end) p = localq;
        if (p != localq_h)
                localq_t = p;           // no overflow
        else error("Overflow");
}
```

The interrupt function assumes that **localq_t** points to an available slot in the queue and puts the real-time data there. It then checks for overflow and updates **localq_t** to point to the next available slot if it's okay or calls an error function otherwise.

```
int
Delete_handler::getX(int& ans)
{
        register int*p = localq_h;
        if (p == localq_t)
                return 0;
        ans = *p;
        if (++p == localq_end) p = localq;
        localq_h = p;
        return 1;
}
```

The function **getX()** assigns the next datum to its argument and returns "1," or returns "0" and leaves its argument alone if no data is available. A call to **getX()** may be interrupted, but it has been designed so that no corruption of the queue will result.

```
class Delete_printer : public task {
        Delete_handler*handler;
public:
        Delete_printer();
};
```

Delete_printer() is a task that will create a **Delete_handler** and print whatever data is received.

```
Delete_printer::Delete_printer()
: handler(new Delete_handler)
{
        for (;;) {
                wait(handler);
                inti;
                while (handler->getX(i))
                cout << i << "\n";
        }
}
```

Note that each time the printer task is scheduled, it prints all the available data from the delete handler.

Implementation Details

The approach taken was to minimize the impact to the scheduler and to isolate as much as possible the machine and operating system dependent parts of the implementation. There is a system-dependent function, **sigFunc()**, which catches each signal for which an **Interrupt_handler** exists. When the signal is sent, **sigFunc()** calls the appropriate **interrupt()** function. It then atomically puts the address of a dedicated alerter task in a static, private cell of the scheduler and rearms the signal and returns. At the next entry to the scheduler, that cell is checked and if it is non-zero, the alerter task is scheduled. The alerter task alerts all pending interrupt handlers and returns to the scheduler. Tasks that were waiting for interrupt handlers are then eligible to run.

The other system-dependent parts of the implementation are the constructor and destructor for class **Interrupt_handler**. Its constructor takes the signal number as argument (it might be an interrupt vector address in another system). If some other interrupt handler already existed for that signal, it is saved (and alerted if it was pending), and otherwise the UNIX system function **signal()** is called to

associate **sigFunc()** with the signal. The destructor undoes the action of the constructor, restoring the previous signal routine if necessary.

Example Programs

tcreate.c

The following program repeatedly creates a task and waits for it to terminate. It would be possible to time creation of new tasks without waiting for them to terminate, but because of the limited number of processes that can exist under the UNIX system, the corresponding UNIX system program would fail.

```
#include "task.h"

class Child : public task                    // user task declaration
{
public:
                Child(int);                  // task constructor declaration
};

Child::Child(int i)                          // user task constructor definition
: ("Child")                                  // argument to base class constructor
{
        resultis(i);                         // terminate task execution
}

main()
{
        for (register int i = 10000; i--; ) {
                Child*  c = new Child(i);    // create a task
                c->result();                 // wait for it to terminate
                delete c;                    // clean up
        }
        thistask->resultis(0);               // exit from main task
}
```

ucreate.c

The following C program repeatedly forks a UNIX process and waits for it to terminate.

```
main()
{
        register int i;
        for (i=10000; i--; )
                if (fork() == 0)
                        exit(0);             // child process
                else
                        wait((int*)0);       // parent process
}
```

tswitch.c

The following program uses two semaphores (described under "Semaphores") to alternate control between a parent and child task.

```
#define K 10000
#include "task.h"

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

Semaphore sema1;                                // for signals from main to Child
Semaphore sema2;                                // for signals from Child to main

Child::Child()
: ("Child")
{
        for (register int n = K / 2; n--; ) {
                sema1.wait();                   // wait for a signal from main
                sema2.signal();                 // send it back
        }
        resultis(0);
}

main()
{
        new Child;
        sema1.signal();                         // send the first signal
        for (register int n = K / 2; n--; ) {
                sema2.wait();                   // wait for a signal from Child
                sema1.signal();                 // send it back
        }
        thistask->resultis(0);
}
```

uswitch.c

The following C program uses a UNIX system signal to force alternation between two UNIX system processes. The program is a little strange in that its main routine consists of an infinite loop of **pause()** calls. Unfortunately the utility of **wait()** and **pause()** for signal handling is limited because it is always possible that a signal has been received just as the **wait()** or **pause()** is being executed.

```
#include <signal.h>
#define K10000
int     otherpid;
int     received;
int     child;
void
sig()                                           /* signal-catching routine.  called */
```

```
                                        /* when a signal is received        */
{
        signal(SIGTERM, sig);           /* arrange to catch the next signal */
        received++;
        if (child && received >= K/2) exit();
        kill(otherpid, SIGTERM);        /* send it back */
        if (!child && received >= K/2) exit();
}

main()
{
        signal(SIGTERM, sig);           /* arrange to catch the signal */
        if ((otherpid = fork()) == 0) {  /* create the child process */
                otherpid = getppid();    /* get parent process id */
                child = 1;               /* this is the child */
                kill(otherpid, SIGTERM); /* send the first signal */
        }
        for(;;)
                pause();
}
```

real_timer.c

In addition to the robot application, the system was implemented on the UNIX Operating System using signals as interrupts. A class **Real_timer**, modelled on the original class **timer** was built.

```
class Real_timer : public object {
        friend class Alarm_handler;
        int     state;              // RUNNING, IDLE, TERMINATED
        long    time;               // initially delay, then alarm time
        void    insert(int);        // put on chain
        void    remove();           // remove from chain & make IDLE
        void    resume();           // called when time is up
public:
                Real_timer(int);
                ~Real_timer();
        int     pending();
        void    reset(int);
        void    print(int, int =0);
};
```

Instead of simulated clock ticks, class **Real_timer** measures time in seconds. It is based on the following handler for the alarm signal and a task that maintains the list of unexpired **Real_timer** instances.

```
class Alarm_handler : public task {
        friend Real_timer;
        Real_timer*         chain;
        Interrupt_handler*  bell;
        void                add_timer(Real_timer*);
        void                remove_timer(Real_timer*);
public:
                Alarm_handler();
```

```
};

