

AT&T C++ Library Manual Pages

203935
100703-002

TRADEMARKS

Microtec[®], the Microtec logo, Nanokernel[®], Spectra[®], VRTX[®], VRTX32[®], XRAY[®], XSH[®], and Xtrace[®] are registered trademarks of Mentor Graphics Corporation.

BSPBuilder[™], FastStart[™], IFX[™], *logio*[™], SNX[™], Source Explorer[™], the Spectra logo, Target Manager[™], TNX[™], ToolBuilder[™], Virtual Target[™], VRTXmc[™], VRTXsa[™], Xconfig[™], Xpert[™], Xpert Profiler[™], XRAY In-Circuit Debugger[™], and XRAY In-Circuit Debugger Monitor[™] are trademarks of Mentor Graphics Corporation.

All other trademarks mentioned in this document are trademarks of their respective owners.

RESTRICTED RIGHTS LEGEND

U.S. Government Restricted Rights. The software programs and related documentation have been developed entirely at private expense and are commercial computer software provided with RESTRICTED RIGHTS. Use, duplication or disclosure by the U.S. Government or a U.S. Government subcontractor is subject to the restrictions set forth in the license agreement under which the software programs and documentation were obtained pursuant to DFARS 227.7202-3(a) or as set forth in subparagraphs (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clause at FAR 52.227-19, as applicable.

Mentor Graphics Corporation
880 Ridder Park Dr.
San Jose, CA 95131

Copyright © 1987-1999 Mentor Graphics Corporation. All rights reserved. No part of this publication may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical or otherwise, without prior written permission of Mentor Graphics Corporation.

Contents

Preface

About this Manual	vii
Notational Conventions	viii
Questions and Suggestions	viii

1 Stream Library

Changes from the Original AT&T Manual	1-1
filebuf—Buffer for File I/O.....	1-2
fstream—iostream and streambuf Specialized to Files	1-6
ios—Input/Output Formatting.....	1-10
iostream—Buffering, Formatting, and Input/Output	1-20
istream—Formatted and Unformatted Input	1-24
manipulators—iostream out of Band Manipulations	1-31
ostream—Formatted and Unformatted Output	1-35
streambuf—Interface for Derived Classes	1-41
streambuf—Public Interface of Character Buffering Class	1-48
strstreambuf—streambuf Specialized to Arrays	1-53
stdiobuf—iostream Specialized to stdio FILE	1-56
strstream—iostream Specialized to Arrays	1-57

Tables

Table P-1.	Notational Conventions	viii
------------	------------------------------	------

Preface

The Embedded Software Division of Mentor Graphics Corporation provides tools that can be used individually to enhance existing development environments or together to provide a complete and highly integrated embedded software development solution.

About this Manual

The *AT&T C++ Library Manual Pages*, provided with release 2.1 of the AT&T C++ Language System, are reproduced from the *UNIX System V AT&T C++ Language System Release 2.1 Library Manual*. Only the manual pages that are relevant to the Microtec C++ compiler have been reproduced.

- Chapter 1, *Stream Library*, describes functions that let you format input and output from C++ programs. These functions can be found in the following manual pages:

filebuf	streambuf(prot)
fstream	streambuf(pub)
ios	strstreambuf
istream	stdiobuf
manipulators	strstream
ostream	

This manual is intended for reference purposes only. You should be familiar with the C++ programming language and environment in order to fully utilize these manual pages.

Notational Conventions

This manual uses the notational conventions shown in Table P-1 (unless otherwise noted).

Table P-1. Notational Conventions

Symbol	Name	Usage
{ }	Curly Braces	Enclose a list from which you must choose an item.
[]	Square Brackets	Enclose optional items.
...	Ellipsis	Indicates that you may repeat the preceding item zero or more times.
	Vertical Bar	Separates alternative items in a list.
	Punctuation	Punctuation such as commas (,) and colons (:) must be entered as shown.
	Typewriter Font	Represents code or user input in interactive examples.
	<i>Italics</i>	Represents a descriptive item that should be replaced with an actual item.
	Bold	Represents elements that need to stand out from the main body of text.
->	Arrow	Refers to a sequence of windows. For example, Window1 -> Window2 -> Window3 refers to Window3, which is opened from Window2, which is opened from Window1.

Questions and Suggestions

Mentor Graphics is committed to providing its customers with quality software development and RTOS tools and support services. Our commitment continues beyond your purchase of the product throughout your development life cycle.

If you have questions or suggestions regarding this product, please contact your Mentor Graphics support representative. Contact numbers are listed on the back cover of this document.

The original AT&T stream library manual pages are printed on the following pages. They have been reformatted to conform with the Embedded Software Division document style. Their content is unaltered, except for the changes noted in the following section.

Changes from the Original AT&T Manual

Minor spelling and punctuation errors have been corrected.

Cross-references have been added throughout the chapter.

The original AT&T manual contained two man pages named `streambuf` (abbreviated `sbuf`). The two were distinguished by **.prot** and **.pub** suffixes in the headers of each manual page. Since these distinctions were lost in reformatting, the two `streambuf` functions are now distinguished in the text by either (prot) or (pub) following `streambuf`.

- `streambuf` — Interface for Derived Classes

Was: `sbuf.prot`

Now: `streambuf (prot)`

- `streambuf` — Public Interface of Character Buffering Class

Was: `sbuf.pub`

Now: `streambuf (pub)`

filebuf—Buffer for File I/O

Example

```
#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 other stuff, see ios—Input/Output Formatting ...
} ;

#include <fstream.h>

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

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

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

Description

`filebufs` specialize `streambufs` to use a file as a 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 *streambuf—Public Interface of Character Buffering Class* and *streambuf—Interface for Derived Classes*) is allocated automatically if one is not specified explicitly with a constructor or a call to `setbuf`. `filebuf`s can also be made unbuffered with certain arguments to the constructor or `setbuf`, in which case a system call is made for each character that is read or written. The **get** and **put** pointers into the reserve area are conceptually tied together; they behave as a single pointer. Therefore, the descriptions below refer to a single **get/put** pointer.

In the descriptions below, assume:

- `f` is a `filebuf`
- `pfb` is a `filebuf*`
- `psb` is a `streambuf*`
- `i`, `d`, `len`, and `prot` are `ints`
- `name` and `ptr` are `char*s`
- `mode` is an `int` representing an `open_mode`
- `off` is a `streamoff`
- `p` and `pos` are `streampos`
- `dir` is a `seek_dir`

Constructors

<code>filebuf()</code>	Constructs an initially closed <code>filebuf</code> .
<code>filebuf(d)</code>	Constructs a <code>filebuf</code> connected to file descriptor <code>d</code> .
<code>filebuf(d, p, len)</code>	Constructs a <code>filebuf</code> connected to file descriptor <code>d</code> and initialized to use the reserve area starting at <code>p</code> and containing <code>len</code> bytes. If <code>p</code> is null or <code>len</code> is zero or less, the <code>filebuf</code> will be unbuffered.

Members

<code>pfb=f.attach(d)</code>	Connects <code>f</code> to an open file descriptor, <code>d</code> . <code>attach</code> normally returns <code>&f</code> , but returns 0 if <code>f</code> is already open.
<code>pfb=f.close()</code>	Flushes any waiting output, closes the file descriptor, and disconnects <code>f</code> . Unless an error occurs, <code>f</code> 's error state will be cleared. <code>close</code> returns <code>&f</code> unless errors occur, in which case it returns 0. Even if errors occur, <code>close</code> leaves the file descriptor and <code>f</code> closed.
<code>i=f.fd()</code>	Returns <code>i</code> , the file descriptor <code>f</code> is connected to. If <code>f</code> is closed, <code>i</code> is EOF .

<code>i=f.is_open()</code>	Returns non-zero when <code>f</code> is connected to a file descriptor, and zero otherwise.
<code>pfb=f.open(name, mode, prot)</code>	Opens file <code>name</code> and connects <code>f</code> to it. If the file does not already exist, an attempt is made to create it with protection mode <code>prot</code> , unless <code>ios::nocreate</code> is specified in <code>mode</code> . By default, <code>prot</code> is <code>filebuf::openprot</code> , which is 0644. Failure occurs if <code>f</code> is already open. <code>open</code> normally returns <code>&f</code> , but if an error occurs it returns 0. The members of <code>open_mode</code> are bits that may be strung together with "or". (Because the stringing together with "or" returns an <code>int</code> , <code>open</code> takes an <code>int</code> rather than an <code>open_mode</code> argument.) The meanings of these bits in <code>mode</code> are described in detail in <i>fstream—iostream and streambuf Specialized to Files</i> .
<code>p=f.seekoff(off, dir, mode)</code>	Moves the get/put pointer as designated by <code>off</code> and <code>dir</code> . It may fail if the file that <code>f</code> 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). <code>off</code> is interpreted as a count relative to the place in the file specified by <code>dir</code> as described in <i>streambuf—Public Interface of Character Buffering Class</i> . <code>mode</code> is ignored. <code>seekoff</code> returns <code>p</code> , the new position, or EOF if a failure occurs. The position of the file after a failure is undefined.
<code>p=f.seekpos(pos, mode)</code>	Moves the file to a position <code>pos</code> as described in <i>streambuf—Public Interface of Character Buffering Class</i> . <code>mode</code> is ignored. <code>seekpos</code> normally returns <code>pos</code> , but on failure it returns EOF .
<code>psb=f.setbuf(ptr, len)</code>	Sets up the reserve area as <code>len</code> bytes beginning at <code>ptr</code> . If <code>ptr</code> is null or <code>len</code> is less than or equal to 0, <code>f</code> will be unbuffered. <code>setbuf</code> normally returns <code>&f</code> . However, if <code>f</code> is open and a buffer has been allocated, no changes are made to the reserve area or to the buffering status, and <code>setbuf</code> returns 0.
<code>i=f.sync()</code>	Attempts to force the state of the get/put pointer of <code>f</code> to agree (be synchronized) with the state of the file <code>f.fd</code> . 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, <code>sync</code>

returns 0, but it returns **EOF** if synchronization is not possible.

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

Caveats

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

There is no way to force atomic reads.

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

See Also

`streambuf (prot)`, `streambuf (pub)`, `fstream`

fstream—iostream and streambuf Specialized to Files

Example

