

---

# Preface

The C/C++ development system CDS++ provides you with C++ class libraries for data-stream-oriented I/O and complex mathematics that are compatible to Cfront V3.0.3. The Cfront C++ libraries were last released with the SNI C++ V3.1 B/C (Reliant UNIX) compiler.


The Cfront C++ classes for complex mathematics (*libcomplex.a* or *libcomplex.so* library) and stream-oriented I/O (*libC.a* or *libC.so* library) are available if the program is compiled and linked in the Cfront C++ mode of the CDS++ compiler (*-X d* option).

The Cfront C++ classes for stream-oriented I/O also currently function as the I/O interface in the ANSI C++ modes of the CDS++ compiler (*-X w*, *-X e* options) since the standard I/O in conformance with ANSI/ISO C++ will not be available until a later version of the CDS ++ development system. The modules are integrated in the standard C++ libraries *libCstd.a* and *libCstd.so*.

A thread-safe version of all libraries is also available. These can only be used in conjunction with the thread package of the product DCE (Reliant UNIX) as of V2.0.

## Notational conventions

In this manual the following conventions are used for statement formats and user entries.

<i>italics</i>	Commands, invariable file names and other constant terms; for example the names of files, parameters, etc.
constant width	In format specifications: sample names for files, parameters, etc. In examples: input and output on screen
\$	System prompt, ready for user entries
␣	The syntax requires at least one blank character
[]	Entries enclosed in square brackets may be omitted
	Important information
{... ...}	One of the alternatives separated by the   character(s) must be specified

---

# Complex math classes and functions



The following description applies only to the Cfront C++ language mode of the CDS++ compiler.

Please refer to the "Standard C++ Library" manual for a description of the interfaces available for complex mathematics (`<complex>`) in conformance with ANSI/ISO C++ in the ANSI C++ modes.

## cplxintro - Introduction to complex mathematics

The complex math library *libcomplex.a* contains classes, functions, and operators to process data of the user-defined type *complex* in C++ programs.

The complex math library *libcomplex.a* must be specified explicitly, by means of the *-l complex* option at the time of compilation or linkage. Depending on which parts of the library are being used for complex math, it will also be necessary to link in the C math library *libm.a*. This is done by entering the option *-l m* at the end of the call to CC:

```
$ CC code.C -lcomplex -l m
```

Declarations for complex math functions are contained in the header file *complex.h*. This file can be included in the program using the preprocessor directive *#include* as shown below:

---

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

---

The complex math library implements the data type for complex numbers in the class *complex*. This is achieved by overloading the usual input, output, arithmetic, assignment, and comparison operators to work with complex numbers. These operators are discussed in the *cplxops* section (page 11).

Besides the above operators, standard math functions such as exponential, logarithmic, power, and square root functions (see *cplxexp*), and trigonometric functions such as sine, cosine, hyperbolic sine, and hyperbolic cosine (see *cplxtrig*) are also overloaded. Routines to convert between Cartesian and Polar coordinate systems are discussed in the *cplxcartpol* section. Error handling is described under *cplxerr*.

## RETURN VALUES

Functions in the complex math library may return the conventional value pairs (0, 0), (0,  $\pm$ HUGE), ( $\pm$ HUGE, 0), or ( $\pm$ HUGE,  $\pm$ HUGE), when the function is undefined for the given arguments or when the value is not representable. (HUGE is the largest-magnitude single-precision floating point number and is defined in the file *math.h*. The header file *math.h* is included in the file *complex.h*.) In these cases, the external variable *errno* (see “Programmer’s Reference Manual”) is set to the value EDOM or ERANGE.

## FILES

complex.h  
libcomplex.a

## EXAMPLE

The following program fragment in *cplx.in.C* declares a complex variable, initializes it, and then outputs its value:

```
#include <stream.h>
#include <complex.h>

int main()
{
    complex c(1.0, 1.0);
    cout << "The complex number is " << c << "\n";
    return 0;
}
```

Compile *cplx.in.C* with the command:

```
$ CC -X d cplx.in.C -l complex
```

Run the compiled program with:

```
$ a.out
```

```
The complex number is (1,1)
```

Note that the operator << is overloaded for data type *complex* so that complex numbers can be printed easily.

## SEE ALSO

*cplxcartpol* (page 5), *cplxerr* (page 7), *cplxexp* (page 9), *cplxops* (page 11), *cplxtrig* (page 14)

## cplxcartpol - Cartesian/Polar functions

This section describes the *complex* functions used for conversions between Cartesian and Polar coordinate systems.

---

```
#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 */
};
```

---

The following functions are defined for the data type *complex*:

**abs(complex x)**

Returns the absolute value or magnitude of  $x$ .

**arg(complex x)**

Returns the angle of  $x$ , measured in radians in the range  $-\pi$  to  $+\pi$ .

**conj(complex x)**

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

**imag(complex x)**

Returns the imaginary part of  $x$ .

**norm(complex x)**

Returns the square of the magnitude of  $x$ , and is intended for comparison of magnitudes. `complex::norm()` is faster than `complex::abs()`, but is more likely to cause an overflow error.

**polar(double m, double a=0.0);**

Given a pair of polar coordinates, magnitude  $m$ , and angle  $a$  measured in radians, in the range  $-\pi$  to  $+\pi$ .

`real(complex x)`  
Returns the real part of  $x$ .

## EXAMPLE

The following program fragment in *cplxcartpol.C* converts a complex number to the Polar coordinate system and then prints it:

```
#include <stream.h>
#include <complex.h>

main ()
{
    complex d;
    d = polar (10.0, 0.7);
    cout <<real(d)<<" " <<imag(d);
    cout <<"\n";
    return 0;
}
```

Compile *cplxcartpol.C* with the following command:

```
$ CC -X d -o cplxcartpol cplxcartpol.C -l complex -lm
```

Run the compiled program with:

```
$ cplxcartpol
7.64842 6.44218
```

## SEE ALSO

*cplxintro* (page 3), *cplxerr* (page 7), *cplxexp* (page 9), *cplxops* (page 11), *cplxtrig* (page 14)

## cplxerr - Error handling functions

This section describes the error handling function implemented in the C++ complex math library.

---

```
#include <complex.h>

static const complex complex_zero(0,0);
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&);
};
```

---

Users may define their own procedures for handling errors, by incorporating a function named *complex\_error* in their program.

`complex_error(c_exception & x)`

This is invoked when errors are detected in functions from the complex math library.

In the class *c\_exception*, the element *type* is an integer describing the type of error that has occurred; *type* must have one of the following values (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 produced the error. The variables *arg1* and *arg2* are the arguments with which the function was invoked. *retval* is set to the default value that is returned by the function unless the user's *complex\_error* sets it to a different value.

If the user's *complex\_error* function returns a non-zero value, no error message is printed, and *errno* is not set.

The C++ error handling routine `complex_error` only handles errors caused by the use of one of the following four functions:

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

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

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 procedures			
	Types of Errors		
<i>type</i>	SING	OVERFLOW	UNDERFLOW
<i>errno</i>	EDOM	ERANGE	ERANGE
<i>exp()</i> real too large or small imag too large	–	( H, H) (0, 0)	(0, 0) –
<i>log()</i> arg = (0, 0)	M, (H, 0)	–	–
<i>sinh()</i> real too large imag too large	–	( H, H) (0, 0)	– –
<i>cosh()</i> real too large imag too large	–	( H, H) (0, 0)	– –

## SEE ALSO

`cplxintro` (page 3), `cplxcartpol` (page 5), `cplxexp` (page 9), `cplxops` (page 11), `cplxtrig` (page 14), `matherr()` in “Programmer’s Reference Manual”



## cplxexp - Transcendental functions

This section describes the *complex* math functions for calculating natural logarithms, exponentials, square roots, and the value of one argument raised to the power of another argument.

---

```
#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);
};
```

---

The following overloaded math functions are contained in the complex math library:

`exp(complex x)`  
Returns  $e^x$ .

`log(complex x)`  
Returns the natural logarithm of  $x$ .

`pow(complex x, complex y)`  
Returns  $x^y$ .

`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 the following values:

- (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  $\leq 0$ ;
- (-HUGE, HUGE) if the sine is  $> 0$  and the cosine is  $\leq 0$ ;
- (-HUGE, -HUGE) if the sine and cosine are  $\leq 0$ .

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

The function *complex::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 procedures can be changed with the function *complex\_error()*.

## EXAMPLE

The following program fragment in *cplxexp.C* prints a set of complex numbers and their exponential powers.

```
#include <stream.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;
}
```

Compile *cplxexp.C* with the following command:

```
$ CC -X d cplxexp.C -l complex -l m
```

Run the compiled program with:

```
$ a.out
(1,1) (1.46869, 2.28736)
(2,2) (-3.07493, 6.71885)
(3,3) (-19.8845, 2.83447)
```

Note that the operator << is overloaded for data type *complex* so that complex numbers can be printed easily.

## SEE ALSO

*cplxintro* (page 3), *cplxcartpol* (page 5), *cplxerr* (page 7), *cplxops* (page 11), *cplxtrig* (page 14)

## cplxops - Operators

This section describes the arithmetic, comparison, and assignment operators which are overloaded for complex numbers. Note that the entire range of mathematical operations is available for *complex* objects.

