Copyright © 1993 Free Software Foundation, Inc.

libio includes software developed by the University of California, Berkeley.

libio uses floating-point software written by David M. Gay, which includes the following notice:

The author of this software is David M. Gay.
Copyright (c) 1991 by AT&T.

Permission to use, copy, modify, and distribute this software for any purpose without fee is hereby granted, provided that this entire notice is included in all copies of any software which is or includes a copy or modification of this software and in all copies of the supporting documentation for such software.

THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED WARRANTY. IN PARTICULAR, NEITHER THE AUTHOR NOR AT&T MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided also that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions.
1 Introduction

The iostream classes implement most of the features of AT&T version 2.0 iostream library classes, and most of the features of the ANSI X3J16 library draft (which is based on the AT&T design).

This manual is meant as a reference; for tutorial material on iostreams, see the corresponding section of any recent popular introduction to C++.

1.1 Licensing terms for libio

Since the iostream classes are so fundamental to standard C++, the Free Software Foundation has agreed to a special exception to its standard license, when you link programs with libio.a:

As a special exception, if you link this library with files compiled with a GNU compiler to produce an executable, this does not cause the resulting executable to be covered by the GNU General Public License. This exception does not however invalidate any other reasons why the executable file might be covered by the GNU General Public License.

The code is under the GNU General Public License (version 2) for all other purposes than linking with this library; that means that you can modify and redistribute the code as usual, but remember that if you do, your modifications, and anything you link with the modified code, must be available to others on the same terms.

These functions are also available as part of the libg++ library; if you link with that library instead of libio, the GNU Library General Public License applies.

1.2 Acknowledgements

Per Bothner wrote most of the iostream library, but some portions have their origins elsewhere in the free software community. Heinz Seidl wrote the IO manipulators. The floating-point conversion software is by David M. Gay of AT&T. Some code was derived from parts of BSD 4.4, which was written at the University of California, Berkeley.

The iostream classes are found in the libio library. An early version was originally distributed in libg++, and they are still included there as well, for convenience if you need other libg++ classes. Doug Lea was the original author of libg++, and some of the file-management code still in libio is his.

Various people found bugs or offered suggestions. Hongjui Lu worked hard to use the library as the default stdio implementation for Linux, and has provided much stress-testing of the library.
2 Operators and Default Streams

The GNU iostream library, ‘libio’, implements the standard input and output facilities for C++. These facilities are roughly analogous (in their purpose and ubiquity, at least) with those defined by the C ‘stdio’ functions.

Although these definitions come from a library, rather than being part of the “core language”, they are sufficiently central to be specified in the latest working papers for C++.

You can use two operators defined in this library for basic input and output operations. They are familiar from any C++ introductory textbook: << for output, and >> for input. (Think of data flowing in the direction of the “arrows”.)

These operators are often used in conjunction with three streams that are open by default:

- **ostream cout**: The standard output stream, analogous to the C stdout.
- **istream cin**: The standard input stream, analogous to the C stdin.
- **ostream cerr**: An alternative output stream for errors, analogous to the C stderr.

For example, this bare-bones C++ version of the traditional “hello” program uses << and cout:

```cpp
#include <iostream.h>

int main(int argc, char **argv)
{
    cout << "Well, hi there.\n";
    return 0;
}
```

Casual use of these operators may be seductive, but—other than in writing throwaway code for your own use—it is not necessarily simpler than managing input and output in any other language. For example, robust code should check the state of the input and output streams between operations (for example, using the method `good`). See Section 3.1.1 [Checking the state of a stream], page 5. You may also need to adjust maximum input or output field widths, using manipulators like `setw` or `setprecision`.

**<<**

Write output to an open output stream of class ostream. Defined by this library on any object of a C++ primitive type, and on other classes of the library. You can overload the definition for any of your own applications’ classes.

Returns a reference to the implied argument *this (the open stream it writes on), permitting statements like

```
cout << "The value of i is " << i << "\n";
```
The GNU C++ Iostream Library

Operator on istream

Read input from an open input stream of class istream. Defined by this library on primitive numeric, pointer, and string types; you can extend the definition for any of your own applications’ classes.

Returns a reference to the implied argument *this (the open stream it reads), permitting multiple inputs in one statement.
3 Stream Classes

The previous chapter referred in passing to the classes `ostream` and `istream`, for output and input respectively. These classes share certain properties, captured in their base class `ios`.

3.1 Shared properties: class `ios`

The base class `ios` provides methods to test and manage the state of input or output streams.

`ios` delegates the job of actually reading and writing bytes to the abstract class `streambuf`, which is designed to provide buffered streams (compatible with C, in the GNU implementation). See Chapter 5 [Using the `streambuf` layer], page 23, for information on the facilities available at the `streambuf` level.

**Constructor**

`ios::ios ([streambuf* sb [, ostream* tie]])`

The `ios` constructor by default initializes a new `ios`, and if you supply a `streambuf sb` to associate with it, sets the state `good` in the new `ios` object. It also sets the default properties of the new object.

You can also supply an optional second argument `tie` to the constructor: if present, it is an initial value for `ios::tie`, to associate the new `ios` object with another stream.