```
#include <fstream.h>

typedef long streamoff, streampos;
class ios {
public:
    enum seek_dir { beg, cur, end } ;
    enum Description
open_mode{in,out,ate,app,trunc,nocreate,noreplace } ;
    enum io_state { goodbit=0, eofbit, failbit, badbit } ;
    // and other stuff, see ios—Input/Output Formatting ...
} ;

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

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

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

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

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

```

        fstream(int fd, char* p, int l) ;

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

```

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

In the following descriptions, assume:

- `f` is any of `ifstream`, `ofstream`, or `fstream`
- `pfb` is a `filebuf*`
- `psb` is a `streambuf*`
- `name` and `ptr` are `char*s`
- `i`, `fd`, `len`, and `prot` are `ints`
- `mode` is an `int` representing an `open_mode`

Constructors

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

`xstream()` Constructs an unopened `xstream`.

`xstream(name, mode, 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` will indicate failure in case the open fails.

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

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

Member functions

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

`f.close` Closes any associated `filebuf` and thereby breaks the connection of the `f` to a file. `f`'s error state is cleared except on failure.

A failure occurs when the call to `f.rdbuf->close` fails.

`f.open(name, mode, prot)`

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

`ios::app`

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

`ios::ate`

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

`ios::in`

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

`ios::out`

The file is opened for output. `ios::out` is implied by construction and opens of `ofstreams`. For `fstream`, it says that

output operations are to be allowed. `ios::out` may be specified even if `prot` does not permit output.

`ios::trunc`

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

`ios::nocreate`

If the file does not already exist, the open will fail.

`os::noreplace`

If the file already exists, the open will fail.

`pfb=f.rdbuf`

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

`psb=f.setbuf(p,len)`

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

See Also

`filebuf`, `istream`, `ios`, `ostream`, `streambuf` (pub)

ios—Input/Output Formatting

Example

```
#include <iostream.h>

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

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

```

        ostream*    tie(ostream*);
        long  unsetf(long);
        int   width();
        int   width(int);
        static int  xalloc();
protected:
        ios();
        init(streambuf*);
private:
        ios(ios&);
        void  operator=(ios&);
};

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

```

Description

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

Several enumerations are declared in class `ios`, `open_mode`, `io_state`, `seek_dir`, and format flags, to avoid polluting the global name space. The `io_states` are described on this manual page under *Error States*. The format fields are also described on this page, under *Formatting*. The `open_modes` are described in detail in *fstream—iostream and streambuf Specialized to Files* under `open`. The `seek_dirs` are described in *streambuf—Public Interface of Character Buffering Class* under `seekoff`.

In the following descriptions assume:

- `s` and `s2` are `ioss`
- `sr` is an `ios&`
- `sp` is a `ios*`
- `i`, `oi`, `j`, and `n` are `ints`
- `l`, `f`, and `b` are `longs`
- `c` and `oc` are `chars`
- `osp` and `oosp` are `ostream*s`
- `sb` is a `streambuf*`

- `pos` is a `streampos`
- `off` is a `streamoff`
- `dir` is a `seek_dir`
- `mode` is an `int` representing an `open_mode`
- `fct` is a function with type `ios& (*) (ios&)`
- `vp` is a `void*&`

Constructors and assignment

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

`ios(sr)`
`s2=s` Copying of `ioss` is not well-defined in general, therefore the constructor and assignment operators are private so that the compiler will complain about attempts to copy `ios` objects. Copying pointers to `iostreams` is usually what is desired.

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

Error States

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

`i=s.rdstate()` Returns the current error state.

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

`i=s.good()` Returns non-zero if the error state has no bits set, zero otherwise.

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

`i=s.fail()` Returns non-zero if either `badbit` or `failbit` is set in the error state, zero otherwise. Normally this indicates that some

extraction or conversion has failed, but the stream is still usable. That is, once the `failbit` is cleared, I/O on `s` can usually continue.

```
i=s.bad()
```

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

Operators

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

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

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

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

Formatting

An `ios` has a format state that is used by input and output operations to control the details of formatting operations. For other operations, the format state has no particular effect and its components may be set and examined arbitrarily by user code. Most formatting details are controlled by using the **`flags()`**, **`setf()`**, and **`unsetf()`** functions to set the following flags, which are declared in an enumeration in class `ios`. Three other components of the format state are controlled separately with the functions **`fill()`**, **`width()`**, and **`precision()`**.

