English

Fujitsu Software BS2000

C++ Library Functions

Reference Manual

Valid for:
C++ V4.0

Edition June 2023

Comments… Suggestions… Corrections…

The User Documentation Department would like to know your opinion on this manual. Your feedback helps us to optimize our documentation to suit your individual needs.

Feel free to send us your comments by e-mail to: bs2000.info@fujitsu.com.

Certified documentation according to DIN EN ISO 9001:2015

To ensure a consistently high quality standard and user-friendliness, this documentation was created to meet the regulations of a quality management system which complies with the requirements of the standard DIN EN ISO 9001:2015.

Copyright and Trademarks

Table of Contents

C++ Library Functions

1 Preface

This manual describes all the classes, functions and operators provided under the BS2000 operating system by the C++ standard library (C++ V2.1) for complex math and stream-oriented input/output. The C++ standard library is part of the CRTE Common Runtime Environment.

Familiarity with the C and C++ programming languages and the BS2000 operating
system is an essential requirement for using this manual effectively.

The chapter "Using the C++ library functions" contains general information onthe C++ SYSLNK.CRTE.CPP library and the link between the C++ library and the C runtime system.

Chapters "Complex math classes and functions"C and "Classes and functions for input/output" provide a detailed description of the C++ library functions.
The structure of these chapters reflects the usual structure of SINIX C++ library descriptions. Thus the same section headers are used (under SINIX C++, these correspond to the section names of the man pages).

In the body of the text, references to other publications are made using abbreviated titles; the full titles are listed in the "References" section at the back of the manual.
Notes on how to order manuals are given at the end of the same section.

The more important books and manuals are:

The "C++ User Guide", which describes how to compile, link and start a C++ program using the C++(BS2000) V2.1 compiler and other BS2000 operating system components. This includes a description of the use of the CRTE libraries in the compilation and linkage phases for a program.

The "C Library Functions" manual, which describes all the C functions and macros provided by the C runtime system. In addition, this manual contains information concerning file processing, buffering, etc. which is of importance for C++ I/O as well (internally, C++ I/O is handled by the C runtime system, see also "Relationship between the C++ library and the C runtime system") .

The "CRTE User Guide", which contains information on the concept and use of the Common Runtime Environment which includes, among other things, the C, C++, COBOL85 and ILCS runtime systems.

The second edition of "The C++ Programming Language" by Bjarne Stroustrup, whichdescribes the C++-specific language scope of the C++(BS2000) V2.1 compiler.

2 Using the C++ library functions

This chapter provides information concerning the following topics:

- The CRTE SYSLNK.CRTE.CPP library
- Relationship between the C++ library and the C runtime system

2.1 The CRTE SYSLNK.CRTE.CPP library

The SYSLNK.CRTE.CPP library provided with CRTE V1.0 contains the following:

- Standard header elements for the C++ library functions (type S)

complex.h	stdiostream.h
iomanip.h	generic.h
fstream.h	new.h
iostream.h	strstream.h

  These header (or include) files are included in the program with the preprocessor directive #include during compilation. For a detailed description, please refer to your C++ User Guide.

- C++ library function modules (LLMs, type L)
  These modules contain the code for all C++ library functions for complex math and standard I/O.
  Either the modules are permanently (statically) linked to the C++ program using BINDER or they are linked dynamically using DBL. For a detailed description, please refer to your C++ User Guide.

2.2 Relationship between the C++ library and the C runtime system

The C runtime system included in the CRTE SYSLNK.CRTE library is a prerequisite for the use of the C++ library functions. The library containing the C runtime system must be specified both for compiling and for linking a C++ program using the C++ library functions (please refer to your C++ User Guide).

Functions such as the C++ standard I/O functions are actually implemented by internal calls to various C runtime system input/output routines.

Except for record-oriented input/output, the C++ input/output functions can perform any file access method that can be performed by the C library functions.
Whenever there are differences between KR and ANSI functionality in the C runtime system, ANSI functionality is always applicable to the execution of C++ library functions.

Please refer to the "C Library Functions" manual for more detailed information on file processing, ANSI functionality, buffering, etc.

3 Complex math classes and functions

This chapter provides information concerning the following topics:

3.1 cplxintro Introduction to complex mathematics in C++

This section contains an overview of the classes, functions and operators provided by the C++ complex math library.

#include <complex.h>
class complex;

Declarations for all complex math operators and functions are contained in the <complex.h> header file.

The data type for complex numbers is implemented as a class named *complex*.

Overloaded versions of the following operators and math functions are available for processing complex numbers:

- standard input/output operators and arithmetic, assignment, and comparison operators; see section "cplxops Operators"
- standard math functions such as exponential, logarithmic, power, and square root functions; see section "cplxexp Transcendental functions"
- trigonometric functions (sine, cosine, hyperbolic sine, and hyperbolic cosine); see section "cplxtrig Trigonometric and hyperbolic functions".

Routines to convert between Cartesian and Polar coordinate systems are discussed in section "cplxcartpol Cartesian/Polar functions".
Error handling is described in section "cplxerr Error handling functions".

3.2 cplxcartpol Cartesian/Polar functions

This section describes the Cartesian and Polar functions in the class *complex*.

```
#include <complex.h>
class complex
{
public:
    friend double      abs(complex);
    friend double      arg(complex);
    friend complex     conj(complex);
    friend double      imag(const complex&);
    friend double      norm(complex);
    friend complex     polar(double, double = 0.0);
    friend double      real(const complex&);
/* other declarations */
};
```

double d = abs(complex x)

Returns the absolute value or magnitude of *x*.

double d = norm(complex x)

Returns the square of the magnitude of *x*, and is intended for comparison of magnitudes. The *norm()* function is faster than *abs()*. With *norm()*, however, an overflow error is more likely since the square of the magnitude is returned.

double d = arg(complex x)

Returns the angle of *x*, measured in radians in the range -pi to pi.

complex y = conj(complex x)

Returns the complex conjugate of *x*. If *x* is specified in the form *(real, imag)*, then *conj(x)* is identical to *(real, -imag)*.

complex y = polar(double m, double a=0.0);

Returns a value of type *complex*, given a pair of polar coordinates: magnitude *m*, and angle *a*, measured in radians.

double d = real(complex &x)

Returns the real part of *x*.

double d = imag(complex &x)

Returns the imaginary part of *x*.

EXAMPLE

The following program converts a complex number to the Polar coordinate system and then prints it:

```
#include <iostream.h>
#include <complex.h>
 main ()
 {
   complex d;
   d = polar (10.0, 0.7);
   cout <<real(d)<<" "<<imag(d);
   cout <<"\n";
   return 0;
 }
```

The result of executing the program is:

```
7.64842 6.44218

%  CCM0998 CPU time used: 0.0006 seconds
```

SEE ALSO

cplxerr, *cplxerr*, *cplxops*, *cplxtrig*

3.3 cplxerr Error handling functions

This section describes the error handling function used for complex math in C++.

#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);

};

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);

};

---

int i = complex_error(c_exception & x)

The error handling function *complex_error* is called if an error occurs for one of the following four functions:

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

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

In the class *c_exception*, 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 <*complex.h*>):

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 producedthe error. The variables *arg1* and *arg2* are the arguments with which the function wasinvoked. *retval* is set to the default value that is returned by the function unless theuser's *complex_error* sets it to a different value.

If the user's *complex_error* function returns a non-zero value, no error message isprinted, and *errno* is not set.

If the user does not supply a function called *complex_error*, the default error handlingroutines described under the heading "RETURN VALUES" with the respective functionsare invoked upon error. Default error handling is also summarized in the table below. Inevery case, *errno* is set to EDOM or ERANGE and the program continues.

The following abbreviations are used in the table below:

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

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

EXAMPLE The following program declares a complex number using the default constructor, which gives (0.0, 0.0), and then calls the *log()* function with (0.0, 0.0). This produces an error since *log(0.0, 0.0)* is undefined. The *complex_error()* function is called to handle the error.

```
#include <iostream.h>
#include <complex.h>
#include <stdlib.h>
int complex_error(c_exception & p)
{
  cerr << "Error when processing ";
  cerr << p.name << " ( " << p.arg1 << " )\n";
  exit (1);
  return 0; /* NOT REACHED */
}
main()
{
  complex c;
  c = log (c);
  return 0;
}
```

The result of executing the program is:

```
Error when processing log ( ( 0, 0) )
%  CCM0998 CPU time used: 0.0005 seconds
%  CCM0999 exit 1
```

SEE ALSO

cplxcartpol, *cplxexp*, *cplxops*, *cplxtrig*

3.4 cplxexp Transcendental functions

This section describes the transcendental functions in the class *complex*.

```
#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);
};
```

complex z = exp(complex x)

Returns e^x.

complex z = log(complex x)

Returns the natural logarithm of *x*.

complex z = pow(complex x, complex y)

Returns x^y.

complex z = sqrt(complex x)

Returns the square root of *x*, contained in the first or fourth quadrants of the complex plane.

RETURN VALUES

exp returns (0.0, 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* is > 0;

(HUGE, -HUGE) if the cosine is > 0 and the sine is 0;

(-HUGE, HUGE) if the sine is > 0 and the cosine is 0;

(-HUGE, -HUGE) if the sine and cosine are 0.

This section describes the transcendental functions in the class *complex*.

```
#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);

};
```

---

complex z = exp(complex x)

Returns e^x.

complex z = log(complex x)

Returns the natural logarithm of *x*.

complex z = pow(complex x, complex y)

Returns x^y.

complex z = sqrt(complex x)

Returns the square root of *x*, contained in the first or fourth quadrants of the complex plane.

RETURN VALUES

exp returns (0.0, 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* is > 0;
- (HUGE, -HUGE), if the cosine is > 0 and the sine is <= 0
- (-HUGE, HUGE), if the sine is > 0 and the cosine is <= 0
- (-HUGE, -HUGE), if the sine and cosine are <= 0 sind.

In all these cases, *errno* is set to ERANGE.

log returns (-HUGE, 0.0) and sets *errno* to EDOM when *x* is (0.0, 0.0). A message indicating SING error is printed on the standard error output.

These error handling routines can be changed with the *complex_error()* function (see *section* "cplxerr Error handling functions").