Alarm_handler     alarm_handler;              // the only instance

Alarm_handler::Alarm_handler()
: ("Alarm_handler"), chain(0)
{
        sleep();
        for(;;) {
                for (long now = time(0); chain && chain->time <= now;
                                        chain = (Real_timer*)chain->o_next)
                        chain->resume();  // alert the timer
                if (chain) {
                        alarm(chain->time - now);
                        wait(bell);
                } else {
                        bell->forget(thistask);
                        delete bell;
                        sleep();
                }
        }
}
```

The **Interrupt_handler** pointed to by **Alarm_handler::bell** only exists while there are pending **Real_timer** objects. The **Alarm_handler** task runs after an alarm signal, and after alerting any timers that have expired, if there are any unexpired timers, it resets the alarm and waits.

A Porting Guide for the C++ Coroutine Library

NOTE

This section is taken directly from a paper by Stacey Keenan.

Introduction

The C++ coroutine library, commonly known as the task library after its header file, **task.h**, provides multiple threads of control within one UNIX system process. Each thread of control is a coroutine, or task, and each task runs until it explicitly gives up the processor; there is no pre-emption. Implementing concurrency requires knowledge of hardware-dependent and compiler-dependent runtime features, especially calling sequence and stack frame layout; hence the library is target-dependent and must be ported explicitly to each supported compiler/processor platform.[16] The target-dependent parts of the library are isolated in four files. Release 2.0 of the C++ Language System supplies the task library for the AT&T 3B20, AT&T WE32000 family (e.g., 3B2, 3B15), AT&T 6386 WGS and DEC VAX processors, and the Sun-2 and Sun-3 Workstations (Sun compilers on Motorola 68000 family processors).

This paper describes the implementation of the task library, with particular emphasis on task creation and task switching, where target-dependent code is needed. The existing implementations for the 3B, VAX, and Sun Workstation processors are used as examples.[17] The scope of this paper is limited by the similarity of the runtime models supported by these targets. Targets diverging from these models, like mainframe or RISC-style processors, are likely to present porting difficulties not addressed in this paper. It is assumed that the reader has access to the source code for the library. This paper does not describe how to use the task library; see "A Set of C++ Classes for Co-routine Style Programming" and "Extending the C++ Task System for Real-Time Control" for user-level information. "Task Switching Fundamentals" provides background needed to understand the workings of the task library. "Implementation of Task Switching" describes how the task library creates new tasks and switches among them, including details about the target-dependent functions **swap()** and **fudge_return()**. The final sections discuss source file organization and miscellaneous hints for porting the library.

Task Switching Fundamentals

The C++ task library provides non-preemptive scheduling for tasks. A task runs until it explicitly gives up the processor to allow another task to run. Typically, a task will give up the processor when it tries to perform an action that cannot yet be done, for example, if it tries to put an object on a full queue, or to get an object from an empty queue. When this happens, the task is put to sleep.[18] The scheduler then chooses to run the next task on the ready-to-run list, **sched::runchain**.

When a task is put to sleep, or suspended, the task system must save the state of the task so that it may be resumed later. On the targets described here, this involves saving the task's stack and hardware registers, including the non-volatile registers and the frame pointer (and the argument pointer on some targets). A task switch is the process of saving the state of one task, and restoring the state of another.

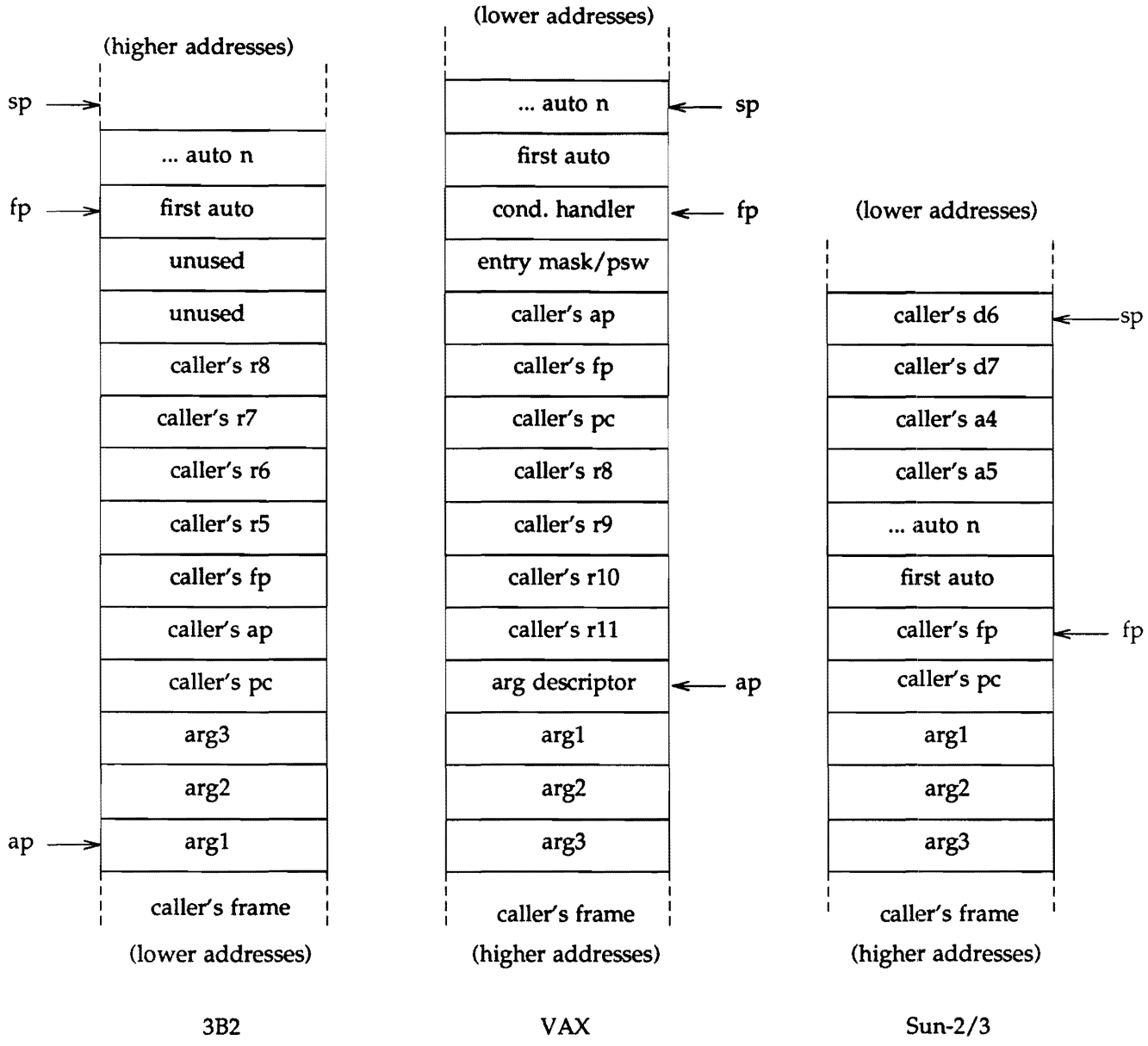
Stack Frames

Some familiarity with the C runtime environment and the target implementation of stacks is needed to understand the details of task creation and switching. A C function call sets up a new stack frame for the function. A stack frame contains the arguments to the function, the saved hardware state of the calling function, and any automatic variables used by the function. Figure 2-1 illustrates the stack frames built on the 3B2, the VAX, and the Sun-2/3 targets for a function called with three arguments and saving four registers. These stack frames are described here to provide a base for later discussions on the internals of the task library.

On a 3B2, the argument pointer (ap) points to the start of the arguments to the function, the frame pointer (fp) points to the start of the automatics of the function, and the stack pointer (sp) points to the next available space in the stack. The caller's registers are saved between the arguments and the automatics. Previous stack frames can be accessed via the frame pointer: The old frame pointer, argument pointer, and program counter (pc) are always a fixed distance below the frame pointer. Stacks grow up, toward higher memory addresses.

On a VAX, stacks grow down, toward lower memory, although the figures in this paper will show the low memory on top and relative positions on the stack will be described in terms of the pictures (e.g., above means higher in the picture, at a lower memory address). The argument pointer points to a longword containing the number of arguments that have been pushed on the stack. Arguments are pushed in reverse order, so that the first argument is stored one word below the ap. The frame pointer points to a condition handler, above which are the automatics of the function. The stack pointer points to the last assigned word in the stack. The word just under the frame pointer contains a procedure entry mask, which tells which registers were saved in the frame. Saved user registers and the old frame pointer, argument pointer, and program counter are stored between the argument and frame pointers.

Figure 2-1: Stack Frames on a 3B2, a VAX, and a Sun-2/3 for a Function Taking 3 Arguments and Saving 4 Registers



The stack on the Sun-2/3 Workstation grows down, toward lower memory. This target has no argument pointer. Arguments, saved registers, and automatics are all referenced as offsets from the frame pointer. Arguments are pushed on the stack in reverse order, followed by the return pc and the old frame pointer. The frame pointer points at the old frame pointer. Space for automatics is reserved above the frame pointer. Saved registers are pushed after the reserved space, and the stack pointer points to the last saved register. The 68000 processor has both data (dx) and address (ax) registers. In

this example, two of each type are saved.

On entry, a function first saves all the registers that it might use.[19] On function exit, the same number of registers are restored from the register save area of the stack frame. On some targets, like the VAX, stack frames are self-describing: one can tell how many registers are saved in the frame (and where they are) from the frame itself (by looking at the entry mask). Thus, the function return sequence on a VAX consists of a single, simple instruction: **ret**. The 3B and Sun-2/3 targets do not have self-describing stack frames. This means that "return" instructions on these targets need to specify how many registers to restore. When (as happens in the task system) one needs to restore registers without returning through the normal return sequence, one can only find out how many registers were saved on the stack by looking at the save instruction at the beginning of the function.

To switch to a new task, the task system needs to know what the new frame pointer (and argument pointer on the 3B targets) should be and from where to restore all the non-volatile registers.[20] The task library explicitly saves the frame pointer and argument pointer of the function to be returned to, swap(), in the task object as **t_framep** and **t_ap**. The non-volatile registers are stored in **swap**'s stack frame.

DEDICATED and SHARED Tasks

Tasks can be of one of two modes: DEDICATED or SHARED. DEDICATED tasks each have their own stack, of some fixed size, allocated from the free store. SHARED tasks share a single stack, of some fixed size. When a SHARED task is about to resume execution, if its stack space is occupied by another task,[21] the portion of the stack that is in use by the other (suspended) task is copied out to a save area, and the resuming task's stack is copied from its save area back into the stack. Because the in-use part of the stack is less than the allocated size of the stack, the user can save space by using SHARED stacks, at a cost in execution speed. Additionally, some targets and operating systems do not allow the stack pointer to point into the UNIX process data segment; on these systems SHARED tasks must be used.[22]

Implementation of Task Switching

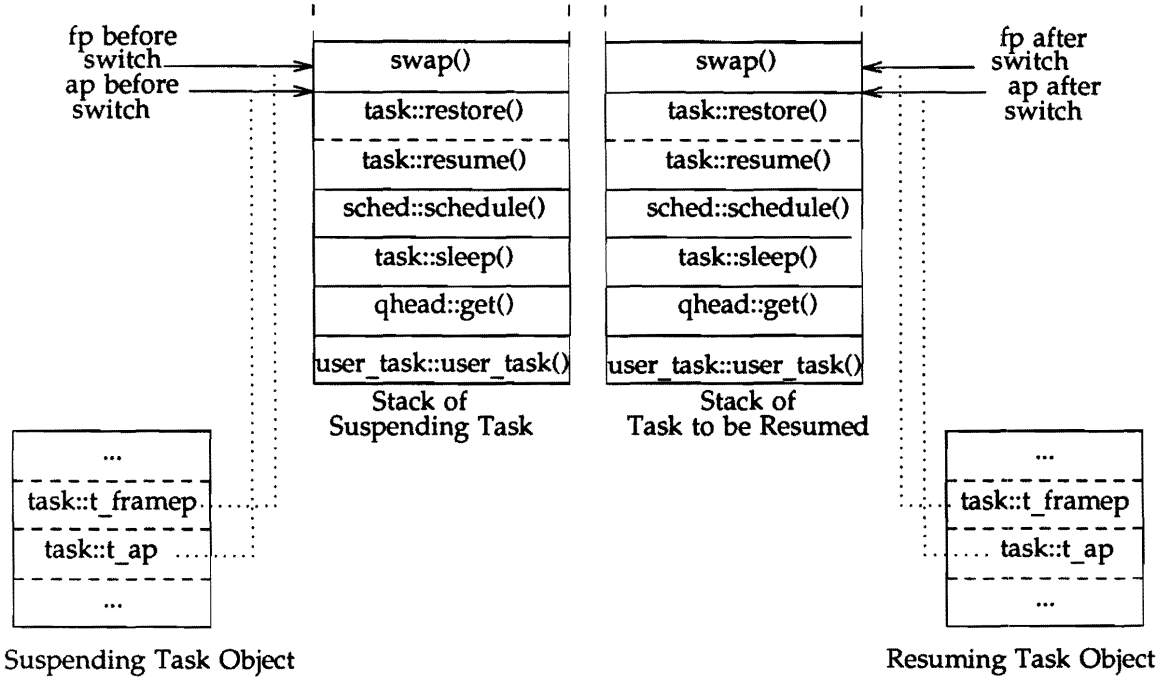
There are two general contexts in which a task switch occurs: when a parent task creates a new child task and switches to it, and when a task suspends and the scheduler chooses a new task to run. The stacks of both the suspending and resuming tasks look different in each of these situations. Task creation differs from a switch to a suspended task in two ways. First, in task creation a runtime environment for the new task must be set up before the switch can take place. Second, task creation causes the parent task to be suspended and the new task to run immediately, bypassing any other tasks waiting on the run chain. This is the only case where a task switch takes place without a call to the scheduler to choose the next task to run. These two contexts are described below.

Task Switches Between Suspending and Resuming Tasks

In task switches from a suspending to a resuming task (i.e., switches other than those to newly created tasks), the function that causes the running task to block (**qhead::get()** in Figure 2-2) calls **task::sleep()**, which in turn calls the scheduler, **sched::schedule()**. After selecting the next task to run, the scheduler calls **task::resume()**[23] for the resuming task. The function **task::resume()** calls **task::restore()**, an inline function whose purpose is to call the appropriate version of **swap()** (**swap()** for DEDICATED tasks, sswap() for SHARED tasks) with the appropriate arguments.

Figure 2-2 shows examples of the stacks for a suspending and a resuming task, both of type **user_task** (**user_task::user_task()** is the constructor and "main" function of the task). Each box in the stack represents a stack frame; the frames for **task::resume()** and **task::restore()** are separated by a dashed line because **task::restore()** is an inline function, and therefore doesn't really have its own stack frame.

Figure 2-2: A Task Switch from a Suspending to a Resuming Task (DEDICATED)



Switching Between DEDICATED Tasks: swap()

The two **swap** functions do the real work of performing a task switch. They are written in assembly language because they manipulate hardware registers. The **swap()** function saves the state of the suspending task (labeled **running** in the code)[24] and restores the state of the resuming task (labeled **to-run**). Saving the state of the suspending task involves first saving all the non-volatile registers in **swap**'s stack frame, then saving the current frame pointer, which defines **swap**'s frame, and the argument pointer, if necessary, in the suspending task's task object, in members **t_framep** and **t_ap**. Then **swap()** overwrites the hardware frame pointer and argument pointer with the values saved in the resuming task's **t_framep** and **t_ap**. Now the to-run task is running; **swap()** returns, restoring all the registers that were saved when the to-run task was suspended. Note that **swap's** save is done on the suspending task's stack, and the restore is done on the resuming task's stack. This is because save and restore instructions are executed relative to the frame pointer, which was modified in the middle of **swap()**. Figure 2-2 illustrates a task switch on a 3B target. The **swap()** hardware frame and argument pointers are shown both before and after the switch.

Switching Between SHARED Tasks: sswap()

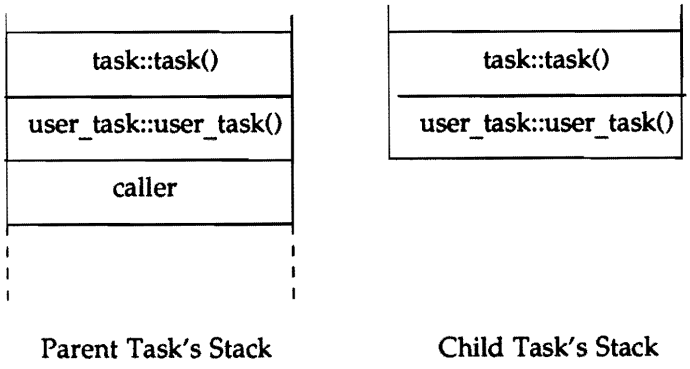
The function **sswap()** is like **swap()**, but has additional code for SHARED tasks to copy task stacks out of and into the shared stack area.[25] There are three tasks that are relevant during a SHARED task switch: the suspending task, the resuming task, and the task that last occupied the stack space that the resuming task now wants to occupy (the target stack). This "prevOnStack" task is often the same as the suspending task, but that is not necessarily the case.[26]

The **sswap()** function first saves all the non-volatile registers in its stack frame, then saves the frame pointer (and argument pointer, if necessary) of the suspending task in that task's task object, just as **swap()** does. It also calculates and saves the height of the stack in the **t_size** member of the task object. Next, it allocates space and copies the contents of the target stack to that space, which becomes "prevOnStack's" save area (pointed to by task member **t_savearea**). Next, **sswap()** copies the resuming task's saved stack back from its **t_savearea** to the target stack, and deletes the space. Finally, sswap() restores the resuming task's **t_framep** (and **t_ap**, if necessary) to be the hardware frame and argument pointers, and the resuming task is running. As in **swap()**, **sswap()** returns, restoring all the registers saved in the resuming task's **sswap** frame.

New Task Creation

To use the task library, the user derives a class, which I will refer to as class **user_task**, from the base class **task**. The "main" program for the user task will be the constructor **user_task::user_task()**. The first thing **user_task::user_task()** does is to call the base class constructor, **task::task()**. The constructor **task::task()** initializes the private data for the new task, acquires stack space[27] in which the task will run, initializes the stack with the top two frames of the parent task's stack (as illustrated in Figure 2-3), inserts the parent task on the run chain, and switches to the new task, which runs immediately.

Figure 2-3: Creating a New Task's Stack



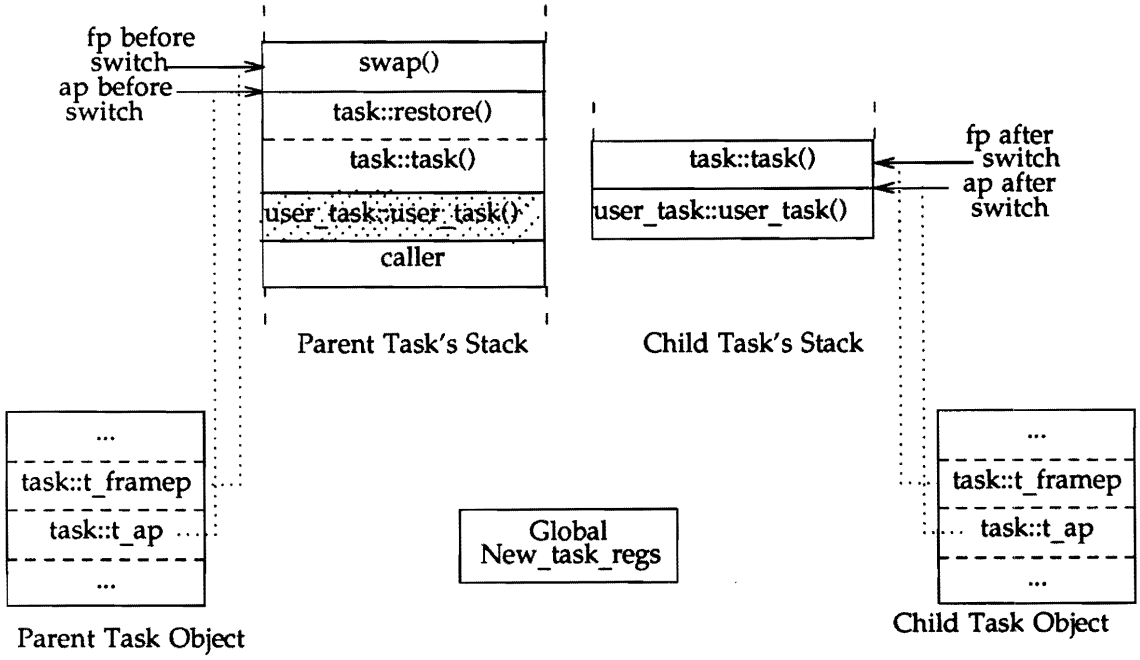
After initializing the new task's stack, the parent task continues execution in **task::task()**. Notice that the parent's stack contains a frame for **user_task::user_task()**, the child's "main"; the parent task needs to skip over that frame when it returns from **task::task()**. To arrange this, **task::task()** calls a function, **task::fudge_return()**, to alter **task::task**'s stack frame so that it returns to **user_task::user_task**'s caller (restoring any registers saved in the skipped frame as well). This change to the parent's stack is shown in Figure 2-4 with dotted lines through the **user_task::user_task()** frame. The **fudge_return** function will be described in detail under "Fudging the Parent Stack."

swap() for Children

When a new task is created, its stack does not have an instance of **swap()** on it; **task::task()** is the top frame. It is **task::task**'s responsibility to arrange for the hardware state of **user_task::user_task()** to be restored when the child begins execution there. Therefore, **task::task()** saves the frame and argument pointers for the child's **task::task()** frame in the child's **t_framep** and **t_ap** of its task object. Then **task::task()** saves all the registers as they were when **user_task::user_task()** called **task::task()** in a global variable, **New_task_regs**.[28] Getting these registers right, no matter how many registers were saved in **user_task::user_task** or **task::task()**, is a bit tricky. We first copy all the current hardware registers into **New_task_regs** and then overwrite any of those that are used by **task::task()** with those saved in **task::task**'s frame. This is done with a macro, **SAVE_CHILD_REGS**, which calls **SAVE_REGS()** to do the first step, and **save_saved_regs()** to do the second step.

Then the parent calls **task::restore**, which calls **swap()** with a NEW_CHILD argument. Given this argument, **swap()** explicitly restores the registers that were saved in **New_task_regs**, instead of restoring the registers saved in the frame. See Figure 2-4. When **swap()** returns, the return is effectively from **task::task()**, as that is where the frame pointer points, and then the child task is executing in **user_task::user_task()**. On the 3Bs, the assembly language return instruction specifies how many registers to restore. Because the necessary registers have been restored from **New_task_regs**, **swap()** restores no registers saved in **task::task()**'s frame on its return. The VAX return instruction determines the number of registers saved in the frame by looking at the entry mask under the frame pointer, therefore, when **swap()** returns, the registers saved in **task::task**'s frame are restored. Since these registers are the same as those saved by **save_saved_regs()**, **save_saved_regs()** is unnecessary on the VAX.

Figure 2-4: A Task Switch to a New Child (DEDICATED)



sswap() for Children

New SHARED tasks don't need to copy in a new stack, nor do they need to reset the hardware frame and argument pointers. Their stacks are already in place, since a new SHARED task runs in its parent's stack. However, the parent task needs to call **sswap()** to save its state and to copy its active stack to its save area. Therefore, **task::restore()** and **sswap()** are called with a **NEW_CHILD** argument, and sswap() has a branch for new children to skip the "copy in" part.

Fudging the Parent Stack

As mentioned above, **fudge_return** is called by **task::task()** to modify the parent stack so that the parent does not return to **user_task::user_task()**. Rather, the parent skips the **user_task::user_task()** frame and returns to **user_task::user_task**'s caller (**main()** in Figure 2-4). This routine is highly machine- and compiler-dependent. It depends on call/return and save/restore conventions of both the compiler and the machine. The left side of Figure 2-5 shows a hypothetical example of a parent stack when **fudge_return()** is first called. Portions of three stack frames are shown:

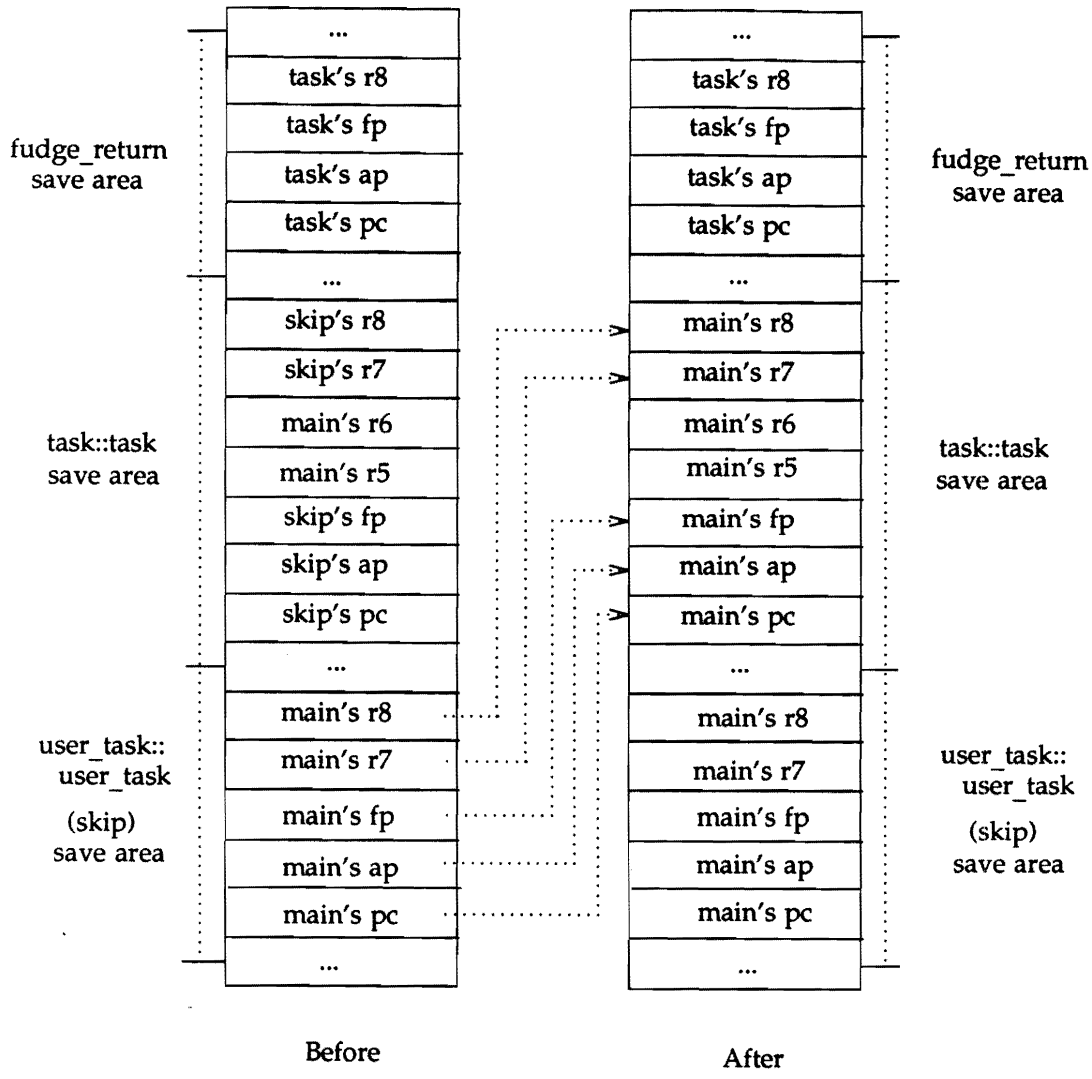
- at the bottom is the register save area for **user_task::user_task()**, containing the saved state of **main()** (i.e., "main's r8" refers to the value of hardware register r8 in **main()** before **user_task::user_task()** was called). In this example, **user_task::user_task()** uses, and therefore saves, two registers, which on a 3B2 would be registers r7 and r8.
- in the middle is the save area for **task::task()**, containing the saved state of **user_task::user_task()** or **skip()**, as it is labeled in the diagram and in the **fudge_return()** code. In this example, **task::task()** uses and saves four registers, r5 through r8.[29]
- at the top is the save area for **fudge_return()**, containing the saved state of **task::task()**. In this example, **fudge_return()** uses and saves just one register, r8.

The ellipses in the diagram represent function arguments, automatics, and unused words in the stack frames. The **fudge_return()** function must copy up the relevant elements from **skip**'s stack frame to task::task's stack frame, so that when **task::task**'s return instruction is executed, the parent will find itself back in **main()** (in this example), with the hardware registers restored to the values they had before **skip()** was called. The stack on the right side of Figure 2-5 represents the same parent stack after **fudge_return** has altered the stack. The dotted arrows show where the elements from **skip**'s save area have been copied.

In the 3B, VAX, and Sun-2/3 implementations, **fudge_return()** overwrites the program counter, frame pointer, and argument pointer (for 3B targets only) saved in **task::task**'s frame with those saved in **skip**'s frame. This causes **task::task()** to return to **main()**.

Restoring **main()**'s registers is trickier. It requires knowing the layout of the save area for at least skip() and **task::task()**, and sometimes for **fudge_return()** as well. Ways of determining the frame layout are discussed under "Finding Where Registers Are Saved: Framelayout()." For now, assume fudge_return() knows how many registers are saved in each frame.

Figure 2-5: A 3B2 Stack Before and After Fudging

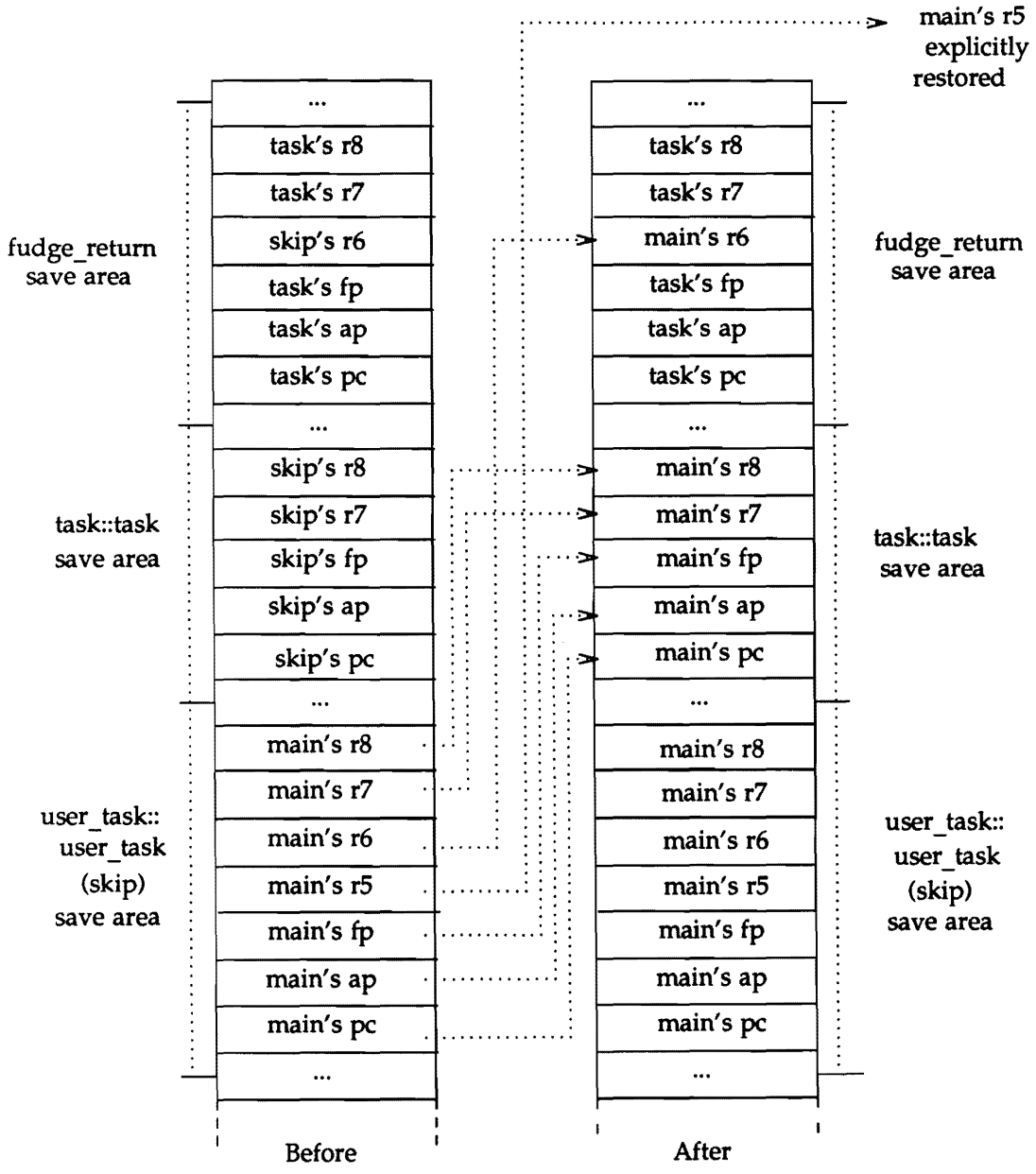


If skip() saved any registers, we must take pains to see that they are restored on **task::task**'s return. If, as is the case in the example in Figure 2-5, all the registers saved in **skip**'s frame are also saved in task::task's frame, this is simple. We just copy the saved **skip()** registers over the corresponding task::task() registers, leaving any additional saved **task::task()** registers in place. There is room in task::task's frame for these registers and, in the case of the 3B and Sun-2/3 targets,[30] **task::task**'s restore instruction will restore all the registers we care about.

There are various difficulties with restoring the "extra" registers when **skip()** saves registers that task::task() does not save. On some targets, such as the VAX and Sun-2/3, there is no room in the frame for the additional registers; on other targets, such as the 3Bs, **task::task**'s restore instruction won't restore any extra registers, although the save area is always large enough to hold extras. Figure

2-6 shows a parent stack frame where the **skip()** frame contains four saved registers, the **task::task** frame contains only two saved registers, and the **fudge_return()** frame contains three saved registers. In this example, r5 and r6 are "extra."

Figure 2-6: Fudging When user_task::user_task() Uses More Registers than task::task



If **fudge_return()** saved any of the "extra" registers, then we can overwrite those with the corresponding saved **skip** registers. In Figure 2-6, **skip()** saved r6 (**main's r6**), **task::task()** did not, but **fudge_return()** also saved r6. Therefore, **fudge_return()** will overwrite the r6 in its save area with the r6 from **skip's** save area. When **fudge_return()** returns, r6 will be restored to the value it had when **main()** last executed, which is what we want. Because **task::task()** did not save r6, we know that it will not disturb its value.

Neither **task::task()** nor **fudge_return()** saved the other extra register, r5, in this example. Therefore, to ensure that when **task::task()** returns, r5 has the value it had in **main()**, and not the value it had in skip() (its current value), **fudge_return()** must explicitly set the hardware register r5 to the value saved in **skip's** frame (main's r5). This is safe to do, because none of the intervening functions use r5. The function **fudge_return()** calls an assembly language function to overwrite r5 (or any other extra registers). After **fudge_return()** and **task::task()** return, all the registers will have the values they had when main() last executed on the parent stack.

There is one final step: arranging for the stack pointer to be in the right place after **task::task** returns. This depends on the way the target executes a return. Without some adjustment, the stack pointer will be set one frame too high (at the top of **skip's** frame instead of at the top of **main's** frame).

On the VAX, a return instruction restores the frame and argument pointers from those saved in the stack, pops the saved registers off the stack, and adds the number of arguments that are on the stack (as given in the argument descriptor, see Figure 2-1) to the stack pointer. We can cause the stack pointer to be restored correctly by adjusting the argument descriptor in **task::task's** frame to include all the words in the **skip** frame in addition to the arguments. In other words, **fudge_return()** alters task::task's frame to look as though there is a big argument list.

On the 3Bs, a return instruction restores the frame and argument pointers from those saved on the stack, but the stack pointer is given the value of the argument pointer of the returning function. This presents a problem for a fudged parent stack: when we return from **task::task()**, the frame and argument pointers are reset to point to **main's** frame, as we wanted, but the new stack pointer points where task::task's argument pointer was, which is higher than needed and wastes space.[31] What we want is to have the stack pointer point to where **skip's** argument pointer was. We arrange for this with an assembly language function, **FUDGE_SP()**,[32] which is defined for the 3Bs to take an argument, the skip() argument pointer, and to reset the current argument pointer (**task::task's**) to the argument. FUDGE_SP() is called just before **task::task()** returns on the parent side. Once **FUDGE_SP()** is called, no arguments to **task::task()** can be referenced. The **task::task()** constructor returns the **this** pointer, which is its implicit first argument. The **this** argument is usually in a register, but if it is not, task::task will need to reference it through the now-changed argument pointer when it sets the return value. Therefore, **FUDGE_SP()** also copies the value of **task::task's** first argument to be user_task::user_task's first argument, to ensure that **task::task's** return value will be set properly.

The Sun-2/3 targets have a similar problem to that described above for the 3B targets. The solution, however, is different. The Sun-2/3 compiler typically generates a function return sequence of three instructions: **movem, unlk,** and **rts.** The **movem** instruction restores the registers denoted by a mask and uses an offset from the frame pointer to find the register save area. The **unlk** instruction resets the frame pointer to be the one saved in the stack, and also resets the stack pointer to point at the saved return program counter on the stack. Finally, the **rts** instruction pops the program counter off the stack, leaving the stack pointer pointing at the top of the frame of the function that called the returning function. As with the 3B targets, after a parent task (whose stack has been fudged) returns from task::task() to **main()** (in the example), the stack pointer points to the top of the skipped frame.

We compensate for this with a variation in **FUDGE_SP()** and **fudge_return()** on the Sun-2/3 targets.[33] Instead of overwriting **task::task**'s return pc with **skip**'s return pc, **fudge_return()** overwrites **task::task**'s return pc with the address of an assembly language function, **fudge_sp()**. When the parent task returns through **task::task()**, it calls **FUDGE_SP()**, which sets a global variable, **Skip_pc_p**, to point to **skip**'s return pc in the stack. Then **task::task()** returns to **fudge_sp()**,[34] which sets the stack pointer to **Skip_pc_p**, and executes an **rts** instruction, which pops **skip's** saved return pc off the stack, leaving the stack pointer at the top of **main()**'s frame.

Finding where registers are saved: FrameLayout()

As mentioned above, fudging the parent stack requires knowing the layout of the stack frames surrounding the one to be fudged.[35] This is not a problem for targets with self-describing stack frames, such as the VAX. Targets that do not have self-describing stack frames, such as the 3B and Sun-2/3, include a structure, defined in the source file **fudge.c**, called **FrameLayout**. **FrameLayout** has different members, depending on the target. It always has a constructor, which initializes the members so that fudge_return() has the information it needs to modify the parent stack.

FrameLayout for the 3B Processors

On the 3B2 and 3B20 targets the layout of saved registers follows from the number of registers saved by the function. On both targets, the size of the save area is invariant; if fewer than all the registers are saved, some slots in the save area will be unused and contain garbage values. The number of registers saved is found by looking at the save instruction of the function in question. By convention, the save instruction is the first instruction of the function. The easiest way to find the save instruction for a given function, *f*, is by dereferencing a pointer to the function. However, when *f* is a constructor, as both **task::task()** and **user_task::user_task()** are, one cannot take its address. In this case, one can find the save instruction for *f* by using the pointer to the return pc saved in the *f*'s frame, backing up one instruction to find the instruction to call *f*, and following the destination argument of the call to find the save instruction.

On the 3B targets, **FrameLayout** contains one element: **n_saved**, which represents the number of registers saved in the frame. The **FrameLayout** constructor finds **n_saved** for the frame denoted by its frame pointer argument. **FrameLayout::FrameLayout()** uses the frame pointer to find the return pc, which points to the instruction after the call to the denoted function. It backs up one instruction to get a pointer to the call instruction,[36] then decodes the call instruction (using a function called **call_dst_ptr()**) to get a pointer to the function denoted by the frame pointer argument. Finally, it decodes the save instruction (pointed to by the function pointer) to find the number of registers saved in the frame.

FrameLayout for the Sun-2/3 Target

On the Sun-2/3 target, **FrameLayout** contains two elements: **offset**, the offset of the top of the register save area from the frame pointer, and **mask**, the bit mask denoting which registers were saved. The **FrameLayout** constructor for the Sun-2/3 initializes the structure by a method similar to that described above for the 3B targets, which involves following the return pc to find the call, and decoding the call to find the destination of the call. Finally, it decodes the instructions in the function prologue (which can vary), to find the **mask** and the **offset**.

Source File Organization

The target-dependent parts of the task library are isolated in four source files:

hw_stack.h
: contains target-dependent macro, const, structure, and function declarations for each supported target (surrounded by **#ifdefs**).

hw_stack.c
: contains definitions of target-dependent functions for each supported target (surrounded by **#ifdefs**). Many of these are short assembly language functions which set or return hardware registers.

fudge.c
: There is a version of **fudge.c** for each supported target, currently: **fudge.c.3b**, **fudge.c.vax**, **fudge.c.386**, and **fudge.c.68k**.[37] These files contain definitions of **task::fudge_return()** and **FrameLayout::FrameLayout()** (for the targets that need it).

swap.s
: There is a version of **swap.s** for each supported target, currently: **swap.s.3b**, **swap.s.vax**, **swap.s.386**, and **swap.s.68k**. These files contain the assembly language functions **swap()** and **sswap()**.

Hints for Porting the Task Library to Other Processors

- Draw pictures (like those in Figure 2-1) of the stack frame layout for the target to which you are porting. Detailed pictures of the register save areas of several frames on the stack, like those in Figures 2-5 and 2-6, are helpful in writing **fudge_return()**.
- Become familiar with the sequence of operations in function calls and returns. Write and compile some sample C or C++ programs and look at the generated code to see what kinds of call and return sequences the compiler generates, in what order registers are used, and so forth. A fast way to write the copy in and copy out loops for **sswap()** is to write them in C, compile them with the **-S** option, and transcribe the generated code into **sswap()**.
- The implementation of the task library was designed to be both maintainable and, as far as possible, portable across both machines and compilers. These goals are sometimes mutually exclusive, and in those cases, we aimed for maintainability and portability across different compilers for the same machine (where possible). Some porters may want to write some of the assembly language functions in **hw_stack.c** as macros that depend on positional parameters and compiler conventions. For example, **FP()** returns the frame pointer for the calling function. This could also be written for the 3B targets as a macro that takes as an argument the first automatic variable of the function and returns the address of that variable, or for the VAX takes the same argument and returns the address of that variable minus one. This only works if the macro is given the **first** automatic as an argument, if the compiler assigns automatics in the order in which they are declared, and if the optimizer leaves the automatic on the stack, even if it is never read nor written.

Footnotes

1. The original version of this paper was written in 1980 by B. Stroustrup and revised in 1982 by him. Since then both the task library and C++ (then known as "C with Classes") have changed substantially, but the interface to the task library has been left intact. This has allowed old programs to run with new versions of the library, but has prevented any updating of the style of the interface, which does not conform to current tastes.

   This version of the paper has been revised by J. E. Shopiro to reflect the present state of affairs. I have added a few notes (in sans-serif type) where changes have been significant, and have made numerous syntactic changes, etc., without further comment.

2. Many of the member functions are inline, but their definitions are not shown here to prevent clutter. Class **task** is derived from class **sched** which is derived from class **object**. Class **object** is a simple base class used by most classes in the task system. It contains some of the pointers used by the task system's internal "house-keeping." Class **object** is described under "The **object** Class."

3. The class may have other member functions, of course, which may be called by the constructor or by any other function according to the usual rules of C++.

4. When the first task is created, **main()** automatically becomes a task itself.

5. It is a fairly simple job to add a new kind of task that returns some other datatype.

6. The handling of run time errors will be described below.

7. Thus **qhead::pending()** returns 1 if the queue is empty and 0 otherwise. Correspondingly, **qtail::pending() returns 1 if the queue is full and 0 otherwise.**

8. The default maximum size for a queue is 10000. That is, the queue can hold up to 10000 pointers to objects. It does not, however, pre-allocate space.

9. The original task package had a number of global variables, including **thistask**, **task_chain**, and **clock**. They are now all macros which expand to inline functions that return the values of private static variables. Thus programs that just read the values will be unaffected, but programs that try to set them (which was always illegal) will fail to compile.

10. **Waitlist()** is an example of a function whose form does not satisfy current esthetic standards.

11. In a quasi-parallel system this will only be true provided no infinite loop without task system calls exists. Such a loop constitutes an error that only a system with true parallelism or time slicing can recover from.

12. Coroutines can exchange control among themselves more freely than ordinary functions and procedures. In the usual function calling discipline, when one procedure (more precisely, one instance of a procedure) executes a procedure call, a new instance of the called procedure is created, and the calling procedure waits until the called procedure (and any procedures *it* may call) returns. A procedure instance is initiated when the procedure is called and is destroyed when it returns. When one coroutine (coroutine instance) initiates another it need not wait for the new coroutine to end, but instead it can be resumed while the new coroutine is still active. A running coroutine can relinquish control to any waiting coroutine without abandoning its state and later regain control and continue from where it left off.

13. Class **object** is the base class of most classes in the task system. We use the `typewriter` font for programming language constructs.

14. Semaphores which are used for mutual exclusion are initialized with one excess signal so that the first lock call will succeed.

15. But watch out for deadlock.

16. To the extent that the target hardware dictates subroutine linkage and stack frame layout, the compiler is less important. Some machines, like the 3Bs and the VAX, support a particular stack frame; the task library is largely independent of the compiler on these machines. The 68000, however, does not support a specific stack frame arrangement; the task library on this machine also depends on the compiler conventions for the stack frame. The word *target* will be used in this paper to denote an instance of either a processor or a compiler/processor platform.

17. The stack frame layout on the AT&T 6386 WGS is similar to that on the Sun-2/3 Workstations. The task library port is also similar on these targets.

18. See "A Set of C++ Classes for Co-routine Style Programming" or "Extending the C++ Task System for Real-Time Control" for details. The ways in which a task is put to sleep and awakened are target-independent.

19. This is true for our example targets. Some targets may use a caller save convention rather than a callee save convention.

20. It may not be immediately obvious that *all* registers must be saved on a task switch. Consider a task A, which has a function *f* that uses all the registers. It calls another function, *g*, which uses less than all the registers, say two, and therefore only saves two registers in its save area. If a task switch occurs before *g* returns, and task B uses all the registers, it will destroy those needed by task A's function *f*.

21. It can happen that a SHARED task will resume execution without having ever been displaced by another task sharing the same stack.

22. For example, DEDICATED tasks do not work with 3B2s running versions of the UNIX system earlier than SVR3.

23. The function **resume()** is virtual, with definitions for tasks and timers. Only tasks are relevant here.

24. If the suspending task is TERMINATED, then **swap()** does not save its state.

25. Writing the code for stack copying of SHARED tasks in assembly language adds more complexity than we would like to the job of porting the task library. It would be possible to call a C function to copy out the suspended task's stack to its save area. However, copying the resuming task's stack back in presents a problem: If the resuming task's stack is taller than the stack on which we are executing, a copy-in will overwrite the current stack frame. The **sswap()** function is careful to move all the data it needs from the frame into registers, so that if the frame is overwritten, **sswap()** can still complete successfully. But if **sswap()** called a C function to do the copy-in, that function might overwrite its own stack frame, making it impossible to return to **sswap()** to finish the task switch. So long as the copy-in must be written as part of **sswap()**, it seems little more trouble to write the complementary copy-out in assembly language as well.

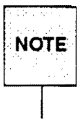
26. When the prevOnStack task and the resuming task are the same, **restore()** calls **swap()**, rather than **sswap()**, to do the task switch, as no stack copying is necessary.

27. The constructor **task::task()** only acquires stack space for DEDICATED tasks, that is, tasks that have their own stack. SHARED tasks will need space in which to save the current (or parent) task's active stack; **sswap()** takes care of that, as described above.

28. Only one child is activated at a time — remember, no pre-emption — and the child runs immediately, so it is safe to put these registers in a global, and more space-efficient than keeping them as part of the task object.
29. Note that, in Figure 2-5, the saved r5 and r6 in **task::task**'s frame are labeled "main's r5" and "main's r6" rather than "skip's r5" and "skip's r6." This is because in this example, **skip()** does not use r5 or r6; **main()** was the last function to use r5 and r6. Therefore, the values of r5 and r6 saved in **task::task**'s frame are the values that r5 and r6 had when **main()** was running.
30. The restore instruction for the VAX doesn't specify which registers to restore.
31. In the case of a task that repeatedly spawned children, the stack pointer would grow unnecessarily, eventually causing the stack to overflow. Each time the parent task returned from **task::task**, the stack pointer would be an additional frame higher than needed, and a new call to **task::task** would start building the next frame where the stack pointer pointed.
32. **FUDGE_SP()** is defined as a do-nothing macro for the VAX.
33. The AT&T 6386 WGS port of the task library also uses this technique.
34. When **task::task()** returns, the hardware registers are restored to the values they had in **main()** and the frame pointer is set to the value it had in **main()**, but the program counter is set to **fudge_sp()**.
35. Some of these frames are for user functions, so we cannot rely on techniques which require the C++ code for the function to be written so as to generate code that creates frames with some particular layout.
36. Because 3B instructions can be of various sizes, one cannot deterministically "back up" one instruction. **FrameLayout::FrameLayout()** subtracts each possible instruction size from the return pc and decodes the resulting pointer to check for a call opcode and legal operands. There is a small possibility, reduced by familiarity with the compiler, that these heuristic methods could yield more than one candidate call instruction.
37. The .68k suffix used for the Sun-2/3 target is something of a misnomer. These files were written specifically for Sun compiler/68K platforms; they will not necessarily work on all 68K platforms, for example, the AT&T compiler for the 68K. However, the **#ifdefs** in the source files say **#ifdef mc68000**.

3 Iostream Examples

Iostream Examples



This chapter is taken directly from a paper by Jerry Schwarz.

Abstract

The iostream library supports formatted I/O in C++. This document, 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. The streambuf class supports alternate sources and sinks of characters.

The manual pages for the iostream library can be found in Appendix A.

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++ (as of 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 document 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 document 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.

This document 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.

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 upper level is responsible for formatting. 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 document 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 **y** 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 document 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 document 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 or'ed 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 ;
        }
```