```
skipws
```

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

left
right
internal

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

dec
oct
hex

These flags control the conversion base of a value. The conversion base is 10 (decimal) if `dec` is set, but if `oct` or `hex` is set, conversions are done in octal or hexadecimal, respectively. If none of these is set, insertions are in decimal, but extractions are interpreted according to the C++ lexical conventions for integral constants. These fields are collectively identified by the static member, `ios::basefield`. The flag `dec` is set by default in `ios::init(streambuf*)`, and a manipulator must be used to change the conversion base flags to another value; see *manipulators—iostream out of Band Manipulations*. The manipulators `hex`, `dec`, and `oct` can also be used to set the conversion base; see *Built-in Manipulators* below.

showbase

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

showpos

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

uppercase

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

showpoint

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

`scientific`
`fixed`

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

`unitbuf`

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

`stdio`

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

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

`oc=s.fill(c)`

Sets the "fill character" format state variable to `c` and returns the previous value. `c` will be used as the padding character, if one is necessary (see *width*, below). The default fill or padding character is a space. The positioning of the fill character is determined by the `right`, `left`, and `internal` flags; see above. A parameterized manipulator, `setfill`, is also available for setting the fill character; see *manipulators—iostream out of Band Manipulations*.

`c=s.fill()`

Returns the "fill character" format state variable.

`l=s.flags()`

Returns the current format flags.

<code>l=s.flags(f)</code>	Resets all the format flags to those specified in <code>f</code> and returns the previous settings.
<code>oi=s.precision(i)</code>	Sets the "precision" format state variable to <code>i</code> and returns the previous value. This variable controls the number of significant digits inserted by the floating point inserter. The default is 6. A parameterized manipulator, <code>setprecision</code> , is also available for setting the precision; see <i>manipulators—ostream out of Band Manipulations</i> .
<code>i=s.precision()</code>	Returns the "precision" format state variable.
<code>l=s.setf(b)</code>	Turns on in <code>s</code> the format flags marked in <code>b</code> and returns the previous settings. A parameterized manipulator, <code>setiosflags</code> , performs the same function; see <i>manipulators—ostream out of Band Manipulations</i> .
<code>l=s.setf(b,f)</code>	Resets in <code>s</code> only the format flags specified by <code>f</code> to the settings marked in <code>b</code> , and returns the previous settings. That is, the format flags specified by <code>f</code> are cleared in <code>s</code> , then reset to be those marked in <code>b</code> . For example, to change the conversion base in <code>s</code> to be <code>hex</code> , one could write: <code>s.setf(ios::hex,ios::basefield)</code> . <code>ios::basefield</code> specifies the conversion base bits as candidates for change, and <code>ios::hex</code> specifies the new value. <code>s.setf(0,f)</code> will clear all the bits specified by <code>f</code> , as will a parameterized manipulator, <code>resetiosflags</code> ; see <i>manipulators—ostream out of Band Manipulations</i> .
<code>l=s.unsetf(b)</code>	Unsets in <code>s</code> the bits set in <code>b</code> and returns the previous settings.
<code>oi=s.width(i)</code>	Sets the "field width" format variable to <code>i</code> and returns the previous value. When the field width is zero (the default), inserters will insert only as many characters as necessary to represent the value being inserted. When the field width is non-zero, the inserters will insert at least that many characters, using the fill character to pad the value, if the value being inserted requires fewer than field-width characters to be represented. However, the numeric inserters never truncate values, so if the value being inserted will not fit in field-width characters, more than field-width characters will be output. The field width is always interpreted as a 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 or

extraction, and in this sense it behaves as a parameter for insertions and extractions. A parameterized manipulator, `setw`, is also available for setting the width; see *manipulators—iostream out of Band Manipulations*.

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

User-defined Format Flag

Class `ios` can be used as a base class for derived classes that require additional format flags or variables. The `iostream` library provides several functions to do this. The two static member functions, `ios::xalloc` and `ios::bitalloc`, allow several such classes to be used together without interference.

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

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

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

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

Other members

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

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

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

<code>sr<<dec</code> <code>sr>>dec</code>	These set the conversion base format flag to 10.
<code>sr<<hex</code> <code>sr>>hex</code>	These set the conversion base format flag to 16.
<code>sr<<oct</code> <code>sr>>oct</code>	These set the conversion base format flag to 8.
<code>sr>>ws</code>	Extracts whitespace characters. See <i>istream—Formatted and Unformatted Input</i> .
<code>sr<<endl</code>	Ends a line by inserting a newline character and flushing. See <i>ostream</i> .
<code>sr<<ends</code>	Ends a string by inserting a null (0) character. See <i>ostream—Formatted and Unformatted Output</i> .
<code>sr<<flush</code>	Flushes <code>outs</code> . See <i>ostream—Formatted and Unformatted Output</i> .

Several parameterized manipulators that operate on `ios` objects are described in *manipulators—iostream out of Band Manipulations*: `setw`, `setfill`, `setprecision`, `setiosflags`, and `resetiosflags`.

The `streambuf` associated with an `ios` may be manipulated by other methods than through the `ios`. For example, characters may be stored in a 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 extraction operations, the extraction (input) will proceed normally.

Caveats

The need for `sync_with_stdio` is a wart.

The old stream package did this as a default but, in the `iostream` package, unbuffered `stdiobufs` are too inefficient to be the default.

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

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

See Also

`complex`, `streambuf`, `istream`, `ostream`, `manipulators`

iostream—Buffering, Formatting, and Input/Output

Example

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

class Iostream_init ;

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

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

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

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

Description

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

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

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

Core Classes

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

<code>streambuf</code>	This is the base class for buffers. It supports insertion (also known as storing or putting) and extraction (also known as fetching or getting) of characters. Most members are in-lined for efficiency. The public interface of class <code>streambuf</code> is described in and the protected interface (for derived classes) is described in <i>streambuf—Interface for Derived Classes</i> .
<code>ios</code>	This class contains state variables that are common to the various stream classes, for example, error states and formatting states. See <i>ios—Input/Output Formatting</i> .
<code>istream</code>	This class supports formatted and unformatted conversion from sequences of characters fetched from <code>streambufs</code> . See <i>istream—Formatted and Unformatted Input</i> .
<code>ostream</code>	This class supports formatted and unformatted conversion to sequences of characters stored into <code>streambufs</code> . See <i>ostream—Formatted and Unformatted Output</i> .
<code>iostream</code>	This class combines <code>istream</code> and <code>ostream</code> . It is intended for situations in which bidirectional operations (inserting into and extracting from a single sequence of characters) are desired. See <i>ios—Input/Output Formatting</i> .
<code>istream_withassign</code> <code>ostream_withassign</code> <code>iostream_withassign</code>	These classes add assignment operators and a constructor with no operands to the corresponding class without assignment. The predefined streams (see below) <code>cin</code> , <code>cout</code> , <code>cerr</code> , and <code>clog</code> , are objects of these classes. See <i>istream—Formatted and Unformatted Input</i> , <i>ostream—Formatted and Unformatted Output</i> , and <i>ios—Input/Output Formatting</i> .
<code>Iostream_init</code>	This class is present for technical reasons relating to initialization. It has no public members. The <code>Iostream_init</code> constructor initializes the predefined streams (listed below). Because an object of this class is declared in the <code>iostream.h</code>

header file, the constructor is called once each time the header is included (although the real initialization is only done once), and therefore the predefined streams will be initialized before they are used. In some cases, global constructors may need to call the `iostream_init` constructor explicitly to ensure the standard streams are initialized before they are used.

Predefined streams

The following streams are predefined:

<code>cin</code>	The standard input (file descriptor 0).
<code>cout</code>	The standard output (file descriptor 1).
<code>cerr</code>	Standard error (file descriptor 2). Output through this stream is unit-buffered, which means that characters are flushed after each inserter operation. (See <code>ostream::osfx</code> in <i>ostream—Formatted and Unformatted Output</i> and <code>ios::unitbuf</code> in <i>ios—Input/Output Formatting</i> .)
<code>clog</code>	This stream is also directed to file descriptor 2, but unlike <code>cerr</code> its output is buffered.

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

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

Classes derived from streambuf

Classes derived from `streambuf` define the details of how characters are produced or consumed. Derivation of a class from `streambuf` (the protected interface) is discussed in *streambuf—Interface for Derived Classes*. The available buffer classes are:

<code>filebuf</code>	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 <i>filebuf—Buffer for File I/O</i> .
<code>stdiobuf</code>	This buffer class supports I/O through stdio FILE structs. It is intended for use when mixing C and C++ code. New code

should prefer to use `filebufs`. See *stdiobuf—iostream Specialized to stdio FILE*.

`strstreambuf` This buffer class stores and fetches characters from arrays of bytes in memory (i.e., strings). See *strstreambuf—streambuf Specialized to Arrays*.

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

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

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

`istrstream`
`ostrstream` These classes support in-core formatting. They use a `strstreambuf`. See *strstream—iostream Specialized to Arrays*.

`stdiostream` This class specializes `iostream` for stdio **FILES**.

Caveats

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

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

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

See Also

`ios`, `streambuf` (prot), `streambuf` (pub), `filebuf`, `stdiobuf`, `strstreambuf`, `istream`, `ostream`, `fstream`, `strstream`, manipulators

istream—Formatted and Unformatted Input

Example

```
#include <iostream.h>

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

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

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

```



```

        streampos    tellg();
        istream&      operator>>(char*);
        istream&      operator>>(char&);
        istream&      operator>>(short&);
        istream&      operator>>(int&);
        istream&      operator>>(long&);
        istream&      operator>>(float&);
        istream&      operator>>(double&);
        istream&      operator>>(unsigned char*);
        istream&      operator>>(unsigned char&);
        istream&      operator>>(unsigned short&);
        istream&      operator>>(unsigned int&);
        istream&      operator>>(unsigned long&);
        istream&      operator>>(streambuf*);
        istream&      operator>>(istream& (*)(istream&));
        istream&      operator>>(ios& (*)(ios&));
    };

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

    extern istream_withassign cin;