---

```
#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);
};
```

---

The operators have their conventional precedences. In the following descriptions for *complex* 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 complex number  $y$ .

 $x -= y$ 

Complex number  $x$  is assigned the value of the arithmetic difference of itself and complex number  $y$ .

 $x *= y$ 

Complex number  $x$  is assigned the value of the arithmetic product of itself and complex number  $y$ .

 $x /= y$ 

Complex number  $x$  is assigned the value of the arithmetic quotient of itself and complex number  $y$ .



The above 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.

## EXAMPLE

The following program fragment in *cplxops.C* defines the complex variables *d* and *c*, divides *d* by *c*, and then prints the values of *c* and *d*:

```
#include <stream.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;
}
```

Compile *cplxops.C* with the following command:

```
$ CC -X d cplxops.C -l complex -l m
```

Run the compiled program with:

```
$ a.out
(2, 2) (5.25, 0.25)
(2, 2) (1.375, -1.25)
```

## SEE ALSO

*cplxintro* (page 9), *cplxcartpol* (page 5), *cplxerr* (page 7), *cplxexp* (page 9), *cplxtrig* (page 14)

## 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);
};
```

---

The following trigonometric functions are defined for *complex* objects:

`sin(complex x)`  
Returns the sine of  $x$ .

`cos(complex x)`  
Returns the cosine of  $x$ .

`sinh(complex x)`  
Returns the hyperbolic sine of  $x$ .

`cosh(complex x)`  
Returns the hyperbolic cosine of  $x$ .

### RETURN VALUES

If the imaginary part of  $x$  causes an overflow, `complex::sinh()` and `complex::cosh()` return the value (0.0, 0.0). When the real part is large enough to cause an overflow, the functions `complex::sinh()` and `complex::cosh()` return the following values:

- (HUGE, HUGE) if the cosine and sine of the imaginary part of  $x$  are  $\geq 0$ ;
- (HUGE, -HUGE) if the cosine is  $\geq 0$  and the sine is  $< 0$ ;
- (-HUGE, HUGE) if the sine is  $\geq 0$  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 procedures may be changed with the function `complex_error()` (see `cplxerr`).

## EXAMPLE

The following program fragment in *cplxtrig.C* prints a range of complex numbers and the corresponding values calculated by the function *complex::cosh()*:

```
#include <stream.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;
}
```

Compile *cplxtrig.C* with the following command:

```
$ CC -X d cplxtrig.C -l complex -l m
```

Run the compiled program with:

```
$ a.out
```

The result of executing the program:

```
(0, 0) (1, 0)
(1, 1) (0.83373, 0.988898)
(2, 2) (-1.56563, 3.29789)
```

Note that the operator `<<` is overloaded for data type *complex* so that complex numbers can be printed easily.

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

## SEE ALSO

*cplxintro* (page 3), *cplxcartpol* (page 5), *cplxerr* (page 7), *cplxexp* (page 9), *cplxops* (page 11)





---

## Classes and functions for stream I/O



The following description is currently valid for both the Cfront C++ language mode and the ANSI C++ language modes of the CDS++ compiler.

I/O interfaces that conform to ANSI/ISO C++ will be available in the ANSI C++ modes in a later version of the CDS++ development system. At that time, you will be able to find a description of the interfaces in the appropriate manual for the standard C++ library.

### iosintro - Introduction to buffering, formatting, and input/output

The libraries *libC.a* (Cfront C++ mode) and *libCstd.a* (ANSI C++ modes) contains classes and functions for formatted and non-formatted I/O in C++ programs.

This section describes the primary mechanism used to implement input and output in C++ programs.

The C++ *iostream* package declared in various header files consists primarily of a collection of classes. Although originally intended only to support input/output, the package now supports related activities such as incore formatting. This package is a mostly source-compatible extension of the earlier datastream-oriented I/O package.

---

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

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

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

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

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

---

In the *iostream* package, there are some functions which return characters, but which use *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 derived from the core classes. Both groups of classes are listed below.

## Core classes

The core of the *iostream* package comprises the following classes:

### streambuf

This is the base class for buffers. It supports the output (storing) and input (extraction) of characters. Most members are inlined for efficiency. The public interface of the class *streambuf* is described in *sbufpub* and the protected interface (for derived classes) is described in *sbufprot*.

### ios

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 *streambufs*. See *istream*.

**ostream**

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

**iostream**

This class combines *istream* and *ostream*. It is intended for situations in which bidirectional operations (i.e. output to or input 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 the corresponding class without assignment. The predefined streams (see below) *cin*, *cout*, *cerr*, and *clog*, are objects of these classes. See *istream*, *ostream*, and *ios*.

**iostream\_init**

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

**Predefined streams**

The following streams are predefined:

**cin**

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

**cout**

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

**cerr**

The standard error stream (file descriptor 2). Output through this stream is unbuffered, which means that characters are flushed from the buffer after each output (inserter 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 them 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 can use 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 or consumed. 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. Members support opening, closing, and seeking. Common uses do not require the program to manipulate file descriptors. See page 22.

#### *stdiobuf*

This buffer class supports I/O through *stdio* FILE structures. It is intended for use when mixing C and C++ code. New code should prefer to use *filebufs*. See page 77.

#### *strstreambuf*

This buffer class stores and fetches characters from arrays of bytes in memory (i.e., strings). See page 74.

### Classes derived from *istream*, *ostream*, and *iostream*

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

#### *ifstream*

#### *ofstream*

#### *fstream*

These classes support formatted I/O to and from files. They use a *filebuf* to do the I/O.

#### *istrstream*

#### *ostrstream*

These classes support “in core” formatting. They use a *sbuf*. See page 79.

#### *stdiostream*

This class specializes *iostream* for *stdio* FILEs.

## BUGS

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

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

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

## SEE ALSO

*filebuf* (page 22), *fstream* (page 26), *ios* (page 30), *istream* (page 41), *manip* (page 48), *ostream* (page 53), *sbufprot* (page 61), *sbufpub* (page 69), *strstream* (page 79), *ssbuf* (page 74), *stdiobuf* (page 77)

## filebuf - Buffer for file input/output

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

---

```

#include <iostream.h>

typedef long streamoff, streampos;

class ios
{
public:
    enum          io_state {goodbit=0, eofbit=1, failbit=2, badbit=4, hardfail=0200};
    enum          seek_dir {beg, cur, end};
    enum          open_mode {in, out, ate, app, trunc, nocreate, noreplace};
    // 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(int d);
                                filebuf(int d, char* p, int l);
    filebuf*      attach(int d);
    filebuf*      close();
                                ~filebuf();
    int           fd();
    int           is_open();
    filebuf*      open(const char *name, int mode, int prot=openprot);
    virtual streampos seekoff(streamoff, ios::seek_dir, int);
    virtual streambuf* setbuf(char* p, int len);
    virtual int    sync();
    virtual int    overflow(int=EOF);
    virtual int    underflow();
};

```



All the members and member functions identified below in italics which are not specified by any different class belong to the *filebuf* class.

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

The reserve area (or buffer, see page 69 and page 61) is allocated automatically if it is not specified explicitly with a constructor or a call to *setbuf()*. *filebufs* can also be made unbuffered with certain arguments to the constructor or *setbuf()*, in which case a system call is made for each character that is read or written, (this is very resource intensive and not recommended for normal use).

## Constructors

*filebuf*()

Constructs an initially closed *filebuf*.

*filebuf*(int *d*)

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

*filebuf*(int *d*, char\* *p*, int *l*)

Constructs a *filebuf* connected to file descriptor *d*, and initialized to use the reserve area starting at *p* and containing *l* bytes. If *p* is *NULL*, or *l* is zero or less, the *filebuf* is unbuffered.

## Members

*f.attach*(int *d*)

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

*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.

*f.fd*()

Returns the file descriptor to which *f* is connected. If *f* is closed, *fd* returns EOF.

*f.is\_open*()

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

**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 with protection mode *prot*, unless *ios::nocreate* is specified in *mode*. By default, *prot* is *filebuf::openprot*, which is 0644. Failure occurs if *f* is already open. *open()* normally returns *&f*, but if an error occurs it returns 0. The members of *ios::open\_mode* are bits that may be or'ed together. (Because the or'ing returns an *int*, *open()* takes an *int* rather than an *ios::open\_mode* argument.) The meanings of these bits in *mode* are described in detail in *fstream* on page 26

**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* on page 69. *mode* is ignored. *seekoff()* returns the new position, or EOF if a failure occurs. The position in the file after a failure is undefined.

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

Sets up the reserve area as *len* bytes beginning at *p*. If *p* 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 the buffering status, and *setbuf()* returns the value *NULL*.

**f.sync()**

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

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.



## EXAMPLE

The following program fragment in *filebuf.C* 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 <stream.h>
#include <fstream.h>
#include <osfcn.h>

int main()
{
    filebuf b;      /* constructor with no parameters called */
    if (b.attach(1))
    {
        cout << "have attached filebuf b to file descriptor 1\n";
        b.detach();
    }
    else
    {
        cerr << "can't attach filebuf to file descriptor 1\n";
        exit(1);    /* error return */
    }
    return 0;
}
```

Compile *filebuf.C* with the following command:

```
$ CC -X d filebuf.C # Cfront C++ mode
```

or

```
$ CC -X w filebuf.C # ANSI C++ mode
```

Run the compiled program with:

```
$ a.out
have attached filebuf b to file descriptor 1
```

## BUGS

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

There is no way to force atomic (unsplittable) reads.