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 of 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 **ostreams** 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), IOMANIP(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 **streambufs**, pointed at by **in** and **out**. Methods for constructing **streambufs** 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**. EOFs returned while fetching tend to mean that the streambuf has run out of characters from the ultimate producer. EOFs returned when storing tend to signal real errors. Because, unlike iostreams, streambufs 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()** 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 **streambufs** 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 streambufs, 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 be sent to the ultimate consumer.

In some circumstances, such as when streambufs 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 queuelike, 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 filelike, which means that when one pointer moves the other is adjusted to point to the same place. So a filelike class behaves as if there were only one pointer. Other possibilities are logically possible, but do not seem to be as useful.

A queuelike 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 **strstreambufs** 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 streambufs. 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 **chars** 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 << '\t' ;
        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 (**seeks**).
- 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 document 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 **streambufs** 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. For compatibility a **stream.h** is supplied that includes **iostream.h** and other headers that are required for compatibility and 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**. **streambufs** created for this purpose can usually be replaced directly by **strstreambufs** 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 **filebufs**. 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 streambufs (declared in **stdiostream.h**) are provided to make the connection.

The old usage:

```
FILE* stdiofile ;
filebuf fb(stdiofile) ;
istream in(stdiofile) ;
ostream out(stdiofile) ;
```

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 ;
```

A Appendix A

Manual Pages for C++ Class Libraries

Complex Library Manual Pages

NAME

complex – introduction to C++ complex mathematics library

SYNOPSIS

```
#include <complex.h>
class  complex;
```

DESCRIPTION

This section describes functions and operators found in the C++ Complex Mathematics Library, *libcomplex.a*. These functions are not automatically loaded by the C++ compiler, CC(1); however, the link editor searches this library under the **-lcomplex** option. Declarations for these functions may be found in the **#include** file **<complex.h>**.

The Complex Mathematics library implements the data type of complex numbers as a class, `complex`. It overloads the standard input, output, arithmetic, assignment, and comparison operators, discussed in the manual pages for cplxops(3C++). It also overloads the standard exponential, logarithm, power, and square root functions, discussed in cplxexp(3C++), and the trigonometric functions of sine, cosine, hyperbolic sine, and hyperbolic cosine, discussed in cplxtrig(3C++), for the class `complex`. Routines for converting between Cartesian and polar coordinate systems are discussed in cartpol(3C++). Error handling is described in cplxerr(3C++).

FILES

INCDIR/complex.h
LIBDIR/libcomplex.a

SEE ALSO

cartpol(3C++), cplxerr(3C++), cplxops(3C++), cplxexp(3C++), and cplxtrig(3C++).
Stroustrup, B., "Complex Arithmetic in C++," C++ Translator Release 2.0 Documentation.

DIAGNOSTICS

Functions in the Complex Mathematics Library (3C++) may return the conventional values **(0, 0)**, **(0, ±HUGE)**, **(±HUGE, 0)**, or **(±HUGE, ±HUGE)**, when the function is undefined for the given arguments or when the value is not representable. **(HUGE** is the largest-magnitude single-precision floating-point number and is defined in the file **<math.h>**. The header file <math.h> is included in the file **<complex.h>**.) In these cases, the external variable *errno* [see *intro*(2)] is set to the value EDOM or ERANGE.

NAME

cartesian/polar – functions for the C++ Complex Math Library

SYNOPSIS

```
#include <complex.h>

class complex {

public:

        friend double       abs(complex);
        friend double       arg(complex);
        friend complex      conj(complex);
        friend double       imag(complex);
        friend double       norm(complex);
        friend complex      polar(double, double = 0);
        friend double       real(complex);
};
```

DESCRIPTION

The following functions are defined for `complex`, where:
— **d**, **m**, and **a** are of type `integer` and
— **x** and **y** are of type `complex`.

d = abs(x) Returns the absolute value or magnitude of **x**.

d = norm(x) Returns the square of the magnitude of **x**. It is faster than **abs**, but more likely to cause an overflow error. It is intended for comparison of magnitudes.

d = arg(x) Returns the angle of **x**, measured in radians in the range $-\pi$ to π.

y = conj(x) Returns the conjugation of **x**. That is, if **x** is *(real, imag)*, then **conj(x)** is *(real, −imag)*.

y = polar(m, a) Creates a complex given a pair of polar coordinates, magnitude **m**, and angle **a**, measured in radians in the range $-\pi$ to π.

d = real(x) Returns the real part of **x**.

d = imag(x) Returns the imaginary part of **x**.

SEE ALSO

CPLX.INTRO(3C++), cplxerr(3C++), cplxops(3C++), cplxexp(3C++), and cplxtrig(3C++).

NAME

complex_error – error-handling function for the C++ Complex Math Library

SYNOPSIS

```
#include <complex.h>

class c_exception
{
                int             type;
                char            *name;
                complex         arg1;
                complex         arg2;
                complex         retval;

public:

                c_exception( char *n, const complex& a1, const complex& a2 = complex_zero );

                friend int      complex_error( c_exception& );

                friend complex  exp( complex );
                friend complex  sinh( complex );
                friend complex  cosh( complex );
                friend complex  log( complex );
};
```

DESCRIPTION

In the following description of the `complex` error handling routine,
— d is of type `double` and
— x is of type `complex`.

d = complex_error(x&) Invoked by functions in the C++ Complex Mathematics Library when errors are detected.

Users may define their own procedures for handling errors, by defining a function named complex_error in their programs. **complex_error** must be of the form described above.

The element *type* is an integer describing the type of error that has occurred, from the following list of constants (defined in the header file):

SING	argument singularity
OVERFLOW	overflow range error
UNDERFLOW	underflow range error

The element *name* points to a string containing the name of the function that incurred the error. The variables *arg1* and *arg2* are the arguments with which the function was invoked. *Retval* is set to the default value that will be returned by the function unless the user's **complex_error** sets it to a different value.

If the user's **complex_error** function returns non-zero, no error message will be printed, and *errno* will not be set.

If **complex_error** is not supplied by the user, the default error-handling procedures, described with the complex math functions involved, will be invoked upon error. These procedures are also summarized in the table below. In every case, *errno* is set to EDOM or ERANGE and the program continues.

Note that complex math functions call functions included in the math library which has its own error handling routine, *matherr(3M)*. Users may also override this routine by supplying their own version.

DEFAULT ERROR HANDLING PROCEDURES			
	Types of Errors		
type	SING	OVERFLOW	UNDERFLOW
errno	EDOM	ERANGE	ERANGE
EXP:			
real too large/small	-	(±H, ±H)	(0, 0)
imag too large	-	(0, 0)	-
LOG:			
arg = (0, 0)	M, (H, 0)	-	-
SINH:			
real too large	-	(±H, ±H)	-
imag too large	-	(0, 0)	-
COSH:			
real too large	-	(±H, ±H)	-
imag too large	-	(0, 0)	-

ABBREVIATIONS	
M	Message is printed (EDOM error).
(H, 0)	(HUGE, 0) is returned.
(±H, ±H)	(±HUGE, ±HUGE) is returned.
(0, 0)	(0, 0) is returned.

SEE ALSO

CPLX.INTRO(3C++), matherr(3M), cartpol(3C++), cplxops(3C++), cplxexp(3C++), and cplxtrig(3C++).

NAME

exp, log, pow, sqrt – exponential, logarithm, power, square root functions for the C++ complex library

SYNOPSIS

```
#include <complex.h>

class complex {

public:
        friend complex      exp(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      sqrt(complex);
};
```

DESCRIPTION

The following math functions are overloaded by the complex library, where:
— x, y, and z are of type `complex`.

z = exp(x)	Returns e^x.
z = log(x)	Returns the natural logarithm of **x**.
z = pow(x, y)	Returns x^y.
z = sqrt(x)	Returns the square root of **x**, contained in the first or fourth quadrants of the complex plane.

SEE ALSO

CPLX.INTRO(3C++), cartpol(3C++), cplxerr(3C++), cplxops(3C++), and cplxtrig(3C++).

DIAGNOSTICS

exp returns **(0, 0)** when the real part of x is so small, or the imaginary part is so large, as to cause overflow. When the real part is large enough to cause overflow, exp returns **(HUGE, HUGE)** if the cosine and sine of the imaginary part of x are positive, **(HUGE, –HUGE)** if the cosine is positive and the sine is not, **(–HUGE, HUGE)** if the sine is positive and the cosine is not, and **(–HUGE, –HUGE)** if neither sine or cosine is positive. In all these cases, *errno* is set to ERANGE.

log returns **(–HUGE, 0)** and sets *errno* to EDOM when *x* is **(0, 0)**. A message indicating SING error is printed on the standard error output.

These error-handling procedures may be changed with the function **complex_error** (cplxerr(3C++)).

NAME

complex_operators: operators for the C++ complex math library

SYNOPSIS

```
#include <complex.h>

class complex {

public:
        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);
};
```

DESCRIPTION

The basic arithmetic operators, comparison operators, and assignment operators are overloaded for complex numbers. The operators have their conventional precedences. In the following descriptions for `complex` operators,
— x, y, and z are of type `complex`.

Arithmetic operators:

z = x + y	Returns a `complex` which is the arithmetic sum of complex numbers *x* and *y*.
z = -x	Returns a `complex` which is the arithmetic negation of complex number *x*.
z = x - y	Returns a `complex` which is the arithmetic difference of complex numbers *x* and *y*.
z = x * y	Returns a `complex` which is the arithmetic product of complex numbers *x* and *y*.
z = x / y	Returns a `complex` which is the arithmetic quotient of complex numbers *x* and *y*.

Comparison operators:

x == y	Returns non-zero if complex number *x* is equal to complex number *y*; returns 0 otherwise.
x != y	Returns non-zero if complex number *x* is not equal to complex number *y*; returns 0 otherwise.

Assignment operators:

x += y	Complex number *x* is assigned the value of the arithmetic sum of itself and complex number *y*.
x -= y	Complex number *x* is assigned the value of the arithmetic difference of itself and complex number *y*.

x *= y — Complex number *x* is assigned the value of the arithmetic product of itself and complex number *y*.

x /= y — Complex number *x* is assigned the value of the arithmetic quotient of itself and complex number *y*.

WARNING

The assignment operators do not produce a value that can be used in an expression. That is, the following construction is syntactically invalid,

```
complex    x, y, z;
x = ( y += z );
```

whereas,

```
x = ( y + z );

x = ( y == z );
```

are valid.

SEE ALSO

CPLX.INTRO(3C++), cartpol(3C++), cplxerr(3C++), cplxexp(3C++), and cplxtrig(3C++).

NAME

cplxtrig – trigonometric and hyperbolic functions for the C++ complex library

SYNOPSIS

```
#include <complex.h>

class complex {

public:
        friend complex      sin(complex);
        friend complex      cos(complex);

        friend complex      sinh(complex);
        friend complex      cosh(complex);
};
```

DESCRIPTION

The following trigonometric functions are defined for `complex`, where:
— x and y are of type `complex`.

y = sin(x) Returns the sine of **x**.

y = cos(x) Returns the cosine of **x**.

y = sinh(x) Returns the hyperbolic sine of **x**.

y = cosh(x) Returns the hyperbolic cosine of **x**.

SEE ALSO

CPLX.INTRO(3C++), cartpol(3C++), cplxerr(3C++), cplxops(3C++), and cplxexp(3C++).

DIAGNOSTICS

If the imaginary part of *x* would cause overflow **sinh** and **cosh** return **(0, 0)** . When the real part is large enough to cause overflow, **sinh** and **cosh** return **(HUGE, HUGE)** if the cosine and sine of the imaginary part of x are non-negative, **(HUGE, –HUGE)** if the cosine is non-negative and the sine is less than 0, **(–HUGE, HUGE)** if the sine is non-negative and the cosine is less than 0, and **(–HUGE, –HUGE)** if both sine and cosine are less than 0. In all these cases, *errno* is set to **ERANGE**.

These error-handling procedures may be changed with the function **complex_error** (cplxerr(3C++)).

Task Library Manual Pages

NAME

task – coroutines, multiple threads of control, C++ task library

SYNOPSIS

```
#include <task.h>

class object;
class sched : public object;
class timer : public sched;
class task  : public sched;

class qhead : public object;
class qtail : public object;

class Interrupt_handler : public object;

class histogram;
class randint;
class urand : public randint;
class erand : public randint;
```

DESCRIPTION

The C++ task library provides facilities for writing programs with multiple threads of control within one UNIX system process. Each thread of control is a `task` or coroutine. Each `task` is an instance of a user-defined class derived from class `task`, and the main program of the `task` is the constructor of its class. A `task` can be suspended and resumed without interfering with its internal state. Each `task` runs until it explicitly gives up the processor; there is no pre-emption.

Most classes in the task system are derived from the base class `object`. The base class `sched` is responsible for scheduling and for the functionality that is common to `task`s and `timer`s. Class `sched` is meant to be used strictly as a base class, that is, it is illegal to create objects of class `sched`. Class `task` must also be used only as a base class. The programmer must derive a class from class `task`, and provide a constructor to serve as the `task`'s main program. The task system can be used for writing event-driven simulations. `task`s execute in a simulated time frame. Objects of class `timer` provide a facility for implementing time-outs and other time-dependent phenomena. Classes `task`, `timer`, `sched`, and `object` and their public member functions are described on the task(3C++) manual page.

Classes `qhead` and `qtail` enable a wide range of message-passing and data-buffering schemes to be implemented simply. These classes are described on the queue(3C++) manual page.

Class `Interrupt_handler` provides an interface for writing classes that can wait for external events using UNIX system signals. These classes are described on the interrupt(3C++) manual page.

Class `histogram` aids data gathering. Classes `randint`, `urand`, and `erand` provide random number generation. These four classes are described on the tasksim(3C++) manual page.

SEE ALSO

task(3C++), queue(3C++), interrupt(3C++), tasksim(3C++)

Stroustrup, B. and Shopiro, J. E., "A Set of C++ Classes for Co-routine Style Programming," in *AT&T C++ Language System Release 2.0 Library Manual.*

Shopiro, J. E., "Extending the C++ Task System for Real-Time Control," in *AT&T C++ Language System Release 2.0 Library Manual.*

NAME

Interrupt_handler – signal handling for the C++ task library

SYNOPSIS

```
#include <task.h>

class Interrupt_handler : public object {
        virtual void    interrupt();
public:
                        Interrupt_handler(int);
                        ~Interrupt_handler();
        objtype         o_type();
        int             pending();
};
```

DESCRIPTION

Class `Interrupt_handler` allows `tasks` to wait for external events in the form of UNIX system signals. Class `Interrupt_handler` is derived from class `object` so that `tasks` can wait for `Interrupt_handler` objects. Class `object` is described on the task(3) manual page.

The public member functions supplied in the task system class `Interrupt_handler` are listed and described below. The following symbols are used:

ih an `Interrupt_handler` object

i an `int`

eo an `objtype` enumeration

Interrupt_handler ih(i);

Constructs a new `Interrupt_handler` object, **ih**, which is to wait for a signal number **i**. (See signal(2).) Once an `Interrupt_handler` object has been established for a particular signal, when that signal occurs, the private, virtual **interrupt()** function is called at real time. When it returns, control will resume at the point where the current `task` was interrupted. That is, signals do not cause the current `task` to be pre-empted. When the currently running `task` is suspended, a special, built-in task, the *interrupt alerter* will be scheduled. This `task` alerts the `Interrupt_handler` (and any others that have received interrupts since the *interrupt alerter* last ran), and thereby makes any `tasks` waiting for those `Interrupt_handlers` runnable. As long as any `Interrupt_handler` exists, the scheduler will wait for an interrupt, rather than exiting when the *run chain* becomes empty.

void interrupt()

The private, virtual function, **Interrupt_handler::interrupt()** is a null function, but because it is virtual, the programmer can specify the action to be taken at interrupt time by defining an **interrupt()** function in a class derived from `Interrupt_handler`.

eo = ih.o_type()

Returns the class type of the object (`object::INTHANDLER`). **o_type()** is a virtual function.

i = ih.pending()

Returns TRUE except the first time it is called after a signal occurs.

DIAGNOSTICS

See task(3).

BUGS

UNIX System V Releases 3.1 and 3.2 (SVR3.1 and SVR3.2) for the Intel 386 machine will not call a signal handler when the current `task` is running on a stack in the free store, that is, when the current `task` has a DEDICATED stack. If you need to use the signal handling mechanisms on that configuration, you cannot use tasks which have DEDICATED stacks. In this case, compile the task library with _SHARED_ONLY defined, which will make SHARED the default mode for `tasks`. (Note: it is insufficient to declare all `tasks` as SHARED without compiling a _SHARED_ONLY version of the task library, because the internal system `task`, the *interrupt alerter* is DEDICATED by default.)

SEE ALSO

TASK.INTRO(3C++), task(3C++), queue(3C++), tasksim(3C++), signal(2)

Stroustrup, B. and Shopiro, J. E., "A Set of C++ Classes for Co-routine Style Programming," in *AT&T C++ Language System Release 2.0 Library Manual.*

Shopiro, J. E., "Extending the C++ Task System for Real-Time Control," in *AT&T C++ Language System Release 2.0 Library Manual.*

NAME

queue – qheads and qtails for the C++ task library

SYNOPSIS