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

```

Description

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

In the following descriptions assume that:

- ins is an istream
- inwa is an istream_withassign
- insp is a istream*
- c is a char&
- delim is a char
- ptr is a char* or unsigned char*
- sb is a streambuf&
- i, n, len, d, and need are ints
- pos is a streampos

- `off` is a `streamoff`
- `dir` is a `seek_dir`
- `manip` is a function with type `istream& (*)(istream&)`

Constructors and assignment

<code>istream(sb)</code>	Initializes <code>ios</code> state variables and associates buffer <code>sb</code> with the <code>istream</code> .
<code>istream_withassign()</code>	Does no initialization.
<code>inswa=sb</code>	Associates <code>sb</code> with <code>inswa</code> and initializes the entire state of <code>inswa</code> .
<code>inswa=ins</code>	Associates <code>ins->rdbuf</code> with <code>inswa</code> and initializes the entire state of <code>inswa</code> .

Input prefix function

`i = ins.ipfx(need)` If `ins`'s error state is non-zero, returns zero immediately. If necessary (and if it is non-null), any `ios` tied to `ins` is flushed (see the description `ios::tie` in *ios—Input/Output Formatting*). Flushing is considered necessary if either `need==0` or if there are fewer than `need` characters immediately available. If `ios::skipws` is set in `ins.flags` and `need` is zero, then leading whitespace characters are extracted from `ins`. `ipfx` returns zero if an error occurs while skipping whitespace; otherwise it returns non-zero.

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

Formatted input functions (extractors)

`ins>>x` Calls `ipfx(0)` and if that returns non-zero, extracts characters from `ins` and converts them according to the type of `x`. It stores the converted value in `x`. Errors are indicated by setting the error state of `ins`. `ios::failbit` means that characters in `ins` were not a representation of the required type. `ios::badbit` indicates that attempts to extract characters failed. `ins` is always returned.

The details of conversion depend on the values of `ins`'s format state flags and variables (see *ios—Input/Output Formatting*) and the type of `x`. Except that extractions that use

width reset it to 0, the extraction operators do not change the value of `ostream`'s format state. Extractors are defined for the following types, with conversion rules as described below.

`char*, unsigned char*`

Characters are stored in the array pointed at by `x` until a whitespace character is found in `ins`. The terminating whitespace is left in `ins`.

If `ins.width` is non-zero it is taken to be the size of the array, and no more than `ins.width()-1` characters are extracted. A terminating null character (0) is always stored (even when nothing else is done because of `ins`'s error status). `ins.width` is reset to 0.

`char&, unsigned char&`

A character is extracted and stored in `x`.

`short&, unsigned short&,
int&, unsigned int&,
long&, unsigned long&`

Characters are extracted and converted to an integral value according to the conversion specified in `ins`'s format flags. Converted characters are stored in `x`. The first character may be a sign (+ or -). After that, if `ios::oct`, `ios::dec`, or `ios::hex` is set in `ins.flags`, the conversion is octal, decimal, or hexadecimal respectively. Conversion is terminated by the first "non-digit," which is left in `ins`. Octal digits are the characters '0' to '7'. Decimal digits are the octal digits plus '8' and '9'. Hexadecimal digits are the decimal digits plus the letters 'a' through 'f' (in either upper or lower case). If none of the conversion base format flags is set, then the number is interpreted according to C++ lexical conventions. That is, if the first characters (after the optional sign) are `0x` or `0X`, a hexadecimal conversion is performed on following hexadecimal digits. Otherwise, if the first character is a 0, an octal conversion is performed, and in all other cases a decimal conversion is performed. `ios::failbit` is set if there are no digits (not counting the 0 in `0x` or `0X`) during hex conversion) available.

`float&, double&`

Converts the characters according to C++ syntax for a float or double, and stores the result in `x`. `ios::failbit` is set if

there are no digits available in `ins` or if it does not begin with a well-formed floating point number.

The type and name (`operator>>`) of the extraction operations are chosen to give a convenient syntax for sequences of input operations. The operator overloading of C++ permits extraction functions to be declared for user-defined classes. These operations can then be used with the same syntax as the member functions described here.

`ins>>sb`

If `ios.ipfx(0)` returns non-zero, extracts characters from `ios` and inserts them into `sb`. Extraction stops when **EOF** is reached. Always returns `ins`.

Unformatted input functions

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

`insp=&ins.get(ptr,len,delim)`

Extracts characters and stores them in the byte array beginning at `ptr` and extending for `len` bytes. Extraction stops when `delim` is encountered (`delim` is left in `ins` and not stored), when `ins` has no more characters, or when the array has only one byte left. `get` always stores a terminating null, even if it doesn't extract any characters from `ins` because of its error status. `ios::failbit` is set only if `get` encounters an end of file before it stores any characters.

`insp=&ins.get(c)` Extracts a single character and stores it in `c`.

`insp=&ins.get(sb,delim)`

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

`i=ins.get()`

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

`insp=&ins.getline(ptr,len,delim)`

Does the same thing as `ins.get(ptr,len,delim)` with the exception that it extracts a terminating `delim` character from `ins`. In case `delim` occurs when exactly `len` characters have been extracted, termination is treated as being due to the array being filled, and this `delim` is left in `ins`.

`insp=&ins.ignore(n,d)`

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

`insp=&ins.read(ptr,n)`

Extracts `n` characters and stores them in the array beginning at `ptr`. If end of file is reached before `n` characters have been extracted, `read` stores whatever it can extract and sets `ios::failbit`. The number of characters extracted can be determined via `ins.gcount`.

Other members

`i=ins.gcount()`

Returns the number of characters extracted by the last unformatted input function. Formatted input functions may call unformatted input functions and thereby reset this number.

`i=ins.peek()`

Begins by calling `ins.ipfx(1)`. If that call returns zero or if `ins` is at end of file, it returns **EOF**. Otherwise, it returns the next character without extracting it.

`insp=&ins.putback(c)`

Attempts to back up `ins.rdbuf`. `c` must be the character before `ins.rdbuf`'s **get** pointer. (Unless other activity is modifying `ins.rdbuf`, this is the last character extracted from `ins`.) If it is not, the effect is undefined. `putback` may fail (and set the error state). Although it is a member of `istream`, `putback` never extracts characters, so it does not call `ipfx`. It will, however, return without doing anything if the error state is non-zero.

`i=&ins.sync()`

Establishes consistency between internal data structures and the external source of characters. Calls `ins.rdbuf->sync`, which is a virtual function, so the details depend on the derived class. Returns **EOF** to indicate errors.

`ins>>manip`

Equivalent to `manip(ins)`. Syntactically this looks like an extractor operation, but semantically it does an arbitrary operation rather than converting a sequence of characters and storing the result in `manip`. A predefined manipulator, `ws`, is described below.

Member functions related to positioning

`insp=&ins.seekg(off,dir)`

Repositions `ins.rdbuf`'s **get** pointer. See *streambuf—Public Interface of Character Buffering Class* for a discussion of positioning.

`insp=&ins.seekg(pos)`

Repositions `ins.rdbuf`'s **get** pointer. See *streambuf—Public Interface of Character Buffering Class* for a discussion of positioning.

`pos=ins.tellg()`

The current position of `ios.rdbuf`'s **get** pointer. See *streambuf—Public Interface of Character Buffering Class* for a discussion of positioning.

Manipulator

`ins>>ws`

Extracts whitespace characters.

`ins>>dec`

Sets the conversion base format flag to 10. See *ios—Input/Output Formatting*.

`ins>>hex`

Sets the conversion base format flag to 16. See *ios—Input/Output Formatting*.

`ins>>oct`

Sets the conversion base format flag to 8. See *ios—Input/Output Formatting*.

Caveats

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

See Also

`ios`, `streambuf` (`pub`), manipulators

manipulators—iostream out of Band Manipulations

Example