As the operating system does not usually report failures of the system call *seek()* (e.g. on a *tty* device), nor does a *filebuf*.

## SEE ALSO

*fstream* (page 26), *sbufprot* (page 61), *sbufpub* (page 69)  
*lseek* in the "Programmer's Reference Manual"

## fstream - Specialization of iostream and streambuf for files

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

---

```

#include <fstream.h>

class fstreambase : virtual public ios
{
public:
    fstreambase();
    fstreambase(const char* name, int mode, int prot=filebuf::openprot);
    fstreambase(int fd);
    fstreambase(int fd, char* p, int l);
    ~fstreambase();
    void open(const char* name, int mode, int prot=filebuf::openprot);
    void attach(int fd);
    int detach();
    void close();
    void setbuf(char* p, int l);
    filebuf* rdbuf() { return &buf; }

private:
    filebuf buf;

protected:
    void verify(int);

};

class ifstream : public fstreambase, public istream
{
public:
    ifstream();
    ifstream(const char* name, int mode=ios::in, int prot=filebuf::openprot);
    ifstream(int fd);
    ifstream(int fd, char* p, int l);
    ~ifstream();
    filebuf* rdbuf() { return fstreambase::rdbuf(); }
    void open(const char* name, int mode=ios::in, int prot=filebuf::openprot);

};

class ofstream : public fstreambase, public ostream
{
public:

```

```

        ofstream();
        ofstream(const char* name, int mode=ios::out, int prot=filebuf::openprot);
        ofstream(int fd);
        ofstream(int fd, char* p, int l);
        ~ofstream();
    filebuf*   rdbuf() { return fstreambase::rdbuf(); }
    void      open(const char* name, int mode=ios::out, int prot=filebuf::openprot);
};

class fstream : public fstreambase, public iostream
{
public:
        fstream();
        fstream(const char* name, int mode, int prot=filebuf::openprot);
        fstream(int fd);
        fstream(int fd, char*p, int l);
        ~fstream();
    filebuf*   rdbuf() { return fstreambase::rdbuf(); }
    void      open(const char* name, int mode, int prot=filebuf::openprot);
};

```

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



All the members and member functions identified below in italics which are not specified by any different class belong to one of the above classes.

## Constructors

In *xstream*, *x* is either *if*, *of*, or *f* so that *xstream* stands for: *ifstream*, *ofstream*, or *fstream*

The constructors for *xstream* are:

*xstream*()

Constructs an unopened *xstream*.

*xstream*(char\* name, int mode, int prot)

Constructs an *xstream* and opens file *name* using *mode* as the open mode and *prot* as the protection mode. By default, *prot* is *filebuf::openprot*, which is 0644. The error state (*io\_state*) of the constructed *xstream* indicates failure in case the *open* fails.

*xstream*(int fd)

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

`xstream(int fd, char* p, int l)`

Constructs an *xstream* connected to file descriptor *fd*, and, in addition, initializes the associated *filebuf* to use the *l* bytes at *p* as the reserve area. If *p* is null or *l* is 0, the *filebuf* is unbuffered.

## Member functions

`f.attach(int fd)`

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

`f.detach()`

Breaks the connection between *f* and the file descriptor and releases the file descriptor. Before the connection is broken, the data buffered for output are flushed out.

`f.close()`

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

`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 with protection mode *prot* unless *ios::nocreate* is set. By default, *prot* is *filebuf::openprot*, which is 0644. Failure occurs if *f* is already open, or the system call *open()* fails. *ios::failbit* is set in *f*'s error status on failure. The members of *open\_mode* are bits that may be or'ed together. (Because the or'ing returns an *int*, *open()* takes an *int* rather than an *open\_mode* argument.)

The meanings of these bits in *mode* are:

*ios::app*

A seek to the end of file is performed. Subsequent data written to the file is always 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 exist, an open error occurs.

**ios::out**

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

**ios::trunc**

If the file already exists, its contents 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**

This variable is used to control files which are opened. If the file does not already exist, the *open()* fails.

**ios::noreplace**

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

**f.rdbuf()**

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

**f.setbuf(char\* p, int l)**

Has the usual effect of a *setbuf()* (see page 22), 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.

**SEE ALSO**

*filebuf* (page 22), *ios* (page 30), *istream* (page 41), *ostream* (page 53), *sbufpub* (page 69)  
*close()*, *open()* in the “Programmer’s Reference Manual”

## 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=1, failbit=2, badbit=4, hardfail=0200};
    enum        open_mode {in, out, ate, app, trunc, nocreate, noreplace};
    enum        seek_dir {beg, cur, end};

    /* flags for controlling format */
    enum
    {
        skipws=01,
        left=02, right=04, internal=010,
        dec=020, oct=040, hex=0100,
        showbase=0200, showpoint=0400,
        uppercase=01000, showpos=02000,
        scientific=04000, fixed=010000,
        unitbuf=020000, stdio=040000
    };

    static const long basefield; /* dec | oct | hex */
    static const long adjustfield; /* left | right | internal */
    static const long floatfield; /* scientific | fixed */

public:
    ios(streambuf*);
    virtual ~ios();
    int bad() const;
    static long bitalloc();
    void clear(int i=0);
    int eof() const;
    int fail() const;
    char fill() const;
    char fill(char);
    long flags() const {return x_flags;}
    long flags(long);
    int good() const;
    long& iword(int);
    int operator!() const;
```

```

operator    const void*() const;
operator    void*();
int         precision() const;
int         precision(int);
streambuf*  rdbuf();
void* &     pword(int);
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(int);
static int  xalloc();

protected:
            ios();
            void    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&);

```



All the members and member functions identified below in italics which are not specified by any different class belong to the *ios* class.

The stream classes derived from class *ios* provide a high level interface that supports transferring formatted and unformatted information into and out of *streambufs*.

Several enumerations are declared in class *ios* (*open\_mode*, *io\_state*, *seek\_dir*), plus format flags, to avoid burdening the global name space with these details. The *io\_states* are described in this section under “Error States” . The format fields are also described in this section under “Formatting” . The *open\_modes* are described in detail in *fstream* under *open()*. The *seek\_dirs* are described in *sbufpub* under *seekoff()* .

## 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)

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 *istreams* 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 as *protected*. Therefore *ios::init()* is declared *protected* and must be used for initialization of derived classes.

## Error states

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

*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())*.

*s.good()*

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

*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.

*s.fail()*

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



### 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!()* *const* and *operator const void\*()* *const*. The latter converts an *ios* to a pointer so that it can be compared to *NULL*. The conversion returns the value *NULL* 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. For other operations the format state has no particular effect and its components may be set and examined arbitrarily by user code. Most formatting details are controlled by using the *flags()*, *setf()*, and *unsetf()* functions to set the following flags, which are declared in an enumeration in class *ios*. Three other components of the format state are controlled separately with the functions *fill()*, *width()*, and *precision()*.

### skipws

If *skipws* is set, whitespace is skipped on input. This applies to numeric, character and string inputs.

In case of string input the extraction will stop at the first white space character. In the special 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 is 10 (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, numeric insertions are in decimal, but extractions (inputs) 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 floating point conversion.

scientific

fixed

These flags control the format to which a floating point value is converted for insertion into a stream.

- If *scientific* is set, the value is converted to 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 only if the exponent resulting from the conversion is less than -4 or greater than or equal to the *precision*.
- 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 *unitbuf* is set, a flush is performed by *ostream::osfx()* after each insertion. Unit buffering provides a compromise between buffered output and unbuffered output. Performance is better under unit buffering than unbuffered output, which makes a system call for each character output. Unit buffering makes a system call for each insertion operation, and doesn't require the user to call *ostream::flush()*.

stdio

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

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

s.fill(char c)

Sets the “fill character” format state variable to *c* and returns the previous value. *c* is used 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* (page 48).

The “fill character” has no effect on input.

s.fill()

Returns the “fill character” format state variable.

`s.flags()`

Returns the current format flags.

`s.flags(long f)`

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

`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 (output). The default is 6. A parameterized manipulator, *setprecision* is also available for setting the precision, see *manip* (page 48).

`s.precision()`

Returns the “precision” format state variable.

`s.setf(long b)`

Turns on in *s* the format flags marked in *b* and returns the previous settings. All other flags are left unchanged. A parameterized manipulator, *setiosflags* performs the same function, see *manip* (page 48).

`s.setf(long setbits, long field)`

Resets in *s* only the format flags specified by *field* to the settings marked in *setbits*, and returns the previous settings. That is, the format flags specified by *field* are cleared in *s*, then reset to be those marked in *setbits*. 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, field)* clears all the bits specified by *field*, as does a parameterized manipulator, *resetiosflags* (see *manip*).

`s.unsetf(long b)`

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

`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, values are never truncated during numeric insertions, so if the value being inserted does not fit in *field-width* 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* (page 41). When the field width is non-zero, it is taken to be the size of the array, and no more than *field-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*).

`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.

`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.

`ios::xalloc()`

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

`s.iword(int i)`

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

`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

`s.rdbuf()`

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

`ios::sync_with_stdio()`

Solves problems that arise when mixing *stdio* and *iostreams*. The first time it is called it resets the standard *iostreams* (*cin*, *cout*, *cerr*, *clog*; see *iosintro* (page 17)) to be streams using *stdiobufs*.

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` and `ios::stdio` above). Invoking `sync_with_stdio()` degrades performance a variable amount, depending on the length of the strings being inserted.



Shorter strings cause a greater performance degradation.

`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 non-zero and an `ios` needs more characters or has characters to be consumed, the `ios` pointed at by the tie variable is flushed. By default, `cin` is tied initially to `cout` so that attempts to get more characters from standard input result in flushing standard output. Additionally, `cerr` and `clog` are tied to `cout` by default. For other `ios`s, the tie variable is set to zero by default.

`s.tie()`

Returns the “tie” variable.

### Built-in manipulators

Some convenient manipulators (functions that take an `ios&`, an `istream&`, or an `ostream&` and return their argument, (see `manip`)) 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`.