EXAMPLE The following program prints a set of complex numbers and their exponential powers.

```
#include <iostream.h>
#include <complex.h>
main()
{
  complex c;
  for (c = complex(1.0,1.0); real(c) < 4.0; c += complex(1.0,1.0))
  {
    cout <<c<<" "<<exp(c)<<"\n";
  }
  return 0;
}
```

The result of executing the program is:

```
( 1, 1) ( 1.46869, 2.28736)
( 2, 2) ( -3.07493, 6.71885)
( 3, 3) ( -19.8845, 2.83447)
%  CCM0998 CPU time used: 0.0012 seconds
```

Note that complex numbers can be printed by using the << operator and can be processed as easily as numbers of type *float* or *double*.

SEE ALSO

cplxcartpol, *cplxerr*, *cplxops*, *cplxtrig*

3.5 cplxops Operators

This section describes the basic input/output, arithmetic, comparison, and assignment operators.

```
#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);

};

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

The operators have their conventional precedences. In the following descriptions for operators, *x, y*, and *z* are variables of class *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 a non-zero integer if complex number *x* is equal to complex number *y*; returns 0 otherwise.

x != y

Returns a non-zero integer 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 complexnumber *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.In other words, the following construction is syntactically invalid:

```
complex x, y, z;

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

The following lines, by contrast:

```
x = (y + z);

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

are valid.

Input/output operators:

Output and input of complex numbers may be performed using the << and >> operators, respectively.

Output format:

(real, imag)

Input format:

```
Input            corresponds to       complex number

-----------------------------------------------------
Zahl                    |             (Zahl, 0)
(Zahl)                  |             (Zahl, 0)
(Zahl1, Zahl2)          |             (Zahl1, Zahl2)
```

EXAMPLE The following program defines the complex numbers *c* and *d*, divides *d* by *c*, and then prints the values of *c* and *d*:

```
#include <iostream.h>
#include <complex.h>
main()
{
  complex c,d;
  d = complex(10.0, 11.0);
  c = complex (2.0, 2.0);
  while (norm(c) < norm(d))
  {
    d /= c;
    cout << c << " " <<d << "\n";
  }
  return 0;
}
```

The result of executing the program is:

```
( 2, 2)  ( 5.25, 0.25)
( 2, 2)  ( 1.375, -1.25)
%  CCM0998 CPU time used: 0.0009 seconds
```

SEE ALSO

cplxcartpol, *cplxerr*, *cplxexp*, *cplxtrig*

3.6 cplxtrig Trigonometric and hyperbolic functions

This section describes the trigonometric and hyperbolic functions for the data type*complex*.

#include <complex.h>

class complex
{

public:

friend complex sin(complex);
friend complex cos(complex);

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

---

complex y = sin(complex x)

Returns the sine of *x*.

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

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

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

RETURN VALUES

This section describes the trigonometric and hyperbolic functions for the data type *complex*.

#include <complex.h>

class complex
{

public:

friend complex sin(complex);
friend complex cos(complex);

friend complex sinh(complex);
friend complex cosh(complex);

};

---

complex y = sin(complex x)

Returns the sine of *x*.

complex y = cos(complex x)

Returns the cosine of *x*.

complex y = sinh(complex x)

Returns the hyperbolic sine of *x*.

complex y = cosh(complex x)

Returns the hyperbolic cosine of *x*.

RETURN VALUES

If the imaginary part of *x* causes an overflow, *sinh* and *cosh* return (0.0, 0.0). When the real part is large enough to cause an overflow, *sinh* and *cosh* return:

- (HUGE, HUGE) if the cosine and sine of the imaginary part of *x* are >= 0;
- (HUGE, -HUGE) if the cosine is >= 0 and the sine is < 0;
- (-HUGE, HUGE) if the sine is >= 0 and the cosine is < 0.
- (-HUGE, -HUGE) if both sine and cosine are < 0.

In all these cases, *errno* is set to ERANGE.
These error handling routines may be changed with the function *complex_error()* (see *cplxerr*).

EXAMPLE The following program prints a range of complex numbers with their associated *cosh()* values:

```
#include <iostream.h>
#include <complex.h>
main()
{
  complex c;
  while (norm(c) < 10.0)
  {
    cout << c <<" " <<cosh(c) << "\n";
    c += complex(1.0, 1.0);
  }
  return 0;
}
```

The result of executing the program is:

```
( 0, 0)  ( 1, 0)
( 1, 1)  ( 0.83373, 0.988898)
( 2, 2)  ( -1.56563, 3.29789)
%  CCM0998 CPU time used: 0.0017 seconds
```

Note

The result of the *cosh()* function is a *complex* object.

The constants of type *double* (e.g. 10.0, 1.0 etc) are used to construct complex numbers.

SEE ALSO

cplxcartpol, *cplxerr*, *cplxexp*, *cplxops*

4 Classes and functions for input/output

This chapter provides information concerning the following topics:

- iosintro Introduction to buffering, formatting, and input/output
- filebuf Buffer for file input/output
- fstream Specialization of iostream and streambuf for files
- ios Base class for input/output
- istream Formatted and unformatted input
- manip iostream manipulation
- ostream Formatted and unformatted output
- sbufprot Protected interface of class streambuf
- sbufpub Public interface of class streambuf
- sstreambuf Specialization of streambuf for arrays
- stdiobuf Specialization of iostream for stdio FILEs
- strstream Specialization of iostream for arrays

4.1 iosintro Introduction to buffering, formatting, and input/output

This section describes the mechanisms that may be used to implement input and output in C++. The standard I/O library is written in C++ and shows the power of this programming language.

The C++ *iostream* package consists primarily of a collection of classes declared in the following header files:
<iostream.h>, <fstream.h>, <strstream.h>, <stdiostream.h>, <iomanip.h>. Although originally intended only to support input/output, the package now supports related activities such as byte array processing.

```
#include <iostream.h>

typedef long streampos, streamoff;

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;

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

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

#include <strstream.h>
class strstreambuf : public streambuf;
class strstreambase : virtual public ios;
class istrstream : public strstreambase, public istream;
class ostrstream : public strstreambase, public ostream;
class strstream : public strstreambase, public iostream;

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

#include <iomanip.h>
```

In the iostream package, there are some functions which return characters, but whichuse *int* as a return type. *int* is used so that all possible characters in the machine character set can be returned, as well as the value EOF as an error indication. A character is usually stored in a location of type *char* or *unsigned char*.

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

The header file <*iomanip.h*> supplies macro definitions which programmers can useto define new parameterized manipulators (see *manip*).

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 *sbufpub* and the protected interface (for derived classes) is described in*sbufprot*.

ios

> This is the base class for *stream* input/output in C++. This class contains state variables that are common to the various stream classes, for example, error states and formatting states. See *ios*.

istream

> This class supports formatted and unformatted conversion from sequences of characters fetched from *streambuf*s. See *istream*.

ostream

> This class supports formatted and unformatted conversion to sequences of characters stored into *streambuf*s. See *ostream*.

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*.

istream_withassign
ostream_withassign
iostream_withassign

> These classes add assignment operators and a constructor with no operands to thecorresponding class without assignment. The predefined streams (see below) *cin, cout, cerr*, and *clog*, are objects of these classes. See *istream, ostream,* and *ios*.

Predefined streams

The following streams are predefined:

cin

> The standard input (file descriptor 0), similar to *stdin* in the C language.

cout

The standard output (file descriptor 1), similar to *stdout* in the C language.

cerr

Standard error (file descriptor 2). Output through this stream is unit-buffered, which means that characters are passed to the C runtime system after each insert operation. (See *ostream::osfx()* in *ostream* and *ios::unitbuf* in *ios*.) It is like *stderr* in the C language.

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 causes *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 canuse these, or they 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 orconsumed. Derivation of a class from *streambuf* (the *protected interface*) is discussed in *sbufprot*. The available buffer classes are:

filebuf

This buffer class supports I/O through file descriptors. Member functions support opening, closing, and seeking. Common uses do not require the program to manipulate file descriptors. See *filebuf*.

stdiobuf

This buffer class supports I/O through stdio FILE structs. It is intended for use whenmixing C and C++ code. New code should prefer to use *filebuf*s. See *stdiobuf*.

strstreambuf

This buffer class stores and fetches characters from arrays of bytes in memory (i.e.,strings). See *sstreambuf*.

Classes derived from istream, ostream, and iostream

Classes derived from *istream, ostream*, and *iostream* specialize the core classes for use with particular kinds of *streambuf*s. 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 *filebuf*s. See *fstream*.

fstreambase

The member functions common to all three classes are defined in class *fstreambase*.

istrstream
ostrstream
strstream

These classes support the processing of arrays of bytes (e.g. strings), and use class *strstreambuf* (see *strstream*).

strstreambase

The member functions common to these classes are defined in class *strstreambase*.

stdiostream

This class specializes *iostream* for *stdio* FILEs (see *stdiobuf*).

BUGS Performance of programs that copy from *cin* to *cout* can sometimes be improved by breaking the tie between *cin* and *cout* and doing explicit flushes of *cout*.
Some member functions of *streambuf* and *ios* (not discussed in this section) are present only for backward compatibility with the stream package.

SEE ALSO

filebuf , *fstream*, *ios*, *istream*, *manip*, *ostream*, *sbufprot*, *sbufpub*, *strstream*, *sstreambu*), *stdiobuf*

4.2 filebuf Buffer for file input/output

This section describes how class *filebuf* should be used.

```
#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,

                                        bin, tabexp};

        // and lots of other class members, see ios ...

};

#include <fstream.h>

class filebuf : public streambuf
{
public:

        static const int openprot; /* default protection for open*/

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

filebuf*            attach(int d);
filebuf*            close();
int                 fd();
int                 is_open();
filebuf*            open(const char *name, int omode, int prot=openprot);

virtual int         overflow(int=EOF);
virtual streampos   seekoff(streamoff, ios::seek_dir, int omode);

virtual streambuf*  setbuf(char* p, int len);
virtual int         sync();
virtual int         underflow();

};
```

This section describes how class *filebuf* should be used.

```
#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,

                                bin, tabexp};

   // and lots of other class members, see ios ...