```

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

class SMANIP(T) {
    SMANIP(T)( ios& (*)(ios&,T), T);
    friend ostream& operator>>(istream&, SMANIP(T)&);
    friend ostream& operator<<(ostream&, SMANIP(T)&);
};
class SAPP(T) {
    SAPP(T)( ios& (*)(ios&,T));
    SMANIP(T) operator()(T);
};
class IMANIP(T) {
    IMANIP(T)( istream& (*)(istream&,T), T);
    friend istream& operator>>(istream&, IMANIP(T)&);
};
class IAPP(T) {
    IAPP(T)( istream& (*)(istream&,T));
    IMANIP(T) operator()(T);
};
class OMANIP(T) {
    OMANIP(T)( ostream& (*)(ostream&,T), T);
    friend ostream& operator<<(ostream&, OMANIP(T)&);
};
class OAPP(T) {
    OAPP(T)( ostream& (*)(ostream&,T));
    OMANIP(T) operator()(T);
};
class IOMANIP(T) {
    IOMANIP(T)( iostream& (*)(iostream&,T), T);
    friend istream& operator>>(iostream&, IOMANIP(T)&);
    friend ostream& operator<<(iostream&, IOMANIP(T)&);
};

class IOAPP(T) {
    IOAPP(T)( iostream& (*)(iostream&,T));
    IOMANIP(T) operator()(T);
};

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

```

Description

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

```
cout << flush
```

to flush `cout`. Several `iostream` classes supply manipulators: see *ios—Input/Output Formatting*, *istream—Formatted and Unformatted Input*, and *ostream—Formatted and Unformatted Output*. `flush` is a simple manipulator; some manipulators take arguments, such as the predefined `ios` manipulators, `setfill` and `setw` (see below). The header file **`iomanip.h`** supplies macro definitions which programmers can use to define new parameterized manipulators.

Ideally, the types relating to manipulators would be parameterized as "templates." The macros defined in **`iomanip.h`** are used to simulate templates. `IOMANIPdeclare(T)` declares the various classes and operators. (All code is declared inline so that no separate definitions are required.) Each of the other `T`s is used to construct the real names and therefore must be a single identifier. Each of the other macros also requires an identifier and expands to a name.

In the following descriptions, assume:

- `t` is a `T`, or type name
- `s` is an `ios`
- `i` is an `istream`
- `o` is an `ostream`
- `io` is an `iostream`
- `f` is an `ios& (*) (ios&)`
- `if` is an `istream& (*) (istream&)`
- `of` is an `ostream& (*) (ostream&)`
- `iof` is an `iostream& (*) (iostream&)`
- `n` is an `int`
- `l` is a `long`

```
s<<SMANIP(T)(f,t)
```

```
s>>SMANIP(T)(f,t)
```

```
s<<SAPP(T)(f)(t)
```

```
s>>SAPP(T)(f)(t)
```

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


```
i>>IMANIP(T)(if,t)
i>>IAPP(T)(if)(t)
```

Returns `if(i,t)`.

```
o<<OMANIP(T)(of,t)
o<<OAPP(T)(of)(t)
```

Returns `of(o,t)`.

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

Returns `iof(io,t)`.

`iomanip.h` contains two declarations, `IOMANIPdeclare(int)` and `IOMANIPdeclare(long)`, and some manipulators that take an `int` or a `long` argument. These manipulators all have to do with changing the format state of a stream; see *ios—Input/Output Formatting* for further details.

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

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

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

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

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

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

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

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

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

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

Caveats

Syntax errors will be reported if `IOMANIPdeclare(T)` occurs more than once in a file with the same `T`.

See Also

ios, istream, ostream

ostream—Formatted and Unformatted Output

Example

```
#include <iostream.h>

typedef long streamoff, streampos;
class ios {
public:
    enum seek_dir { beg, cur, end };
    enum open_mode { in, out, ate, app, trunc, nocreate,
noreplace } ;
    enum { skipws=01,
        left=02, right=04, internal=010,
        dec=020, oct=040, hex=0100,
        showbase=0200, showpoint=0400, uppercase=01000,
showpos=02000,
        scientific=04000, fixed=010000,
        unitbuf=020000, stdio=040000 };
    // and other stuff, see ios—Input/Output Formatting ...
} ;

class ostream : public ios {
public:
    ostream(streambuf*);
    ostream& flush();
    int opfx();
    ostream& put(char);
    ostream& seekp(streampos);
    ostream& seekp(streamoff, seek_dir);
    streampos tellp();
    ostream& write(const char* ptr, int n);
    ostream& write(const unsigned char* ptr, int n);
    ostream& operator<<(const char*);
    ostream& operator<<(char);
    ostream& operator<<(short);
    ostream& operator<<(int);
    ostream& operator<<(long);
    ostream& operator<<(float);
    ostream& operator<<(double);
    ostream& operator<<(unsigned char);
    ostream& operator<<(unsigned short);
    ostream& operator<<(unsigned int);
    ostream& operator<<(unsigned long);
    ostream& operator<<(void*);
    ostream& operator<<(streambuf*);
    ostream& operator<<(ostream& (*)(ostream&));
    ostream& operator<<(ios& (*)(ios&));
};
class ostream_withassign {
    ostream_withassign();
```

```

        ostream&    operator=(istream&);
        ostream&    operator=(streambuf*);
    };

    extern ostream_withassign cout;
    extern ostream_withassign cerr;
    extern ostream_withassign clog;

    ostream&    endl(ostream&) ;
    ostream&    ends(ostream&) ;
    ostream&    flush(ostream&) ;

    ios& dec(ios&) ;
    ios& hex(ios&) ;
    ios& oct(ios&) ;

```

Description

ostreams support insertion (storing) into a `streambuf`. These are commonly referred to as output operations. The `ostream` member functions and related functions are described below.

In the following descriptions, assume:

- `outs` is an `ostream`
- `outswa` is an `ostream_withassign`
- `outsp` is an `ostream*`
- `c` is a `char`
- `ptr` is a `char*` or `unsigned char*`
- `sb` is a `streambuf*`
- `i` and `n` are `ints`
- `pos` is a `streampos`
- `off` is a `streamoff`
- `dir` is a `seek_dir`
- `manip` is a function with type `ostream& (*)(ostream&)`

Constructors and assignment

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

`ostream_withassign()` Does no initialization. This allows a file static variable of this type (`cout`, for example) to be used before it is constructed, provided it is assigned to first.

`outswa=sb` Associates `sb` with `swa` and initializes the entire state of `outswa`.

`inswa=ins` Associates `ins->rdbuf` with `swa` and initializes the entire state of `outswa`.

Output prefix function

`i=outs.opfx()` If `outs`'s error state is nonzero, returns immediately. If `outs.tie` is non-null, it is flushed. Returns non-zero except when `outs`'s error state is nonzero.

Output suffix function

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

`osfx` is called by all predefined inserters, and should be called by user-defined inserters as well, after any direct manipulation of the `streambuf`. It is not called by the binary output functions.

Formatted output functions (inserters)

`outs<<x` First calls `outs.opfx` and if that returns 0, does nothing. Otherwise, inserts a sequence of characters representing `x` into `outs.rdbuf`. Errors are indicated by setting the error state of `outs`. `outs` is always returned.

`x` is converted into a sequence of characters (its representation) according to rules that depend on `x`'s type and `outs`'s format state flags and variables (see *ios—Input/Output Formatting*). Inserters are defined for the following types, with conversion rules as described below:

`char*` The representation is the sequence of characters up to (but not including) the terminating null of the string `x` points at.

any integral type except char and unsigned char
If `x` is positive the representation contains a sequence of decimal, octal, or hexadecimal digits with no leading zeros according to whether `ios::dec`, `ios::oct`, or `ios::hex`, respectively, is set in `ios`'s format flags. If none of those flags are set, conversion defaults to decimal. If `x` is zero, the representation is a single zero character(0). If `x` is negative, decimal conversion converts it to a minus sign (-) followed by decimal digits. If `x` is positive and `ios::showpos` is set,

decimal conversion converts it to a plus sign (+) followed by decimal digits. The other conversions treat all values as unsigned. If `ios::showbase` is set in `ios`'s format flags, the hexadecimal representation contains `0x` before the 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—Input/Output Formatting*.) The default value for `outs.precision` is 6. If neither `ios::scientific` nor `ios::fixed` is set, either fixed or scientific notation is chosen for the representation, depending on the value of `x`.

`char, unsigned char` No special conversion is necessary.

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

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

Unformatted output functions

`outsp=&outs.put(c)` Inserts `c` into `outs.rdbuf`. Sets the error state if the insertion fails.

`outsp=&outs.write(s,n)` Inserts the `n` characters starting at `s` into `outs.rdbuf`. These

characters may include zeros (i.e., `s` need not be a null terminated string).

Other member functions

`outsp=&outs.flush()`

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

`outs<<manip`

Equivalent to `manip(outs)`. Syntactically this looks like an insertion operation, but semantically it does an arbitrary operation rather than converting `manip` to a sequence of characters as do the insertion operators. Predefined manipulators are described below.

Positioning functions

`outsp=&ins.seekp(off,dir)`

Repositions `outs.rdbuf`'s **put** pointer. See *streambuf—Public Interface of Character Buffering Class* for a discussion of positioning.

`outsp=&outs.seekp(pos)`

Repositions `outs.rdbuf`'s **put** pointer. See *streambuf—Public Interface of Character Buffering Class* for a discussion of positioning.

`pos=outs.tellp()`

The current position of `outs.rdbuf`'s **put** pointer. See *streambuf—Public Interface of Character Buffering Class* 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—Input/Output Formatting*.

`outs<<hex`

Sets the conversion base format flag to 16. See *ios—Input/Output Formatting*.

`outs<<oct`

Sets the conversion base format flag to 8. See *ios—Input/Output Formatting*.

See Also

`ios`, `streambuf (pub)`, manipulators

streambuf—Interface for Derived Classes

Example