```
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: 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 queue-like *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 two extraction operations, the extraction (input) can be continued normally.

## EXAMPLE

The following program fragment in *ios.C* uses some data members of class *ios* to change the output format of both integers and doubles on *cout*:

```
#include <stream.h>
#include <math.h>

void someoutput()
{
    int i;
    const int N = 12;
    /* better to use const int than hash define          */
    for (i = 1; i < N; i += 2)
    {
        cout << i << " " << pow( (double) i, (double) i) << "\n";
    }
    cout << "\n";
}

int main()
{
    someoutput();
    /* show default formats for integers and doubles    */
    cout.setf( ios::fixed, ios::floatfield);
    /* set the output format for floats and doubles to fixed */
    someoutput();

    cout.setf( ios::oct, ios::basefield);
    /* set the output format of integers to octal      */
    someoutput();

    return 0;
}
```

The program requires the C math library, so the file *ios.C* must be compiled with the *-l m* option:

```
$ CC -X d ios.C -l m # Cfront C++ mode
```

or

```
$ CC -X w ios.C -l m # ANSI C++ mode
```

Run the compiled program with:

```
$ a.out
```

```
1 1
3 27
5 3125
7 823543
9 3.8742e+08
11 2.85312e+11
```

```
1 1.000000
3 27.000000
5 3125.000000
7 823543.000000
9 387420489.000000
11 285311670610.999877
```

```
1 1.000000
3 27.000000
5 3125.000000
7 823543.000000
11 387420489.000000
13 285311670610.999877
```

The precision of these results depends on the machine used.

## BUGS

The old stream package had a constructor that took a *FILE\** argument. This is now replaced by *stdiostream*. To avoid having *iostream.h* depend on *stdio.h*, the old constructor is no longer declared.

The old *stream* package allowed copying of streams. This is disallowed by the new *iostream* package. However, objects of type *istream\_withassign*, *ostream\_withassign*, and *iostream\_withassign* can be assigned to. Old code using copying can usually be rewritten to use pointers or these classes. (The standard streams *cin*, *cout*, *cerr*, and *clog* are members of “*withassign*” classes, so they can be assigned to, as in *cin=inputfstream*.)

## SEE ALSO

*iosintro* (page 17), *istream* (page 41), *manip* (page 48), *ostream* (page 53), *sbufprot* (page 61), *sbufpub* (page 69)



## 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};
    /* flags for controlling format */
    enum
    {
        skipws=01,
        left=02, right=04, internal=010,
        dec=020, oct=040, hex=0100,
        showbase=0200, showpoint=0400,
        uppercase=01000, showpos=02000,
        scientific=04000, fixed=010000,
        unitbuf=020000, stdio=040000
    };
    // 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& c);
    istream&   get(char& c);
    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 n=1,int d=EOF);
    int        ipfx(int need=0);
    int        peek();
    istream&   putback(char);
};
```

```

istream& read(char* ptr, 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 {
    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.



All the members and member functions identified below in italics which are not specified by any different class belong to the *istream* class.

## 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 defined with an assignment.

## Input prefix function

`ins.ipfx(int need)`

If *ins*'s error state is non-zero, returns zero immediately. If necessary (and if it is non-null), any *ios* tied to *ins* is flushed (see the description *ios::tie()* in *ios*, page 30). Flushing is considered necessary if either *need*  $\neq$  0 or if there are fewer than *need* characters immediately available. If *ios::skipws* is set in *ins.flags()* and *need* is zero, then leading whitespace characters are extracted from *ins*.

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

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

## Formatted input functions (extractors)

`ins>>x`

Calls *ipfx(0)* and if that returns non-zero, extracts characters from *ins* and converts them according to the type of *x*. It stores the converted value in *x*. Errors are indicated by setting the error state of *ins*. If *ios::failbit* is set, this means that characters in *ins* did not match the required type. If *ios::badbit* is set, this 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*, page 30) and the type of *x*. Apart from extractors which use a *width* and reset the field width to 0, extraction operators do not alter the format state value of *ostream*. Extractors are defined for the following 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 the conversion specified in *ins*'s format flags. Converted characters are stored in *x*. The first character may be a sign (+ or -). After that, if *ios::oct*, *ios::dec*, or *ios::hex* is set in *ins.flags()*, the conversion is octal, decimal, or hexadecimal, respectively. Conversion is terminated by the first "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 on following hexadecimal digits. Otherwise, if the first character is a 0, an octal conversion is performed, and in all other cases a decimal conversion is performed. *ios::failbit* is set if there are no digits (not counting the 0 in *0x* or *0X* during hex conversion) available.

float&, double&

Converts the characters according to C++ syntax for a *float* or *double*, and stores the result in *x*. *ios::failbit* is set if there are no digits available in *ins* or if it does not begin with a well formed floating point number.



*skipws* should be left set during the extraction of numerical values. Otherwise an error can occur.

The type and name of the extraction functions 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, characters are extracted from *ios* and inserted into *sb*. Extraction stops when the end-of-file (EOF) is reached. Always returns *ins*.

## Unformatted input functions

These functions call *ipfx(1)* and proceed only if it returns non-zero:

`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. Input terminates when the *delim* character 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.

`ins.get(char & c)`

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

`ins.get(char* 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 the *delim* character (which it leaves in *ins*). *ios::failbit* is set if it stops because the store into *sb* fails.

`ins.get()`.

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

`ins.getline(char* ptr, int len, char 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*. If *delim* occurs after exactly *len* characters have been extracted, termination is treated as being due to the array being filled, and this *delim* is left in *ins*.

`ins.ignore(int n, int d)`

Extracts and throws away up to *n* characters. Extraction stops prematurely if the character *d* is extracted or end of file is reached. If *d* is EOF it can never cause termination.

`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 has already extract and sets *ios::failbit*. The number of characters extracted can be determined via *ins.gcount()*.

## Other members

`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.

`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.

`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.

`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 executes 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

`ins.seekg(streamoff off, seek_dir dir)`

Repositions `ins.rdbuf()`'s `get` pointer. See `sbufpub` (page 69), for a discussion of positioning.

`ins.seekg(streampos pos)`

Repositions `ins.rdbuf()`'s `get` pointer. See `sbufpub` (page 69) for a discussion of positioning.

`ins.tellg()`

Returns the current position of `ios.rdbuf()`'s `get` pointer. See `sbufpub` (page 69) for a discussion of positioning.

## Manipulators

`ins>>ws`

Extracts whitespace characters.

`ins>>dec`

Sets the conversion base format flag to 10. See *ios* (page 30).

`ins>>hex`

Sets the conversion base format flag to 16. See *ios* (page 30).

`ins>>oct`

Sets the conversion base format flag to 8. See *ios*, (page 30).

## EXAMPLE

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

```
#include <iostream.h>

const int N = 80;
char text[N];           // text buffer

int main()
{
    int i;
    cout << " Please enter text :\n";
    cin.getline(text, N);    // get at most N characters
    i = cin.gcount() - 1;
    while (i)
    {
        cout << text [--i];    // prints line in the reverse order
    }
    cout << endl;
    return 0;              // successful return
}
```

The result of executing the program is:

```
Please enter text:
TOM
MOT
```

## BUGS

There is no overflow detection on conversion of integers. There should be, and overflow should cause the error state to be set.