```
#include <task.h>

enum qmodetype { EMODE, WMODE, ZMODE };

class qhead : public object {
public:
                        qhead(qmodetype =WMODE, int =10000);
                        ~qhead();
        qhead*          cut();
        object*         get();
        objtype         o_type();
        int             pending();
        void            print(int, int =0);
        int             putback(object*);
        int             rdcount();
        int             rdmax();
        qmodetype       rdmode();
        void            setmode(qmodetype);
        void            setmax(int);
        void            splice(qtail*);
        qtail*          tail();
};

class qtail : public object {
public:
                        qtail(qmodetype =WMODE, int =10000);
                        ~qtail();
        qtail*          cut();
        qhead*          head();
        objtype         o_type();
        int             pending();
        void            print(int, int =0);
        int             put(object*);
        int             rdspace();
        int             rdmax();
        qmodetype       rdmode();
        void            splice(qhead*);
        void            setmode(qmodetype);
        void            setmax(int);
};
```

DESCRIPTION

Classes `qhead` and `qtail` enable a wide range of message-passing and data-buffering schemes to be implemented simply with the C++ task system. Both classes are derived from the base class `object`, which is described on the task(3) manual page. In general, class `qhead` provides facilities for taking objects off a queue, and class `qtail` provides facilities for putting objects on a queue. The objects transmitted through a queue must be of class `object` or of some class derived from it.

A queue is a data structure with an associated list of `objects` in first-in, first-out order. Each queue also has associated `qhead` and `qtail` objects attached (one of each). No public functions are provided to operate on queues directly. Rather all access to a queue is through either the attached `qhead` or the attached `qtail`. To create a queue, the programmer must declare a `qhead` object and then use that object to call **qhead::tail()** or must declare a `qtail` object and then use that object to call **qtail::head()**. For example:

```
qhead qh;
qtail* qtp = qh.tail();
```

Once the queue is established, `objects` are added to it with **qtail::put()** and `objects` are removed from it with **qhead::get()**.

Objects derived from class `object` have definitions of when they are *ready* and *pending* (not ready). `qhead` objects are ready when the queue is not empty and pending when the queue is empty. `qtail` objects are ready when the queue is not full, and pending when the queue is full.

Queues have three attributes: mode, maximum size, and count. The size and count attributes apply to the queue itself, while the mode attribute applies independently to the `qhead` and `qtail` of a queue. These attributes are described below.

Both classes `qhead` and `qtail` have a mode (set by the constructor) that controls what happens when an object of that class is pending. The default is WMODE (wait mode). With WMODE, a `task` that executes **qhead::get()** on an empty queue will be suspended until that queue becomes non-empty. Similarly, with WMODE a `task` that executes **qtail::put()** on a full queue will be suspended until that queue has room for the `object` to be added to the queue. In EMODE (error mode), calling **qhead::get()** for an empty queue or calling **qtail::put()** for a full queue will cause a run time error. In ZMODE (zero mode), if **qhead::get()** is executed on an empty queue it will return the NULL pointer instead of a pointer to an object. In ZMODE, if **qtail::put()** is executed on a full queue, it will return 0 instead of 1. The modes of a queue's head and tail need not be the same. Classes `qhead` and `qtail` both provide a function, **setmode()**, which will reset the mode.

Queues also have a maximum size, which is set to 10000 by default. That is, the queue can hold up to 10000 pointers to objects. It does not, however, preallocate space. The size of a queue can be reset with either **qhead::setmax()** or **qtail::setmax()**.

The count is the number of objects on a queue.

Both the `qhead` and `qtail` constructors optionally take mode and size arguments.

The public member functions supplied in the task system classes `qhead` and `qtail` are listed and described in the next two sections. The following symbols are used:

qh a `qhead` object

qt a `qtail` object

t a `task` object

qhp
a pointer to a `qhead`

qtp a pointer to a `qtail`

op a pointer to an `object`

tp a pointer to a `task`

i, j `ints`

eo an `objtype` enumeration

eq a `qmodetype` enumeration

Class qhead

Class `qhead` has one form of constructor:

qhead qh(eq, j)

Constructs a `qhead` object, **qh**. Both arguments are optional and have default values. **eq** represents the *mode* (see above), which can be WMODE, EMODE, or ZMODE. WMODE is the default. **j** represents the maximum length of the queue attached to **qh**; the default is 10000.

The public member functions of class `qhead` are (in alphabetical order):

qhp = qh.cut()

Splits **qh** in two. **qhead::cut()** returns a pointer to a new `qhead`, which is attached to the original queue. `objects` that are already on the queue and `objects` that are **qtail::put()** on the original queue, must be retrieved via **qhp**. **qhead::cut()** modifies **qh** to point to a new empty queue. A new `qtail` must be established for **qh** (with **qh.tail()**). `objects` that are **qtail::put()** to the new `qtail`, can be retrieved via a **qh.get()** .

Thus, **qhead::cut()** can be used to insert a filter into an existing queue, without changing the appearance of the queue to anyone using it, and without halting the flow of `objects` through the queue. The filter will intercept `objects` that are **qtail::put()** on the original `qtail` when it does a **qhead::get()** on the new `qhead`. Then the filter can **qtail::put()** `objects` on the new `qtail`, where execution of **qhead::get()** on the original `qhead` will retrieve them. In other words, the filter `task` uses the newly established `qhead` and `qtail`, while other `tasks` continue to **put()** and **get()** from the original `qtail` and `qhead`. **qhead::splice()** can be used to restore the queue to its original configuration.

op = qh.get()

Returns a pointer to the `object` at the head of the queue, if the queue is not empty. If the queue is empty, **qhead::get()** 's behavior depends on the mode of **qh**. In WMODE, a `task` that executes **qhead::get()** on an empty queue will be suspended until that queue becomes non-empty, when the operation can complete successfully. In EMODE, it will cause a run time error. In ZMODE, it will return the NULL pointer instead of a pointer to an `object`.

eo = qh.o_type()

Returns the class type of the object (`object::QHEAD`). **o_type()** is a virtual function.

i = qh.pending()

Returns TRUE if the queue attached to **qh** is empty, and FALSE otherwise. **pending()** is a virtual function.

qh.print(i)

Prints the contents of **qh** on `stdout`. It calls the **print()** function for the `object` base class. **i** specifies the amount of information to be printed. It can be 0, for the minimum amount of information, or VERBOSE, for more information. A second integer argument is for internal use and defaults to 0. **print()** is a virtual function.

i = qh.putback(op)

Puts the `object` denoted by **op** back on the head of the queue attached to **qh**, and returns 1 on success. This allows a `qhead` to operate as a stack. A `task` calling **qhead::putback()** competes for queue space with `tasks` using **qtail::put()**. Calling **qhead::putback()** for a full queue causes a run time error in both EMODE and

WMODE, and returns NULL in ZMODE.

i = qh.rdcount()
Returns the current number of `objects` in the queue attached to **qh**.

i = qh.rdmax()
Returns the maximum size of the queue attached to **qh**.

eq = qh.rdmode()
Returns the current mode of **qh**, WMODE, EMODE, or ZMODE.

qh.setmode(eq)
Sets the mode of **qh** to **eq**, which can be WMODE, EMODE, or ZMODE.

qh.setmax(i)
Sets the maximum size of the queue attached to **qh** to **i** It is legal to decrease the maximum below the current number of `objects` on the queue. Doing so means that no more `objects` can be put on the queue until the queue has been drained below the new limit.

qh.splice(qtp)
Reverses the action of a previous **qhead::cut()**. **qhead::splice()** merges the queue attached to **qh** with the queue attached to **qtp**. The list of `objects` on the latter queue precede those on the former queue in the merged list. **qhead::splice()** deletes **qh** and **qtp**. **qh** is meant to be a `qhead` that was previously **cut()**, and **qtp** is meant to be the pointer returned by that **cut()**. If in merging the queues **qhead::splice()** causes an empty queue to become non-empty or a full queue to become non-full, it will alert all `tasks` waiting for that state change, and add them to the scheduler's *run chain.* (See **object::alert()** on the task(3) manual page.)

qtp = qh.tail()
Creates a `qtail` object for the queue attached to **qh** (if none exists) and returns a pointer, **qtp**, to the new `qtail` object.

Class qtail

Class `qtail` has one form of constructor:

qtail qt(eq, j)
Constructs a `qtail` object, **qt**. Both arguments are optional and have default values. **eq** represents the *mode* (see above), which can be WMODE, EMODE, or ZMODE. WMODE is the default. **j** represents the maximum length of the queue attached to **qt**; the default is 10000.

The public member functions of class `qtail` are (in alphabetical order):

qtp = qt.cut()
Splits the queue to which it is applied in two. **qtail::cut()** returns a pointer to a new `qtail`, which is attached to the original queue. `objects` already on the original queue can still be retrieved with a **qhead::get()** to the original `qhead`. (This is the primary functional difference between **qhead::cut()** and **qtail::cut()**.) **qtail::cut()** modifies **qt** to point to a new empty queue. A new `qhead` must be established for **qt**. `objects` that are **qtail::put()** to **qt** must be retrieved via the new `qhead`. `objects` that are **qtail::put()** to **qtp** will be retrieved via the original `qhead`.

Thus, **qtail::cut()** can be used to insert a filter into an existing queue, without changing the appearance of the queue to anyone using it, and without halting the flow of `objects` through the queue. The filter will intercept `objects` that are **qtail::put()** on the original `qtail` when it does a **qhead::get()** on the new `qhead`. Then the filter can **qtail::put()** `objects` on the new `qtail`, where execution of **qhead::get()** on the original `qhead` will retrieve them. In other words, the filter `task` uses the newly

established qhead and qtail, while other tasks continue to put() and get() from the original qtail and qhead. qtail::splice() can be used to restore the queue to its original configuration.

qhp = qt.head()
Creates a qhead object for the queue attached to qt (if none exists) and returns a pointer to the new qhead object.

eo = qt.o_type()
Returns the class type of the object (object::QTAIL). o_type() is a virtual function.

i = qt.pending()
Returns TRUE if the queue attached to qt is full, and FALSE otherwise. pending() is a virtual function.

qt.print(i)
Prints the contents of qt on stdout. It calls the print() function for the object base class. i specifies the amount of information to be printed. It can be 0, for the minimum amount of information, or VERBOSE, for more information. A second integer argument is for internal use and defaults to 0. print() is a virtual function.

i = qt.put(op)
Adds the object denoted by op to the tail of the queue attached to qt, and returns 1 on success. If the queue is full, qtail::put() 's behavior depends on the mode of qt. In WMODE, a task that executes qtail::put() on a full queue will be suspended until that queue becomes non-full, when the operation can complete successfully. In EMODE, it will cause a run time error. In ZMODE, it will return NULL.

i = qt.rdspace()
Returns the number of objects that can be inserted into the queue attached to qt before it becomes full.

i = qt.rdmax()
Returns the maximum size of the queue attached to qt.

eq = qt.rdmode()
Returns the current mode of qt, WMODE, EMODE, or ZMODE.

qt.splice(qhp)
Reverses the action of a previous qtail::cut(). qtail::splice() merges the queue attached to qt with the queue attached to qhp. The list of objects on the former queue precede those on the latter queue in the merged list. qtail::splice() deletes qt and qhp. qt is meant to be a qtail that was previously cut(), and qhp is meant to be the pointer returned by that cut(). If in merging the queues qtail::splice() causes an empty queue to become non-empty or a full queue to become non-full, it will alert all tasks waiting for that state change, and add them to the scheduler's *run chain*. (See object::alert() on the task(3) manual page.)

qt.setmode(eq)
Sets the mode of qt to eq, which can be WMODE, EMODE, or ZMODE.

qt.setmax(i)
Sets the maximum size of the queue attached to qt to i. It is legal to decrease the maximum below the current number of objects on the queue. Doing so means that no more objects can be put on the queue until the queue has been drained below the new limit.

DIAGNOSTICS
See task(3).

SEE ALSO

TASK.INTRO(3C++), task(3C++), interrupt(3C++), tasksim(3C++)

Stroustrup, B. and Shopiro, J. E., "A Set of C++ Classes for Co-routine Style Programming," in *AT&T C++ Language System Release 2.0 Library Manual.*

NAME

task – coroutines, multiple threads of control, C++ task library

SYNOPSIS

```
#include <task.h>

typedef int (*PFIO)(int,object*);
typedef void (*PFV)();
extern  int _hwm;

class object {
public:
        enum objtype { OBJECT, TIMER, TASK, QHEAD, QTAIL, INTHANDLER };

        object*         o_next;
                        object();
                        ~object();
        void            alert();
        void            forget(task*);
        virtual objtype o_type();
        virtual int     pending();
        virtual void    print(int, int =0);
        void            remember(task*);
        static int      task_error(int, object*);
        int             task_error(int);
        static task*    this_task();
        static  PFIO    error_fct;
};

class sched : public object {
protected:
                        sched();
public:
        enum statetype { IDLE=1, RUNNING=2, TERMINATED=4 };

        static  task*   clock_task;
        void            cancel(int);
        int             dont_wait();
        static long     get_clock();
        sched*          get_priority_sched();
        static sched*   get_run_chain();
        static int      get_exit_status();
        int             keep_waiting();
        int             pending();
        void            print(int, int =0);
        statetype       rdstate();
        long            rdtime();
        int             result();
        void            setclock(long);
        static void     set_exit_status(int);
        virtual void    setwho(object*);
        static  PFV     exit_fct;
};
```

```
#define DEFAULT_MODE     DEDICATED

class task : public sched {
public:
        enum modetype { DEDICATED=1, SHARED=2 };
protected:
                        task(char* name=0, modetype mode=DEFAULT_MODE, int stacksize=SIZE);
public:
        task*           t_next;
        char*           t_name;
                        ~task();
        void            cancel(int);
        void            delay(int);
        static task*    get_task_chain();
        objtype         o_type();
        int             preempt();
        void            print(int, int =0);
        void            resultis(int);
        void            setwho(object*);
        void            sleep(object* =0);
        void            wait(object*);
        int             waitlist(object* ...);
        int             waitvec(object**);
        object*         who_alerted_me();
};

class timer : public sched {
public:
                        timer(int);
                        ~timer();
        objtype         o_type();
        void            print(int, int =0);
        void            reset(int);
        void            setwho(object*);
};
```

DESCRIPTION

A `task` is an object with an associated program and thread of control. To use the task system, the programmer must derive a class from class `task`, and supply a constructor to serve as the `task`'s main program. Control in the task system is based on a concept of operations which may succeed immediately or be blocked, and `objects` which may be *ready* or *pending* (not ready). When a `task` executes a blocking operation on an `object` that is ready, the operation succeeds immediately and the `task` continues running, but if the `object` is pending, the `task` waits. Control then returns to the scheduler, which chooses the next `task` from the ready list or *run chain*. Eventually, the pending `object` may become ready, and it will notify all the `tasks` that are waiting for it, causing the waiting `tasks` to be put back on the *run chain* .

A `task` can be in one of three states:

RUNNING The `task` is running or it is ready to run.

IDLE The `task` is waiting for a pending `object`.

TERMINATED The `task` has completed its work. It cannot be resumed, but its result can be retrieved.

The function **sched::rdstate()** returns the state. These states are enumerations of type `statetype`. These enumerations are in the scope of class `sched`.

Most classes in the task system are derived from class `object`. Each different kind of `object` can have its own way of determining whether it is ready, which makes it easy to add new capabilities to the system. However, each kind of `object` can have only one criterion for readiness (although it may have several blocking operations). The criterion for readiness is defined by the virtual function **pending()**. For all classes derived from `object`, **pending()** returns TRUE if the `object` is not ready. This invariant should be maintained for user-defined derived classes as well.

Each pending `object` contains a list (the *remember chain*) of the `tasks` that are waiting for it. When a `task` attempts an operation on a pending `object` (that is, it calls a blocking function), that `task` is put on the *remember chain* for the `object` via **object::remember()**, and the `task` is suspended. When the state of an `object` changes from pending to ready, **object::alert()** must be called for the `object`. (Note, this is done for classes in the task system. Programmers who write classes for which `tasks` can wait, must ensure that **object::alert()** is called on a state change.) **alert()** changes the state of all `tasks` "remembered" by the `object` from IDLE to RUNNING and puts them on the scheduler's *run chain*.

The base class, `sched`, is responsible for scheduling and for the functionality that is common to `tasks` and `timers`. Class `sched` can only be used as a base class, that is, it is illegal to create objects of class `sched`. Class `sched` also provides facilities for measuring simulated time. A unit of simulated time can represent any amount of real time, and it is possible to compute without consuming simulated time. The system clock is initialized to 0 and can be set with **sched::setclock()** once only. Thereafter, only a call to **task::delay()** will cause the clock to advance. **sched::getclock()** can be used to read the clock.

Class `timer` provides a facility for implementing time-outs and other time-dependent phenomena. A `timer` is similar to a `task` with a constructor consisting of the single statement:

```
delay(d);
```

That is, when a `timer` is created it simply waits for the number of time units given to it as its argument, and then wakes up any `tasks` waiting for it. A `timer`'s state can be either RUNNING or TERMINATED.

A `task` cannot return a value with the usual function return mechanism. Instead, a `task` sets the value of its *result* (using **task::resultis()** or **task::cancel()**), at which time the `task` becomes TERMINATED. Then this result can be retrieved by other `tasks` via a call to **sched::result()**.

The `task` constructor takes three optional arguments: a name, a mode, and a stacksize. The name is a character string pointer, which is used to initialize the class `task` variable `t_name`. This name can be used to provide more readable output and does not affect the behavior of the `task`.

The mode argument can be DEDICATED (the default when none is specified) or SHARED, (the enumerations of type `modetype` in class `task`'s scope). DEDICATED `tasks` each have their own stack, allocated from the free store. SHARED `tasks` share stack space with the `task` that creates them. When a SHARED `task` is running, it occupies the shared stack space, while copies of the active portions of the other `tasks`' stacks occupy save areas. SHARED `tasks` trade speed for space: they use less storage than DEDICATED `tasks` use, but task switches among SHARED `tasks` often involve copying stacks to and from the save area.

The stacksize argument to the `task` constructor represents the maximum space that a `task`'s stack can occupy. The default is 750 machine words. Overflowing the stack is a fatal error.

When an object of a class derived from class `task` is created, its constructor becomes a new `task` that runs in parallel with the other `tasks` that have been created. When the first `task` is created, `main()` automatically becomes a `task` itself.

The public member functions supplied in the task system classes `task`, `object`, `sched`, and `timer` are listed and described in the next four sections. The following symbols are used:

t a `task` object

o an `object` object

s a `sched` object

tm a `timer` object

op a pointer to an `object`

tp a pointer to a `task`

sp a pointer to a `sched`

cp a pointer to a `char`

i, j `ints`

l a `long int`

eo an `objtype` enumeration

es a `statetype` enumeration

em a `modetype` enumeration

Class Task

Class `task` has one form of constructor, which is protected:

task t(cp, em, j)

Constructs a `task` object, **t**. All three arguments are optional and have default values. If **cp** is given, the character string it points to is used as **t**'s name. **em** represents the *mode* (see above), and can be DEDICATED or SHARED. DEDICATED is the default. The default mode can be changed to SHARED by recompiling the task library with _SHARED_ONLY defined. See the NOTES section. **j** represents the maximum size of **t**'s stack; the default is 750 machine words.

Most public member functions of class `task` are conditional or unconditional requests for suspension. They are (in alphabetical order):

t.cancel(i)

Puts **t** into the TERMINATED state, without suspending the calling `task` (that is, without invoking the scheduler), and sets **t**'s result (or "return value") to **i**.

t.delay(i)

Suspends **t** for the time specified by **i**. A delayed `task` is in the RUNNING state. **t** will resume at the current time on the task system clock + **i**. Only a call to **delay()** causes the clock to advance.

tp = task::get_task_chain()
tp = t.get_task_chain()

Returns a pointer to the first `task` on the list of all `tasks` (linked by `t_next` pointers).

eo = t.o_type()

Returns the class type of **t** (`object::TASK`). **o_type()** is a virtual function.

i = t.preempt()

Suspends RUNNING `task` **t**, making it IDLE. Returns the number of time units left in **t**'s delay. Calling **preempt()** for an IDLE or TERMINATED `task` causes a runtime error.

t.print(i)

Prints the contents of **t** on `stdout`. The first argument, **i**, specifies the amount of information to be printed. It can be 0, for the minimum amount of information, VERBOSE, for more information, CHAIN, for information about each object on the chain of all `tasks`, or STACK, for information about the runtime stack. These argument constants can be combined with the or operator, e.g., `print(VERBOSE|CHAIN)`. A second integer argument is for internal use and defaults to 0. **print()** is a virtual function.

t.resultis(i)

Sets the result (or "return value") of **t** to be the value of **i** and puts **t** in the TERMINATED state. The result can be examined by calling **t.result()** (**result()** is a member function of class `sched`). `tasks` cannot return a value using the usual function return mechanism. A call to **task::resultis()** should appear at the end of every `task` constructor body (unless the constructor will execute infinitely). A `task` is *pending* (see **sched::pending()**) until it is TERMINATED.

t.setwho(op)

Records the `object` denoted by **op** as the one that alerted **t** when it was IDLE. ***op** is meant to be the `object` whose state change from pending to ready caused **t** to be put back on the *run chain*. This information can be retrieved with **task::who_alerted_me()**.

t.sleep(op)
t.sleep()

Suspends **t** unconditionally (puts the **t** in the IDLE state). The **op** argument is optional. If **task::sleep()** is given a pointer to a pending `object` as an argument, **t** will be "remembered" by the denoted `object`, so that when that `object` becomes ready, **t** will be "alerted" and put back on the *run chain* (via **object::alert()**). If no argument is given to **task::sleep()**, the event that will cause **t** to be resumed is unspecified. Contrast **sleep()** with **wait()**, which suspends a `task` conditionally. **task::sleep()** does not check whether the `object` denoted by **op** is pending.

t.wait(op)

If **op** points to a pending `object`, then **t** will be suspended (put in the IDLE state) until that `object` is ready. If **op** points to an `object` that is not pending (that is ready), then **t** will not be suspended at all. Any class derived from class `object` that is ever going be waited for must have rules for when it is pending and ready. Each `object` can only have one definition of pending.

i = t.waitlist(op ...)

Suspends **t** to wait for one of a list of `objects` to become ready. **waitlist()** takes a list of `object` pointers terminated by a 0 argument. If any of the arguments points to a "ready" `object`, then **t** will not be suspended at all. **waitlist()** returns when one of the `objects` on the list is ready. It returns the position in the list of the `object` that caused the return, with positions numbered starting from 0. Note that `objects` on the list other than the one denoted by the return value might also be ready.

i = t.waitvec(op*)

Is the same as **waitlist()**, except that it takes as an argument the address of a vector holding a list of `object` pointers.

op = t.who_alerted_me()
Returns a pointer to the `object` whose state change from *pending* to *ready* caused t to be put back on the *run chain* (put in the RUNNING state).

_hwm = 1;
Causes the task system to keep track of the "high water mark" for each `task`'s stack; that is, the most stack ever used by each `task`. This information is printed by **task::print(STACK)**. This information is intended primarily for debugging purposes, and will affect performance speed. **_hwm** must be set before any `tasks` whose high water marks are of interest are created. Note that two `tasks` are created by a static constructor: the internal Interrupt_alerter `task` and the "main" `task`. If you need accurate information about the high water mark for "main," then **_hwm** must be set by a static constructor which is called before that for the Interrupt_alerter `task`.

Class Object

Class `object` has one form of constructor:

object o;
Construct an `object` object, **o**, which is not on any lists. The constructor takes no arguments.

Public member functions of class `object` are (in alphabetical order):

o.alert()
Changes the state of all `tasks` "remembered" by **o** from IDLE to RUNNING, puts them on the scheduler's *run chain*, and removes them from **o**'s *remember chain*.

o.forget(tp)
Removes all occurrences of the `task` denoted by **tp** from **o**'s *remember chain*.

eo = o.o_type()
Returns the class type of the object, **o** (`object::OBJECT`). **o_type()** is a virtual function.

i = o.pending()
Returns the ready status of an `object`. It returns FALSE if **o** is ready, and TRUE if it is pending. Classes derived from class `object` must define **pending()** if they are to be waited for. **object::pending()** returns TRUE by default. **pending()** is a virtual function.

o.print(i)
Prints the contents of **o** on `stdout`. It is called by the **print()** functions for classes derived from `object`. See **task::print()** for a description of the arguments. **print()** is a virtual function.

o.remember(tp)
Adds the `task` denoted by **tp** to **o**'s *remember chain*. Remembered `tasks` will be alerted when **o**'s state becomes ready.

i = object::task_error(i, op)
i = o.task_error(i, op)
The central error function called by task system functions when a run time error occurs. **i** represents the error number (see the DIAGNOSTICS section for a list of error numbers and their meanings). **op** is meant to be a pointer to the `object` which called **task_error()** or 0. **object::task_error()** examines the variable `error_fct`, and if this variable denotes a function, that function will be called with **i** and **op** as arguments, respectively. (See `error_fct`, below.) Otherwise, **i** will be given as an argument to **print_error()**, which will print an error message on `stderr` and call **exit(i)**, terminating the program. The non-static, single argument form of **task_error()** is obsolete, but

remains for compatibility.

tp = object::this_task()
tp = o.this_task()
Returns a pointer to the `task` that is currently running.

PFIO user-defined-error-function;
error_fct = user-defined-error-function
error_fct is a pointer to a function that returns an `int` and takes two arguments: an `int` representing the error number and an `object*` representing the `object*` that called **task_error**. If **error_fct** is set, **task_error()** will call the **user-defined-error-function** with the error number and the `object*` as arguments. (The `object*` will be 0 if **task_error** was not called by an `object`.) If **user-defined-error-function** does not return 0, **task_error()** will call **exit(i)**. If the **user-defined-error-function** does return 0, **task_error()** will retry the operation that caused the error.

Class Sched

Both class `task` and class `timer` are derived from class `sched`. Class `sched` provides one form of constructor, which is protected:

sched s;
Constructs a `sched` object, **s**, initialized to be IDLE and to have a 0 delay.

Class `sched` is responsible for the functionality that is common to `tasks` and `timers`. Class `sched` provides the following public member functions:

s.cancel(i)
Puts **s** into the TERMINATED state, without suspending the caller (that is, without invoking the scheduler), and sets the result of **s** to be **i**.

i = s.dont_wait()
Returns the number of times **keep_waiting()** has been called, minus the number of times **dont_wait()** has been called (excluding the current call). If these functions are used as intended, the return value represents the number of `objects` that were waiting for external events before the current call. See **keep_waiting()**. See interrupt(3C++) for a description of how `tasks` can wait for external events.

l = sched::get_clock()
l = s.get_clock()
Returns the value of the task system clock.

i = sched::get_exit_status()
i = s.get_exit_status()
Returns the *exit status* of the task program. When a task program terminates normally (that is, **task_error** is not called), the program will call **exit(i)**, where **i** is the value passed by the last caller of **sched::set_exit_status()**.

sp = s.get_priority_sched()
Returns a pointer to a system `task`, `interrupt_alerter`, if a signal that was being waited for has occurred. If no interrupt has occurred, **get_priority_sched()** returns 0.

sp = sched::get_run_chain()
sp = s.get_run_chain()
Returns a pointer to the *run chain*, the linked list of ready `sched` objects (`tasks` and `timers`).

i = s.keep_waiting()
Returns the number of times **keep_waiting()** has been called (not counting the current call), minus the number of times **dont_wait()** has been called. **keep_waiting()** is meant to be called when an `object` that will wait for an external event is created. For example, it is called when an `Interrupt_handler` object is created by the `Interrupt_handler` constructor (see interrupt(3C++)). The inverse function, **dont_wait()**, should be called when such an `object` is deleted. **keep_waiting()** causes the scheduler to keep waiting (not to exit) when there are no runnable `tasks` (because an external event may make an IDLE `task` runnable).

i = s.pending()
Returns FALSE if **s** (`task` or `timer`) is in the TERMINATED state, TRUE otherwise. **pending()** is a virtual function.

s.print(i)
Prints the contents of **s** on `stdout`. It is called by the **print()** functions for classes derived from `sched`. See **task::print()** and **timer::print()** for a description of the arguments. **print()** is a virtual function.

es = s.rdstate()
Returns the state of **s**: RUNNING, IDLE, or TERMINATED.

l = s.rdtime()
Returns the clock time at which **s** is to run.

i = s.result()
Returns the result of **s** (as set by **task::resultis()**, **task::cancel()**, or **sched::cancel()**). If **s** is not yet TERMINATED, the calling `task` will be suspended to wait for **s** to terminate. If a `task` calls **result()** for itself, it will cause a run time error.

sched::setclock(l)
s.setclock(l)
Initializes the system clock to the time given by **l**. Causes a run time error if used more than once.

sched::set_exit_status(i)
s.set_exit_status(i)
Sets the *exit status* of the task program. When a task program terminates normally (that is, **task_error** is not called), the program will call **exit(i)**, where **i** is the value passed by the last caller of **set_exit_status()**.

s.setwho(op)
Is a virtual function defined for `tasks` and `timers`; see its definition for those classes. The argument is meant to be a pointer to the `object` that caused **s** to be alerted.

PFV user-defined-exit-function;
exit_fct = user-defined-exit-function
exit_fct is a pointer to a function taking no arguments and returning void. If set, the task system scheduler will call the **user-defined-exit-function** before the program exits.

clock_task = tp;
Sets **tp** to be a `task` that will be scheduled each time the system clock advances, before any other `tasks`. The **clock_task** must be IDLE when it is resumed by the scheduler. The **clock_task** can suspend itself by calling **task::sleep()** to ensure this.

Class Timer

Class `timer` provides one form of constructor:

timer tm(i);
Constructs a `timer` object, **tm**, and inserts it on the scheduler's *run chain*.

The following public member functions are provided for `timers`:

eo = tm.o_type()
Returns the class type of the object (`object::TIMER`). **o_type()** is a virtual function.

tm.reset(i)
Resets **tm**'s delay to **i**. This makes repeated use of `timers` possible. A `timer` can be reset even when it is TERMINATED.

tm.setwho(op)
Is defined to be null for `timers`. **setwho()** is a virtual function.

tm.print(i)
Prints the contents of **tm** on `stdout`. The argument is ignored. **print()** is a virtual function.

FILES

LIBDIR/libtask.a

NOTES

The task library is supplied only for the following machines: WE32000-series machines (e.g., the AT&T 3B2), AT&T 3B20, AT&T 6386 WGS, Sun-2 and Sun-3, and the VAX. It must be ported to work on other platforms.

WARNINGS

Beware of optimizing compilers that inline constructors for classes derived from class `task`!

Although the task library was engineered to be as free as possible from dependencies on compilation systems and dynamic call chains, it does depend on the existence of stack frames for the `task` constructor and constructors for classes derived from class `task`. If these constructors are inlined by an optimizing compiler, unpredictable behavior will result.

For related reasons, although you must derive a class from class `task` to use the task library, you can only have one level of derivation from class `task`. That is, the system will not work reliably if you derive a class from a class derived from class `task`.

BUGS

DEDICATED tasks are implemented by building task stacks in the free store. Because UNIX System V Release 2 (SVR2) for the WE32000-series machines does not allow stack pointers to point into the free store, DEDICATED `tasks` cannot be used on these machines with SVR2. In such cases, compile the task library with _SHARED_ONLY defined, which will make SHARED the default mode for `tasks`. (Note: it is insufficient to declare all `tasks` as SHARED without compiling a _SHARED_ONLY version of the task library, because there is an internal system `task` (the *interrupt alerter* `task, see interrupt(3C++)`) which is DEDICATED by default.)

UNIX System V Releases 3.1 and 3.2 (SVR3.1 and SVR3.2) for the Intel 386 machine will not call a signal handler when the current `task` is running on a stack in the free store, that is, when the current `task` has a DEDICATED stack. If you need to use the signal handling mechanisms (described on the tasksim(3C++) manual page) on that configuration, you cannot use tasks which have DEDICATED stacks. In this case, compile the task library with _SHARED_ONLY defined, which will make SHARED the default mode for `tasks`.

DIAGNOSTICS

When a task system function encounters a run time error, it calls **object::task_error()**, with one of the following error numbers as an argument. The table below lists the run time errors the task system detects, the associated error messages, and explanations of the errors.

	Error Name	Message	Explanation
1	E_OLINK	"object::delete(): has chain"	Attempt to delete an object which remembers a task.
2	E_ONEXT	"object::delete(): on chain"	Attempt to delete an object which is still on some chain.
3	E_GETEMPTY	"qhead::get(): empty"	Attempt to get from an empty queue in E_MODE.
4	E_PUTOBJ	"qtail::put(): object on other queue"	Attempt to put an object already on some queue.
5	E_PUTFULL	"qtail::put(): full"	Attempt to put to a full queue in E_MODE.
6	E_BACKOBJ	"qhead::putback(): object on other queue"	Attempt to putback an object already on some queue.
7	E_BACKFULL	"qhead::putback(): full"	Attempt to putback to a full queue in E_MODE.
8	E_SETCLOCK	"sched::setclock(): clock!=0"	Clock was non-zero when setclock() was called.
9	E_CLOCKIDLE	"sched::schedule(): clock_task not idle"	The clock_task was not IDLE when the clock was advanced.
10	E_RESTERM	"sched::schedule: terminated"	Attempt to resume a TERMINATED task.
11	E_RESRUN	"sched::schedule: running"	Attempt to resume a RUNNING task.
12	E_NEGTIME	"sched::schedule: clock<0"	Negative argument to delay().
13	E_RESOBJ	"sched::schedule: task or timer on other queue"	Attempt to resume task or timer already on some queue.
14	E_HISTO	"histogram::histogram(): bad arguments"	Bad arguments for histogram constructor.
15	E_STACK	"task::restore(): stack overflow"	Task run time stack overflow.
16	E_STORE	"new: free store exhausted"	No more free store--new() failed.
17	E_TASKMODE	"task::task(): bad mode"	Illegal mode argument for task constructor.
18	E_TASKDEL	"task::~task(): not terminated"	Attempt to delete a non-TERMINATED task.
19	E_TASKPRE	"task::preempt(): not running"	Attempt to preempt a non-RUNNING task.

Error Name	Message	Explanation
20 E_TIMERDEL	"timer::~timer(): not terminated"	Attempt to delete a non-TERMINATED timer.
21 E_SCHTIME	"schedule: bad time"	Scheduler run chain is corrupted: bad time.
22 E_SCHOBJ	"sched object used directly (not as base)"	Sched object used directly instead of as a base class.
23 E_QDEL	"queue::~queue(): not empty"	Attempt to delete a non-empty queue.
24 E_RESULT	"task::result(): thistask->result()"	A task attempted to obtain its own result().
25 E_WAIT	"task::wait(): wait for self"	A task attempted to wait() for itself to TERMINATE.
26 E_FUNCS	"FrameLayout::FrameLayout(): function start"	Internal error--cannot determine the call frame layout.
27 E_FRAMES	"FrameLayout::FrameLayout(): frame size"	Internal error--cannot determine frame size.
28 E_REGMASK	"task::fudge_return(): unexpected register mask"	Internal error--unexpected register mask.
29 E_FUDGE_SIZE	"task::fudge_return(): frame too big"	Internal error--fudged frame too big.
30 E_NO_HNDLR	"sigFunc - no handler for signal"	No handler for the generated signal.
31 E_BADSIG	"illegal signal number"	Attempt to use a signal number that is out of range.
32 E_LOSTHNDLR	"Interrupt_handler::~Interrupt_handler(): signal handler not on chain"	

SEE ALSO

TASK.INTRO(3C++), interrupt(3C++), queue(3C++), tasksim(3C++)

Stroustrup, B. and Shopiro, J. E., "A Set of C++ Classes for Co-routine Style Programming," in *AT&T C++ Language System Release 2.0 Library Manual.*

Shopiro, J. E., "Extending the C++ Task System for Real-Time Control," in *AT&T C++ Language System Release 2.0 Library Manual.*

Keenan, S. A., "A Porting Guide for the C++ Coroutine Library," in *AT&T C++ Language System Release 2.0 Library Manual.*

NAME

tasksim – histograms and random numbers for simulations with C++ tasks

SYNOPSIS

```
#include <task.h>

class histogram {
public:
        int     l, r;
        int     binsize;
        int     nbin;
        int*    h;
        long    sum;
        long    sqsum;
                histogram(int nb =16, int left =0, int right =16);
        void    add(int bin);
        void    print();
};

class randint {
public:
                randint(long seed =0);
        int     draw();
        float   fdraw();
        void    seed(long);
};

class urand : public randint {
public:
        int     low, high;
                urand(int lo, int hi);
        int     draw();
};

class erand : public randint {
public:
        int     mean;
                erand(int m);
        int     draw();
};
```

DESCRIPTION

The C++ task library can be used to program simulations. To support such applications, the library supplies classes to ease data gathering and random number generation.

The public member functions supplied in the task system classes `histogram`, `randint`, `urand`, and `erand` are listed and described in the next two sections. The following symbols are used:

h a `histogram` object

ri a `randint` object

ur a `urand` object

er a `erand` object

i, nb, left, right, lo, hi, m
`ints`

`l` a `long int`

`f` a `float`

Histograms

Class `histogram` provides simple facilities to generate histograms.

Class `histogram` has one form of constructor:

histogram h(nb, left, right);
Constructs a `histogram` object, **h**. A histogram consists of `nbin` bins, `h[0], ... h[nbin-1]`, covering a range `l` to `r` of integers. The optional arguments to the `histogram` constructor correspond to the number of bins (`nbins`), and the left (`l`) and right (`r`) ends of the range, respectively. By default, **nb** is 16, **left** is 0, and **right** is 16, in other words, there are 16 bins covering a range from 0 to 16.

h.add(i)
Adds one to the ith bin. The sum of the integers added is maintained in `sum`, and the sum of their squares is maintained in `sqsum`. If **i** is outside the range `l-r`, the range is extended by either decreasing `l` or increasing `r`. The number of bins however, remains constant, so the size of the range covered by a bin is doubled each time the size of the range is doubled.

h.print()
Prints the numbers of entries for each non-empty bin in **h**.

Random Number Generation

Classes `randint`, `urand`, and `erand` provide basic facilities for generating random numbers, and can serve as a paradigm for other, application-specific generators.

Each object of class `randint` provides an independent sequence of random numbers.

Class `randint` has one form of constructor:

randint ri(l);
Constructs a `randint` object, **ri**. The argument is optional, and defaults to 0. If l is given, it is used to seed **ri**.

i = ri.draw()
Returns an random `int` in the range from 0 to *largest_positive_integer*. Integers returned by **randint::draw()** are uniformly distributed in that range.

f = ri.fdraw()
Returns `floats` that are uniformly distributed in the interval 0 to 1.

ri.seed(l)
Reinitializes a generator with the seed l.

Classes `urand` and `erand` are both derived from class `randint`.

urand ur(lo, hi);
Constructs a `urand` object, **ur**. **lo** and **hi** define the range from `low` to `high` for the distribution of numbers generated by this object.

i = ur.draw()
Returns a random `int` in the range `low` to `high`. Integers returned from **urand::draw()** will be uniformly distributed in the range.

erand er(i)
Constructs an `erand` object, **er**, with **i** as the `mean` for the distribution of random

numbers generated.

i = er.draw()

Returns a random `int`. Integers returned from **erand::draw()** will be exponentially distributed around the `mean`. **erand::draw()** uses **log()** from the C math library, so programs using it must be loaded with `-lm`.

DIAGNOSTICS

See task(3).

SEE ALSO

TASK.INTRO(3C++), task(3C++), interrupt(3C++), queue(3C++)

Stroustrup, B. and Shopiro, J. E., "A Set of C++ Classes for Co-routine Style Programming," in *AT&T C++ Language System Release 2.0 Library Manual.*

Iostream Library Manual Pages

NAME

iostream – buffering, formatting and input/output

SYNOPSIS

```
#include <iostream.h>
class streambuf ;
class ios ;
class istream : virtual public ios ;
class ostream : virtual public ios ;
class iostream : public istream, public ostream ;
class istream_withassign : public istream ;
class ostream_withassign : public ostream ;
class iostream_withassign : public iostream ;

class Iostream_init ;

extern istream_withassign cin ;
extern ostream_withassign cout ;
extern ostream_withassign cerr ;
extern ostream_withassign clog ;

#include <fstream.h>
class filebuf : public streambuf ;
class fstream : public iostream ;
class ifstream : public istream ;
class ofstream : public ostream ;

#include <strstream.h>
class strstreambuf : public streambuf ;
class istrstream : public istream ;
class ostrstream : public ostream ;

#include <stdiostream.h>
class stdiobuf : public streambuf ;
class stdiostream : public ios ;
```

DESCRIPTION

The C++ iostream package declared in `iostream.h` and other header files consists primarily of a collection of classes. Although originally intended only to support input/output, the package now supports related activities such as incore formatting. This package is a mostly source-compatible extension of the earlier stream I/O package, described in *The C++ Programming Language* by Bjarne Stroustrup.

In the iostream man pages, *character* refers to a value that can be held in either a `char` or `unsigned char`. When functions that return an `int` are said to return a character, they return a positive value. Usually such functions can also return `EOF` (-1) as an error indication. The piece of memory that can hold a character is referred to as a *byte*. Thus, either a `char*` or an `unsigned char*` can point to an array of bytes.

The iostream package consists of several core classes, which provide the basic functionality for I/O conversion and buffering, and several specialized classes derived from the core classes. Both groups of classes are listed below.

Core Classes

The core of the iostream package comprises the following classes:

`streambuf`
: This is the base class for buffers. It supports insertion (also known as *storing* or *putting*) and extraction (also known as *fetching* or *getting*) of characters. Most members are inlined for efficiency. The public interface of class `streambuf` is described in *sbuf.pub(3C++)* and the protected interface (for derived classes) is described in *sbuf.prot(3C++)*.

`ios`
: This class contains state variables that are common to the various stream classes, for example, error states and formatting states. See *ios(3C++)*.

`istream`
: This class supports formatted and unformatted conversion from sequences of characters fetched from `streambufs`. See *istream(3C++)*.

`ostream`
: This class supports formatted and unformated conversion to sequences of characters stored into `streambufs`. See *ostream(3C++)*.

`iostream`
: This class combines `istream` and `ostream`. It is intended for situations in which bidirectional operations (inserting into and extracting from a single sequence of characters) are desired. See *ios(3C++)*.

`istream_withassign`
`ostream_withassign`
`iostream_withassign`
: These classes add assignment operators and a constructor with no operands to the corresponding class *without assignment*. The predefined streams (see below) `cin`, `cout`, `cerr`, and `clog`, are objects of these classes. See *istream(3C++)*, *ostream(3C++)*, and *ios(3C++)*.

`Iostream_init`
: This class is present for technical reasons relating to initialization. It has no public members. The `Iostream_init` constructor initializes the predefined streams (listed below). Because an object of this class is declared in the iostream.h header file, the constructor is called once each time the header is included (although the real initialization is only done once), and therefore the predefined streams will be initialized before they are used. In some cases, global constructors may need to call the Iostream_init constructor explicitly to ensure the standard streams are initialized before they are used.

Predefined streams

The following streams are predefined:

`cin` The standard input (file descriptor 0).

`cout` The standard output (file descriptor 1).

`cerr` Standard error (file descriptor 2). Output through this stream is unit-buffered, which means that characters are flushed after each inserter operation. (See **ostream::osfx()** in *ostream(3C++)* and `ios::unitbuf` in *ios(3C++)*.)

`clog` This stream is also directed to file descriptor 2, but unlike `cerr` its output is buffered.

`cin`, `cerr` and `clog` are tied to `cout` so that any use of these will cause `cout` to be flushed.

In addition to the core classes enumerated above, the iostream package contains additional classes derived from them and declared in other headers. Programmers may use these, or may

choose to define their own classes derived from the core iostream classes.

Classes derived from `streambuf`

Classes derived from `streambuf` define the details of how characters are produced or consumed. Derivation of a class from `streambuf` (the *protected interface*) is discussed in *sbuf.prot*(3C++). The available buffer classes are:

`filebuf`
: This buffer class supports I/O through file descriptors. Members support opening, closing, and seeking. Common uses do not require the program to manipulate file descriptors. See *filebuf*(3C++).

`stdiobuf`
: This buffer class supports I/O through stdio `FILE` structs. It is intended for use when mixing C and C++ code. New code should prefer to use `filebufs`. See *stdiobuf*(3C++).

`strstreambuf`
: This buffer class stores and fetches characters from arrays of bytes in memory (i.e., strings). See *ssbuf*(3C++).

Classes derived from `istream`, `ostream`, and `iostream`

Classes derived from `istream`, `ostream`, and `iostream` specialize the core classes for use with particular kinds of `streambufs`. These classes are:

`ifstream`
`ofstream`
`fstream`
: These classes support formatted I/O to and from files. They use a `filebuf` to do the I/O. Common operations (such as opening and closing) can be done directly on streams without explicit mention of `filebufs`. See *fstream*(3C++).

`istrstream`
`ostrstream`
: These classes support "in core" formatting. They use a `strstreambuf`. See *strstream*(3C++).

`stdiostream`
: This class specializes `iostream` for stdio `FILEs`. See *stdiostream.h*.

CAVEATS

Parts of the `streambuf` class of the old stream package that should have been private were public. Most normal usage will compile properly, but any code that depends on details, including classes that were derived from `streambufs` will have to be rewritten.

Performance of programs that copy from `cin` to `cout` may sometimes be improved by breaking the tie between `cin` and `cout` and doing explicit flushes of `cout`.

The header file `stream.h` exists for compatibility with the earlier stream package. It includes `iostream.h`, `stdio.h`, and some other headers, and it declares some obsolete functions, enumerations and variables. Some members of `streambuf` and `ios` (not discussed in these man pages) are present only for backward compatibility with the stream package.

SEE ALSO

ios(3C++), sbuf.pub(3C++), sbuf.prot(3C++), filebuf(3C++), stdiobuf(3C++), ssbuf(3C++), istream(3C++), ostream(3C++), fstream(3C++), strstream(3C++), manip(3C++)

NAME

filebuf – buffer for file I/O.

SYNOPSIS

```
#include <iostream.h>

typedef long streamoff, streampos;
class ios {
public:
                enum            seek_dir { beg, cur, end };
                enum            open_mode { in, out, ate, app, trunc, nocreate, noreplace } ;
                // and lots of other stuff, see ios(3C++) ...
} ;

#include <fstream.h>

class filebuf : public streambuf {
public:
                static const int openprot ; /* default protection for open */

                                filebuf() ;
                                filebuf(int d);
                                filebuf(int d, char*  p, int len) ;

                filebuf*        attach(int d) ;
                filebuf*        close();
                int             fd();
                int             is_open();
                filebuf*        open(char *name, int omode, int prot=openprot) ;
                streampos       seekoff(streamoff, seek_dir, int omode) ;
                streampos       seekpos(streampos, int omode) ;
                streambuf*      setbuf(char* p, int len) ;
                int             sync() ;
};
```

DESCRIPTION

`filebuf`s specialize `streambuf`s to use a file as source or sink of characters. Characters are consumed by doing writes to the file, and are produced by doing reads. When the file is seekable, a `filebuf` allows seeks. At least 4 characters of putback are guaranteed. When the file permits reading and writing, the `filebuf` permits both storing and fetching. No special action is required between gets and puts (unlike stdio). A `filebuf` that is connected to a file descriptor is said to be *open*. Files are opened by default with a protection mode of `openprot`, which is 0644.

The *reserve area* (or buffer, see *sbuf.pub(3C++)* and *sbuf.prot(3C++)*) is allocated automatically if one is not specified explicitly with a constructor or a call to **setbuf()**. `filebuf`s can also be made *unbuffered* with certain arguments to the constructor or **setbuf()**, in which case a system call is made for each character that is read or written. The *get* and *put* pointers into the reserve area are conceptually tied together; they behave as a single pointer. Therefore, the descriptions below refer to a single get/put pointer.

In the descriptions below, assume:
— **f** is a `filebuf`.
— **pfb** is a `filebuf*`.
— **psb** is a `streambuf*`.
— **i**, **d**, **len**, and **prot** are `int`s.

— **name** and **ptr** are `char*`s.
— **mode** is an `int` representing an `open_mode`.
— **off** is a `streamoff`.
— **p** and **pos** are `streampos`'s.
— **dir** is a `seek_dir`.

Constructors:

filebuf()
Constructs an initially closed `filebuf`.

filebuf(d)
Constructs a `filebuf` connected to file descriptor **d**.

filebuf(d, p, len)
Constructs a `filebuf` connected to file descriptor **d** and initialized to use the reserve area starting at **p** and containing **len** bytes. If **p** is null or **len** is zero or less, the `filebuf` will be unbuffered.

Members:

pfb=f.attach(d)
Connects **f** to an open file descriptor, **d**. **attach()** normally returns **&f**, but returns 0 if **f** is already open.

pfb=f.close()
Flushes any waiting output, closes the file descriptor, and disconnects **f**. Unless an error occurs, **f**'s error state will be cleared. **close()** returns **&f** unless errors occur, in which case it returns `0`. Even if errors occur, **close()** leaves the file descriptor and **f** closed.

i=f.fd() Returns **i**, the file descriptor **f** is connected to. If **f** is closed, **i** is `EOF`.

i=f.is_open()
Returns non-zero when **f** is connected to a file descriptor, and zero otherwise.

pfb=f.open(name, mode, prot)
Opens file **name** and connects **f** to it. If the file does not already exist, an attempt is made to create it with protection mode **prot**, unless `ios::nocreate` is specified in **mode**. By default, **prot** is `filebuf::openprot`, which is 0644. Failure occurs if **f** is already open. **open()** normally returns **&f**, but if an error occurs it returns 0. The members of `open_mode` are bits that may be or'ed together. (Because the or'ing returns an `int`, `open()` takes an `int` rather than an `open_mode` argument.) The meanings of these bits in **mode** are described in detail in *fstream*(3C++).

p=f.seekoff(off, dir, mode)
Moves the get/put pointer as designated by **off** and **dir**. It may fail if the file that **f** is attached to does not support seeking, or if the attempted motion is otherwise invalid (such as attempting to seek to a position before the beginning of file). **off** is interpreted as a count relative to the place in the file specified by **dir** as described in *sbuf.pub*(3C++). **mode** is ignored. **seekoff()** returns **p**, the new position, or `EOF` if a failure occurs. The position of the file after a failure is undefined.

p=f.seekpos(pos, mode)
Moves the file to a position **pos** as described in *sbuf.pub*(3C++). **mode** is ignored. **seekpos()** normally returns **pos**, but on failure it returns `EOF`.

psb=f.setbuf(ptr, len)
Sets up the reserve area as **len** bytes beginning at **ptr**. If **ptr** is null or **len** is less than or equal to 0, **f** will be unbuffered. **setbuf()** normally returns **&f**. However, if **f** is open and a buffer has been allocated, no changes are made to the reserve area or to

the **buffering status, and setbuf() returns 0.**

i=f.sync()

Attempts to force the state of the get/put pointer of f to agree (be synchronized) with the state of the file **f.fd()**. This means it may write characters to the file if some have been buffered for output or attempt to reposition (seek) the file if characters have been read and buffered for input. Normally, **sync()** returns 0, but it returns EOF if synchronization is not possible.

Sometimes it is necessary to guarantee that certain characters are written together. To do this, the program should use **setbuf()** (or a constructor) to guarantee that the reserve area is at least as large as the number of characters that must be written together. It can then call **sync()**, then store the characters, then call sync() again.

CAVEATS

attach() and the constructors should test if the file descriptor they are given is open, but I can't figure out a portable way to do that.

There is no way to force atomic reads.

The UNIX system does not usually report failures of seek (e.g. on a tty), so a filebuf does not either.

SEE ALSO

sbuf.pub(3C++), sbuf.prot(3C++), fstream(3C++)

NAME

fstream – iostream and streambuf specialized to files

SYNOPSIS