```
#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 other stuff, see ios—Input/Output Formatting ...
} ;

class streambuf {
public:
    streambuf() ;
    streambuf(char* p, int len);
    void dbp() ;
protected:
    int allocate();
    char* base();
    int blen();
    char* eback();
    char* ebuf();
    char* egptr();
    char* ep_ptr();
    void gbump(int n);
    char* gp_ptr();
    char* pbase();
    void pbump(int n);
    char* pp_ptr();
    void setg(char* eb, char* g, char* eg);
    void setp(char* p, char* ep);
    void setb(char* b, char* eb, int a=0);
    int unbuffered();
    void unbuffered(int);
    virtual int doallocate();
    virtual ~streambuf() ;
public:
    virtual int pbackfail(int c);
    virtual int overflow(int c=EOF);
    virtual int underflow();
    virtual streambuf*
        setbuf(char* p, int len);
    virtual streampos
        seekpos(streampos, int =ios::in|ios::out);
    virtual streampos
        seekoff(streamoff, seek_dir,
int=ios::in|ios::out);
```

```

        virtual int sync();
};

```

Description

`streambufs` implement the buffer abstraction described in *streambuf—Public Interface of Character Buffering Class*. However, the `streambuf` class itself contains only basic members for manipulating the characters; normally a class derived from `streambuf` will be used. This man page describes the interface needed by programmers coding a derived class. Broadly speaking, there are two kinds of member functions described here. The non-virtual functions are provided for manipulating a `streambuf` in ways that are appropriate in a derived class. Their descriptions reveal details of the implementation that would be inappropriate in the public interface. The virtual functions permit the derived class to specialize the `streambuf` class in ways appropriate to the specific sources and sinks that it is implementing. The descriptions of the virtual functions explain the obligations of the virtuals of the derived class. If the virtuals behave as specified, the `streambuf` behaves as specified in the public interface. However, if the virtuals do not behave as specified, the `streambuf` may not behave properly, and an `iostream` (or any other code) that relies on proper behavior of the `streambuf` may not behave properly either.

In the following descriptions assume:

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

Constructors

<code>streambuf</code>	Constructs an empty buffer corresponding to an empty sequence.
<code>streambuf(b, len)</code>	Constructs an empty buffer and then sets up the reserve area to be the <code>len</code> bytes starting at <code>b</code> .

The Get, Put, and Reserver area

The protected members of `streambuf` present an interface to derived classes organized around three areas (arrays of bytes) managed cooperatively by the base and derived classes: the get, put, and reserve areas (or buffers). The get and the put areas

are normally disjoint, but they may both overlap the reserve area, whose primary purpose is to be a resource in which space for the put and get areas can be allocated. The get and the put areas are changed as characters are put into and gotten from the buffer, but the reserve area normally remains fixed. The areas are defined by a collection of `char*` values. The buffer abstraction is described in terms of pointers that point between characters, but the `char*` values must point at `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

<code>ptr=sb->base</code>	Returns a pointer to the first byte of the reserve area. Space between <code>sb->base</code> and <code>sb->ebuf</code> is the reserve area.
<code>ptr=sb->eback</code>	Returns a pointer to a lower bound on <code>sb->gptr</code> . Space between <code>sb->eback</code> and <code>sb->gptr</code> is available for putback.
<code>ptr=sb->ebuf</code>	Returns a pointer to the byte after the last byte of the reserve area.
<code>ptr=sb->egptr</code>	Returns a pointer to the byte after the last byte of the get area.
<code>ptr=sb->eptr</code>	Returns a pointer to the byte after the last byte of the put area.
<code>ptr=sb->gptr</code>	Returns a pointer to the first byte of the get area. The available characters are those between <code>sb->gptr</code> and <code>sb->egptr</code> . The next character fetched will be <code>*(sb->gptr)</code> unless <code>sb->egptr</code> is less than or equal to <code>sb->gptr</code> .
<code>ptr=sb->pbase</code>	Returns a pointer to the put area base. Characters between <code>sb->pbase</code> and <code>sb->pptr</code> have been stored into the buffer and not yet consumed.
<code>ptr=sb->pptr</code>	Returns a pointer to the first byte of the put area. The space between <code>sb->pptr</code> and <code>sb->eptr</code> is the put area and characters will be stored here.

Functions for setting the pointers

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

`sb->setb(b, eb, i)` Sets `base` and `ebuf` to `b` and `eb` respectively. `i` controls whether the area will be subject to automatic deletion. If `i` is non-zero, then `b` will be deleted when `base` is changed by another call of `setb`, or when the destructor is called for `*sb`. If `b` and `eb` are both null, then we say that there is no reserve

area. If `b` is non-null, there is a reserve area even if `eb` is less than `b` and so the reserve area has zero length.

`sb->setp(p, ep)` Sets `pptr` to `p`, `pbase` to `p`, and `epptr` to `ep`.

`sb->setg(eb, g, eg)`
Sets `eback` to `eb`, `gptr` to `g`, and `egptr` to `eg`.

Other non-virtual members

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

`i=sb->blen` Returns the size (in chars) of the current reserve area.

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

`sb->gbump(n)` Increments `gptr` by `n` which may be positive or negative. No checks are made on whether the new value of `gptr` is in bounds.

`sb->pbump(n)` Increments `pptr` by `n` which may be positive or negative. No checks are made on whether the new value of `pptr` is in bounds.

`sb->unbuffered(i)`
`i=sb->unbuffered` There is a private variable known as `sb`'s buffering state. `sb->unbuffered(i)` sets the value of this variable to `i` and `sb->unbuffered` returns the current value. This state is independent of the actual allocation of a reserve area. Its primary purpose is to control whether a reserve area is allocated automatically by `allocate`.

Virtual member functions

Virtual functions may be redefined in derived classes to specialize the behavior of streambufs. This section describes the behavior that these virtual functions should

have in any derived classes; the next section describes the behavior that these functions are defined to have in base class `streambuf`.

`i=sb->doallocate` Is called when `allocate` determines that space is needed. `doallocate` is required to call `setb` to provide a reserve area or to return **EOF** if it cannot. It is only called if `sb->base` is zero and `sb->unbuffered` is zero.

`i=overflow(c)` Is called to consume characters. If `c` is not **EOF**, `overflow` also must either save `c` or consume it. Usually it is called when the put area is full and an attempt is being made to store a new character, but it can be called at other times. The normal action is to consume the characters between `pbase` and `pptr`, call `setp` to establish a new put area and, if `c!=EOF`, store it (using `sputc`). `sb->overflow` should return **EOF** to indicate an error; otherwise it should return something else.

`i=sb->pbackfail(c)` Is called when `eback` equals `gptr` and an attempt has been made to putback `c`. If this situation can be dealt with (e.g., by repositioning an external file), `pbackfail` should return `c`; otherwise it should return **EOF**.

`pos=sb->seekoff(off, dir, mode)`
Repositions the **get** and/or **put** pointers (i.e., the abstract **get** and **put** pointers, not `pptr` and `gptr`). The meanings of `off` and `dir` are discussed in *streambuf—Public Interface of Character Buffering Class*. `mode` specifies whether the **put** pointer (`ios::out` bit set) or the **get** pointer (`ios::in` bit set) is to be modified. Both bits may be set in which case both pointers should be affected. A class derived from `streambuf` is not required to support repositioning. `seekoff` should return **EOF** if the class does not support repositioning. If the class does support repositioning, `seekoff` should return the new position or **EOF** on error.

`pos=sb->seekpos(pos, mode)`
Repositions the `streambuf` **get** and/or **put** pointer to `pos`. `mode` specifies which pointers are affected as for `seekoff`. Returns `pos` (the argument) or **EOF** if the class does not support repositioning or an error occurs.

`sb=sb->setbuf(ptr, len)`
Offers the array at `ptr` with `len` bytes to be used as a reserve area. The normal interpretation is that if `ptr` or `len` are zero,

then this is a request to make the `sb` unbuffered. The derived class may use this area or not as it chooses. It may accept or ignore the request for unbuffered state as it chooses. `setbuf` should return `sb` if it honors the request. Otherwise, it should return 0.

`i=sb->sync`

Is called to give the derived class a chance to look at the state of the areas, and synchronize them with any external representation. Normally `sync` should consume any characters that have been stored into the put area, and if possible give back to the source any characters in the get area that have not been fetched. When `sync` returns there should not be any unconsumed characters, and the get area should be empty. `sync` should return **EOF** if some kind of failure occurs.

`i=sb->underflow`

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

`i=sb->streambuf::doallocate`

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

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

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

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

Returns **EOF**.

`pos=sb->streambuf::seekpos(pos, mode)`

Returns `sb>seekoff(streamoff(pos), ios::beg, mode)`. Thus to define seeking in a derived class, it is frequently only necessary to define `seekoff` and use the inherited `streambuf::seekpos`.