};

#include <fstream.h>

class filebuf : public streambuf
{
public:

   static const int openprot; /* Standardschutzmodus für Öffnen*/

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

   filebuf*          attach(int d);
   filebuf*          close();
   int               fd();
   int               is_open();
   filebuf*          open(const char *name, int omode, int prot=openprot);

   virtual int       overflow(int=EOF);
   virtual streampos seekoff(streamoff, ios::seek_dir, int omode);
   virtual streambuf* setbuf(char* p, int len);
   virtual int       sync();
   virtual int       underflow();

};
```

*filebuf*s specialize *streambuf*s to use a file as source or sink of characters. Charactersare consumed by doing writes to the file, and are produced by doing reads. When thefile is seekable, a *filebuf* allows seeks. 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*. No protection mode is used for files in BS2000.

The *reserve area* (or buffer, see *sbufpub* and *sbufprot*) is allocated automatically if it 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 each character is passed to the C runtime system for each read or write. Unbufferedinput/output is not as fast as buffered input/output. The *get* and *put* pointers into the reserve area are conceptually tied together and 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*.
- *mode* is an *int* representing an *open_mode*.

Constructors

filebuf()

> Constructs an initially closed *filebuf*.

filebuf(int d)

> Constructs a *filebuf* connected to file descriptor *d*.

filebuf(int d, char * p, int 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* is unbuffered.

Members (non-virtual)

filebuf * pfb=f.attach(int d)

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

filebuf * pfb=f.close()

> Flushes any waiting output, closes the file descriptor, and disconnects *f*. Unless an error occurs, *f*'s error state is 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.

int i=f.fd()

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

int i=f.is_open()

> Returns non-zero when *f* is connected to a file descriptor, and zero otherwise.

filebuf * pfb=f.open(char* name, int mode, int prot)

Opens file *name* and connects *f* to it. If the file does not already exist, an attempt is made to create it, unless *ios::nocreate* or *ios::in* is specified in *mode*. The *prot* parameter is ignored under BS2000. Failure occurs if *f* is already open. On success,*open()* returns *&f*. If an error occurs it returns 0. The members of *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*. *name* can take any of the values described for the C library functions *open()* and *fopen()*. This name is passed to the C runtime system. Thus any kind of control (except for record-oriented input/output) is possible when opening files in C++. Please refer to section *fstream* for a list of possible values for *name*, and to the "C Library Functions" manual for more detailed information on file processing.

Virtual members

int i=f.overflow(int c)

Please refer to section *sbufprot* (streambuf::overflow()) for a general description.
For *filebuf*s this means:
The contents of the buffer is written to the associated file if *f* is attached to an openfile. Thus a new *put* area becomes available.
On error, EOF is returned.
On success, 0 is returned.

streampos sp=f.seekoff(streamoff off, ios::seek_dir dir, int 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*sbufpub*. *mode* is ignored. *seekoff()* returns *sp*, the new position, or EOF if a failure occurs. The position of the file after a failure is undefined. Relative seeks in text filesare invalid under BS2000 unless *off* = 0.

streambuf * psb=f.setbuf(char * ptr, int 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* is 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 thebuffering status, and *setbuf()* returns 0.

int i=f.sync()

Attempts to force the state of the get/put pointer of *f* to agree (be synchronized) withthe 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. However, *sync()* does not guarantee that the writes made were flushed to disk.
In BS2000, *sync()* passes the contents of the buffer to the C runtime system.
Synchronization uses relative seeks of the C runtime system. This is not possible fortext files under BS2000. Therefore, the *sync* function can only be used for binary files.

Note

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.

int i=f.underflow();

Please refer to section *sbufprot* (streambuf::underflow()) for a general description.
For *filebuf*s this means:
When the *get* area is empty, the associated file is read if *f* is attached to an open file (padding the *get* area). On success, the next character is returned.
On error, EOF is returned.

EXAMPLE

The following program tries to attach a variable of type *filebuf* to file descriptor 1, which is a *cout*, and then prints a message showing the success or failure of the *attach()*:

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
int main()
{
  filebuf b;     /* constructor with no parameters called */
  if (b.attach(1))
  {
    static char str[] = "have attached filebuf b to file descriptor 1\n";
    b.sputn(str, sizeof(str)-1);
  }
  else
  {
    cerr << "can't attach filebuf to file descriptor 1\n";
    exit(1);      /* error return */
  }
  return 0;
}
```

The result of executing the program is:

```
have attached filebuf b to file descriptor 1

%  CCM0998 CPU time used: 0.0003 seconds
```

BUGS *attach()* and the constructors should test if the file descriptor they are given is open.

There is no way to force atomic reads.

SEE ALSO

fstream, *sbufprot*, *sbufpub*

lseek() in the C runtime system

4.3 fstream Specialization of iostream and streambuf for files

This section describes the classes *ifstream*, *ofstream*, and *fstream*, which provide low level operations on files and streams.

```
#include <iostream.h>

class ios
{
public:

    enum  seek_dir {beg, cur, end};
    enum  open_mode {in, out, ate, app, trunc, nocreate, noreplace,

                                bin, tabexp};

    enum  io_state {goodbit=0, eofbit, failbit, badbit};
    // see ios for other class members ...

};

#include <fstream.h>

class fstreambase : virtual public ios
{
public:

    fstreambase();
    ~fstreambase();
    fstreambase(const char* name, int mode, int prot=filebuf::openprot);
    fstreambase(int fd);
    fstreambase(int fd, char * p, int l);

    void      attach(int fd);
    void      close();
    void      open(const char* name, int mode, int prot=filebuf::openprot);

    filebuf*  ();
    void      setbuf(char* p, int l);

};

class ifstream : public fstreambase, public istream
{
public:

    ifstream();
    ~ifstream();
    ifstream(const char* name, int mode=ios::in, int prot=filebuf::openprot);
    ifstream(int fd);
    ifstream(int fd, char* p, int l);
```

```
    void        open(const char* name, int mode=ios::in, int prot=filebuf::openprot);

    filebuf*    rdbuf();

};

class ofstream : public fstreambase, public ostream
{
public:

    ofstream();
    ~ofstream();
    ofstream(const char* name, int mode=ios::out, int prot =filebuf::openprot);ofstream(int fd);
    ofstream(int fd, char* p, int l);

    void        open(const char* name, int mode=ios::out, int prot=filebuf::openprot);

    filebuf*    rdbuf();

};

class fstream : public fstreambase, public iostream
{
public:

    fstream();
    ~fstream();
    fstream(const char* name, int mode, int prot =filebuf::openprot);
    fstream(int fd);
    fstream(int fd, char* p, int l);

    void        open(const char* name, int mode, int prot=filebuf::openprot);

    filebuf*    rdbuf();

};
```

Base class *fstreambase* contains the standard definitions for constructors and memberfunctions for derived classes.

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

In the following descriptions, assume that:

- *f* is any of *ifstream*, *ofstream*, or *fstream*.
- *mode* is an *int* representing an *open_mode*.

Constructors

In *xstream*, *x* is either:

- *if*

- *of*, or
- *f*,

so that *xstream* stands for:

- *ifstream*
- *ofstream*, or
- *fstream*.

The constructors for *xstream* are:

*x*stream()

Constructs an unopened *xstream*.

*x*stream(char * name, int mode, int prot)

Constructs an *xstream* and opens file *name* using *mode* as the open mode. For a description of parameters *name* and *mode*, see under *open()* below. *prot* is ignored under BS2000.
The error state *(io_state)* of the constructed *xstream* indicates failure in case the *open* fails.

*x*stream(int fd)

Constructs an *xstream* connected to file descriptor *fd*, which must be already open.

*x*stream(int fd, char * ptr, int len)

Constructs an *xstream* 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* is unbuffered.

Member functions

void f.attach(int fd)

Connects *f* to the file descriptor *fd*. A failure occurs when *f* is already connected to afile. A failure sets *ios::failbit* in *f*'s error state.

void f.close()

Closes any associated *filebuf* and thereby breaks the connection of the *f* to a file. *f*'serror state is cleared except on failure, which is when the C runtime system detects afailure in the system call *close()*.

void f.open(char * name, int mode, int prot)

Opens file *name* and connects *f* to it. If the file does not already exist, an attempt is made to create it unless *ios::nocreate* or *ios::in* is set. Failure occurs if *f* is already open, or the C runtime system call *open()* fails. *ios::failbit* is set in *f*'s error status on failure. *prot* is ignored under BS2000.

name may be:

- any valid BS2000 file name
- "link=*linkname*", where *linkname* is a BS2000 link name
- "(SYSDTA)", "(SYSOUT)", "(SYSLST)" for the appropriate system file
- "(SYSTERM)" for terminal input/output
- "(INCORE)" for a temporary binary file that is only set up in virtual memory

For more detailed information, please refer to the "C Library Functions" manual.

The members of *open_mode* are bits that may be or'ed together. (Because the or'ingreturns 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 appended to the end of file. *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 *ifstream*s. For *fstream*s it indicates that input operations should be allowed if possible. It 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. If the file does not already exist, the *open()* fails.

ios::out

The file is opened for output. *ios::out* is implied by construction and opens of *ofstream*s. For *fstream* it says that output operations are to be allowed.

ios::trunc

If the file already exists, its contents are 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()* fails.

ios::noreplace

If the file already exists, the *open()* fails.

ios::bin

The file is opened as a binary file. If this parameter is omitted the file is opened as a text file.

ios::tabexp

Is ignored for binary files and input files.
For text files, the tab character (\t) is converted to the corresponding number of spaces. Tab positions are spaced 8 columns apart (1, 9, 17, ...). When this parameter is omitted, the tab character is mapped as the corresponding EBCDIC value in the text file (see the "C Library Functions" manual).

Note

If *open()* of class *istream* is used, *ios::in* is always set.
If *open()* of class *ostream* is used, *ios::out* is always set.

filebuf * 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.

void f.setbuf(char * p, int len)

Has the usual effect of a *setbuf()* (see *filebuf()*), offering space for a reserve area or requesting unbuffered I/O. An error occurs if *f* is open or the call to *f.rdbuf()->setbuf* fails.