```
#include <fstream.h>

typedef long streamoff, streampos;
class ios {
public:
                enum        seek_dir { beg, cur, end } ;
                enum        open_mode { in, out, ate, app, trunc, nocreate, noreplace } ;
                enum        io_state { goodbit=0, eofbit, failbit, badbit } ;
                // and lots of other stuff, see ios(3C++) ...
} ;

class ifstream : istream {
                            ifstream() ;
                            ifstream(char* name, int =ios::in, int prot =filebuf::openprot) ;
                            ifstream(int fd) ;
                            ifstream(int fd, char* p, int l) ;

                void        attach(int fd) ;
                void        close() ;
                void        open(char* name, int =ios::in, int prot=filebuf::openprot) ;
                filebuf*    rdbuf() ;
                void        setbuf(char* p, int l) ;
};

class ofstream : ostream {
                            ofstream() ;
                            ofstream(char* name, int =ios::out, int prot =filebuf::openprot) ;
                            ofstream(int fd) ;
                            ofstream(int fd, char* p, int l) ;

                void        attach(int fd) ;
                void        close() ;
                void        open(char* name, int =ios::out, int prot=filebuf::openprot) ;
                filebuf*    rdbuf() ;
                void        setbuf(char* p, int l) ;
};

class fstream : iostream {
                            fstream() ;
                            fstream(char* name, int mode, int prot =filebuf::openprot) ;
                            fstream(int fd) ;
                            fstream(int fd, char* p, int l) ;

                void        attach(int fd) ;
                void        close() ;
                void        open(char* name, int mode, int prot=filebuf::openprot) ;
                filebuf*    rdbuf() ;
                void        setbuf(char* p, int l) ;
};
```

DESCRIPTION

`ifstream`, `ofstream`, and `fstream` specialize `istream`, `ostream`, and `iostream`, respectively, to files. That is, the associated `streambuf` will be a `filebuf`.

In the following descriptions, assume
— **f** is any of `ifstream`, `ofstream`, or `fstream`.
— **pfb** is a `filebuf*`.
— **psb** is a `streambuf*`.
— **name** and **ptr** are `char*`s.
— **i**, **fd**, **len**, and **prot** are `int`s.
— **mode** is an `int` representing an `open_mode`.

Constructors

The constructors for *x*`stream`, where *x* is either `if`, `of` or `f`, are:

***x*stream()**
Constructs an unopened *x*`stream`.

***x*stream(name, mode, prot)**
Constructs an *x*`stream` and opens file **name** using **mode** as the open mode and **prot** as the protection mode. By default, **prot** is `filebuf::openprot`, which is 0644. The error state (`io_state`) of the constructed *x*`stream` will indicate failure in case the **open** fails.

***x*stream(d)**
Constructs an *x*`stream` connected to file descriptor **d**, which must be already open.

***x*stream(d,ptr,len)**
Constructs an *x*`stream` connected to file descriptor **fd**, and, in addition, initializes the associated `filebuf` to use the **len** bytes at **ptr** as the reserve area. If **ptr** is null or **len** is 0, the `filebuf` will be unbuffered.

Member functions

f.attach(d)
Connects **f** to the file descriptor **d**. A failure occurs when **f** is already connected to a file. A failure sets `ios::failbit` in **f**'s error state.

f.close()
Closes any associated `filebuf` and thereby breaks the connection of the **f** to a file. **f**'s error state is cleared except on failure. A failure occurs when the call to **f.rdbuf()->close()** fails.

f.open(name,mode,prot)
Opens file **name** and connects **f** to it. If the file does not already exist, an attempt is made to create it with protection mode **prot** unless `ios::nocreate` is set. By default, **prot** is `filebuf::openprot`, which is 0644. Failure occurs if **f** is already open, or the call to **f.rdbuf()->open()** fails. `ios::failbit` is set in **f**'s error status on failure. The members of `open_mode` are bits that may be or'ed together. (Because the or'ing returns an `int`, `open()` takes an `int` rather than an `open_mode` argument.) The meanings of these bits in **mode** are

`ios::app`
A seek to the end of file is performed. Subsequent data written to the file is always added (appended) at the end of file. On some systems this is implemented in the operating system. In others it is implemented by seeking to the end of the file before each write. `ios::app` implies `ios::out`.

`ios::ate`
A seek to the end of the file is performed during the **open()**. `ios::ate` does not imply `ios::out`.

`ios::in`
The file is opened for input. `ios::in` is implied by construction and opens of `ifstreams`. For `fstreams` it indicates that input operations should be allowed if possible. Is is legal to include `ios::in` in the modes of an `ostream` in which case it implies that the original file (if it exists) should not be truncated.

`ios::out`
The file is opened for output. `ios::out` is implied by construction and opens of `ofstreams`. For `fstream` it says that output operations are to be allowed. `ios::out` may be specified even if **prot** does not permit output.

`ios::trunc`
If the file already exists, its contents will be truncated (discarded). This mode is implied when `ios::out` is specified (including implicit specification for `ofstream`) and neither `ios::ate` nor `ios::app` is specified.

`ios::nocreate`
If the file does not already exist, the **open()** will fail.

`ios::noreplace`
If the file already exists, the **open()** will fail.

pfb=f.rdbuf()
Returns a pointer to the `filebuf` associated with f. **fstream::rdbuf()** has the same meaning as **iostream::rdbuf()** but is typed differently.

psb=f.setbuf(p,len)
Has the usual effect of a **setbuf()** (see *filebuf(3C++)*), offering space for a reserve area or requesting unbuffered I/O. Normally the returned **psb** is **f.rdbuf()**, but it is 0 on failure. A failure occurs if **f** is open or the call to **f.rdbuf()->setbuf** fails.

SEE ALSO
filebuf(3C++), istream(3C++), ios(3C++), ostream(3C++), sbuf.pub(3C++)

NAME

ios – input/output formatting

SYNOPSIS

```
#include <iostream.h>

class ios {
public:
                enum            io_state { goodbit=0, eofbit, failbit, badbit };
                enum            open_mode { in, out, ate, app, trunc, nocreate, noreplace };
                enum            seek_dir { beg, cur, end };
                /* flags for controlling format */
                enum            { skipws=01,
                                  left=02,  right=04, internal=010,
                                  dec=020, oct=040, hex=0100,
                                  showbase=0200, showpoint=0400, uppercase=01000, showpos=02000,
                                  scientific=04000, fixed=010000,
                                  unitbuf=020000, stdio=040000 };
                static const long basefield;
                                /* dec|oct|hex */
                static const long adjustfield;
                                /* left|right|internal */
                static const long floatfield;
                                /* scientific|fixed */
public:
                                ios(streambuf*);

                int             bad();
                static long     bitalloc();
                void            clear(int state =0);
                int             eof();
                int             fail();
                char            fill();
                char            fill(char);
                long            flags();
                long            flags(long);
                int             good();
                long&           iword(int);
                int             operator!();
                                operator void*();
                int             precision();
                int             precision(int);
                streambuf*      rdbuf();
                void* &         pword(int);
                int             rdstate();
                long            setf(long setbits, long field);
                long            setf(long);
                static void     sync_with_stdio();
                ostream*        tie();
                ostream*        tie(ostream*);
                long            unsetf(long);
                int             width();
                int             width(int);
                static int      xalloc();
```

```
protected:
                        ios();
                        init(streambuf*);
private:
                        ios(ios&);
            void        operator=(ios&);
};

            /* Manipulators */
ios&        dec(ios&) ;
ios&        hex(ios&) ;
ios&        oct(ios&) ;
ostream&    endl(ostream& i) ;
ostream&    ends(ostream& i) ;
ostream&    flush(ostream&) ;
istream&    ws(istream&) ;
```

DESCRIPTION

The stream classes derived from class `ios` provide a high level interface that supports transferring formatted and unformatted information into and out of `streambufs`. This manual page describes the operations common to both input and output.

Several enumerations are declared in class `ios`, `open_mode`, `io_state`, `seek_dir`, and format flags, to avoid polluting the global name space. The `io_states` are described on this manual page under "Error States." The format fields are also described on this page, under "Formatting." The `open_modes` are described in detail in *fstream(3C++)* under **open()**. The `seek_dirs` are described in *sbuf.pub(3C++)* under **seekoff()**.

In the following descriptions assume:

- — **s** and **s2** are `ioss`.
- — **sr** is an `ios&`.
- — **sp** is a `ios*`.
- — **i**, **oi** **j**, and **n** are `ints`.
- — **l**, **f**, and **b** are `longs`.
- — **c** and **oc** are `chars`.
- — **osp** and **oosp** are `ostream*`s.
- — **sb** is a `streambuf*`.
- — **pos** is a `streampos`.
- — **off** is a `streamoff`.
- — **dir** is a `seek_dir`.
- — **mode** is an `int` representing an `open_mode`.
- — **fct** is a function with type `ios& (*)(ios&)`.
- — **vp** is a `void*&`.

Constructors and assignment:

ios(sb) The `streambuf` denoted by **sb** becomes the `streambuf` associated with the constructed **ios**. If **sb** is null, the effect is undefined.

ios(sr)
s2=s

Copying of `ioss` is not well-defined in general, therefore the constructor and assignment operators are private so that the compiler will complain about attempts to copy `ios` objects. Copying pointers to `iostreams` is usually what is desired.

ios()

init(sb)

Because class ios is now inherited as a virtual base class, a constructor with no arguments must be used. This constructor is declared protected. Therefore ios::init(streambuf*) is declared protected and must be used for initialization of derived classes.

Error States

An **ios** has an internal *error state* (which is a collection of the bits declared as `io_states`). Members related to the error state are:

i=s.rdstate()

Returns the current error state.

s.clear(i)

Stores **i** as the error state. If **i** is zero, this clears all bits. To set a bit without clearing previously set bits requires something like **s.clear(ios::badbit|s.rdstate())**.

i=s.good()

Returns non-zero if the error state has no bits set, zero otherwise.

i=s.eof()

Returns non-zero if `eofbit` is set in the error state, zero otherwise. Normally this bit is set when an end-of-file has been encountered during an extraction.

i=s.fail()

Returns non-zero if either `badbit` or `failbit` is set in the error state, zero otherwise. Normally this indicates that some extraction or conversion has failed, but the stream is still usable. That is, once the `failbit` is cleared, I/O on **s** can usually continue.

i=s.bad()

Returns non-zero if **badbit** is set in the error state, zero otherwise. This usually indicates that some operation on **s.rdbuf()** has failed, a severe error, from which recovery is probably impossible. That is, it will probably be impossible to continue I/O operations on **s**.

Operators

Two operators are defined to allow convenient checking of the error state of an `ios`: **operator!()** and **operator void*()**. The latter converts an `ios` to a pointer so that it can be compared to zero. The conversion will return 0 if `failbit` or `badbit` is set in the error state, and will return a pointer value otherwise. This pointer is not meant to be used. This allows one to write expressions such as:

```
if ( cin ) ...

if ( cin >> x ) ...
```

The ! operator returns non-zero if `failbit` or `badbit` is set in the error state, which allows expressions like the following to be used:

```
if ( !cout ) ...
```

Formatting

An `ios` has a *format state* that is used by input and output operations to control the details of formatting operations. For other operations the format state has no particular effect and its components may be set and examined arbitrarily by user code. Most formatting details are

controlled by using the **flags()**, **setf()**, and **unsetf()** functions to set the following flags, which are declared in an enumeration in class `ios`. Three other components of the format state are controlled separately with the functions **fill()**, **width()**, and **precision()**.

`skipws`
: If `skipws` is set, whitespace will be skipped on input. This applies to scalar extractions. When `skipws` is not set, whitespace is not skipped before the extractor begins conversion. As a precaution against looping, zero width fields are considered a bad format by the extractors, so if the next character is whitespace and the skip variable is not set, the arithmetic extractors will signal an error.

`left`
`right`
`internal`
: These flags control the padding of a value. When `left` is set, the value is left-adjusted, that is, the fill character is added after the value. When `right` is set, the value is right-adjusted, that is, the fill character is added before the value. When `internal` is set, the fill character is added after any leading sign or base indication, but before the value. Right-adjustment is the default if none of these flags is set. These fields are collectively identified by the static member, `ios::adjustfield`. The fill character is controlled by the fill() function, and the width of padding is controlled by the **width()** function.

`dec`
`oct`
`hex`
: These flags control the conversion base of a value. The conversion base is 10 (decimal) if `dec` is set, but if `oct` or `hex` is set, conversions are done in octal or hexidecimal, respectively. If none of these is set, insertions are in decimal, but extractions are interpreted according to the C++ lexical conventions for integral constants. These fields are collectively identified by the static member, `ios::basefield`. The manipulators **hex**, **dec**, and **oct**, can also be used to set the conversion base, see "Built-in Manipulators" below.

`showbase`
: If `showbase` is set, insertions will be converted to an external form that can be read according to the C++ lexical conventions for integral constants. `showbase` is unset by default.

`showpos`
: If `showpos` is set, then a "+" will be inserted into a decimal conversion of a postive integral value.

`uppercase`
: If `uppercase` is set, then an uppercase "X" will be used for hexadecimal conversion when `showbase` is set, or an uppercase "E" will be used to print floating point numbers in scientific notation.

`showpoint`
: If `showpoint` is set, trailing zeros and decimal points appear in the result of a floating point conversion.

`scientific`
`fixed`
: These flags control the format to which a floating point value is converted for

insertion into a stream. If `scientific` is set, the value is converted using scientific notation, where there is one digit before the decimal point and the number of digits after it is equal to the *precision* (see below), which is six by default. An uppercase "E" will introduce the exponent if `uppercase` is set, a lowercase "e" will appear otherwise. If `fixed` is set, the value is converted to decimal notation with *precision* digits after the decimal point, or six by default. If neither `scientific` nor `fixed` is set, then the value will be converted using either notation, depending on the value: scientific notation will be used only if the exponent resulting from the conversion is less that -4 or greater than the precision. If `showpoint` is not set, trailing zeroes are removed from the result and a decimal point appears only if it is followed by a digit. `scientific` and `fixed` are collectively identified by the static member, `ios::floatfield`.

`unitbuf`
When set, a flush is performed by **ostream::osfx()** after each insertion. Unit buffering provides a compromise between buffered output and unbuffered output. Performance is better under unit buffering than unbuffered output, which makes a system call for each character output. Unit buffering makes a system call for each insertion operation, and doesn't require the user to call **ostream::flush()**.

`stdio` When set, `stdout` and `stderr` are flushed by **ostream::osfx()** after each insertion.

The following functions use and set the format flags and variables.

oc=s.fill(c)
Sets the "fill character" format state variable to **c** and returns the previous value. **c** will be used as the padding character, if one is necessary (see **width()**, below). The default fill or padding character is a space. The positioning of the fill character is determined by the `right`, `left`, `internal` flags, see above. A parameterized manipulator, **setfill** is also available for setting the fill character, see *manip(3C++)*.

c=s.fill()
Returns the "fill character" format state variable.

l=s.flags()
Returns the current format flags.

l=s.flags(f)
Resets all the format flags to those specified in **f** and returns the previous settings.

oi=s.precision(i)
Sets the "precision" format state variable to **i** and returns the previous value. This variable controls the number of significant digits inserted by the floating point inserter. The default is 6. A parameterized manipulator, **setprecision** is also available for setting the precision, see *manip(3C++)*.

i=s.precision()
Returns the "precision" format state variable.

l=s.setf(b)
Turns on in **s** the format flags marked in **b** and returns the previous settings. A parameterized manipulator, **setiosflags** performs the same function, see *manip(3C++)*.

l=s.setf(b,f)
Resets in **s** only the format flags specified by **f** to the settings marked in **b**, and returns the previous settings. That is, the format flags specified by **f** are cleared in **s**, then reset to be those marked in **b**. For example, to change the conversion base in **s** to be `hex`, one could write: **s.setf(ios::hex,ios::basefield)** `ios::basefield` specifies the conversion base bits as candidates for change, and `ios::hex` specifies the new value. **s.setf(0,f)** will clear all the bits specified by **f**, as will a parameterized manipulator, **resetiosflags**, see *manip(3C++)*.

l=s.unsetf(b)
Unsets in **s** the bits set in **b** and returns the previous settings.

oi=s.width(i)
Sets the "field width" format variable to **i** and returns the previous value. When the field width is zero (the default), inserters will insert only as many characters as necessary to represent the value being inserted. When the field width is non-zero, the inserters will insert at least that many characters, using the fill character to pad the value, if the value being inserted requires fewer than field-width characters, to be represented. However, the numeric inserters never truncate values, so if the value being inserted will not fit in field-width characters, more than field-width characters will be output. The field width is always interpreted as a mininum number of characters; there is no direct way to specify a maximum number of characters. The field width format variable is reset to the default (zero) after each insertion or extraction, and in this sense it behaves as a parameter for insertions and extractions. A parameterized manipulator, **setw** is also available for setting the width, see *manip(3C++)*.

i=s.width()
Returns the "field width" format variable.

User-defined Format Flags
Several functions are provided to allow users to derive classes from class `ios` that require additional format flags or variables. The two static member functions **ios::xalloc** and **ios::bitalloc**, allow several such classes to be used together without interference.

b=ios::bitalloc()
Returns a `long` with a single, previously unallocated, bit set. This allows users who need an additional flag to acquire one, and pass it as an argument to **ios::setf()**, for example.

i=ios::xalloc()
Returns a previously unused index into an array of words available for use as format state variables by derived classes.

l=s.iword(i)
When **i** is an index allocated by **ios::xalloc**, **iword()** returns a reference to the ith user-defined word.

vp=s.pword(i)
When **i** is an index allocated by **ios::xalloc**, **pword()** returns a reference to the ith user-defined word. **pword()** is the same as **iword** except that it is typed differently.

Other members:

sb=s.rdbuf()
Returns a pointer to the `streambuf` associated with **s** when **s** was constructed.

ios::sync_with_stdio()
Solves problems that arise when mixing stdio and iostreams. The first time it is called it will reset the standard iostreams (`cin`, `cout`, `cerr`, `clog`) to be streams using `stdiobufs`. After that input and output using these streams may be mixed with input and output using the corresponding `FILE`s (`stdin`, `stdout`, and `stderr`) and will be properly synchronized. **sync_with_stdio()** makes `cout` and `cerr` unit buffered (see `ios::unitbuf` and `ios::stdio` above). Invoking **sync_with_stdio()** degrades performance a variable amount, depending on the length of the strings being inserted (shorter strings incur a larger performance hit).

oosp=s.tie(osp)
Sets the "tie" variable to **osp**, and returns its previous value. This variable supports automatic "flushing" of `ioss`. If the tie variable is non-null and an `ios` needs more characters or has characters to be consumed, the `ios` pointed at by the tie variable is flushed. By default, `cin` is tied initially to `cout` so that attempts to get more characters from standard input result in flushing standard output. Additionally, `cerr` and `clog` are tied to `cout` by default. For other `ioss`, the tie variable is set to zero by default.

osp=s.tie()
Returns the "tie" variable.

Built-in Manipulators:
Some convenient manipulators (functions that take an `ios&`, an `istream&`, or an `ostream&` and return their argument, see *manip(3C++)*) are:

sr<<dec
sr>>dec
These set the conversion base format flag to 10.

sr<<hex
sr>>hex
These set the conversion base format flag to 16.

sr<<oct
sr>>oct
These set the conversion base format flag to 8.

sr>>ws Extracts whitespace characters. See *istream(3C++)*.

sr<<endl
Ends a line by inserting a newline character and flushing. See *ostream(3C++)*.

sr<<ends
Ends a string by inserting a null(0) character. See *ostream(3C++)*.

sr<<flush
Flushes **outs**. See *ostream(3C++)*.

Several parameterized manipulators that operate on `ios` objects are described in *manip(3C++)*: **setw, setfill, setprecision, setiosflags,** and **resetiosflags.**

The `streambuf` associated with an `ios` may be manipulated by other methods than through the `ios`. For example, characters may be stored in a queuelike `streambuf` through an `ostream` while they are being fetched through an `istream`. Or for efficiency some part of a program may choose to do `streambuf` operations directly rather than through the `ios`. In most cases the program does not have to worry about this possibility, because an `ios` never saves information about the internal state of a `streambuf`. For example, if the `streambuf` is repositioned between extraction operations the extraction (input) will proceed normally.

CAVEATS

The need for **sync_with_stdio** is a wart. The old stream package did this as a default, but in the iostream package unbuffered `stdiobufs` are too inefficient to be the default.

The stream package had a constructor that took a **FILE*** argument. This is now replaced by `stdiostream`. It is not declared even as an obsolete form to avoid having `iostream.h` depend on `stdio.h`.

The old stream package allowed copying of streams. This is disallowed by the iostream package. However, objects of type `istream_withassign`, `ostream_withassign`, and `iostream_withassign` can be assigned to. Old code using copying can usually be rewritten to use pointers or these classes. (The standard streams `cin`, `cout`, `cerr`, and `clog` are members of "withassign" classes, so they can be assigned to, as in `cin = inputfstream`.)

SEE ALSO

IOS.INTRO(3C++), streambuf(3C++), istream(3C++), ostream(3C++), manip(3C++).

NAME

istream – formatted and unformatted input

SYNOPSIS

```
#include <iostream.h>

typedef long streamoff, streampos;
class ios {
public:
                enum            seek_dir { beg, cur, end };
                enum            open_mode { in, out, ate, app, trunc, nocreate, noreplace } ;
                /* flags for controlling format */
                enum            { skipws=01,
                                  left=02,  right=04, internal=010,
                                  dec=020, oct=040, hex=0100,
                                  showbase=0200, showpoint=0400, uppercase=01000, showpos=02000,
                                  scientific=04000, fixed=010000,
                                  unitbuf=020000, stdio=040000 };
                // and lots of other stuff, see ios(3C++) ...
} ;

class istream : public ios {
public:
                                istream(streambuf*);
                int             gcount();
                istream&        get(char* ptr, int len, char delim='\n');
                istream&        get(unsigned char* ptr,int len, char delim='\n');

                istream&        get(unsigned char&);
                istream&        get(char&);
                istream&        get(streambuf& sb, char delim ='\n');
                int             get();
                istream&        getline(char* ptr, int len, char delim='\n');
                istream&        getline(unsigned char* ptr, int len, char delim='\n');
                istream&        ignore(int len=1,int delim=EOF);
                int             ipfx(int need=0);
                int             peek();
                istream&        putback(char);
                istream&        read(char* s, int n);
                istream&        read(unsigned char* s, int n);
                istream&        seekg(streampos);
                istream&        seekg(streamoff, seek_dir);
                int             sync();
                streampos       tellg();

                istream&        operator>>(char*);
                istream&        operator>>(char&);
                istream&        operator>>(short&);
                istream&        operator>>(int&);
                istream&        operator>>(long&);
                istream&        operator>>(float&);
                istream&        operator>>(double&);
                istream&        operator>>(unsigned char*);
                istream&        operator>>(unsigned char&);
```