```
pos=sb->streambuf::seekoff(off, dir, mode)
```

Returns **EOF**.

```
sb=sb->streambuf::setbuf(ptr, len)
```

Will honor the request when there is no reserve area.

```
i=sb->streambuf::sync
```

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

```
i=sb->streambuf::underflow
```

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

Caveats

The constructors are public for compatibility with the old stream package. They ought to be protected.

The interface for unbuffered actions is awkward. It's hard to write `underflow` and `overflow` virtuals that behave properly for unbuffered `streambufs` without special casing. Also, there is no way for the virtuals to react sensibly to multi-character **gets** or **puts**.

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

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

See Also

`streambuf (pub)`, `ios`, `istream`, `ostream`

streambuf—Public Interface of Character Buffering Class

Example

```
#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 other stuff ... See ios—Input/Output Formatting
} ;

class streambuf {
public :

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

Description

The `streambuf` class supports buffers into which characters can be inserted (put) or from which characters can be fetched (gotten). Abstractly, such a buffer is a sequence of characters together with one or two pointers (a **get** and/or a **put** pointer) that define the location at which characters are to be inserted or fetched. The pointers should be thought of as pointing between characters rather than at them. This makes it easier to understand the boundary conditions (a pointer before the first character or after the last). Some of the effects of getting and putting are defined by this class but most of the details are left to specialized classes derived from `streambuf`. (See `filebuf`—*Buffer for File I/O*, `strstreambuf`—*streambuf Specialized to Arrays*, and `stdiobuf`—*istream Specialized to stdio FILE*.)

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 (e.g., see *strstream—iostream Specialized to Arrays* and *strstreambuf—streambuf Specialized to Arrays*) 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. File-like buffers (e.g., `filebuf`, see *filebuf—Buffer for File I/O*) 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 *streambuf—Interface for Derived Classes*, which describes the protected interface.

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

- `i`, `n`, and `len` are ints
- `c` is an int. It always holds a "character" value or **EOF**. A "character" value is always positive even when `char` is normally sign extended
- `sb` and `sbl` are `streambuf*s`
- `ptr` is a `char*`
- `off` is a `streamoff`
- `pos` is a `streampos`
- `dir` is a `seek_dir`
- `mode` is an int representing an `open_mode`

Public member functions:

- | | |
|-----------------------------------|--|
| <code>i=sb->in_avail</code> | Returns the number of characters that are immediately available in the get area for fetching. <code>i</code> characters may be fetched with a guarantee that no errors will be reported. |
| <code>i=sb->out_waiting</code> | Returns the number of characters in the put area that have not been consumed (by the ultimate consumer). |

`c=sb->sbumpc` Moves the **get** pointer forward one character and returns the character it moved past. Returns **EOF** if the **get** pointer is currently at the end of the sequence.

`pos=sb->seekoff(off, dir, mode)` Repositions the **get** and/or **put** pointers. `mode` specifies whether the **put** pointer (`ios::out` bit set) or the **get** pointer (`ios::in` bit set) is to be modified. Both bits may be set in which case both pointers should be affected. `off` is interpreted as a byte offset. (Notice that it is a signed quantity.) The meanings of possible values of `dir` are:

`ios::beg`

The beginning of the stream.

`ios::cur`

The current position.

`ios::end`

The end of the stream (end of file.)

Not all classes derived from `streambuf` support repositioning. `seekoff` will return **EOF** if the class does not support repositioning. If the class does support repositioning, `seekoff` will return the new position or **EOF** on error.

`pos=sb->seekpos(pos, mode)` Repositions the `streambuf` **get** and/or **put** pointer to `pos`. `mode` specifies which pointers are affected as for `seekoff`. Returns `pos` (the argument) or **EOF** if the class does not support repositioning or an error occurs. In general, a `streampos` should be treated as a "magic cookie" and no arithmetic should be performed on it. Two particular values have special meaning:

`streampos(0)`

The beginning of the file.

`streampos(EOF)`

Used as an error indication.

`c=sb->sgetc` Returns the character after the **get** pointer. Contrary to what most people expect from the name, it does not move the **get** pointer. Returns **EOF** if there is no character available.

`sb1=sb->setbuf(ptr, len, i)` Offers the `len` bytes starting at `ptr` as the reserve area. If `ptr`

is null or `len` is zero or less, then an unbuffered state is requested. Whether the offered area is used, or a request for unbuffered state is honored depends on details of the derived class. `setbuf` normally returns `sb`, but if it does not accept the offer or honor the request, it returns 0.

`i=sb->sgetn(ptr, n)`

Fetches the `n` characters following the **get** pointer and copies them to the area starting at `ptr`. When there are fewer than `n` characters left before the end of the sequence, `sgetn` fetches whatever characters remain. `sgetn` repositions the **get** pointer following the fetched characters and returns the number of characters fetched.

`c=sb->snextc`

Moves the **get** pointer forward one character and returns the character following the new position. It returns **EOF** if the pointer is currently at the end of the sequence or is at the end of the sequence after moving forward.

`i=sb->sputbackc(c)` Moves the **get** pointer back one character. `c` must be the current content of the sequence just before the **get** pointer. The underlying mechanism may simply back up the **get** pointer or may rearrange its internal data structures so the `c` is saved. Thus the effect of `sputbackc` is undefined if `c` is not the character before the **get** pointer. `sputbackc` returns **EOF** when it fails. The conditions under which it can fail depend on the details of the derived class.

`i=sb->sputc(c)`

Stores `c` after the **put** pointer, and moves the **put** pointer past the stored character; usually this extends the sequence. It returns **EOF** when an error occurs. The conditions that can cause errors depend on the derived class.

`i=sb->sputn(ptr, n)`

Stores the `n` characters starting at `ptr` after the **put** pointer and moves the **put** pointer past them. `sputn` returns `i`, the number of characters stored successfully. Normally `i` is `n`, but it may be less when errors occur.

`sb->stoss`

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

`i=sb->sync`

Establishes consistency between the internal data structures and the external source or sink. The details of this function depend on the derived class. Usually this "flushes" any char-

acters that have been stored but not yet consumed, and "gives back" any characters that may have been produced but not yet fetched. `sync` returns **EOF** to indicate errors.

Caveats

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

Requiring the program to provide the previously fetched character to `sputback` is probably a botch.