EXAMPLE

The following program opens the file *#TEMP*. On success, text is written to the file.

```
#include <fstream.h>
#include <iostream.h>
int main()
{
  static char * name = "#TEMP";
  ofstream q(name, ios::out);
  cout << " File " << name;
  if (q.ios::failbit)
  {
     cout << " is open.\n";
     q << "This is the first line of file " << name << ".\n";
  }
  else
  {
     cout << " could not be opened.\n";
     exit (1);
  }
  return 0;
}
```

The result of executing the program is:

```
File #TEMP is open.
%  CCM0998 CPU time used: 0.0395 seconds
```

SEE ALSO

filebuf, *ios*, *istream*, *ostream*, *sbufpub*
close(), open() in the C runtime system

4.4 ios Base class for input/output

This section describes the operators that are common to both input and output.

```
#include <iostream.h>

class ios
{
public

   enum  io_state {goodbit=0, eofbit, failbit, badbit};

   enum  open_mode {in, out, ate, app, trunc, nocreate, noreplace,

                                tabexp};

   enum  seek_dir {beg, cur, end};

   /* Format control flags */
   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*);
   virtual ~ios();
```

```
    int             bad() const;
    static long     bitalloc();
    void            clear(int state=0);
    int             eof() const;
    int             fail() const;
    char            fill() const;

    char            fill(char);
    long            flags() const;
    long            flags(long);
    int             good() const;
    long&           iword(int);
    int             operator!() const;
                    operator void*();
                    operator const void*() const;

    int             precision() const;
    int             precision(int);
    void* &         pword(int);
    streambuf*      rdbuf();
    int             rdstate() const;
    long            setf(long setbits, long field);
    long            setf(long);
    static void     sync_with_stdio();
    ostream*        tie();
    ostream*        tie(ostream*);
    long            unsetf(long);
    int             width() const;
    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&);
```

The stream classes derived from class *ios* provide a high level interface that supportstransferring formatted and unformatted information into and out of *streambuf*

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 inthis section under "Error states". The format fields are also described in this sectionunder "Formatting". The *open_mode*s are described in detail in *fstream* under *open()*.
The *seek_dir*s are described in *sbufpub* under *seekoff()*.

In the following descriptions, assume:

- *s* and *s2* are *ios*
- *sr* is an *ios&*.
- *mode* is an *int* representing an *open_mode*.

Constructors and assignment

ios(streambuf * sb)

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

ios(ios& sr)
s2=s

> Copying of *ios*s is not well-defined in general, therefore the constructor and assignment operators are private so that the compiler complains about attempts to copy *ios* objects. Copying pointers to *iostream*s is usually what is required.

ios()
init(streambuf * 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_state*s). Members related to the error state are:

int i=s.rdstate()

> Returns the current error state.

s.clear(int 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()).*

int i=s.good()

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

int 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.

int 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.

int 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 is probably impossible to continue I/O operations on *s*.

Operators

Two operators are defined to allow convenient checking of the error state of an *ios* object: *operator!()* and *operator void*()* or *operator const void*() const*. The latter converts an *ios* to a pointer so that it can be compared to zero. The conversion returns0 if *failbit* or *badbit* is set in the error state, and returns a pointer value otherwise. This pointer is not meant to be used. This allows you 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 such as 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. The format state 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 is skipped on input. This applies to scalar extractions.
When *skipws* is not set, whitespace is not skipped before the extractor begins conversion. In this case zero width fields should not be used, as a precaution againstlooping. So if the next character is whitespace and the *skipws* variable is not set, thearithmetic extractors signal an error.
If *skipws* is not set and numeric input is attempted, and the first character of the input is white space, the extraction will fail.
In case of string input the extraction will stop at the first white space character. In thespecial case that the first character in the input stream is white space, nothing will be extracted.
In both cases the input stream will be read until the first white space character is found and then no further.

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 an integer value. The conversion base is10 (decimal) if *dec* is set, but if *oct* or *hex* is set, conversions are done in octal or hexadecimal, 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 are converted to an external form that can be read according to the C++ lexical conventions for integral constants. This means octals are preceded by the character ’0’, and hexadecimals are preceded by the string ’0x’(cf. *uppercase*). *showbase* is unset by default.

showpos

If *showpos* is set, then a plus character ’+’ is inserted into a decimal conversion of a positive integral value.

uppercase

If uppercase is set, then an uppercase *X* is used for hexadecimal output when *showbase* is set, or an uppercase *E* is 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 floatingpoint 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.
- If *uppercase* is set, an uppercase *E* introduces the exponent; a lowercase *e* appears 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 is converted using either notation, depending on the value: scientific notation is used if the exponent
  resulting from the conversion is less than -4 or greater than or equal to the precision. Otherwise, decimal notation is used.
- If *showpoint* is not set, trailing zeros 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. Performanceis better under unit buffering than unbuffered output, which makes a C runtime
system call for each character output. Unit buffering makes a C runtime system call for each insertion operation, and doesn't require the user to call *ostream::flush()*.
In BS2000, a call to *ostream::flush()* terminates the record and starts a new record.

stdio

When set, *stdout* and *stderr* are flushed by *ostream::osfx()* after each insertion.
In BS2000 this means that the current line (record) is terminated and subsequent data is written to a new line (record).

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

char oc=s.fill(char c)

Sets the "fill character" format state variable to *c* and returns the previous value. *c* isused as the padding character, if 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*.

Note

The "fill character" has no effect on input.

char c=s.fill()

Returns the "fill character" format state variable.

long l=s.flags()

Returns the current format flags.

long l=s.flags(long f)

Resets all the format flags to those specified in *f* and returns the previous settings.

int oi=s.precision(int 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 availablefor setting the precision, see *manip*.

int i=s.precision()

Returns the "precision" format state variable.

long l=s.setf(long b)

Turns on in *s* the format flags marked in *b* and returns the previous settings. All otherflags are left unchanged. A parameterized manipulator, *setiosflags* performs the samefunction, see *manip* .

long l=s.setf(long b, long 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*, you could write:

```
s.setf(ios::hex, ios::basefield)
```

Any previous settings to oct or dec will be cleared by this.

ios::basefield specifies the conversion base bits as candidates for change, and *ios::hex* specifies the new value. *s.setf(0, f)* clears all the bits specified by *f*, as does a parameterized manipulator, *resetiosflags* (see *manip*).

long l=s.unsetf(long b)

Unsets in *s* the bits set in *b* and returns the previous settings.

int oi=s.width(int i)

Sets the "field width" format variable to *i* and returns the previous value.

This has two different meanings for either output or input streams:

- *Output*: When the field width is zero (the default), inserters only insert as many characters as necessary to represent the value being inserted. When the field width is non-zero, the inserters insert at least that many characters.

  If the value being inserted requires fewer than field-width characters to be represented, the fill character is used to pad the value. However, the numeric inserters never truncate values, so if the value being inserted does not fit in fieldwidth characters, more than field-width characters are output.

  The field width is always interpreted as a minimum 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.
- *Input*: A setting of the field width applies only for the extraction of *char** and *unsigned char**, see *istream*. When the field width is non-zero, it is taken to be the size of the array, and no more than *width-1* characters are extracted.

  The field width format variable is reset to the default (zero) after each extraction.

A parameterized manipulator, *setw* is also available for setting the width (see *manip*).

int i=s.width()

> Returns the "field width" format variable.

User-defined format flags

Several functions are provided to allow users to derive classes from the base 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.

long 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 then pass it as an argument to *ios::setf()*, for example.

int i=ios::xalloc()

> Returns a previously unused index into an array of words available for use as format state variables by derived classes.

long & l=s.iword(int i)

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

void*& vp=s.pword(int i)

> When *i* is an index allocated by *ios::xalloc*, *pword()* returns a reference to the *i*th user-defined word. *pword()* is similar to *iword*, except that it has a different return type.

Other members

streambuf* sb=s.rdbuf()

> Returns a pointer to the *streambuf* associated with *s* when *s* was constructed.

static void ios::sync_with_stdio()

Solves problems that arise when mixing stdio and iostreams. The first time it is calledit resets the standard iostreams (*cin, cout, cerr, clog*; see *iosintro* to be streams using *stdiobuf*s. After that input and output using these streams may be mixed with input and output using the corresponding FILEs (*stdin, stdout*, and *stderr*) and is properly synchronized. *sync_with_stdio()* makes *cout* and *cerr* unit buffered (see *ios::unitbuf* ans *ios::stdio* above). Invoking *sync_with_stdio()* degrades performance.
Since, in BS2000, output to stdout (SYSOUT) implies a subsequent change of line, behaviour for C++ input/output is not as expected for synchronization with C input/output: Each C++ output is written to a separate line. If synchronization is not used, the order of output is undefined.

Note

Unit buffering for standard input/output files under BS2000 causes each read or written unit to close the current record and start reading or writing for the next record.

ostream * oosp=s.tie(ostream * osp)

Sets the "tie" variable to *osp*, and returns its previous value. This variable supports automatic "flushing" of *ios*s. If the tie variable is nonnull and an *ios* needs more characters or has characters to be consumed, the *ios* pointed at by the tie variable isflushed. By default, *cin* is tied initially to *cout* so that attempts to get more charactersfrom standard input result in flushing standard output. Additionally, *cerr* and *clog* aretied to *cout* by default. For other *ios*s, the tie variable is set to zero by default.

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

Note

In the C runtime system, text output files are flushed before stdin (SYSDTA) is read.*iostream::tie* affects how the C++ buffer contents are passed to the C runtime system. Information passing from the C runtime system buffer to the file is not affected by the value of 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*)) 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*.

sr<<endl
Ends a line by inserting a newline character and flushing. See *ostream*.
Under BS2000, writing a newline character to a text file implies a change of record.

sr<<ends

Ends a string by inserting a null(0) character. See *ostream*.

sr<<flush

Flushes *sr*. See *ostream*.

Several parameterized manipulators that operate on *ios* objects are described in *manip: setbase, setw, setfill, setprecision, setiosflags*, and *resetiosflags*.

The *streambuf* associated with an *ios* can be manipulated by other methods than through the *ios*. For example, characters can 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 operationsthe extraction (input) proceeds normally.