## SEE ALSO

*ios* (page 30), *manip* (page 48), *sbufpub* (page 69)

## manip - ostream manipulation

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

---

```

#include <iostream.h>
#include <iomanip.h>
IOMANIPdeclare(T);

class SMANIP(T) {
    SMANIP(T)(ios& (*f)(ios&, T), T a) : fct(f), arg(a) { }
    friend ostream& operator>>(ostream& o, const SMANIP(T)& m) {
        ios* s = &o; (*m.fct)(*s,m.arg); return o; }
    friend ostream& operator<<(ostream& o, const SMANIP(T)& m) {
        ios* s = &o; (*m.fct)(*s,m.arg); return o; }
};

class SAPP(T) {
public:
    SAPP(T)(ios& (*f)(ios&,T)) : fct(f) { }
    SMANIP(T) operator()(T a) {return SMANIP(T)(fct,a); }
};

class IMANIP(T) {
public:
    IMANIP(T)(ostream& (*f)(ostream&, T), T a) : fct(f), arg(a) { }
    friend ostream& operator>>(ostream& s, const IMANIP(T)& m){ return(*m.fct)(s,m.arg); }
};

class IAPP(T) {
public:
    IAPP(T)(ostream& (*f)(ostream&,T)) : fct(f) { }
    IMANIP(T) operator()(T a) {return IMANIP(T)(fct,a); }
};

class OMANIP(T) {
public:
    OMANIP(T)(ostream& (*f)(ostream&, T), T a) : fct(f), arg(a) { }
    friend ostream& operator<<(ostream& s, const OMANIP(T)& m) {
        return(*m.fct)(s,m.arg); }
};

```



```

class OAPP(T) {
public:
    OAPP(T)(ostream& (*f)(ostream&,T)) : fct(f) { }
    OMANIP(T) operator()(T a) {return OMANIP(T)(fct,a); }
};

class IOMANIP(T) {
public:
    IOMANIP(T)(iostream& (*f)(iostream&, T), T a) : fct(f), arg(a) { }
    friend istream& operator>>(istream& s, const IOMANIP(T)& m) {
        return(*m.fct)(s,m.arg); }
    friend ostream& operator<<(ostream& s, const IOMANIP(T)& m) {
        return(*m.fct)(s,m.arg); }
};

class IOAPP(T) {
public:
    IOAPP(T)(iostream& (*f)(iostream&,T)) : fct(f) { }
    IOMANIP(T) operator()(T a) {return IOMANIP(T)(fct,a); }
};

IOMANIPdeclare(int);
IOMANIPdeclare(long);

SMANIP(int)          setbase(int b);
SMANIP(long)         resetiosflags(long b);
SMANIP(long)         setiosflags(long b);
SMANIP(int)          setfill(int f);
SMANIP(int)          setprecision(int p);
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 in an expression a function call containing several insertions or extractions. For example, the predefined manipulator for *ostreams*, *flush*, can be used as follows:

```

cout << flush
to flush cout.

```

Several *ostream* classes supply manipulators, see *ios* (page 30), *istream* (page 41), and *ostream* (page 53). *flush* is a simple manipulator; some manipulators take arguments, such as the predefined *ios* manipulators, *setfill* and *setw* (see below). The header file *iosmanip.h* supplies macro definitions which programmers can use to define new parameterized manipulators.

Ideally, the types relating to manipulators would be parameterized as “templates”. The macros defined in *iosmanip.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 *Ts* (type names) is used to construct the real names and therefore must be a single identifier. Each of the other macros also requires an identifier and is expanded to form a name.

```
s<<SMANIP(T)(ios& (*)(ios&) f, T t)
```

```
s>>SMANIP(T)(ios& (*)(ios&) f, T t)
```

```
s<<SAPP(T)(ios& (*)(ios&) f, (T t))
```

```
s>>SAPP(T)(ios& (*)(ios&) f, (T t))
```

Returns  $f(s,t)$ , where  $s$  is the left operand of the insertion or extractor operator (e.g.  $s$ ,  $i$ ,  $o$ , or  $io$ ).

```
i>>IMANIP(T)(istream& (*)(istream&) isf, T t)
```

```
i>>IAPP(T)(istream& (*)(istream&) isf, (T t))
```

Returns  $isf(i, T t)$ .

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

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

Returns  $of(o,t)$ .

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

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

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

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

Returns  $iof(io,t)$ .

*iosmanip.h* contains the two declarations, *IOMANIPdeclare(int)* and *IOMANIPdeclare(long)*, and also some manipulators that take an *int* or a *long* argument. These manipulators all have to do with changing the format state of a stream, see *ios* (page 30) for further details.

```
o<<setw(int n)
```

```
i>>setw(int n)
```

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

```
o<<setfill(int n)
```

```
i>>setfill(int n)
```

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

```
o<<setprecision(int n)
```

```
i>>setprecision(int n)
```

Sets the precision 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<<resetiosflags(long l)
```

```
i>>resetiosflags(long l)
```

Clears in the stream (*o* or *i*) the format bits specified by *l*. (Calls *o.setf(0, l)* or *i.setf(0, l)*).

## EXAMPLE

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

```
#include <stream.h>
#include <iomanip.h>
#include <string.h>
void testline(const char * const p)
{
    /* put onto cout the parameter string, 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 and a string containing a list of fill */
    /* characters, display the string 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()*.

Compile *manip.C* with the following command:

```
$ CC -X d -o manip manip.C # Cfront C++ mode
or
$ CC -X w -o manip manip.C # ANSI C++ mode
```

Run the compiled program with:

```
$ manip
```

The following output is produced on executing the program:

```
.....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
)))))))))A Test String
```

## BUGS

Syntax errors are reported if *IOMANIPdeclare(T)* occurs more than once in a file with the same *T*.

## SEE ALSO

*ios* (page 30), *istream* (page 41), *ostream* (page 53)

## 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};
    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.



All the members and member functions identified below in italics which are not specified by any different class belong to the *ostream* class.

In the following descriptions, assume:

- *outs* is an *ostream*.
- *outswa* is an *ostream\_withassign*.
- *sb* is a *streambuf*\*

### 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

`outs.opfx()`

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

### Output suffix function

`osfx()`

Performs "suffix" actions before returning from processing insertions. If *ios::unitbuf* is set, *osfx()* flushes the *ostream*. If *ios::stdio* is set, *osfx()* flushes *stdout* and *stderr*.

*osfx()* is called by all predefined output functions, and should also be called by user-defined output functions 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` (page 30)): 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 0, the representation consists of 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 hexadecimal 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` (page 30)). The default value for `outs.precision()` is 6. If neither `ios::scientific` nor `ios::fixed` is set, either fixed or scientific notation is chosen for the representation, depending on the value of `x`.

`char`, `unsigned char`

No special conversion is necessary.



After the representation is determined, padding occurs. If *outs.width()* is greater than 0 and the representation contains fewer than *outs.width()* characters, then enough *outs.fill()* characters are added to bring the total number of characters to *ios.width()*. If *ios::left* is set in *ios*'s format flags, the sequence is left-adjusted, that is, characters are added after the characters determined above. If *ios::right* is set, the padding is added before the characters determined above. If *ios::internal* is set, the padding is added after any leading sign or base indication and before the characters that represent the value. *ios.width()* is reset to 0, but all other format variables are unchanged. The resulting sequence (padding plus representation) is inserted into *outs.rdbuf()*.

`outs<<sb`

If *outs.opfx()* returns non-zero, the sequence of characters that can be fetched from *sb* are inserted into *outs.rdbuf()*. Insertion stops when no more characters can be fetched from *sb*. No padding is performed. Always returns *outs*.

### Unformatted output functions

`outs.put(char c)`

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

`outs.write(char* ptr, 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

`outs.flush()`

Storing characters into a *streambuf* does not always cause them to be consumed (e.g., written to the external file) immediately. *flush()* causes any characters that may have been stored but not yet consumed to be consumed by calling *outs.rdbuf()->sync*.

`outs<<manip`

Equivalent to *manip(outs)*. Syntactically this looks like an insertion operation, but semantically it performs arbitrary operations rather than converting *manip* to a sequence of characters as do the insertion operators.

## Positioning functions

`outs.seekp(streamoff off, seek_dir dir)`

Repositions `outs.rdbuf()`'s `put` pointer. See `sbufpub` (page 69) for a discussion of positioning.

`outs.seekp(streampos pos)`

Repositions `outs.rdbuf()`'s `put` pointer. See `sbufpub` (page 69) for a discussion of positioning.

`outs.tellp()`

Provides the current position of `outs.rdbuf()`'s `put` pointer. See `sbufpub` (page 69) for a discussion of positioning.

## Manipulators

`outs<<endl`

Ends a line by inserting a newline character and flushing.

`outs<<ends`

Ends a string by inserting a null (0) character.

`outs<<flush`

Flushes `outs`.

`outs<<dec`

Sets the conversion base format flag to 10. See `ios` (page 30).

`outs<<hex`

Sets the conversion base format flag to 16. See `ios` (page 30).

`outs<<oct`

Sets the conversion base format flag to 8. See `ios` (page 30).

## EXAMPLE

The following program fragment in *ostream.C* displays a range of different data types in a variety of different formats:

```
#include <stream.h>    /* for cout + other declarations */
#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 = "abcdefghijklmnopqrstuvwxy";
    /* 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 out in field of width 50 */
    return 0;
}
```

Compile *ostream.C* with the following command:

```
$ CC -X d -o ostream ostream.C # Cfront C++ mode
or
$ CC -X w -o ostream ostream.C # ANSI C++ mode
```

Run the compiled program with:

```
$ ostream

50
d
1.2
3.1232
abcdefghijklmnopqrstuvwxy

62
d
1.200000
3.123200

                                abcdefghijklmnopqrstuvwxy
```

Note how the integer *i* has been printed out in two different formats, and the ease by which the format of *double* and *float* values 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* (page 30), *manip* (page 48), *sbufpub* (page 69)

## 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};
    // and many other declarations, 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      pbackfail(int c);
    virtual int      overflow(int c=EOF);
    virtual int      underflow();
    virtual streambuf* setbuf(char* ptr, int len);
    virtual streampos seekpos(streampos, int=ios::in|ios::out);
    virtual streampos seekoff(streamoff, ios::seek_dir, int=ios::in|ios::out);
    virtual int      sync();
};

```

*streambufs* 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 for the specific sources and destinations (for character transfers).