See Also

`ios`, `istream`, `ostream`, `streambuf (prot)`

strstreambuf—streambuf Specialized to Arrays

Example

```

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

class strstreambuf : public streambuf {
public:
    strstreambuf() ;
    strstreambuf(char*, int, char*);
    strstreambuf(int);
    strstreambuf(unsigned char*, int, unsigned
char*);
    strstreambuf(void* (*a)(long), void(*f)(void*));

    void freeze(int n=1) ;
    char* str();
    streambuf* setbuf(char*, int)
};

```

Description

A `strstreambuf` is a `streambuf` that uses an array of bytes (a string) to hold the sequence of characters. Given the convention that a `char*` should be interpreted as pointing just before the `char` it really points at, the mapping between the abstract **get/put** pointers (see *streambuf—Public Interface of Character Buffering Class*) and `char*` pointers is direct. Moving the pointers corresponds exactly to incrementing and decrementing the `char*` values.

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

In the following descriptions assume:

- `ssb` is a `strstreambuf*`
- `n` is an `int`
- `ptr` and `pstart` are `char*s` or `unsigned char*s`
- `a` is a `void* (*)(long)`
- `f` is a `void* (*)(void*)`

Constructors

<code>strstreambuf</code>	Constructs an empty <code>strstreambuf</code> in dynamic mode.
---------------------------	--

This means that space will be automatically allocated to accommodate the characters that are put into the `strstreambuf` (using operators `new` and `delete`). Because this may require copying the original characters, it is recommended that when many characters will be inserted, the program should use `setbuf` (described below) to inform the `strstreambuf`.

`strstreambuf(a, f)`

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

`strstreambuf(n)`

Constructs an empty `strstreambuf` in dynamic mode. The initial allocation of space will be at least `n` bytes.

`strstreambuf(ptr, n, pstart)`

Constructs a `strstreambuf` to use the bytes starting at `ptr`. The `strstreambuf` will be in static mode; it will not grow dynamically. If `n` is positive, then the `n` bytes starting at `ptr` are used as the `strstreambuf`. If `n` is zero, `ptr` is assumed to point to the beginning of a null terminated string and the bytes of that string (not including the terminating null character) will constitute the `strstreambuf`. If `n` is negative, the `strstreambuf` is assumed to continue indefinitely. The **get** pointer is initialized to `ptr`. The **put** pointer is initialized to `pstart`. If `pstart` is null, then stores will be treated as errors. If `pstart` is non-null, then the initial sequence for fetching (the get area) consists of the bytes between `ptr` and `pstart`. If `pstart` is null, then the initial get area consists of the entire array.

Member functions

`ssb->freeze(n)`

Inhibits (when `n` is nonzero) or permits (when `n` is zero) automatic deletion of the current array. Deletion normally occurs when more space is needed or when `ssb` is being destroyed. Only space obtained via dynamic allocation is ever freed. It is an error (and the effect is undefined) to store characters into a `strstreambuf` that was in dynamic allocation mode

and is now frozen. It is possible, however, to thaw (unfreeze) such a `strstreambuf` and resume storing characters.

`ptr=ssb->str`

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

`ssb->setbuf(0,n)`

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

See Also

`streambuf` (pub), `strstream`

stdiobuf—iostream Specialized to stdio FILE

Example

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

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

Description

Operations on a `stdiobuf` are reflected on the associated `FILE`. A `stdiobuf` is constructed in unbuffered mode, which causes all operations to be reflected immediately in the `FILE`. `seekgs` and `seekps` are translated into `fseek`s. `setbuf` has its usual meaning; if it supplies a reserve area, buffering will be turned back on.

Caveats

`stdiobuf` is intended to be used when mixing C and C++ code. New C++ code should prefer to use `filebuf`s, which have better performance.

See Also

`filebuf`, `istream`, `ostream`, `streambuf` (pub)

strstream—iostream Specialized to Arrays

Example

```

#include <strstream.h>

class ios {
public:
    enum open_mode { in, out, ate, app, trunc, nocreate,
noreplace } ;
    // and other stuff, see ios—Input/Output Formatting ...
} ;

class istrstream : public istream {
public:
    istrstream(char*) ;
    istrstream(char*, int) ;
    strstreambuf* rdbuf() ;
} ;

class ostrstream : public ostream {
public:
    ostrstream();
    ostrstream(char*, int, int=ios::out) ;
    int pcount() ;
    strstreambuf* rdbuf() ;
    char* str();
};

class strstream : public strstreambase, public iostream {
public:
    strstream();
    strstream(char*, int, int mode);
    strstreambuf* rdbuf() ;
    char* str();
};

```

Description

`strstream` specializes `iostream` for "incore" operations, that is, storing and fetching from arrays of bytes. The `streambuf` associated with a `strstream` is a `strstreambuf` (see *strstreambuf—streambuf Specialized to Arrays*).

In the following descriptions assume:

- `ss` is a `strstream`
- `iss` is an `istrstream`
- `oss` is an `ostrstream`
- `cp` is a `char*`

- `mode` is an `int` representing an `open_mode`
- `i` and `len` are `ints`
- `ssb` is a `strstreambuf*`

Constructors

`istrstream(cp)` Characters will be fetched from the (null-terminated) string `cp`. The terminating null character will not be part of the sequence. Seeks (`istream::seekg`) are allowed within that space.

`istrstream(cp, len)` Characters will be fetched from the array beginning at `cp` and extending for `len` bytes. Seeks (`istream::seekg`) are allowed anywhere within that array.

`ostrstream` Space will be dynamically allocated to hold stored characters.

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

`strstream` Space will be dynamically allocated to hold stored characters.

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

istrstream members

`ssb = iss.rdbuf` Returns the `strstreambuf` associated with `iss`.

ostrstream members

`ssb = oss.rdbuf` Returns the `strstreambuf` associated with `oss`.

`cp=oss.str` Returns a pointer to the array being used and "freezes" the array. Once `str` has been called, the effect of storing more characters into `oss` is undefined. If `oss` was constructed with

an explicit array, `cp` is just a pointer to the array. Otherwise, `cp` points to a dynamically allocated area. Until `str` is called, deleting the dynamically allocated area is the responsibility of `oss`. After `str` returns, the array becomes the responsibility of the user program.

`i=oss.pcount`

Returns the number of bytes that have been stored into the buffer. This is mainly of use when binary data has been stored and `oss.str` does not point to a null terminated string.

strstream members

`ssb = ss.rdbuf`

Returns the `strstreambuf` associated with `ss`.

`cp=ss.str`

Returns a pointer to the array being used and "freezes" the array. Once `str` has been called, the effect of storing more characters into `ss` is undefined. If `ss` was constructed with an explicit array, `cp` is just a pointer to the array. Otherwise, `cp` points to a dynamically allocated area. Until `str` is called, deleting the dynamically allocated area is the responsibility of `ss`. After `str` returns, the array becomes the responsibility of the user program.

See Also

`strstreambuf`, `ios`, `istream`, `ostream`

A

Arrays

- iostream 1-57
- streambuf 1-53

B

Band manipulations 1-31

Buffering 1-20

C

Character buffering class public interface 1-48

F

File I/O buffer 1-2

filebuf function 1-2

Format flags 1-17

Formatted input 1-24

Formatted output 1-35

Formatting 1-20

fstream function 1-6

I

Input/output formatting 1-10

Input/output support 1-20

Interface for derived classes 1-41

ios function 1-10

iostream function 1-20

- core classes 1-21

- Predefined streams 1-22

iostream specialized to FILE 1-56

istream function 1-24

L

Libraries

- Stream library 1-1

M

manipulators function 1-31

O

ostream function 1-35

P

Public interface of character buffering class
1-48

S

Specializing iostream/streambuf to files 1-6

stdio FILE 1-56

stdiobuf function 1-56

Stream library

- filebuf function 1-2

- fstream function 1-6

- ios function 1-10

- iostream function 1-20

- istream function 1-24

- manipulators function 1-31

- ostream function 1-35

- stdiobuf function 1-56

- streambuf(prot) function 1-41

- streambuf(pub) function 1-48

- strstream function 1-57

- strstreambuf function 1-53

streambuf(prot) function 1-41

streambuf(pub) function 1-48

strstream function 1-57

strstreambuf function 1-53

U

Unformatted input 1-24

Unformatted output 1-35