EXAMPLE The following program uses some data members of class *ios* to change the output format of both integers and *double*s on *cout*:

```
#include <iostream.h>
#include <math.h>
void someoutput()
{
  int i;
  const int N = 12;
  for (i = 1; i < N; i += 2)
  {
    cout << "\t" << i << " " << pow( (double) i, (double) i) << endl;
  }
  cout << "\n";
}
int main()
{
  cout << "Default format :\n";
  someoutput();
  /* show default formats for integers and doubles    */
  cout.setf( ios::fixed, ios::floatfield);
  /* set the output format for floats and doubles to fixed*/
  cout << "The output format for floats and doubles is fixed :\n";
  someoutput();
  cout.setf( ios::oct, ios::basefield);
  /* set the output format for integers to octal             */
  cout << "The output format for integers is octal :\n";
  someoutput();
  return 0;
}
```

The result of executing the program is:

```
 Default format :
         1 1
         3 27
         5 3125
         7 823543
         9 3.8742e+08
         11 2.85312e+11
 The output format for floats and doubles is fixed :
         1 1.000000
         3 27.000000
         5 3125.000000
         7 823543.000000
         9 387420488.999998
         11 285311670610.995117
 The output format for integers is octal :
         1 1.000000
         3 27.000000
         5 3125.000000
         7 823543.000000
         11 387420488.999998
         13 285311670610.995117
%  CCM0998 CPU time used: 0.0066 seconds
```

Note

The precision of these results depends on the machine used.

BUGS

The *iostream* package does not allow copying of streams. However, objects of type *istream_withassign, ostream_withassign*, and *iostream_withassign* can be assigned to. (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

iosintro, *istream*, *manip*, *ostream*, *sbufprot*, *sbufpub*

4.5 istream Formatted and unformatted input

This section describes the *istream* member functions and related functions for formatted and unformatted input.

```
#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,

                              bin, tabexp};

    /* Format control flags */
    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

    };

    // see ios for other class members ...

};

class istream : virtual public ios

{

public:

    istream(streambuf*);
```

```
    virtual     ~istream();
    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, ios::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

{

public:
```

```
                        istream_withassign();
    virtual             ~istream_withassign();
    istream_withassign& operator=(istream&);
    istream_withassign& operator=(streambuf*);
};
extern istream_withassign cin;

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

*istream*s support interpretation of characters fetched from an associated *streambuf*. These are commonly referred to as input or extraction operations.

In the following descriptions assume that:

- *ins* is an *istream*.
- *sb* is a *streambuf**.

Constructors and assignment

istream(streambuf* sb)

> Initializes *ios* state variables and associates buffer *sb* with the *istream*.

istream_withassign()

> Does no initialization.
> *istream_withassign* must be initialized with an assignment.

```
istream_withassign inswa;
streambuf * sb;
inswa=sb
```

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

```
istream_withassign inswa;
inswa=ins
```

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

Input prefix function

int i = ins.ipfx(int need)

If *ins*'s error state is non-zero, returns zero immediately. If necessary (even if it isnon-null), any *ios* tied to *ins* is flushed (see the description *ios::tie()* in section ios. 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* iszero, then leading whitespace characters are extracted from *ins*.

ipfx() returns zero if an error occurs while skipping whitespace; otherwise it returnsnon-zero.

Formatted input functions call *ipfx(0)*, while unformatted input functions call *ipfx(1)*;see below.

Formatted input functions (extractors)

istream ins;
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* did not match 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 and the type of *x*. Extractors are defined for thefollowing types, with conversion rules as described below.

x might have one of the following types:

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 theconversion 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 "nondigit", 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* to *f* (in either uppercase or lowercase). 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 onfollowing hexadecimal digits. Otherwise, if the first character is a *0*, an octalconversion 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.

Note

skipws should not be unset during the extraction of numerical values. Otherwise an error can occur.

The type and name 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:

istream * insp=&ins.get(char * ptr, int len, char 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.

istream * insp=&ins.get(char & c)

Extracts a single character and stores it in *c*.

istream * insp=&ins.get(streambuf & sb, char delim)

Extracts characters from *ins.rdbuf()* and stores them into *sb*. It stops if it encounters end of file, or a store into *sb* fails, or it encounters *delim* (which it leaves in *ins*). *ios::failbit* is set if it stops because the store into *sb* fails.

int i=ins.get().

Extracts a character and returns it. *i* is EOF if extraction encounters end of file.*ios::failbit* is never set.

istream * insp=&ins.getline(char * ptr, int len, int delim)

Does the same thing as *ins.get(char* ptr, int len, char 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*.

istream * insp=&ins.ignore(int n, char 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.

istream * insp=&ins.read(char * ptr, int 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

int 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.

int 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.

istream* insp=&ins.putback(char c)

Attempts to back up *ins.rdbuf()* so that the character *c* can be read later. *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()*. However, it returns without doing anything if the error state is non-zero.

int 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

istream& insp=ins.seekg(streamoff off, ios::seek_dir dir)

Repositions *ins.rdbuf()*'s get pointer. See *sbufpub* for a discussion of positioning.

istream& insp=ins.seekg(streampos pos)

Repositions *ins.rdbuf()*'s get pointer. See *sbufpub* for a discussion of positioning.

streampos pos=ins.tellg()

The current position of *ios.rdbuf()*'s get pointer. See *sbufpub* for a discussion of positioning.

Manipulators

ins>>ws

Extracts whitespace characters.

ins>>dec

Sets the conversion base format flag to 10. See *ios*.

ins>>hex

Sets the conversion base format flag to 16. See *ios*.

ins>>oct

Sets the conversion base format flag to 8. See *ios*.

EXAMPLE The following program reads one line of text, and then prints it in the reverse order.

```
#include <iostream.h>
const int N = 80;
char a[ N];                /* text buffer */
int main()
{
  int i;
  cout << " Please enter text :\n";
  cin.unsetf(ios::skipws);
  cin.getline(text, N);      /* get at most N characters */
  i = cin.gcount() - 1;
  while (i)
  {
    cout << text [--i];      /* prints line in the reverse */
                             /* order */
  }
  return 0;                  /* successful return */
}
```

The result of executing the program is:

```
Please enter text :
TOM
MOT
%  CCM0998 CPU time used: 0.0024 seconds
```

BUGS There is no overflow detection on conversion of integers.

SEE ALSO

ios, *manip*, *sbufpub*

4.6 manip iostream manipulation

This section describes how manipulators are used with *iostream*.

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

IOMANIPdeclare(T);

class SMANIP(T)
{
public:

   SMANIP(T)(ios& (*)(ios&,T), T);
   friend istream& operator>>(istream&, const SMANIP(T)&);
   friend ostream& operator<<(ostream&, const SMANIP(T)&);

};

class SAPP(T)
{
public:

   SAPP(T)(ios& (*)(ios&,T));
   SMANIP(T) operator()(T);

};

class IMANIP(T)
{
public:

   IMANIP(T)(istream& (*)(istream&,T),T);
   friend istream& operator>>(istream&, const IMANIP(T)&);

};

class IAPP(T)
{
public:

   IAPP(T)(istream& (*)(istream&,T));
   IMANIP(T) operator()(T);

};

class OMANIP(T)
{
public:

   OMANIP(T)(ostream& (*)(ostream&,T),T);
   friend  ostream& operator<<(ostream&, const OMANIP(T)&);

};
```

```
class OAPP(T)
{
public:

   OAPP(T)(ostream& (*)(ostream&,T));
   OMANIP(T) operator()(T);

};

class IOMANIP(T)
{
public:

   IOMANIP(T)(iostream& (*)(iostream&,T),T);
   friend istream& operator>>(iostream&, const IOMANIP(T)&);
   friend ostream& operator<<(iostream&, const IOMANIP(T)&);

};

class IOAPP(T)
{
public:

   IOAPP(T)(iostream& (*)(iostream&,T));
   IOMANIP(T) operator()(T);

};

IOMANIPdeclare(int);
IOMANIPdeclare(long);

SMANIP(int)      setbase(int);
SMANIP(long)   resetiosflags(long);
SMANIP(int)      setfill(int);
SMANIP(long)   setiosflags(long);
SMANIP(int)      setprecision(int);
SMANIP(int)      setw(int w);
```

Manipulators are values that may be "inserted into" or "extracted from" streams to achieve some effect (other than to insert or extract values), with a convenient syntax.They enable you to embed a function call in an expression containing series of insertions or extractions. For example, the predefined manipulator for *ostream*s, *flush*,can be used as follows:

```
cout << flush
```

to flush *cout*.

Several *iostream* classes supply manipulators, see *ios*, *istream* and *ostream*. *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." Themacros 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 identifier. 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&, T)*.
- *isf* is an *istream& (*)(istream&, T)*.
- *osf* is an *ostream& (*)(ostream&, T)*.
- *iof* is an *iostream& (*)(iostream&, T)*.
- *n* is an *int*.
- *l* is a *long*.

```
s<<SMANIP(T)( (ios& (*)(ios&, T)) f, T t)
s>>SMANIP(T)( (ios& (*)(ios&, T)) f, T t)
s<<SAPP(T)( (ios& (*)(ios&, T)) f)(T t)
s>>SAPP(T)( (ios& (*)(ios&, T)) f)(T 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)( (istream& (*)(istream&, T)) isf, T t)
i>>IAPP(T)( (istream& (*)(istream&, T)) isf) (T t)
```

Returns *isf(i,t)*.

```
o<<OMANIP(T)( (ostream& (*)(ostream&, T)) osf,T t)
o<<OAPP(T)( (ostream& (*)(ostream&, T)) osf) (T t)
```

Returns *osf(o,t)*.

```
io<<IOMANIP(T)( (iostream& (*)(iostream&, T)) iof,T t)
io>>IOMANIP(T)( (iostream& (*)(iostream&, T)) iof,T t)
io<<IOAPP(T)( (iostream& (*)(iostream&, T)) iof) (T t)
io>>IOAPP(T)( (iostream& (*)(iostream&, T)) iof) (T t)
```

Returns *iof(io,t)*.

<iomanip.h> contains some additional 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* for further details.

```
o<<setbase(int n)
i>>setbase(int n)
```

Sets the conversion base format flag to be *n*.

o<<resetiosflags(long l)
i>>resetiosflags(long l)

The format bits specified by *l* are flushed in the stream (*o* or *i*) (thus calling *o.setf(0, l)* or *i.setf(0, l)*).

o<<setfill(int n)
i>>setfill(int n)

Sets the fill character of the stream (*o* or *i*) to be *n*.

o<<setiosflags(long l)
i>>setiosflags(long l)

Turns on in the stream (*o* or *i*) the format flags marked in *l*. (Calls *o.setf(l)* or *i.setf(l)*).

o<<setprecision(int n)
i>>setprecision(int n)

Sets the precision of the stream (*o* or *i*) to be *n*.

o<<setw(int n)
i>>setw(int n)

Sets the field width of the stream (left-hand operand: *o* or *i*) to *n*.