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 in the public interface. However, if the virtuals do not behave as specified, then the *streambuf* may not behave properly, and an *iostream* (or any other code) that relies on proper behaviour of the *streambuf* may not behave properly either.



All the members and member functions identified below in italics which are not specified by any different class belong to the *streambuf* class.

In the following descriptions assume:

- *sb* is a *streambuf\**.

## Constructors

*streambuf*()

Constructs an empty buffer corresponding to an empty sequence.

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

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

## The get, put, and reserve areas

The *protected* members of *streambuf* present an interface to derived classes organized around three areas (arrays of bytes) managed cooperatively by the base and derived classes. They are the *get area*, the *put area*, and the *reserve area* (or buffer). The *get* and the *put* areas are normally 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* areas can 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 are defined by a collection of *char\** values. The buffer abstraction is described in terms of pointers that point between characters, but the *char\** values must point at *chars*. 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

`sb->base()`

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

`sb->eback()`

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

`sb->ebuf()`

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

`sb->egptr()`

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

`sb->epptr()`

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

`sb->gptr()`

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

`sb->pbase()`

Returns a pointer to the *put* area base. Characters between `sb->pbase()` and `sb->pptr()` have been stored into the buffer and not yet consumed.

`sb->pptr()`

Returns a pointer to the first byte of the *put* area. The space between `sb->pptr()` and `sb->epptr()` is the *put* area and characters are stored here.

## Functions for setting the pointers

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

`sb->setb(char* b, char* eb, int a)`

Sets *base()* and *ebuff()* to *b* and *eb*, respectively. *a* controls whether the area is subject to automatic deletion. If *a* 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.

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

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

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

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

## Other non-virtual members

`sb->allocate()`

Tries to set up a reserve area. If a reserve area already exists or if *sb->unbuffered()* is nonzero, *allocate()* returns 0 without doing anything. If the attempt to allocate space fails, *allocate()* returns EOF, otherwise (allocation succeeds) *allocate()* returns 1. *allocate()* is not called by any non-virtual member function of *streambuf*.

`sb->blen()`

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

`dbp()`

Writes directly on file descriptor 1 information in ASCII about the state of the buffer. It is intended for debugging and nothing is specified about the form of the output. It is considered part of the protected interface because the information it prints can only be understood in relation to that interface, but it is a *public* function so that it can be called anywhere during debugging.

`sb->gbump(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.

`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.



`sb->unbuffered(int i)`

`sb->unbuffered()`

There is a *private* variable known as *sb*'s buffering state.

`sb->unbuffered(int i)` sets the value of this variable to *i* and

`sb->unbuffered()` returns the current value. This state is independent of the actual allocation of a reserve area. Its primary purpose is to control whether a reserve area is allocated automatically by `allocate()`.

## Virtual member functions

Virtual functions may be redefined in derived classes to specialize the behaviour of *streambufs*. 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*.

`sb->doallocate()`

Is called when `allocate()` determines that space is needed. `doallocate()` is used to call `setb()` to provide a reserve area or to return EOF if this is not possible. It is only called if `sb->unbuffered()` is zero and `sb->base()` is zero.

`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 action is to consume the characters between `pbase()` and `pptr()`, call `setp()` to establish a new *put* area, and if *c* != EOF store it (using `sputc()`). `sb->overflow()` should return EOF to indicate an error; otherwise it should return something else.

`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.

`sb->seekoff(streamoff off, seek_dir dir, int mode)`

`seekoff()` is a public virtual member function. A detailed description is given in section *sbufpub* (page 69).

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

`seekpos()` is a public virtual member function. A detailed description is given in section *sbufpub* (page 69).

`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. It may accept or ignore the request for unbuffered state as it chooses. `setbuf()` should return *sb* if it honours the request. Otherwise it should return 0.

`sb->sync()`

`sync()` is a public virtual member function. A detailed description is given in section on *sbufpub* (page 69).

`sb->underflow()`

Is called to supply characters for fetching, i.e. to create a condition in which the *get* area is not empty. If it is called when there are characters in the *get* area it should return the first character. If the *get* area is empty, it should create a 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

`sb->streambuf::doallocate()`

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

`sb->streambuf::overflow(int c)`

Is compatible with the old *stream* package, but that behaviour is not considered part of the specification of the *iostream* package. Therefore, *streambuf::overflow()* should be treated as if it had undefined behaviour. That is, derived classes should always define it.

`sb->streambuf::pbackfail(int c)`

Returns EOF on failure and *c* on success.

`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()*.

`sb->streambuf::seekoff(streamoff off, seekdir dir, int mode)`

Returns EOF.

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

Honours the request when there is no reserve area.

`sb->streambuf::sync()`

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

`sb->streambuf::underflow()`

Is compatible with the old *stream* package, but that behaviour is not considered part of the specification of the *iostream* package. Therefore, *streambuf::underflow()* should be treated as if it had undefined behaviour. That is, it should always be defined in derived classes.

## EXAMPLE

The following program fragment in *sbufprot.C* prints out the machine 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 machine addresses of the *get* and *put* areas.

```
#include <stream.h>                /* For definition of cout */
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;
}
```

Compile *sbufprot.C* with the following command:

```
$ CC -X d -o sbufprot sbufprot.C # Cfront C++ mode
```

or

```
$ CC -X w -o sbufprot sbufprot.C # ANSI C++ mode
```

Run the compiled program with:

```
$ sbufprot
0x804f920
```

Note that the value stored in the pointer will vary between individual machines and versions of SINIX/UNIX, and that *cout* has a default format for pointer values.

## BUGS

The constructors are declared as *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 *get* or *put* operations.

Although the public interface to *streambufs* deals in characters and bytes, the interface to derived classes deals in *chars*. Since a decision had to be made on the types of the real data pointers, it seemed easier to reflect that choice in the types of the *protected* members than to duplicate all the members with both plain and *unsigned char* versions. But perhaps all these uses of *char\** ought to have been implemented by a *typedef*.

The implementation contains a variant of *setbuf()* that accepts a third argument. It is present only for compatibility with the old *stream* package.

## SEE ALSO

*istream* (page 41), *sbufpub* (page 69)

## sbufpub - Public interface of class streambuf

This section describes the *public* member functions of *streambuf*.

---

```
#include <iostream.h>

typedef long streamoff, streampos;

class ios
{
public:
    enum          seek_dir {beg, cur, end};
    enum          open_mode {in, out, ate, app, trunc, nocreate, noreplace};
    // and lots of other classes, see ios (page 30).

};

class streambuf
{
public :
    int          in_avail();
    int          out_waiting();
    int          sbumpc();
    virtual streambuf* setbuf(char* ptr, int len);
    streambuf*   setbuf(unsigned char* ptr, int len);
    streambuf*   setbuf(char* ptr, int len, int count);
    virtual streampos seekpos(streampos, int=ios::in|ios::out);
    virtual streampos seekoff(streamoff, ios::seek_dir, int=ios::in|ios::out);
    int          sgetc();
    int          sgetn(char* ptr, int n);
    int          snextc();
    int          sputbackc(char ch);
    int          sputc(int c);
    int          sputn(const char* ptr, int n);
    void         stoss();
    virtual int   sync();

};
```

---



All the members and member functions identified below in italics which are not specified by any different class belong to the *streambuf* class.

In the following descriptions assume:

- *sb* is a *streambuf\**.

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* (page 22), *sbuf* (page 74), and *stdiobuf* (page 77)).

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* (page 53) and *sbuf* (page 74)) have a *put* and a *get* pointer which move independently of each other. In such buffers characters that are stored are held (i.e., queued) until they are later fetched. Filelike buffers (e.g., *filebuf*, see *filebuf* (page 22)) 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.

### Public member functions

*sb->in\_avail()*

Returns the number of characters that are immediately available in the *get* area for fetching. *i* characters may be fetched with a guarantee that no errors are reported.

*sb->out\_waiting()*

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

sb->sbumpc()

Moves the *get* pointer forward one character and returns the character it moved past. Returns EOF if the *get* pointer is currently at the end of the sequence.

sb->seekoff(streamoff off, seek\_dir dir, int mode)

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

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.

sb->sgetc()

Returns the character after the *get* pointer. Contrary to what most people expect from the name it does **not** move the *get* pointer. Returns EOF if there is no character available.

sb->setbuf(char\* ptr, int len, int count)

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.