```
                istream&        operator>>(unsigned short&);
                istream&        operator>>(unsigned int&);
                istream&        operator>>(unsigned long&);
                istream&        operator>>(streambuf*);
                istream&        operator>>(istream& (*)(istream&));
                istream&        operator>>(ios& (*)(ios&));
};

class istream_withassign : public istream {
                                istream_withassign();
                istream&        operator=(istream&);
                istream&        operator=(streambuf*);
};

extern istream_withassign cin;

istream&        ws(istream&)  ;
ios&            dec(ios&) ;
ios&            hex(ios&) ;
ios&            oct(ios&) ;
```

DESCRIPTION

`istreams` support interpretation of characters fetched from an associated `streambuf`. These are commonly referred to as input or extraction operations. The `istream` member functions and related functions are described below.

In the following descriptions assume that
— **ins** is an `istream`.
— **inwa** is an `istream_withassign`.
— **insp** is a `istream*`.
— **c** is a `char&`
— **delim** is a `char`.
— **ptr** is a `char*` or `unsigned char*`.
— **sb** is a `streambuf&`.
— **i, n, len, d,** and **need** are `int`s.
— **pos** is a `streampos`.
— **off** is a `streamoff`.
— **dir** is a `seek_dir`.
— **manip** is a function with type `istream& (*)(istream&)`.

Constructors and assignment:

istream(sb)
Initializes **ios** state variables and associates buffer **sb** with the `istream`..

istream_withassign()
Does no initialization.

inswa=sb
Associates **sb** with **inswa** and initializes the entire state of **inswa**.

inswa=ins
Associates **ins->rdbuf()** with **inswa** and initializes the entire state of **inswa**.

Input prefix function:

i = ins.ipfx(need)
If **ins**'s error state is non-zero, returns zero immediately. If necessary (and if it is non-null), any `ios` tied to **ins** is flushed (see the description **ios::tie()** in

ios(3C++)). Flushing is considered necessary if either **need==0** or if there are fewer than **need** characters immediately available. If `ios::skipws` is set in **ins.flags()** and **need** is zero, then leading whitespace characters are extracted from **ins**. **ipfx()** returns zero if an error occurs while skipping whitespace; otherwise it returns non-zero.

Formatted input functions call **ipfx(0)**, while unformatted input functions call **ipfx(1)**; see below.

Formatted input functions (extractors):

ins>>x Calls **ipfx(0)** and if that returns non-zero, extracts characters from **ins** and converts them according to the type of **x**. It stores the converted value in **x**. Errors are indicated by setting the error state of **ins**. `ios::failbit` means that characters in **ins** were not a representation of the required type. `ios::badbit` indicates that attempts to extract characters failed. **ins** is always returned.

The details of conversion depend on the values of ins's format state flags and variables (see *ios*(3C++)) and the type of **x**. Except that extractions that use width reset it to 0, the extraction operators do not change the value of `ostream`'s format state. Extractors are defined for the following types, with conversion rules as described below.

`char*, unsigned char*`
: Characters are stored in the array pointed at by **x** until a whitespace character is found in **ins**. The terminating whitespace is left in **ins**. If **ins.width()** is non-zero it is taken to be the size of the array, and no more than **ins.width()-1** characters are extracted. A terminating null character (0) is always stored (even when nothing else is done because of ins's error status). **ins.width()** is reset to 0.

`char&, unsigned char&`
: A character is extracted and stored in **x**.

`short&, unsigned short&,`
`int&, unsigned int&,`
`long&, unsigned long&`
: Characters are extracted and converted to an integral value according to the conversion specified in **ins**'s format flags. Converted characters are stored in **x**. The first character may be a sign (+ or −). After that, if `ios::oct`, `ios::dec`, or `ios::hex` is set in **ins.flags()**, the conversion is octal, decimal, or hexadecimal respectively. Conversion is terminated by the first "non-digit," which is left in **ins**. Octal digits are the characters '0' to '7'. Decimal digits are the octal digits plus '8' and '9'. Hexadecimal digits are the decimal digits plus the letters 'a' through 'f' (in either upper or lower case). If none of the conversion base format flags is set, then the number is interpreted according to C++ lexical conventions. That is, if the first characters (after the optional sign) are `0x` or `0X` a hexadecimal conversion is performed on following hexadecimal digits. Otherwise, if the first character is a `0`, an octal conversion is performed, and in all other cases a decimal conversion is performed. `ios::failbit` is set if there are no digits (not counting the `0` in `0x` or `0X`) during hex conversion) available.

`float&, double&`
: Converts the characters according to C++ syntax for a float or double, and stores the result in **x**. `ios::failbit` is set if there are no digits

available **in** **ins** or if it does not begin with a well formed floating point number.

The type and name(`operator>>`) of the extraction operations are chosen to give a convenient syntax for sequences of input operations. The operator overloading of C++ permits extraction functions to be declared for user-defined classes. These operations can then be used with the same syntax as the member functions described here.

ins>>sb

If **ios.ipfx(0)** returns non-zero, extracts characters from **ios** and inserts them into **sb**. Extraction stops when EOF is reached. Always returns **ins**.

Unformatted input functions:

These functions call **ipfx(1)** and proceed only if it returns non-zero:

insp=&ins.get(ptr,len,delim)

Extracts characters and stores them in the byte array beginning at **ptr** and extending for **len** bytes. Extraction stops when **delim** is encountered (**delim** is left in **ins** and not stored), when **ins** has no more characters, or when the array has only one byte left. **get** always stores a terminating null, even if it doesn't extract any characters from **ins** because of its error status. `ios::failbit` is set only if **get** encounters an end of file before it stores any characters.

insp=&ins.get(c)

Extracts a single character and stores it in **c**.

insp=&ins.get(sb,delim)

Extracts characters from **ins.rdbuf()** and stores them into **sb**. It stops if it encounters end of file or if a store into **sb** fails or if it encounters **delim** (which it leaves in **ins**). `ios::failbit` is set if it stops because the store into **sb** fails

i=ins.get().

Extracts a character and returns it. **i** is `EOF` if extraction encounters end of file. `ios::failbit` is never set.

insp=&ins.getline(ptr,len,delim)

Does the same thing as **ins.get(ptr,len,delim)** with the exception that it extracts a terminating **delim** character from **ins**. In case **delim** occurs when exactly **len** characters have been extracted, termination is treated as being due to the array being filled, and this **delim** is left in **ins**.

insp=&ins.ignore(n,d)

Extracts and throws away up to **n** characters. Extraction stops prematurely if **d** is extracted or end of file is reached. If **d** is `EOF` it can never cause termination.

insp=&ins.read(ptr,n)

Extracts **n** characters and stores them in the array beginning at **ptr**. If end of file is reached before **n** characters have been extracted, **read** stores whatever it can extract and sets `ios::failbit`. The number of characters extracted can be determined via **ins.gcount()**.

Other members are:

i=ins.gcount()

Returns the number of characters extracted by the last unformatted input function. Formatted input functions may call unformatted input functions and thereby reset this number.

i=ins.peek()

Begins by calling **ins.ipfx(1)**. If that call returns zero or if **ins** is at end of file,

it returns `EOF`. Otherwise it returns the next character without extracting it.

insp=&ins.putback(c)
Attempts to back up **ins.rdbuf()**. **c** must be the character before **ins.rdbuf()**'s get pointer. (Unless other activity is modifying **ins.rdbuf()** this is the last character extracted from **ins**.) If it is not, the effect is undefined. **putback** may fail (and set the error state). Although it is a member of `istream`, **putback** never extracts characters, so it does not call **ipfx**. It will, however, return without doing anything if the error state is non-zero.

i=&ins.sync()
Establishes consistency between internal data structures and the external source of characters. Calls **ins.rdbuf()->sync()**, which is a virtual function, so the details depend on the derived class. Returns EOF to indicate errors.

ins>>manip
Equivalent to **manip(ins)**. Syntactically this looks like an extractor operation, but semantically it does an arbitrary operation rather than converting a sequence of characters and storing the result in **manip**. A predefined manipulator, **ws**, is described below.

Member functions related to positioning:

insp=&ins.seekg(off,dir)
Repositions **ins.rdbuf()**'s get pointer. See *sbuf.pub*(3C++) for a discussion of positioning.

insp=&ins.seekg(pos)
Repositions **ins.rdbuf()**'s get pointer. See *sbuf.pub*(3C++) for a discussion of positioning.

pos=ins.tellg()
The current position of **ios.rdbuf()**'s get pointer. See *sbuf.pub*(3C++) for a discussion of positioning.

Manipulator:

ins>>ws
Extracts whitespace characters.

ins>>dec
Sets the conversion base format flag to 10. See *ios*(3C++).

ins>>hex
Sets the conversion base format flag to 16. See *ios*(3C++).

ins>>oct
Sets the conversion base format flag to 8. See *ios*(3C++).

CAVEATS

There is no overflow detection on conversion of integers. There should be, and overflow should cause the error state to be set.

SEE ALSO

ios(3C++), sbuf.pub(3C++), manip(3C++)

NAME

manipulators – iostream out of band manipulations

SYNOPSIS

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

IOMANIPdeclare(T) ;

class SMANIP(T) {
                SMANIP(T)( ios& (*)(ios&,T), T);
                friend          istream& operator>>(istream&, SMANIP(T)&);
                friend          ostream& operator<<(ostream&, SMANIP(T)&);
};
class SAPP(T) {
                SAPP(T)( ios& (*)(ios&,T));
                SMANIP(T) operator()(T);
};
class IMANIP(T) {
                IMANIP(T)( istream& (*)(istream&,T), T);
                friend          istream& operator>>(istream&, IMANIP(T)&);
};
class IAPP(T) {
                IAPP(T)( istream& (*)(istream&,T));
                IMANIP(T) operator()(T);
};
class OMANIP(T) {
                OMANIP(T)( ostream& (*)(ostream&,T), T);
                friend          ostream& operator<<(ostream&, OMANIP(T)&);
};
class OAPP(T) {
                OAPP(T)( ostream& (*)(ostream&,T));
                OMANIP(T) operator()(T);
};
class IOMANIP(T) {
                IOMANIP(T)( iostream& (*)(iostream&,T), T);
                friend          istream& operator>>(iostream&, IOMANIP(T)&);
                friend          ostream& operator<<(iostream&, IOMANIP(T)&);
};
class IOAPP(T) {
                IOAPP(T)( iostream& (*)(iostream&,T));
                IOMANIP(T) operator()(T);
};

IOMANIPdeclare(int);
IOMANIPdeclare(long);

SMANIP(long)  resetiosflags(long);
SMANIP(int)   setfill(int);
SMANIP(long)  setiosflags(long);
SMANIP(int)   setprecision(int);
SMANIP(int)   setw(int w);
```

DESCRIPTION

Manipulators are values that may be "inserted into" or "extracted from" streams to achieve some effect (other than to insert or extract a value representation), with a convenient syntax. They enable one to embed a function call in an expression containing series of insertions or extractions. For example, the predefined manipulator for `ostreams`, **flush**, can be used as follows:

cout << flush

to flush `cout`. Several iostream classes supply manipulators, see *ios(3C++)*, *istream(3C++)*, and *ostream(3C++)*. **flush** is a simple manipulator; some manipulators take arguments, such as the predefined `ios` manipulators, **setfill** and **setw** (see below). The header file `iomanip.h` supplies macro definitions which programmers can use to define new parameterized manipulators.

Ideally, the types relating to manipulators would be parameterized as "templates." The macros defined in `iomanip.h` are used to simulate templates. **IOMANIPdeclare(T)** declares the various classes and operators. (All code is declared inline so that no separate definitions are required.) Each of the other **T**s is used to construct the real names and therefore must be a single identifer. Each of the other macros also requires an identifier and expands to a name.

In the following descriptions, assume:
— **t** is a **T**, or type name.
— **s** is an `ios`.
— **i** is an `istream`.
— **o** is an `ostream`.
— **io** is an `iostream`.
— **f** is an `ios& (*)(ios&)`.
— **if** is an `istream& (*)(istream&)`.
— **of** is an `ostream& (*)(ostream&)`.
— **iof** is an `iostream& (*)(iostream&)`.
— **n** is an `int`.
— **l** is a `long`.

s<<SMANIP(T)(f,t)
s>>SMANIP(T)(f,t)
s<<SAPP(T)(f)(t)
s>>SAPP(T)(f)(t)
Returns **f(s,t)**, where **s** is the left operand of the insertion or extractor operator (i.e., **s**, **i**, **o**, or **io**).

i>>IMANIP(T)(if,t)
i>>IAPP(T)(if)(t)
Returns **if(i,t)**.

o<<OMANIP(T)(of,t)
o<<OAPP(T)(of)(t)
Returns **of(o,t)**.

io<<IOMANIP(T)(iof,t)
io>>IOMANIP(T)(iof,t)
io<<IOAPP(T)(iof)(t)
io>>IOAPP(T)(iof)(t)
Returns **iof(io,t)**.

`iomanip.h` contains two declarations, `IOMANIPdeclare(int)` and `IOMANIPdeclare(long)` and some manipulators that take an `int` or a `long` argument. These manipulators all have to do with changing the format state of a stream, see *ios(3C++)* for further details.

o<<setw(n)
i>>setw(n)
Sets the field width of the stream (left-hand operand: **o** or **i**) to **n**.

o<<setfill(n)
i>>setfill(n)
Sets the fill character of the stream (**o** or **i**, or) to be **n**.

o<<setprecision(n)
i>>setprecision(n)
Sets the precision of the stream (**o** or **i**) to be **n**.

o<<setiosflags(l)
i>>setiosflags(l)
Turns on in the stream (**o** or **i**) the format flags marked in **l**. (Calls **o.setf(l)** or **i.setf(l)**).

o<<resetiosflags(l)
i>>resetiosflags(l)
Clears in the stream (**o** or **i**) the format bits specified by **l**. (Calls **o.setf(0,l)** or **i.setf(0,l)**).

CAVEATS

Syntax errors will be reported if **IOMANIPdeclare(T)** occurs more than once in a file with the same **T**.

SEE ALSO

ios(3C++), istream(3C++), ostream(3C++)

NAME

ostream – formatted and unformatted output

SYNOPSIS

```
#include <iostream.h>

typedef long streamoff, streampos;
class ios {
public:
                enum            seek_dir { beg, cur, end };
                enum            open_mode { in, out, ate, app, trunc, nocreate, noreplace } ;
                enum            { skipws=01,
                                  left=02,  right=04, internal=010,
                                  dec=020, oct=040, hex=0100,
                                  showbase=0200, showpoint=0400, uppercase=01000, showpos=02000,
                                  scientific=04000, fixed=010000,
                                  unitbuf=020000, stdio=040000 };
                // and lots of other stuff, see ios(3C++) ...
} ;

class ostream : public ios {
public:
                                ostream(streambuf*);
                ostream&        flush();
                int             opfx();
                ostream&        put(char);
                ostream&        seekp(streampos);
                ostream&        seekp(streamoff, seek_dir);
                streampos       tellp();
                ostream&        write(const char* ptr, int n);
                ostream&        write(const unsigned char* ptr, int n);
                ostream&        operator<<(const char*);
                ostream&        operator<<(char);
                ostream&        operator<<(short);
                ostream&        operator<<(int);
                ostream&        operator<<(long);
                ostream&        operator<<(float);
                ostream&        operator<<(double);
                ostream&        operator<<(unsigned char);
                ostream&        operator<<(unsigned short);
                ostream&        operator<<(unsigned int);
                ostream&        operator<<(unsigned long);
                ostream&        operator<<(void*);
                ostream&        operator<<(streambuf*);
                ostream&        operator<<(ostream& (*)(ostream&));
                ostream&        operator<<(ios& (*)(ios&));
};

class ostream_withassign {
                                ostream_withassign();
                istream&        operator=(istream&);
                istream&        operator=(streambuf*);
};
```

```
extern ostream_withassign cout;
extern ostream_withassign cerr;
extern ostream_withassign clog;

ostream&        endl(ostream&) ;
ostream&        ends(ostream&) ;
ostream&        flush(ostream&) ;
ios&            dec(ios&) ;
ios&            hex(ios&) ;
ios&            oct(ios&) ;
```

DESCRIPTION

`ostreams` support insertion (storing) into a `streambuf`. These are commonly referred to as output operations. The `ostream` member functions and related functions are described below.

In the following descriptions, assume:
— **outs** is an `ostream`.
— **outswa** is an `ostream_withassign`.
— **outsp** is an `ostream*`.
— **c** is a `char`.
— **ptr** is a `char*` or `unsigned char*`.
— **sb** is a `streambuf*`
— **i** and **n** are `int`s.
— **pos** is a `streampos`.
— **off** is a `streamoff`.
— **dir** is a `seek_dir`.
— **manip** is a function with type `ostream& (*)(ostream&)`.

Constructors and assignment:

ostream(sb)
Initializes **ios** state variables and associates buffer **sb** with the `ostream`.

ostream_withassign()
Does no initialization. This allows a file static variable of this type (**cout** for example) to be used before it is constructed, provided it is assigned to first.

outswa=sb
Associates **sb** with **swa** and initializes the entire state of **outswa**.

inswa=ins
Associates **ins->rdbuf()** with **swa** and initializes the entire state of **outswa**.

Output prefix function:

i=outs.opfx()
If **outs**'s error state is nonzero, returns immediately. If **outs.tie()** is non-null, it is flushed. Returns non-zero except when **outs**'s error state is nonzero.

Output suffix function:

osfx() Performs "suffix" actions before returning from inserters. If `ios::unitbuf` is set, **osfx()** flushes the `ostream`. If `ios::stdio` is set, **osfx()** flushes `stdout` and `stderr`.

osfx() is called by all predefined inserters, and should be called by user-defined inserters as well, after any direct manipulation of the `streambuf`. It is not called by the binary output functions.

Formatted output functions (inserters):

outs<<x

First calls **outs.opfx()** and if that returns 0, does nothing. Otherwise inserts a sequence of characters representing x into **outs.rdbuf()**. Errors are indicated by setting the error state of **outs**. **outs** is always returned.

x is converted into a sequence of characters (its representation) according to rules that depend on x's type and **outs**'s format state flags and variables (see *ios*(3C++)): Inserters are defined for the following types, with conversion rules as described below.

`char*` The representation is the sequence of characters up to (but not including) the terminating null of the string x points at.

any integral type except `char` and `unsigned char`
If x is positive the representation contains a sequence of decimal, octal, or hexadecimal digits with no leading zeros according to whether `ios::dec`, `ios::oct`, or `ios::hex`, respectively is set in **ios**'s format flags. If none of those flags are set, conversion defaults to decimal. If x is zero, the representation is a single zero character(`0`). If x is negative, decimal conversion converts it to a minus sign (-) followed by decimal digits. If x is positive and `ios::showpos` is set, decimal conversion converts it to a plus sign (+) followed by decimal digits. The other conversions treat all values as unsigned. If `ios::showbase` is set in **ios**'s format flags, the hexadecimal representation contains `0x` before the hexidecimal digits, or `0X` if `ios::uppercase` is set. If `ios::showbase` is set, the octal representation contains a leading 0.

`void*` Pointers are converted to integral values and then converted to hexadecimal numbers as if `ios::showbase` were set.

`float, double`
The arguments are converted according to the current values of **outs.precision()**, **outs.width()** and **outs**'s format flags `ios::scientific`, `ios::fixed`, and `ios::uppercase`. (See *ios(3C++).*) The default value for **outs.precision()** is 6. If neither `ios::scientific` nor `ios::fixed` is set, either fixed or scientific notation is chosen for the representation, depending on the value of x.

`char, unsigned char`
No special conversion is necessary.

After the representation is determined, padding occurs. If **outs.width()** is greater than 0 and the representation contains fewer than **outs.width()** characters, then enough **outs.fill()** characters are added to bring the total number of characters to **ios.width()**. If `ios::left` is set in **ios**'s format flags, the sequence is left-adjusted, that is, characters are added after the characters determined above. If `ios::right` is set, the padding is added before the characters determined above. If `ios::internal` is set, the padding is added after any leading sign or base indication and before the characters that represent the value. **ios.width()** is reset to 0, but all other format variables are unchanged. The resulting sequence (padding plus representation) is inserted into **outs.rdbuf()**.

outs<<sb

If **outs.opfx()** returns non-zero, the sequence of characters that can be fetched from **sb** are inserted into **outs.rdbuf()**. Insertion stops when no more characters can be fetched from **sb**. No padding is performed. Always returns **outs**.

Unformatted output functions:

outsp=&outs.put(c)
Inserts **c** into **outs.rdbuf()**. Sets the error state if the insertion fails.

outsp=&outs.write(s,n)
Inserts the **n** characters starting at **s** into **outs.rdbuf()**. These characters may include zeros (i.e., **s** need not be a null terminated string).

Other member functions:

outsp=&outs.flush()
storing characters into a streambuf does not always cause them to be consumed (e.g., written to the external file) immediately. **flush()** causes any characters that may have been stored but not yet consumed to be consumed by calling **outs.rdbuf()->sync.**

outs<<manip
Equivalent to **manip(outs)**. Syntactically this looks like an insertion operation, but semantically it does an arbitrary operations rather than converting **manip** to a sequence of characters as do the insertion operators. Predefined manipulators are described below.

Positioning functions:

outsp=&ins.seekp(off,dir)
Repositions **outs.rdbuf()**'s put pointer. See *sbuf.pub*(3C++) for a discussion of positioning.

outsp=&outs.seekp(pos)
Repositions **outs.rdbuf()**'s put pointer. See *sbuf.pub*(3C++) for a discussion of positioning.

pos=outs.tellp()
The current position of **outs.rdbuf()**'s put pointer. See *sbuf.pub*(3C++) for a discussion of positioning.

Manipulators:

outs<<endl
Ends a line by inserting a newline character and flushing.

outs<<ends
Ends a string by inserting a null(0) character.

outs<<flush
Flushes **outs**.

outs<<dec
Sets the conversion base format flag to 10. See *ios*(3C++).

outs<<hex
Sets the conversion base format flag to 16. See *ios*(3C++).

outs<<oct
Sets the conversion base format flag to 8. See *ios*(3C++).

SEE ALSO
ios(3C++), sbuf.pub(3C++), manip(3C++)

NAME

streambuf – interface for derived classes

SYNOPSIS

```
#include <iostream.h>

typedef long streamoff, streampos;
class ios {
public:
                enum            seek_dir { beg, cur, end };
                enum            open_mode { in, out, ate, app, trunc, nocreate, noreplace } ;
                // and lots of other stuff, see ios(3C++) ...
} ;

class streambuf {
public:
                                streambuf() ;
                                streambuf(char* p, int len);
                void            dbp() ;
protected:
                int             allocate();
                char*           base();
                int             blen();
                char*           eback();
                char*           ebuf();
                char*           egptr();
                char*           epptr();
                void            gbump(int n);
                char*           gptr();
                char*           pbase();
                void            pbump(int n);
                char*           pptr();
                void            setg(char* eb, char* g, char* eg);
                void            setp(char* p, char* ep);
                void            setb(char* b, char* eb, int a=0);
                int             unbuffered();
                void            unbuffered(int);

                virtual int     doallocate();
                virtual         ~streambuf() ;
public:
                virtual int     pbackfail(int c);
                virtual int     overflow(int c=EOF);
                virtual int     underflow();
                virtual streambuf*
                                setbuf(char* p, int len);
                virtual streampos
                                seekpos(streampos, int =ios::in|ios:out);
                virtual streampos
                                seekoff(streamoff, seek_dir, int =ios::in|ios:out);
                virtual int     sync();
};
```

DESCRIPTION

`streambuf`s implement the buffer abstraction described in *sbuf.pub*(3C++). However, the `streambuf` class itself contains only basic members for manipulating the characters and normally a class derived from `streambuf` will be used. This man page describes the interface needed by programmers who are coding a derived class. Broadly speaking there are two kinds of member functions described here. The non-virtual functions are provided for manipulating a `streambuf` in ways that are appropriate in a derived class. Their descriptions reveal details of the implementation that would be inappropriate in the public interface. The virtual functions permit the derived class to specialize the `streambuf` class in ways appropriate to the specific sources and sinks that it is implementing. The descriptions of the virtual functions explain the obligations of the virtuals of the derived class. If the virtuals behave as specified, the `streambuf` will behave as specified in the public interface. However, if the virtuals do not behave as specified, then the `streambuf` may not behave properly, and an `iostream` (or any other code) that relies on proper behavior of the `streambuf` may not behave properly either.

In the following descriptions assume:
— **sb** is a `streambuf*`.
— **i** and **n** are `int`s.
— **ptr**, **b**, **eb**, **p**, **ep**, **eb**, **g**, and **eg** are `char*`s.
— **c** is an `int` character (positive or `EOF`)).
— **pos** is a `streampos`. (See *sbuf.pub*(3C++).)
— **off** is a `streamoff`.
— **dir** is a `seekdir`.
— **mode** is an `int` representing an `open_mode`.