EXAMPLE The following program shows the use of manipulators (like *setw*) which globally alter output by changing private data members in *cout*:

```
#include <iostream.h>
#include <iomanip.h>
#include <string.h>
void testline(const char * const p)
{
  /* Put the parameter string onto cout and set the field width         */
  /* of the cout stream to be twice the length of the string            */
  int N = 2 * strlen(p);
  cout << setw(N) ;
  cout << p;
}
void someoutput(const char * const p, const char * const q)
{
  /* Given a string p and a string q containing a list of fill          */
  /* characters; display the string p in a variety of fill              */
  /* character contexts                                                 */
  int i;
  int M = strlen(q);
  for (i = 0; i < M; ++i)
  {
    cout << setfill(q[i]);
    testline(p);
  }
}
int main()
{
  someoutput( "A Test String\n", ".,!$%&*()");
  /* Note how the output is right justified for text strings            */
  return 0;
}
```

Note how <*string.h*> must be included to get the function prototype for *strlen()*, and <iomanip.h> must be included to get the prototypes for *setw()* and *setfill()*.

The result of executing the program is:

```
.....................A Test String
,,,,,,,,,,,,,,,,,,,,A Test String
!!!!!!!!!!!!!!!!!!!A Test String
$$$$$$$$$$$$$$$$$$$$A Test String
%%%%%%%%%%%%%%%%%%%%A Test String
&&&&&&&&&&&&&&&&&&&&A Test String
**********************A Test String
((((((((((((((((((((A Test String
))))))))))))))))))))A Test String
%  CCM0998 CPU time used: 0.0006 seconds
```

SEE ALSO

ios, *istream*, *ostream*

4.7 ostream Formatted and unformatted output

This section defines the *ostream* functions for formatted and unformatted output.

```
#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,

                              bin, tabexp};

   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

   };

   //  see ios for other class members

} ;

class ostream : virtual public ios

{

public:
```

```
                ostream(streambuf*);

    virtual     ~ostream();

    ostream&    flush();
    int         opfx();
    void        osfx();
    ostream&    put(char);
    ostream&    seekp(streampos);
    ostream&    seekp(streamoff, ios::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 : public ostream

{

public:

                            ostream_withassign();

    virtual                 ~ostream_withassign();

    ostream_withassign&     operator=(ostream&);
    ostream_withassign&     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&);
```

---

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*
- *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(streambuf * sb)

> Initializes *ios* and *ostream* 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 *outswa* and initializes the entire state of *outswa*.

outswa=outs

> Associates *outs.rdbuf()* with *outswa* and initializes the entire state of *outswa*.

Output prefix function

int outs.opfx()

If *outs*'s error state is non-zero, returns zero immediately. If *outs.tie()* is non-null, the *ios*s associated with *outs* are flushed. Returns non-zero in all other cases.

Output suffix function

void 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*. Under BS2000, flushing stdio and stderr implies, among other things, that the current line (record) is terminated. Subsequent data is written to the next line.

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*). 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, depending on whether *ios::dec, ios::oct*, or *ios::hex* 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*). 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 than0 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, charactersare 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

ostream * outsp=&outs.put(char c)

Inserts *c* into *outs.rdbuf()*. Sets the error state if the insertion fails.

ostream * outsp=&outs.write(char * s, int n)

Inserts the *n* characters starting at *s* into *outs.rdbuf()*. These characters may include zero bytes (i.e., *s* need not be a null-terminated string).

Other member functions

ostream * 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 mayhave been stored but not yet consumed to be consumed by calling *outs.rdbuf()->sync*.
In BS2000, this means that these characters are passed to the C runtime system.

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

ostream * outsp=&outs.seekp(streamoff off, ios::seek_dir dir)

Repositions *outs.rdbuf()*'s put pointer. See *sbufpub* for a discussion of positioning.

ostream * outsp=&outs.seekp(streampos pos)

Repositions *outs.rdbuf()*'s put pointer. See *sbufpub* for a discussion of positioning.

streampos pos=outs.tellp()

The current position of *outs.rdbuf()*'s put pointer. See *sbufpub* for a discussion of positioning.

Manipulators

outs<<endl

Ends a line by inserting a newline character and flushing.
Under BS2000, the newline character is converted to a change-of-record character.

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*.

outs<<hex

Sets the conversion base format flag to 16. See *ios*.

outs<<oct

Sets the conversion base format flag to 8. See *ios*.

EXAMPLE The following program displays a range of different data types in a variety of different formats:

```
#include <iostream.h>
#include <iomanip.h>  /* for setw */
int main()
{
  int i = 50;
  char c = 'd';
  double d = 1.2;
  float f = 3.1232;
  const char * const p = "abcdefghijklmnopqrstuvwxyz";
  /* show the defaults for the various data types first  */
  cout << i << endl;
  cout << c << endl;
  cout << d << endl;
  cout << f << endl;
  cout << p << endl;
  cout << endl;
  cout.setf( ios::oct, ios::basefield);
  cout << i << endl;  /* same number in octal */
  cout << c << endl;
  cout.setf( ios::fixed, ios::floatfield);
  /* use fixed format for floats and doubles */
  cout << d << endl;
  cout << f << endl;  /* above format still holds */
  cout.setf( ios::right, ios::basefield);
  cout << setw( 50) << flush;
  cout << p << endl;  /* put string in field of width 50 */
  return 0;
}
```

The result of executing the program is:

```
50
d
1.2
3.1232
abcdefghijklmnopqrstuvwxyz
62
d
1.200000
3.123199
                         abcdefghijklmnopqrstuvwxyz
%  CCM0998 CPU time used: 0.0009 seconds
```

Note how the integer *i* has been printed out in two different formats, and the ease by which the format of doubles and floats can be controlled. As shown in the first part of *main()*, the output library provides sensible defaults, without the programmer explicitly setting them up.

SEE ALSO

ios, *manip*, *sbufpub*

4.8 sbufprot Protected interface of class streambuf

This section describes the protected and virtual parts of the *streambuf* class; especially interesting for derived classes.

```
#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,

                              bin, tabexp};

    // and lots of other stuff, see ios ...

} ;

class streambuf

{

public:

                streambuf();
                streambuf(char* p, int len);

    virtual     ~streambuf();
    void        dbp();

protected:
```

```
    int             allocate();
    char*           base();
    int             blen() const;
    char*           eback();
    char*           ebuf();
    char*           egptr();
    char*           epptr();
    void            gbump(int n);
    char*           gptr();
    char*           pbase();
    void            pbump(int n);
    char*           pptr();
    void            setb(char* b, char* eb, int a=0);
    void            setg(char* eb, char* g, char* eg);
    void            setp(char* p, char* ep);
    int             unbuffered() const;
    void            unbuffered(int);
    virtual int     doallocate();

public:

  virtual int    overflow(int c=EOF);
  virtual int    pbackfail(int c);

  virtual streampos

                 seekoff(streamoff, ios::seek_dir, int =ios::in|ios:out);

  virtual streampos

                 seekpos(streampos, int =ios::in|ios:out);

  virtual streambuf*

                 setbuf(char* p, int len);

  virtual int    sync();
  virtual int    underflow();

};
```

---

*streambuf*s implement the buffer abstraction described in *sbufpub*. However, the *streambuf* class itself contains only basic members for manipulating the characters and normally a class derived from *streambuf* is used. This section 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* behaves as specified inthe 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 onproper behaviour of the *streambuf* may not behave properly either.

In the following descriptions assume:

- *sb* is a *streambuf**.
- *ptr, b, eb, p, ep, g*, and *eg* are *char**
- *i*, *n*, *len* and *a* are *int*s
- *c* is an *int* character (positive or EOF).
- *pos* is a *streampos* (see sbufpub).
- *off* is a *streamoff*.
- *dir* is a *seek_dir*.
- *mode* is an *int* representing an *open_mode*.

Constructors

streambuf()

> Constructs an empty buffer corresponding to an empty sequence.

streambuf(char * b, int len)

> Constructs an empty buffer and then sets up the reserve area to be the *len* bytes starting at *b*.

The get, put, and reserve area

The protected members of *streambuf* present an interface to derived classes organizedaround three areas (arrays of bytes) managed cooperatively by the base and derivedclasses. They are the *get area*, the *put area*, and the *reserve area* (or buffer). The get and the put areas are normally disjointed, but they may both overlap the reserve area,whose primary purpose is to be a resource in which space for the put and get areascan be allocated. The get and the put areas are changed as characters are put into and taken from the buffer, but the reserve area normally remains fixed. The areas aredefined by a collection of *char** values. The buffer abstraction is described in terms ofpointers 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

char * ptr=sbbase()

> Returns a pointer to the first byte of the reserve area. Space between *sbbase()* and *sb->ebuf()* is the reserve area.

char * ptr=sbeback()

Returns a pointer to a lower bound on *sb->gptr()*. Space between *sb->eback()* and *sb->gptr()* is available for putback.

char * ptr=sbebuf()

Returns a pointer to the byte after the last byte of the reserve area.

char * ptr=sbegptr()

Returns a pointer to the byte after the last byte of the get area.

char * ptr=sbepptr()

Returns a pointer to the byte after the last byte of the put area.

char * ptr=sbgptr()

Returns a pointer to the first byte of the get area. The available characters are thosebetween *sb->gptr()* and *sb->egptr()*. The next character fetched is **(sbgptr())* unless *sb->egptr()* is less than or equal to *sb->gptr()*.

char * ptr=sbpbase()

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.

char * ptr=sbpptr()

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 are stored here.

Functions for setting the pointers

Note

To indicate a particular area (get, put, or reserve) does not exist, all the associated pointers should be set to zero.

void sb->setb(char * b, char * eb, int i)

Sets *base()* and *ebuf()* to *b* and *eb*, respectively. *i* controls whether the area is subject to automatic deletion. If *i* is non-zero, then *b* is 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*, so the reserve area has zero length.

void sb->setp(char * p, char * ep)

Sets *pptr()* to *p, pbase()* to *p*, and *epptr()* to *ep*.

void sb->setg(char * eb, char * g, char * eg)

Sets *eback()* to *eb, gptr()* to *g*, and *egptr()* to *eg*.