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.

`sb->snextc()`

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.

`sb->sputbackc(char ch)`

Moves the *get* pointer back one character. *ch* 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 *ch* is saved. Thus the effect of *sputbackc()* is undefined if *ch* 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.

`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.

`sb->sputn(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.

`sb->stossc()`

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

`sb->sync()`

Establishes consistency between the internal data structures and the external source or sink. The details of this function depend on the derived class. *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()* returns 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 fragment in *sbufpub.C* defines a variable of type *filebuf* attached to *cin* and reads in blocks of characters from that *filebuf* until end of file is reached. For each block that is read in, the size of the buffer that is not used by the read in characters, is printed out:

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

int main(){
    filebuf in_file(0);          /* in_file is connected to cin */
    const int N = 1000;
    /* better to use a const int than a hash define */
    char text_b[N];             /* text buffer */
    cout << in_file.in_avail() << endl;
    /* check the get area before trying to call sgetn */
    while (in_file.sgetn(&text_b[0], N) > 0)
    {
        /* did get some characters */
        cout << in_file.in_avail() << endl;
    }
    return 0;
}
```

Compile *sbufpub.C* with the following command:

```
$ CC -X d -o sbufpub sbufpub.C # Cfront C++ mode
```

or

```
$ CC -X w -o sbufpub sbufpub.C # ANSI C++ mode
```

The program may be given any file as input.

The maximum value returned by the *in\_avail()* member function can be up to 1024, the default buffer size. 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* (page 41), *sbufprot* (page 61)

## ssbuf (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* ptr, int n, char* pstart=0);
    strstreambuf(int);
    strstreambuf(unsigned char* ptr, int n, unsigned char* pstart=0);
    strstreambuf(void* (*a)(long), void(*f)(void*));
    ~strstreambuf();
    void freeze(int n=1);
    char* str();
    virtual streambuf* setbuf(char*, int);
};
```

---

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* (page 69)) and *char\** pointers is direct. Moving the pointers corresponds exactly to incrementing and decrementing the *char\** values.

To accommodate the need for arbitrary length strings *strstreambuf* supports a dynamic mode. When a *strstreambuf* is in dynamic mode, space for the character sequence is allocated as needed. When the sequence is extended too far, it is copied to a new array.



All the members and member functions identified below in italics which are not specified by any different class belong to the *strstreambuf* class.

In the following descriptions assume:

- *ssb* is a *strstreambuf\**.

## Constructors

### stringstream()

Constructs an empty *stringstream* in dynamic mode. This means that space is automatically allocated to accommodate the characters that are put into the *stringstream* (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 *stringstream*.

### stringstream(void (\*a)(long), void\* (\*f)(void\*))

Constructs an empty *stringstream* in dynamic mode. *a* is used as the allocator function in dynamic mode. The argument passed to *a* is a *long* denoting the number of 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.

### stringstream(int n)

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

### stringstream(char\* ptr, int n, char\* pstart)

Constructs a *stringstream* to use the bytes starting at *ptr*. The *stringstream* is in static mode; it does not grow dynamically. If *n* is positive, then the *n* bytes starting at *ptr* are used as the *stringstream*. 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 *stringstream*. If *n* is negative, the *stringstream* 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 *stringstream* that was in dynamic allocation mode and is now frozen. It is possible, however, to thaw (unfreeze) such a *stringstream* and resume storing characters.

ssb->str()

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



Unlike previous versions of C++, *str()* no longer terminates the contents of a stream buffer with a zero (see also page 14).

ssb->setbuf(char\* ptr, 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 fragment in *strstreambuf.C* 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 <stream.h>
#include <string.h>
char* const p = "A very long string indeed "
                " abcdefghijklmnopqrstuvwxyz";

int main()
{
    strstreambuf s(p, 0, 0);
    /* The string p is the strstreambuf. */
    /* The get ptr is to the start of p. */
    char *tp = s.str();
    cout << "length of original string " << strlen(p) << endl;
    cout << "no. of chars in the strstreambuf string " << strlen(tp) << endl;
    return 0;
}
```

Compile *strstreambuf.C* with the following command:

```
$ CC -X d -o strstreambuf strstreambuf.C # Cfront C++ mode
or
$ CC -X w -o strstreambuf strstreambuf.C # ANSI C++ mode
```

Run the compiled program with:

```
$ strstreambuf
length of original string 53
length of strstreambuf string 53
```

Note how the original string length has not changed.

## SEE ALSO

*sbufpub* (page 69), *strstream* (page 79)

## stdiobuf - Specialization of ostream for sdtio FILEs

This section describes *stdiobuf*, which is a class which specializes a *streambuf* to deal with 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 *filebufs*.

---

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

class stdiobuf : public streambuf
{
    FILE*      stdiobuf(FILE* f);
    FILE*      stdiofile();
    virtual    ~stdiobuf();
};
```

---

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.

## EXAMPLE

The following program fragment in *stdiobuf.C* opens the file */etc/passwd* for reading, attaches a variable of type *stdiobuf* to the file, and then prints out a message to show if the *stdiobuf* is attached properly to the file.

```
#include <stdiostream.h>
#include <stdio.h>
#include <stream.h>
#include <osfcn.h>
#include <fcntl.h>

int main()
{
    FILE *qw;
    if (!(qw = fopen("/etc/passwd", "r")))
    {
        cerr << "can't open /etc/passwd\n";
        exit(1);
    }
    stdiobuf s(qw);

    FILE *rt = s.stdiofile();
    if (rt != qw)
    {
        cerr << "Error in stdiofile()" << endl;
    }
    else
    {
        cerr << "stdiofile() is working ok" << endl;
    }

    return 0;
}
```

Compile *stdiobuf.C* with the following command:

```
$ CC -X d -o stdiobuf stdiobuf.C # Cfront C++ mode
```

or

```
$ CC -X w -o stdiobuf stdiobuf.C # ANSI C++ mode
```

Run the compiled program with:

```
$ stdiobuf
stdiofile() is working ok
```

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

## SEE ALSO

*filebuf* (page 22), *istream* (page 41), *ostream* (page 53), *sbufpub* (page 69)

## stringstream (stringstream) - Specialization of istream for arrays

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

---

```
#include <stringstream.h>
#include <istream.h>

class ios {
public:
    enum open_mode {in, out, ate, app, trunc, nocreate, noreplace};
    // and many others, see ios ...
};

class istream : public stringstreambase, public istream {
public:
    istream(char*);
    istream(char*, int);
    istream(const char*);
    istream(const char*, int);
    ~istream();
};

class ostream : public stringstreambase, public ostream {
public:
    ostream();
    ostream(char*, int, int=ios::out);
    ~ostream();
    int pcount();
    char* str();
};

class stringstream : public stringstreambase, public istream {
public:
    stringstream();
    stringstream(char*, int, int mode);
    ~stringstream();
    char* str();
};
```

---

*strstream* specializes *iostream* for “incore” operations, that is, storing and fetching from arrays of bytes. The *streambuf* associated with a *strstream* is a *strstreambuf* (see *ssbuf*).



In the following descriptions assume:

- *spb* is a *strstreambuf*\*.
- *ss* is a *strstream*.
- *iss* is an *istrstream*.
- *oss* is an *ostrstream*.

## 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.

## *istrstream* member function

*iss*.rdbuf()

Returns the *strstreambuf* associated with *iss*.



**ostrstream members****oss.rdbuf()**

Returns the *strstreambuf* associated with *oss*.

**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.

**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 functions****ss.rdbuf()**

Returns the *strstreambuf* associated with *ss*.