Constructors:

streambuf()
Constructs an empty buffer corresponding to an empty sequence.

streambuf(b,len)
Constructs an empty buffer and then sets up the reserve area to be the **len** bytes starting at **b**.

The Get, Put, and Reserver area

The protected members of `streambuf` present an interface to derived classes organized around three areas (arrays of bytes) managed cooperatively by the base and derived classes. They are the *get area*, the *put area*, and the *reserve area* (or buffer). The get and the put areas are normally disjoint, but they may both overlap the reserve area, whose primary purpose is to be a resource in which space for the put and get areas can be allocated. The get and the put areas are changed as characters are put into and gotten from the buffer, but the reserve area normally remains fixed. The areas are defined by a collection of `char*` values. The buffer abstraction is described in terms of pointers that point between characters, but the `char*` values must point at `char`s. To establish a correspondence the `char*` values should be thought of as pointing just before the byte they really point at.

Functions to examine the pointers

ptr=sb->base()
Returns a pointer to the first byte of the reserve area. Space between **sb->base()** and **sb->ebuf()** is the reserve area.

ptr=sb->eback()
Returns a pointer to a lower bound on **sb->gptr()**. Space between **sb->eback()** and **sb->gptr()** is available for putback.

ptr=sb->ebuf()
Returns a pointer to the byte after the last byte of the reserve area.

ptr=sb->egptr()
Returns a pointer to the byte after the last byte of the get area.

ptr=sb->epptr()
Returns a pointer to the byte after the last byte of the put area.

ptr=sb->gptr()
Returns a pointer to the first byte of the get area. The available characters are those between **sb->gptr()** and **sb->egptr()**. The next character fetched will be ***(sb->gptr())** unless **sb->egptr()** is less than or equal to sb->gptr().

ptr=sb->pbase()
Returns a pointer to the put area base. Characters between **sb->pbase()** and **sb->pptr()** have been stored into the buffer and not yet consumed.

ptr=sb->pptr()
Returns a pointer to the first byte of the put area. The space between **sb->pptr()** and **sb->epptr()** is the put area and characters will be stored here.

Functions for setting the pointers

Note that to indicate that a particular area (get, put, or reserve) does not exist, all the associated pointers should be set to zero.

sb->setb(b, eb, i)
Sets **base()** and **ebuf()** to **b** and **eb** respectively. **i** controls whether the area will be subject to automatic deletion. If **i** is non-zero, then **b** will be deleted when **base** is changed by another call of **setb()**, or when the destructor is called for ***sb**. If **b** and **eb** are both null then we say that there is no reserve area. If **b** is non-null, there is a reserve area even if **eb** is less than **b** and so the reserve area has zero length.

sb->setp(p, ep)
Sets **pptr()** to **p**, **pbase()** to **p**, and **epptr()** to **ep**.

sb->setg(eb, g, eg)
Sets **eback()** to **eb**, **gptr()** to **g**, and **egptr()** to **eg**.

Other non-virtual members

i=sb->allocate()
Tries to set up a reserve area. If a reserve area already exists or if **sb->unbuffered()** is nonzero, **allocate()** returns 0 without doing anything. If the attempt to allocate space fails, **allocate()** returns `EOF`, otherwise (allocation succeeds) **allocate()** returns 1. **allocate()** is not called by any non-virtual member function of `streambuf`.

i=sb->blen()
Returns the size (in `chars`) of the current reserve area.

dbp() Writes directly on file descriptor 1 information in ASCII about the state of the buffer. It is intended for debugging and nothing is specified about the form of the output. It is considered part of the protected interface because the information it prints can only be understood in relation to that interface, but it is a public function so that it can be called anywhere during debugging.

sb->gbump(n)
Increments **gptr()** by **n** which may be positive or negative. No checks are made on whether the new value of **gptr()** is in bounds.

sb->pbump(n)
Increments **pptr()** by **n** which may be positive or negative. No checks are made on whether the new value of **pptr()** is in bounds.

sb->unbuffered(i)
i=sb->unbuffered()

There is a private variable known as sb's buffering state. **sb->unbuffered(i)** sets the value of this variable to **i** and **sb->unbuffered()** returns the current value. This state is independent of the actual allocation of a reserve area. Its primary purpose is to control whether a reserve area is allocated automatically by **allocate.**

Virtual member functions

Virtual functions may be redefined in derived classes to specialize the behavior of `streambufs`. This section describes the behavior that these virtual functions should have in any derived classes; the next section describes the behavior that these functions are defined to have in base class `streambuf`.

i=sb->doallocate()

Is called when **allocate()** determines that space is needed. **doallocate()** is required to call **setb()** to provide a reserve area or to return `EOF` if it cannot. It is only called if **sb->unbuffered()** is zero and **sb->base()** is zero.

i=overflow(c)

Is called to consume characters. If **c** is not `EOF`, **overflow()** also must either save **c** or consume it. Usually it is called when the put area is full and an attempt is being made to store a new character, but it can be called at other times. The normal action is to consume the characters between **pbase()** and **pptr()**, call **setp()** to establish a new put area, and if `c!=EOF` store it (using **sputc()**). **sb->overflow()** should return **EOF** to indicate an error; otherwise it should return something else.

i=sb->pbackfail(c)

Is called when **eback()** equals **gptr()** and an attempt has been made to putback **c**. If this situation can be dealt with (e.g., by repositioning an external file), **pbackfail()** should return **c**; otherwise it should return `EOF`.

pos=sb->seekoff(off, dir, mode)

Repositions the get and/or put pointers (i.e., the abstract get and put pointers, not **pptr()** and **gptr()**). The meanings of **off** and **dir** are discussed in *sbuf.pub*(3C++). **mode** specifies whether the put pointer (`ios::out` bit set) or the get pointer (`ios::in` bit set) is to be modified. Both bits may be set in which case both pointers should be affected. A class derived from `streambuf` is not required to support repositioning. **seekoff()** should return `EOF` if the class does not support repositioning. If the class does support repositioning, **seekoff()** should return the new position or `EOF` on error.

pos=sb->seekpos(pos, mode)

Repositions the `streambuf` get and/or put pointer to **pos**. **mode** specifies which pointers are affected as for **seekoff()**. Returns **pos** (the argument) or `EOF` if the class does not support repositioning or an error occurs.

sb=sb->setbuf(ptr, len)

Offers the array at **ptr** with **len** bytes to be used as a reserve area. The normal interpretation is that if **ptr** or **len** are zero then this is a request to make the **sb** unbuffered. The derived class may use this area or not as it chooses. If may accept or ignore the request for unbuffered state as it chooses. **setbuf()** should return **sb** if it honors the request. Otherwise it should return 0.

i=sb->sync()

Is called to give the derived class a chance to look at the state of the areas, and synchronize them with any external representation. Normally **sync()** should

consume any characters that have been stored into the put area, and if possible give back to the source any characters in the get area that have not been fetched. When sync() returns there should not be any unconsumed characters, and the get area should be empty. sync() should return EOF if some kind of failure occurs.

i=sb->underflow()
Is called to supply characters for fetching, i.e., to create a condition in which the get area is not empty. If it is called when there are characters in the get area it should return the first character. If the get area is empty, it should create a nonempty get area and return the next character (which it should also leave in the get area). If there are no more characters available, underflow() should return `EOF` and leave an empty get area.

The default definitions of the virtual functions:

i=sb->streambuf::doallocate()
Attempts to allocate a reserve area using `operator new`.

i=sb->streambuf::overflow(c)
Is compatible with the old stream package, but that behavior is not considered part of the specification of the iostream package. Therefore, **streambuf::overflow()** should be treated as if it had undefined behavior. That is, derived classes should always define it.

i=sb->streambuf::pbackfail(c)
Returns `EOF`.

pos=sb->streambuf::seekpos(pos, mode)
Returns **sb->seekoff(streamoff(pos),ios::beg,mode)**. Thus to define seeking in a derived class, it is frequently only necessary to define **seekoff()** and use the inherited **streambuf::seekpos()**.

pos=sb->streambuf::seekoff(off, dir, mode)
Returns `EOF`.

sb=sb->streambuf::setbuf(ptr, len)
Will honor the request when there is no reserve area.

i=sb->streambuf::sync()
Returns 0 if the get area is empty and there are no unconsumed characters. Otherwise it returns `EOF`.

i=sb->streambuf::underflow()
Is compatible with the old stream package, but that behavior is not considered part of the specification of the iostream package. Therefore, **streambuf::underflow()** should be treated as if it had undefined behavior. That is, it should always be defined in derived classes.

CAVEATS

The constructors are public for compatibility with the old stream package. They ought to be protected.

The interface for unbuffered actions is awkward. It's hard to write **underflow()** and **overflow()** virtuals that behave properly for unbuffered `streambuf()`s without special casing. Also there is no way for the virtuals to react sensibly to multi-character gets or puts.

Although the public interface to `streambufs` deals in characters and bytes, the interface to derived classes deals in `chars`. Since a decision had to be made on the types of the real data pointers, it seemed easier to reflect that choice in the types of the protected members than to duplicate all the members with both plain and unsigned char versions. But perhaps all these

uses of `char*` ought to have been with a typedef.

The implementation contains a variant of **setbuf()** that accepts a third argument. It is present only for compatibility with the old stream package.

SEE ALSO

sbuf.pub(3C++), streambuf(3C++), iostream(3C++)

NAME

streambuf – public interface of character buffering class

SYNOPSIS

```
#include <iostream.h>

typedef long streamoff, streampos;
class ios {
public:
        enum        seek_dir { beg, cur, end };
        enum        open_mode { in, out, ate, app, trunc, nocreate, noreplace } ;
        // and lots of other stuff ... See ios(3C++)
} ;

class streambuf {
public :

        int         in_avail();
        int         out_waiting();
        int         sbumpc();
        streambuf*  setbuf(char* ptr, int len);
        streampos   seekpos(streampos, int =ios::in|ios::out);
        streampos   seekoff(streamoff, seek_dir, int =ios::in|ios::out);
        int         sgetc();
        int         sgetn(char* ptr, int n);
        int         snextc();
        int         sputbackc(char);
        int         sputc(int c);
        int         sputn(const char* s, int n);
        void        stossc();
        virtual int sync();
};
```

DESCRIPTION

The `streambuf` class supports buffers into which characters can be inserted (put) or from which characters can be fetched (gotten). Abstractly, such a buffer is a sequence of characters together with one or two pointers (a get and/or a put pointer) that define the location at which characters are to be inserted or fetched. The pointers should be thought of as pointing between characters rather than at them. This makes it easier to understand the boundary conditions (a pointer before the first character or after the last). Some of the effects of getting and putting are defined by this class but most of the details are left to specialized classes derived from `streambuf`. (See *filebuf(3C++)*, *ssbuf(3C++)*, and *stdiobuf(3C++)*.)

Classes derived from `streambuf` vary in their treatments of the get and put pointers. The simplest are unidirectional buffers which permit only gets or only puts. Such classes serve as pure sources (producers) or sinks (consumers) of characters. Queuelike buffers (e.g., see strstream(3C++) and ssbuf(3C++)) have a put and a get pointer which move independently of each other. In such buffers characters that are stored are held (i.e., queued) until they are later fetched. Filelike buffers (e.g., `filebuf`, see *filebuf(3C++)*) permit both gets and puts but have only a single pointer. (An alternative description is that the get and put pointers are tied together so that when one moves so does the other.)

Most `streambuf` member functions are organized into two phases. As far as possible, operations are performed inline by storing into or fetching from arrays (the *get area* and the *put area*, which together form the *reserve area*, or *buffer*). From time to time, virtual functions are called to deal with collections of characters in the get and put areas. That is, the virtual functions are

called to fetch more characters from the ultimate producer or to flush a collection of characters to the ultimate consumer. Generally the user of a `streambuf` does not have to know anything about these details, but some of the public members pass back information about the state of the areas. Further detail about these areas is provided in *sbuf.prot*(3C++), which describes the protected interface.

The public member functions of the `streambuf` class are described below. In the following descriptions assume:
— **i, n,** and **len** are `int`s.
— **c** is an `int`. It always holds a "character" value or `EOF`. A "character" value is always positive even when `char` is normally sign extended.
— **sb** and **sb1** are `streambuf*`s.
— **ptr** is a `char*`.
— **off** is a `streamoff`.
— **pos** is a `streampos`.
— **dir** is a `seek_dir`.
— **mode** is an `int` representing an `open_mode`.

Public member functions:

i=sb->in_avail()
Returns the number of characters that are immediately available in the get area for fetching. **i** characters may be fetched with a guarantee that no errors will be reported.

i=sb->out_waiting()
Returns the number of characters in the put area that have not been consumed (by the ultimate consumer).

c=sb->sbumpc()
Moves the get pointer forward one character and returns the character it moved past. Returns `EOF` if the get pointer is currently at the end of the sequence.

pos=sb->seekoff(off, dir, mode)
Repositions the get and/or put pointers. **mode** specifies whether the put pointer (`ios::out` bit set) or the get pointer (`ios::in` bit set) is to be modified. Both bits may be set in which case both pointers should be affected. **off** is interpreted as a byte offset. (Notice that it is a signed quantity.) The meanings of possible values of **dir** are

`ios::beg`
The beginning of the stream.

`ios::cur`
The current position.

`ios::end`
The end of the stream (end of file.)

Not all classes derived from `streambuf` support repositioning. **seekoff()** will return `EOF` if the class does not support repositioning. If the class does support repositioning, **seekoff()** will return the new position or `EOF` on error.

pos=sb->seekpos(pos, mode)
Repositions the streambuf get and/or put pointer to **pos**. **mode** specifies which pointers are affected as for **seekoff()**. Returns **pos** (the argument) or `EOF` if the class does not support repositioning or an error occurs. In general a `streampos` should be treated as a "magic cookie" and no arithmetic should be performed on it. Two particular values have special meaning:

streampos(0)
The beginning of the file.

streampos(EOF)
Used as an error indication.

c=sb->sgetc()
Returns the character after the get pointer. Contrary to what most people expect from the name *IT DOES NOT MOVE THE GET POINTER*. Returns EOF if there is no character available.

sb1=sb->setbuf(ptr, len, i)
Offers the **len** bytes starting at **ptr** as the reserve area. If **ptr** is null or **len** is zero or less, then an unbuffered state is requested. Whether the offered area is used, or a request for unbuffered state is honored depends on details of the derived class. **setbuf()** normally returns **sb**, but if it does not accept the offer or honor the request, it returns 0.

i=sb->sgetn(ptr, n)
Fetches the **n** characters following the get pointer and copies them to the area starting at **ptr**. When there are fewer than **n** characters left before the end of the sequence **sgetn()** fetches whatever characters remain. **sgetn()** repositions the get pointer following the fetched characters and returns the number of characters fetched.

c=sb->snextc()
Moves the get pointer forward one character and returns the character following the new position. It returns EOF if the pointer is currently at the end of the sequence or is at the end of the sequence after moving forward.

i=sb->sputbackc(c)
Moves the get pointer back one character. **c** must be the current content of the sequence just before the get pointer. The underlying mechanism may simply back up the get pointer or may rearrange its internal data structures so the **c** is saved. Thus the effect of **sputbackc()** is undefined if **c** is not the character before the get pointer. **sputbackc()** returns EOF when it fails. The conditions under which it can fail depend on the details of the derived class.

i=sb->sputc(c)
Stores **c** after the put pointer, and moves the put pointer past the stored character; usually this extends the sequence. It returns **EOF** when an error occurs. The conditions that can cause errors depend on the derived class.

i=sb->sputn(ptr, n)
Stores the **n** characters starting at **ptr** after the put pointer and moves the put pointer past them. **sputn()** returns **i**, the number of characters stored successfully. Normally **i** is **n**, but it may be less when errors occur.

sb->stossc()
Moves the get pointer forward one character. If the pointer started at the end of the sequence this function has no effect.

i=sb->sync()
Establishes consistency between the internal data structures and the external source or sink. The details of this function depend on the derived class. Usually this "flushes" any characters that have been stored but not yet consumed, and "gives back" any characters that may have been produced but not yet fetched. **sync()** returns EOF to indicate errors.

CAVEATS

setbuf does not really belong in the public interface. It is there for compatibility with the stream package.

Requiring the program to provide the previously fetched character to `sputback` is probably a botch.

SEE ALSO

iostream(3C++), sbuf.prot(3C++)

NAME

strstreambuf – streambuf specialized to arrays

SYNOPSIS

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

class strstreambuf : public streambuf {
public:
                        strstreambuf() ;
                        strstreambuf(char*, int, char*);
                        strstreambuf(int);
                        strstreambuf(unsigned char*, int, unsigned char*);
                        strstreambuf(void* (*a)(long), void(*f)(void*));

        void            freeze(int n=1) ;
        char*           str();
        streambuf*      setbuf(char*, int)
};
```

DESCRIPTION

A `strstreambuf` is a `streambuf` that uses an array of bytes (a string) to hold the sequence of characters. Given the convention that a `char*` should be interpreted as pointing just before the `char` it really points at, the mapping between the abstract get/put pointers (see *sbuf.pub(3C++)*) and `char*` pointers is direct. Moving the pointers corresponds exactly to incrementing and decrementing the `char*` values.

To accommodate the need for arbitrary length strings `strstreambuf` supports a dynamic mode. When a `strstreambuf` is in dynamic mode, space for the character sequence is allocated as needed. When the sequence is extended too far, it will be copied to a new array.

In the following descriptions assume:
— **ssb** is a `strstreambuf*`.
— **n** is an `int`.
— **ptr** and **pstart** are `char*`s or `unsigned char*`s.
— **a** is a `void* (*)(long)`.
— **f** is a `void* (*)(void*)`.

Constructors::

strstreambuf()

Constructs an empty `strstreambuf` in dynamic mode. This means that space will be automatically allocated to accomodate the characters that are put into the `strstreambuf` (using operators `new` and `delete`). Because this may require copying the original characters, it is recommended that when many characters will be inserted, the program should use **setbuf()** (described below) to inform the `strstreambuf`.

strstreambuf(a, f)

Constructs an empty `strstreambuf` in dynamic mode. **a** is used as the allocator function in dynamic mode. The argument passed to **a** will be a `long` denoting the number of bytes to be allocated. If **a** is null, `operator new` will be used. **f** is used to free (or delete) areas returned by **a**. The argument to **f** will be a pointer to the array allocated by **a**. If **f** is null, `operator delete` is used.

strstreambuf(n)

Constructs an empty `strstreambuf` in dynamic mode. The initial allocation

of space will be at least **n** bytes.

strstreambuf(ptr, n, pstart)

Constructs a `strstreambuf` to use the bytes starting at **ptr**. The `strstreambuf` will be in static mode; it will not grow dynamically. If **n** is positive, then the **n** bytes starting at **ptr** are used as the `strstreambuf`. If **n** is zero, **ptr** is assumed to point to the beginning of a null terminated string and the bytes of that string (not including the terminating null character) will constitute the `strstreambuf`. If **n** is negative, the `strstreambuf` is assumed to continue indefinitely. The get pointer is initialized to **ptr**. The put pointer is initialized to **pstart**. If **pstart** is null, then stores will be treated as errors. If **pstart** is non-null, then the initial sequence for fetching (the get area) consists of the bytes between **ptr** and **pstart**. If **pstart** is null, then the initial get area consists of the entire array.

Member functions:

ssb->freeze(n)

Inhibits (when **n** is nonzero) or permits (when **n** is zero) automatic deletion of the current array. Deletion normally occurs when more space is needed or when **ssb** is being destroyed. Only space obtained via dynamic allocation is ever freed. It is an error (and the effect is undefined) to store characters into a `strstreambuf` that was in dynamic allocation mode and is now frozen. It is possible, however, to thaw (unfreeze) such a `strstreambuf` and resume storing characters.

ptr=ssb->str()

Returns a pointer to the first `char` of the current array and freezes **ssb**. If **ssb** was constructed with an explicit array, **ptr** will point to that array. If **ssb** is in dynamic allocation mode, but nothing has yet been stored, **ptr** may be null.

ssb->setbuf(0,n)

ssb remembers **n** and the next time it does a dynamic mode allocation, it makes sure that at least **n** bytes are allocated.

SEE ALSO

sbuf.pub(3C++), strstream(3C++)

NAME

stdiobuf – iostream specialized to stdio FILE

SYNOPSIS

```
#include <iostream.h>
#include <stdiostream.h>
#include <stdio.h>

class stdiobuf : public streambuf {
                                stdiobuf(FILE* f);
                FILE*           stdiofile();
};
```

DESCRIPTION

Operations on a `stdiobuf` are reflected on the associated `FILE`. A `stdiobuf` is constructed in unbuffered mode, which causes all operations to be reflected immediately in the `FILE`. **seekg()**s and **seekp()**s are translated into **fseek()**s. **setbuf()** has its usual meaning; if it supplies a reserve area, buffering will be turned back on.

CAVEATS

`stdiobuf` is intended to be used when mixing C and C++ code. New C++ code should prefer to use `filebuf`s, which have better performance.

SEE ALSO

filebuf(3C++), istream(3C++), ostream(3C++), sbuf.pub(3C++)

NAME

strstream – iostream specialized to arrays

SYNOPSIS

```
#include <strstream.h>

class ios {
public:
                enum            open_mode { in, out, ate, app, trunc, nocreate, noreplace } ;
                // and lots of other stuff, see ios(3C++) ...
} ;

class istrstream : public istream {
public:
                                istrstream(char*) ;
                                istrstream(char*, int) ;
                strstreambuf*   rdbuf() ;
} ;

class ostrstream : public ostream {
public:
                                ostrstream();
                                ostrstream(char*, int, int=ios::out) ;
                int             pcount() ;
                strstreambuf*   rdbuf() ;
                char*           str();
};

class strstream : public strstreambase, public iostream {
public:
                                strstream();
                                strstream(char*, int, int mode);
                strstreambuf*   rdbuf() ;
                char*           str();
};
```

DESCRIPTION

`strstream` specializes `iostream` for "incore" operations, that is, storing and fetching from arrays of bytes. The `streambuf` associated with a `strstream` is a `strstreambuf` (see *ssbuf(3C++)*).

In the following descriptions assume:
— **ss** is a `strstream`.
— **iss** is an `istrstream`.
— **oss** is an `ostrstream`.
— **cp** is a `char*`.
— **mode** is an `int` representing an `open_mode`.
— **i** and **len** are `int`s.
— **ssb** is a `strstreambuf*`.

Constructors

istrstream(cp)

Characters will be fetched from the (null-terminated) string **cp**. The terminating null character will not be part of the sequence. Seeks (**istream::seekg()**) are allowed within that space.

istrstream(cp, len)
Characters will be fetched from the array beginning at **cp** and extending for **len** bytes. Seeks (**istream::seekg()**) are allowed anywhere within that array.

ostrstream()
Space will be dynamically allocated to hold stored characters.

ostrstream(cp,n,mode)
Characters will be stored into the array starting at **cp** and continuing for **n** bytes. If `ios::ate` or `ios::app` is set in **mode**, **cp** is assumed to be a null-terminated string and storing will begin at the null character. Otherwise storing will begin at **cp**. Seeks are allowed anywhere in the array.

strstream()
Space will be dynamically allocated to hold stored characters.

strstream(cp,n,mode)
Characters will be stored into the array starting at **cp** and continuing for **n** bytes. If `ios::ate` or `ios::app` is set in **mode**, **cp** is assumed to be a null-terminated string and storing will begin at the null character. Otherwise storing will begin at **cp**. Seeks are allowed anywhere in the array.

`istrstream` **members**

ssb = iss.rdbuf()
Returns the `strstreambuf` associated with **iss**.

`ostrstream` **members**

ssb = oss.rdbuf()
Returns the `strstreambuf` associated with **oss**.

cp=oss.str()
Returns a pointer to the array being used and "freezes" the array. Once **str** has been called the effect of storing more characters into **oss** is undefined. If **oss** was constructed with an explicit array, **cp** is just a pointer to the array. Otherwise, **cp** points to a dynamically allocated area. Until **str** is called, deleting the dynamically allocated area is the responsibility of **oss**. After **str** returns, the array becomes the responsibility of the user program.

i=oss.pcount()
Returns the number of bytes that have been stored into the buffer. This is mainly of use when binary data has been stored and **oss.str()** does not point to a null terminated string.

`strstream` **members**

ssb = ss.rdbuf()
Returns the `strstreambuf` associated with **ss**.

cp=ss.str()
Returns a pointer to the array being used and "freezes" the array. Once **str** has been called the effect of storing more characters into **ss** is undefined. If **ss** was constructed with an explicit array, **cp** is just a pointer to the array. Otherwise, **cp** points to a dynamically allocated area. Until **str** is called, deleting the dynamically allocated area is the responsibility of **ss**. After **str** returns, the array becomes the responsibility of the user program.

SEE ALSO

strstreambuf(3C++), iostream(3C++)

Index

Index

G

H

I

L

M

N

O

P

Q

R

S

T

U

W

X

Z