Other non-virtual members

int i=sballocate()

Tries to set up a reserve area. If a reserve area already exists or if *sbunbuffered()* is nonzero, *allocate()* returns 0 without doing anything. If the attempt to allocate space fails, *allocate()* returns EOF, otherwise (allocation succeeds) *allocate()* returns *allocate()* is not called by any non-virtual member function of *streambuf*.

int i=sbblen()

Returns the size (in *char*s) of the current reserve area.

void dbp()

Writes directly on file descriptor 1 information in EBCDIC 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.

void sb->gbump(int 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.

void sb->pbump(int 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.

void sb->unbuffered(int i)
int i=sbunbuffered()

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 determine whether a reserve area is allocated automaticallyby *allocate*.

Virtual member functions

Virtual functions may be redefined in derived classes to specialize the behaviour of *streambuf*s. This section describes the behaviour that these virtual functions should have in any derived classes; the next section describes the behaviour that these functions are defined to have in base class *streambuf*.

int i=sbdoallocate()

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.

int i=overflow(int 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 actionis 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.

int i=sb->pbackfail(int 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.

streampos pos=sb->seekoff(streamoff off, seek_dir dir, int mode)

seekoff() is a public virtual member function. A detailed description is given in section *sbufpub*. Repositions the *get* and/or *put* pointers. Not all derived classes support repositioning.

streampos pos=sb->seekpos(streampos pos, int mode)

seekpos() is a public virtual member function. A detailed description is given in section *sbufpub*. Repositions the *get* and/or *put* pointers. Not all derived classes support repositioning.

streambuf * sb=sb->setbuf(char * ptr, int 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 acceptor ignore the request for unbuffered state as it chooses. *setbuf()* should return *sb* if it honours the request. Otherwise it should return 0.

int i=sbsync()

sync() is a public virtual member function. A detailed description is given in section *sbufpub*.

int i=sbunderflow()

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 non-empty 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

int i=sbstreambuf::doallocate()

Attempts to allocate a reserve area using operator *new*.

int i=sb->streambuf::overflow(int c)

streambuf::overflow() should be treated as if it had undefined behaviour. That is, derived classes should always define it.

int i=sb->streambuf::pbackfail(int c)

Returns EOF on failure and *c* on success.

streampos pos=sb->streambuf::seekpos(streampos pos, int 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()*.

streampos pos=sb->streambuf::seekoff(streamoff off, seekdir dir, int mode)

Returns EOF.

streambuf * sb=sb->streambuf::setbuf(char* ptr, int len)

Honours the request when there is no reserve area.

int i=sbstreambuf::sync()

Returns 0 if the get area is empty and there are no unconsumed characters.
Otherwise it returns EOF.

int i=sbstreambuf::underflow()

streambuf::underflow() should be treated as if it had undefined behaviour. That is, it should always be defined in derived classes.

EXAMPLE The program prints the address of the base area of a class derived from a *streambuf*.

The program is an example of displaying memory contents. It could have other trivial member functions like *get_base* which return the addresses of the *get* and *put* areas..

```
#include <iostream.h>
const int N = 20;
class trivial : public streambuf
{
  int a;                /* Some sample data in a class       */
  public:
  trivial() : streambuf(new char[ N], N)
  {
    /* Define trivial constructor by streambuf constructor   */
    a = 0;
  };
   trivial() {};
  /* Assume streambuf destructor will delete the N byte      */
  /* reserve area                                            */
  char * get_base()
  {
    /* We need this function because the streambuf::base()   */
    /* member function is protected.                         */
    /* We don't need the streambuf:: qualifier since scope   */
    /* is ok.                                                */
    return base();
  };
};
int main()
{
  trivial test_var;
  cout << (void *) test_var.get_base() << endl;
  /* We must cast to void * to stop cout displaying the     */
  /* contents of the first byte of the reserve area.        */
  return 0;
}
```

The result of executing the program is:

```
0xc6008
%  CCM0998 CPU time used: 0.0005 seconds
```

Note that the value stored in the pointer may vary, and that *cout* has a default format for pointer values..

BUGS

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 *streambuf*s deals in characters and bytes, the interface to derived classes deals in *char*s. 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 *char* and *unsigned char* versions. But perhaps all these uses of *char** ought to have been with a typedef.

SEE ALSO

istream, sbufpub

4.9 sbufpub Public interface of class streambuf

This section describes the public member functions of *streambuf*. Only objects derived from *streambuf* (e.g. *filebuf*, *strstreambuf*, *stdiobuf*) are to be used in a program rather than plain *streambuf*s!

```
#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,

                                bin, tabexp};

    // and lots of other classes, see ios.

};

class streambuf

{

public:

    int      in_avail();
    int      out_waiting();
    int      sbumpc();
    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 streampos

             seekoff(streamoff, ios::seek_dir, int =ios::in|ios:out);

    virtual streampos

             seekpos(streampos, int =ios::in|ios:out);

    virtual int sync();

};
```

The *streambuf* class supports buffers into which characters can be inserted (*put*) or from which characters can be fetched (*get*). 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 also *filebuf*, *sstreambuf* and *stdiobuf*).

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. Queue-like buffers (see *strstream* and *sstreambuf*) 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*) 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 *sbufprot*, which describes the protected interface.

The public member functions of the *streambuf* class are described below. In the following descriptions, assume that:

- *i*, *n* and *len* are *int*
- *c* is an *int*. *c* holds a "character" value or EOF. A "character" value is always positive even when *char* is normally sign extended.
- *sb* and *sb1* are *streambuf**
- *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

int i=sbin_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 are reported.

int i=sbout_waiting()

> Returns the number of characters in the put area that have not been consumed (by the ultimate consumer).

int c=sbsbumpc()

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.

int c=sbsgetc()

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.

streambuf* sb1=sb->setbuf(char * ptr, int len, int 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 honoured depends on details of the derived class.
setbuf() normally returns *sb*, but if it does not accept the offer or honour the request, it returns 0.

int i=sb->sgetn(char * ptr, int 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.

int c=sbsnextc()

Moves the get pointer forward one character and returns the character following the new position. If the pointer is currently at the end of the sequence or is at the end of the sequence after moving forward, EOF is returned.

int i=sb->sputbackc(int 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.

int i=sb->sputc(int 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.

int i=sb->sputn(const char * ptr, int 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.

void sb stossc()

Moves the get pointer forward one character. If the pointer started at the end of the sequence this function has no effect.

streampos pos=sb->seekoff(streamoff off, ios::seek_dir dir, int mode)

Repositions the get and/or put pointers (i.e. the abstract get and put pointers, not *pptr()* and *gptr()*). *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.)

A class derived from *streambuf* is not required to support repositioning. *seekoff()* returns EOF if the class does not support repositioning. If the class does support repositioning, *seekoff()* returns the new position or EOF on error.

streampos pos=sb->seekpos(streampos pos, int 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 variable of type *streampos* should not have arithmetic performed upon it. Two particular values have special meaning:

streampos(0)

The beginning of the file.

streampos(EOF)

Used as an error indication.

int i=sbsync()

Establishes consistency between the internal data structures and the external source or sink. The details of this function depend on the derived class. *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. In other words, *sync()* "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.

EXAMPLE The following program defines a variable of type *filebuf* attached to *cin* and reads in blocks of characters from that *filebuf* until end of file is reached. Then the program determines the number of characters read in. Each newline character (\n) represents one character:

```
#include <iostream.h>
#include <fstream.h>
int main()
{
  filebuf in_file(0);
  /* in_file is connected to cin */
  const int N = 10;
  int k;
  char text_b[N+1];
  /* text buffer */
  cout << "Please enter " << N << " characters :\n";
  cout.flush();
  k = in_file.sgetn(&text_b[0],N);
  cout << " " << (k+in_file.in_avail()) ;
  cout << " characters have been entered.\n";
       /* Each \n represents one character. */
  return 0;
}
```

The result of executing the program is:

```
Please enter 10 characters :
0123456789
 11 characters have been entered.
%  CCM0998 CPU time used: 0.0040 seconds
```

The user may change the buffer size by calling the *setbuf()* member function. Any buffer size may be set with *setbuf()*.

BUGS *setbuf* does not really belong in the public interface. It is there for compatibility with the stream package.

SEE ALSO

istream, *sbufprot*

4.10 sstreambuf Specialization of streambuf for arrays

This section describes how a string may be used as a stream buffer.

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

class strstreambuf : public streambuf

{

public:

                strstreambuf() ;
                strstreambuf(char*, int, char* pstart=0);
                strstreambuf(int);
                strstreambuf(unsigned char*, int, unsigned char* pstart=0);
                strstreambuf(void* (*a)(long), void(*f)(void*));
                 ~strstreambuf();

   void         freeze(int n=1) ;
   char*        str();

   virtual int  doallocate();
   virtual int  overflow(int);

   virtual streampos

                seekoff(streamoff, ios::seek_dir, int);

   virtual streambuf*

                setbuf(char* p, int n);

   virtual int  underflow();

};
```

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 *sbufpub)* 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 dynamicmode. When a *strstreambuf* is in dynamic mode, space for the character sequence is allocated as needed. When the sequence is extended too far, it is copied to a new array.

In the following descriptions assume:

- *ssb* is a *strstreambuf**.
- *sb* is a *streambuf**.
- *ptr* and *pstart* are *char**s or *unsigned char**

Constructors

strstreambuf()

> Constructs an empty *strstreambuf* in dynamic mode. This means that space is automatically allocated to accommodate 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 are to be inserted,the program should use *setbuf()* (described below) to inform the *strstreambuf*.

strstreambuf(void (*a)(long), void* (*f)(void*))

> Constructs an empty *strstreambuf* in dynamic mode. *a* is used as the allocator function in dynamic mode. The argument passed to *a* is a *long* denoting the numberof bytes to be allocated. If *a* is null, operator *new* is used. *f* is used to free (or delete) areas returned by *a*. The argument to *f* is a pointer to the array allocated by *a*. If *f* is null, operator *delete* is used.

strstreambuf(int n)

> Constructs an empty *strstreambuf* in dynamic mode. The initial allocation of space is at least *n* bytes.

strstreambuf(char * ptr, int n, char * pstart)
strstreambuf(unsigned char * ptr, int n, unsigned char * pstart)

> Constructs a *strstreambuf* to use the bytes starting at *ptr*. The *strstreambuf* is in static mode; it does 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) constitutes 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 are 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(int n)

> Inhibits (when *n* is non-zero) 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.

char* ptr=ssbstr()

> Returns a pointer to the first *char* of the current array and freezes *ssb*. If *ssb* was constructed with an explicit array, *ptr* points to that array. If *ssb* is in dynamic allocation mode, but nothing has yet been stored, *ptr* may be null.

streambuf * sb=ssb->setbuf(char *, int n)

> *ssb* remembers *n* and the next time it does a dynamic mode allocation, it makes sure that at least *n* bytes are allocated.

EXAMPLE The following program declares a variable of type *strstreambuf* and initializes it with string *p*. The *str()* member function is called to ensure that the text string *p* is successfully processed by the *strstreambuf* constructor.