**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 fragment in *strstream.C* 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 <stream.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;
}
```

Compile *strstream.C* with the following command:

```
$ CC -X d -o strstream strstream.C # Cfront C++ mode
```

or

```
$ CC -X w -o strstream strstream.C # ANSI C++ mode
```

Run the compiled program with:

```
$ strstream
A test string to check strstream
```

## SEE ALSO

*istream* (page 41), *sdbuf* (page 74)

---

## Related publications

### Manuals from Siemens Nixdorf Informationssysteme AG

Please apply to your local office for ordering the manuals.

- [1] **C/C++ Compiler V1.0**  
(Reliant UNIX)  
User Guide

*Contents*

- Overview of the C/C++ development system CDS++
- Compiling and linking of C and C++ programs with the commands cc, c89 and CC
- Programming notes and detailed information on: precompiled header files, optimization, binding, C and C++ language support of the compiler (implementation specific behaviour and extensions)

- [2] **Standard C++ Library V1.2**  
User's Guide and Reference

*Contents*

Problem-oriented description and reference work for the ANSI C++ libraries Strings, Containers, Iterators, Algorithms and Numerics.

- [3] **Tools.h++ V7.0**  
User's Guide

*Contents*

Problem-oriented description of the C++ class libraries Tools.h++.

- [4] **Tools.h++ V7.0**  
Class Reference

*Contents*

Reference work for the C++ class libraries Tools.h++.

- [5] Reliant UNIX 5.43  
**Programmers's Reference Manual**

*Contents*

Description of the commands for program development, C library functions and system calls, and a description of a number of header files and C-specific file formats.

## Other publications

- [6] **The C++ Programming Language**  
(2nd Edition)  
by Bjarne Stroustrup

*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.

---

# Index

- operator 11

!= operator 12

\* operator 11

\*= operator 12

+ operator 11

+= operator 12

/ operator 11

/= operator 12

-= operator 12

== operator 12

## A

abs() 5

allocate() 64

angle 5

arg() 5

arithmetic operators 11

assignment operators 11, 12

attach() 23, 28

## B

bad() 33

base() 63

bitalloc() 37

blen() 64

buffer classes 20

buffer extraction 62, 63

## C

C math library 3, 8, 40

c\_exception 7

Cartesian functions 5

cerr 19

cin 19

clear() 32  
clog 19  
close() 23, 28  
comparison operators 11, 12  
complex element functions 5  
complex math library 3  
complex.h 3, 4, 7  
complex\_error() 7, 14  
conj() 5  
conjugation 5  
coordinate systems 3  
core classes (iostream) 18  
cos() 14  
cosh() 8, 14  
cosine 3, 14  
cout 19  
cplxcartpol 3, 5  
cplxerr 3, 7  
cplxexp 3, 9  
cplxintro 3  
cplxops 3, 11  
cplxtrig 14

**D**

dec 34  
difference  
    arithmetic 11  
doallocate() 65

**E**

eback() 63  
ebuf() 63  
EDOM 4, 8, 10  
egptr() 63  
eof() 32  
epptr() 63  
ERANGE 4, 8, 10, 14  
errno 4, 7, 8, 10, 14  
error handling 7  
exp() 8, 9  
exponential function 3, 9  
extraction operation 42

**F**  
fail() 32  
fd() 23  
FILE 77  
file descriptor 0 19  
file descriptor 1 19  
file descriptor 2 19  
filebuf 20, 22  
    member function 23, 24, 25  
filebuf() 23  
fill() 35  
fixed 35  
flags() 36  
flush() 57  
format state 33  
formatted input 41  
formatted output 53  
freeze() 75  
fstream 20, 26  
    member function 28, 29  
fstream() 27  
functions  
    Cartesian 5  
    hyperbolic 14  
    polar 5  
    trigonometric 14

**G**  
gcount() 46  
get area 63, 66, 70  
get pointer 24  
get() 45  
good() 32  
gptr() 63

**H**  
hex 34  
HUGE 4, 8, 9, 14  
hyperbolic cosine 14  
hyperbolic functions 3, 14  
hyperbolic sine 14

**I**  
I/O stream library

- standard 17
- ifstream 20, 26
  - member function 28, 29
- ifstream() 27
- imag() 5
- in\_avail() 70
- init() 32
- input
  - formatted 41
  - unformatted 41
- input operation 42
- input/output 77
- internal 34
- iomanip.h 50, 51
- ios 18, 26, 30
  - format state 33
  - manipulators 38
  - member function 32, 33, 35, 36, 37, 38
- ios() 32
- iosintro 17
- iostream 19, 79
- iostream manipulators 48
- iostream.h 17, 19, 21, 40
- ostream\_init 19
- ostream\_withassign 19
- is\_open() 23
- istream 19, 41
  - manipulators 47
  - member function 43, 45, 46
- istream() 43
- istream\_withassign 19
- istream\_withassign() 43
- istreamstream 20
- istreamstream() 80
- isword() 37

**L**

- left 34
- libC.a 17
- libcomplex.a 3
- library
  - complex math 3
  - math (C) 3
- log() 8, 9



logarithms 3, 9

## M

magnitude 5  
manipulators iostreams 48  
math library  
    C 3, 8, 40  
math.h 4

## N

natural logarithm 9  
negation  
    arithmetic 11  
norm() 5

## O

oct 34  
ofstream 20, 26  
    member function 28, 29  
ofstream() 27  
open() 24, 28  
operator  
    10, 15  
    - 11  
    != 12  
    \* 11  
    \*= 12  
    + 11  
    += 12  
    / 11  
    /= 12  
    -= 12  
    == 12  
    arithmetic 11  
    assignment 11, 12  
    comparison 11, 12  
opfx() 55  
osfx() 55  
ostream 19, 53  
    member function 55, 57, 58  
    output operations 54  
ostream() 55  
ostream\_withassign 19  
ostream\_withassign() 55

ostream 20  
ostream() 80  
out\_waiting() 70  
output  
    formatted 53  
    operations 54  
    unformatted 53  
ostream  
    manipulators 58  
OVERFLOW 7, 8  
overflow() 65

**P**

pbackfail() 65  
pbase() 63  
pcount() 81  
polar functions 5  
polar() 5  
pow() 9  
power 3  
power function 9  
pptr() 63  
precision() 36  
predefined streams 19  
product  
    arithmetic 11  
put area 63, 70  
put pointer 24  
put() 45, 57  
pword() 37

**Q**

quotient  
    arithmetic 11

**R**

rdbuf() 29, 37, 80, 81  
rdstate() 32  
real() 6  
reserve area 63  
right 34

**S**

sbufprot 61

sbufpub 69  
sbumpc() 71  
scientific 35  
seekoff() 24, 65, 71  
seekp() 58  
seekpos() 65, 71  
setbuf() 24, 29, 65, 71, 76  
setf() 36  
sgetc() 71  
sgetn() 71  
showbase 34  
showpoint 34  
showpos 34  
sin() 14  
sine 3, 14  
SING 7, 8, 10  
sinh() 8, 14  
skipws 33  
snextc() 72  
sputbackc() 72  
sputc() 72  
sputn() 72  
sqrt() 9  
square of a magnitude 5  
square root 3, 9  
sstream 79  
sstream entry 79  
sstreambuf 74  
standard error 19  
standard I/O stream library 17  
standard input 19  
standard library  
    for input/output 17  
standard output 19  
stderr 19  
stdin 19  
stdio 35  
stdio.h 21, 40  
stdiobuf 20, 77  
stdiostream 20  
stdiostream.h 20  
stdout 19  
str() 81  
stream.h 21

streambuf 18, 42, 54, 77  
  pointer 64  
streambuf ff 62, 66  
streambuf() 62  
streams  
  predefined 19  
string.h 51  
strstream 80  
strstream ff 79  
strstreambuf 20, 74  
strstreambuf() 75  
sum  
  arithmetic 11  
sync() 24, 66, 72

**T**

tellg() 46  
tellp() 58  
templates 50  
tie() 38  
trigonometric functions 3, 14

**U**

unbuffered() 65  
UNDERFLOW 7, 8  
underflow() 66  
unformatted input 41  
unformatted output 53  
unitbuf 35  
unsetf() 36  
uppercase 34

**W**

whitespace ff 33  
width() 36  
write() 57

**X**

xalloc() 37

---

# Contents

<b>Preface</b> .....	<b>1</b>
Notational conventions .....	2
<b>Complex math classes and functions</b> .....	<b>3</b>
cplxintro - Introduction to complex mathematics .....	3
cplxcartpol - Cartesian/Polar functions .....	5
cplxerr - Error handling functions .....	7
cplxexp - Transcendental functions .....	9
-cplxops - Operators .....	11
-cplxtrig - Trigonometric and hyperbolic functions .....	14
<b>-Classes and functions for stream I/O</b> .....	<b>17</b>
-iosintro - Introduction to buffering, formatting, and input/output .....	17
-filebuf - Buffer for file input/output .....	22
-fstream - Specialization of iostream and streambuf for files .....	26
-ios - Base class for input/output .....	30
-istream - Formatted and unformatted input .....	41
-manip - istream manipulation .....	48
-ostream - Formatted and unformatted output .....	53
-sbufprot - Protected interface of class streambuf .....	61
-sbufpub - Public interface of class streambuf .....	69
-ssbuf (sstreambuf) - Specialization of streambuf for arrays .....	74
-stdiobuf - Specialization of istream for stdio FILES .....	77
-strstream (sstream) - Specialization of istream for arrays .....	79
<b>Related publications</b> .....	<b>83</b>
<b>Index</b> .....	<b>85</b>



## **Cfront C++ Library**

**(Reliant UNIX)**

### **C++ Classes for Complex Math and Stream I/O**

#### **Reference Manual**

*Target group*

C++ programmers who work on Reliant UNIX with the CDS++ development system.

*Contents*

Description of all classes, functions and operations which the CDS++ development system with the C++ libraries compatible with Cfront V3.0.3 provides for complex math and stream I/O.

**Edition: April 1997**

**File:**

BS2000 is registered trademarks of Siemens Nixdorf Informationssysteme AG.

Copyright © Siemens Nixdorf Informationssysteme AG, 1997.

All rights, including rights of translation, reproduction by printing, copying or similar methods, even of parts, are reserved.

Offenders will be liable for damages. All rights, including rights created by patent grant or registration of a utility model or design, are reserved.

Delivery subject to availability; right of technical modifications reserved.