```
#include <strstream.h>
#include <iostream.h>
#include <string.h>
char * const p = "A very long string indeed \
                  abcdefghijlkmnopqrstuvwxyz\n";
int main()
{
  strstreambuf s(p, 0, (char *) NULL);
  /* The string p is the strstreambuf.     */
  /* The get pointer is to the start of p. */
  char *tp = s.str();
  cout << "length of original string " << strlen(p) << endl;
  cout << "length of strstreambuf string " << strlen(tp) << endl;
  return 0;
}
```

The result of executing the program is:

```
length of original string 77
length of strstreambuf string 77
%  CCM0998 CPU time used: 0.0018 seconds
```

Note how the original string length has not changed..

SEE ALSO

sbufpub, *strstream*

4.11 stdiobuf Specialization of iostream for stdio FILEs

This section describes *stdiobuf*, which is a class which specializes a *streambuf* to dealwith the top level input/output structure FILE.

stdiobuf is intended to be used when mixing C and C++ code in the same program. New C++ code should use *filebuf*.

```
#include <iostream.h>
#include <stdiostream.h>
#include <stdio.h>

class stdiobuf : public streambuf

{

public:

                stdiobuf(FILE* f);

   FILE *       stdiofile();

   virtual      ~stdiobuf();

   virtual int  overflow(int c=EOF);
   virtual int  pbackfail(int c);

   virtual streampos

                seekoff(streamoff, ios::seek_dir, int);

   virtual      int sync();
   virtual int  underflow();

};

class stdiostream : public ios

{

public:

                stdiostream(FILE*);

                ~stdiostream();

   stdiobuf *   rdbuf();

};
```

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 is turned back on.

In the following descriptions, assume that:

- *std* is a *stdiobuf*.
- *sts* is a *stdiostream*.
- *fp* is a *FILE **.

Constructors

stdiobuf(FILE * fp)

> Constructs a *stdiobuf* in unbuffered mode, and associates it with *fp*.

stdiostream(FILE * fp)

> Constructs a *stdiostream*, and associates it with *fp*.

stdiobuf members

FILE * fp = std.stdiofile()

> Returns the file pointer connected to *stdiobuf*.

int l = std.overflow(int c)

> Returns EOF if the file connected to *stdiobuf* is closed or *c*=EOF. Calls *putc()* and returns its value otherwise.

int l = std.pbackfail(int c)

> Returns the value of *ungetc()*.

streampos sp = std.seekoff(streamoff p, ios::seek_dir d, int l)

> Parameter *l* is ignored. Returns the value returned by the associated *fseek()*.

int l = std.sync()

> Calls *fflush()* if the last operation was a write access. Returns *fseek()*'s return value for the current position.

int l = std.underflow()

> Returns EOF if the file connected to *stdiobuf* is closed or end-of-file has been encountered. Returns the next character otherwise.

stdiostream member

stdiobuf * std = sts.rdbuf()

> Returns a pointer to the *stdiobuf* connected to *sts*.

EXAMPLE The following program opens the file *#TEMP*, attaches a variable of type *stdiobuf* to this file, and then prints a message to show if the *stdiobuf* is attached properly to the file.

```
#include <stdiostream.h>
#include <stdio.h>
#include <iostream.h>
#include <stdlib.h>
int main()
{
  FILE *qw;
  const char * const name = "#TEMP";
  if (!(qw = fopen(name, "w")))
  {
    cerr << "Can't open " << name << ".\n";
    exit(1);
  }
  stdiobuf s(qw);
  FILE *rt = s.stdiofile();
  if (rt != qw)
  {
    cerr << "Error in stdiofile().\n";
  }
  else
  {
    cerr << "stdiofile() is working ok.\n";
  }
  return 0;
}
```

The result of executing the program is:

```
stdiofile() is working ok.
%  CCM0998 CPU time used: 0.0086 seconds
```

This program shows that the *stdiofile()* member function of *stdiobuf* returns the correct result in this case.

SEE ALSO

filebuf, *istream*, *ostream*, *sbufpub*

4.12 strstream Specialization of iostream for arrays

This section describes class *strstream*, which is a specialization of class *iostream*.*strstream* deals with input and output style operations on arrays of bytes.

```
#include <iostream.h>

class ios

{
public:enum open_mode {in, out, ate, app, trunc, nocreate, noreplace,

                                        bin, tabexp};

// and lots of others, see ios ...

};

#include <strstream.h>

class strstreambase : public virtual ios

{

public:

    strstreambuf*  rdbuf();

};

class istrstream : public strstreambase, public istream

{

public:

                    istrstream(char*);
                    istrstream(char*, int);
                    istrstream(const char*);
                    istrstream(const char*, int);
                     ~istrstream();

};

class ostrstream : public strstreambase, public ostream

{

public:

                ostrstream();
                ostrstream(char*, int, int=ios::out);
                 ~ostrstream();

    int         pcount();
    char*       str();

};
```

```
class strstream : public strstreambase, public iostream

{

public

            strstream();
            strstream(char*, int, int mode);
             ~strstream();

   char*    str();

};
```

---

strstreambase provides the *rdbuf()* member function. Defining *strstreambase*s is noti ntended.

strstream specializes *iostream* for storing and fetching from arrays of bytes. The *streambuf* associated with a *strstream* is a *strstreambuf* (see *sstreambuf*).

In the following descriptions assume:

- *ss* is a *strstream*.
- *iss* is an *istrstream*.
- *oss* is an *ostrstream*.
- *mode* is an *int* representing an *open_mode*.

Constructors

istrstream(char * cp)

> Characters are fetched from the (null-terminated) string *cp*. The terminating null character is not part of the sequence. Seeks (*istream::seekg()*) are allowed within that array.

istrstream(char * cp, int len)

> Characters are fetched from the array beginning at *cp* and extending for *len* bytes.Seeks (*istream::seekg()*) are allowed anywhere within that array.

ostrstream()

> Space is dynamically allocated to hold stored characters.

ostrstream(char * cp, int n, int mode)

> Characters are stored into the array starting at *cp* and continuing for *n* bytes. If *ios::ate* or *ios:: app* is set in *mode*, then *cp* is assumed to be a null-terminated string and storing begins at the null character. Otherwise, storing begins at *cp*. Seeks are allowed anywhere in the array.

strstream()

> Space is dynamically allocated to hold stored characters.

strstream(char * cp, int n, int mode)

Characters are stored into the array starting at *cp* and continuing for *n* bytes. If *ios::ate* or *ios::app* is set in *mode*, then *cp* is assumed to be a null-terminated string and storing begins at the null character. Otherwise, storing begins at *cp*. Seeks are allowed anywhere in the array.

strstreambase members

strstreambuf * ssb = iss.rdbuf()
strstreambuf * ssb = oss.rdbuf()
strstreambuf * ssb = ss.rdbuf()

rdbuf() may be used in derived classes only. Returns the *strstreambuf* connected to *iss*/*oss*/*ss*.

ostrstream members

char * 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.

int 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 member

char * 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

.

EXAMPLE The following program defines a string *str1*, and then reads from the string like an input stream, by using the >> operator. Each character read from the string is printed on *cout*.

```
#include <iostream.h>
#include <strstream.h>
const char * const str1 = "A test string to check strstream\n";
/* Use const to make sure that the string, and the pointer */
/* to it, cannot be changed.                               */
int main()
{
  istrstream is((char*) str1);
  /* Declare variable is using str1 string                */
  is.unsetf(ios::skipws);
  /* By default, an istrstream will skip white space on    */
  /* input. Change the default behaviour by clearing the   */
  /* skipws flag so that it will not skip white space on   */
  /* input.                                                */
  while (EOF != is.peek())
  {
    char c;
    is >> c;
    /* Note how the text string is accessed like an input  */
    /* string.                                             */
    cout << c;
  }
  return 0;
}
```

The result of executing the program is:

```
A test string to check strstream
%  CCM0998 CPU time used: 0.0007 seconds
```

SEE ALSO

istream, *sstreambuf*

5 References

The manuals are available as online manuals, see https://bs2manuals.ts.fujitsu.com.

[1] **CRTE** (BS2000)
Common RunTime Environment
User Guide

Target group
Programmers and system administrators in a BS2000 environment

Contents
Description of the common runtime environment for COBOL85, C and C++ objects and for "foreign language mix"

- components of CRTE
- ILCS program interface
- linkage examples

[2] **C** (BS2000)
C Library Functions
Reference Manual

Target group
C users working with BS2000
Contents

- Descriptions of all C functions and macros provided by the C runtime library
- `Basic information, programming notes and examples for: file processing, STXIT and contingency routines, locality`

[3] **C++** (BS2000)
C++ Compiler
User Guide

Target group
Cand C++ users in a BS2000 environment
Contents

- Description of all activities concerned with the creation of executable C and C++ programs: compilation, linking, loading, debugging
- Programming notes and additional information on: program runtime control, file processing, event handling, locale concept, language interfacing, C and C++ language features of the C++ compiler

[4] **The C++ Programming Language**
by Bjarne Stroustrup

Target group
C++ programmers and programmers wishing to learn C++.
Contents
This standard work by C++ originator Bjarne Stroustrup includes an introduction to C and C++ with a large number of examples, three chapters on software development using C++ and a complete reference manual.