**Destructor**

`ios::~ios ()`

The `ios` destructor is virtual, permitting application-specific behavior when a stream is closed—typically, the destructor frees any storage associated with the stream and releases any other associated objects.

3.1.1 Checking the state of a stream

Use this collection of methods to test for (or signal) errors and other exceptional conditions of streams:

**Method**

`ios::operator void* () const`

You can do a quick check on the state of the most recent operation on a stream by examining a pointer to the stream itself. The pointer is arbitrary except for its truth value; it is true if no failures have occurred (`ios::fail` is not true).

For example, you might ask for input on `cin` only if all prior output operations succeeded:

```cpp
if (cout)
{
    // Everything OK so far
    cin >> new_value;
    ...
}
```
The GNU C++ Iostream Library

Method

**ios::operator ! () const**

In case it is more convenient to check whether something has failed, the operator
! returns true if `ios::fail` is true (an operation has failed). For example, you
might issue an error message if input failed:

```cpp
if (!cin)
{
    // Oops
    cerr << "Eh\n";
}
```

**iostate ios::rdstate () const**

Return the state flags for this stream. The value is from the enumeration `iostate`. You can test for any combination of:

- **goodbit** There are no indications of exceptional states on this stream.
- **eofbit** End of file.
- **failbit** An operation has failed on this stream; this usually indicates bad
  format of input.
- **badbit** The stream is unusable.

**void ios::setstate (iostate state)**

Set the state flag for this stream to `state in addition to` any state flags already

See **ios::clear** to set the stream state without regard to existing state flags.
See **ios::good**, **ios::eof**, **ios::fail**, and **ios::bad**, to test the state.

**int ios::good () const**

Test the state flags associated with this stream; true if no error indicators are
set.

**int ios::bad () const**

Test whether a stream is marked as unusable. (Whether `ios::badbit` is set.)

**int ios::eof () const**

True if end of file was reached on this stream. (If `ios::eofbit` is set.)

**int ios::fail () const**

Test for any kind of failure on this stream: *either* some operation failed, or
the stream is marked as bad. (If either `ios::failbit` or `ios::badbit` is set.)

**void ios::clear (iostate state)**

Set the state indication for this stream to the argument `state`. You may call
**ios::clear** with no argument, in which case the state is set to `good` (no errors
pending).

See **ios::good**, **ios::eof**, **ios::fail**, and **ios::bad**, to test the state; see
**ios::set** or **ios::setstate** for an alternative way of setting the state.
3.1.2 Choices in formatting

These methods control (or report on) settings for some details of controlling streams, primarily to do with formatting output:

**char ios::fill () const**
Method
Report on the padding character in use.

**char ios::fill (char padding)**
Method
Set the padding character. You can also use the manipulator `setfill`. See Section 3.1.3 [Changing stream properties in expressions], page 9.
Default: blank.

**int ios::precision () const**
Method
Report the number of significant digits currently in use for output of floating point numbers.
Default: 6.

**int ios::precision (int signif)**
Method
Set the number of significant digits (for input and output numeric conversions) to `signif`.
You can also use the manipulator `setprecision` for this purpose. See Section 3.1.3 [Changing stream properties using manipulators], page 9.

**int ios::width () const**
Method
Report the current output field width setting (the number of characters to write on the next `<<` output operation).
Default: 0, which means to use as many characters as necessary.

**int ios::width (int num)**
Method
Set the input field width setting to `num`. Return the previous value for this stream.
This value resets to zero (the default) every time you use `<<`; it is essentially an additional implicit argument to that operator. You can also use the manipulator `setw` for this purpose. See Section 3.1.3 [Changing stream properties using manipulators], page 9.

**fmtflags ios::flags () const**
Method
Return the current value of the complete collection of flags controlling the format state. These are the flags and their meanings when set:

- **ios::dec**
- **ios::oct**
- **ios::hex**
What numeric base to use in converting integers from internal to display representation, or vice versa: decimal, octal, or hexadecimal, respectively. (You can change the base using the manipulator `setbase`, or any of the manipulators `dec`, `oct`, or `hex`; see Section 3.1.3 [Changing stream properties in expressions], page 9.)
On input, if none of these flags is set, read numeric constants according to the prefix: decimal if no prefix (or a ‘.’ suffix), octal if a ‘0’ prefix is present, hexadecimal if a ‘0x’ prefix is present.

Default: dec.

`ios::fixed`

Avoid scientific notation, and always show a fixed number of digits after the decimal point, according to the output precision in effect. Use `ios::precision` to set precision.

`ios::left`

`ios::right`

`ios::internal`

Where output is to appear in a fixed-width field; left-justified, right-justified, or with padding in the middle (e.g. between a numeric sign and the associated value), respectively.

`ios::scientific`

Use scientific (exponential) notation to display numbers.

`ios::showbase`

Display the conventional prefix as a visual indicator of the conversion base: no prefix for decimal, ‘0’ for octal, ‘0x’ for hexadecimal.

`ios::showpoint`

Display a decimal point and trailing zeros after it to fill out numeric fields, even when redundant.

`ios::showpos`

Display a positive sign on display of positive numbers.

`ios::skipws`

Skip white space. (On by default).

`ios::stdio`

Flush the C `stdio` streams `stdout` and `stderr` after each output operation (for programs that mix C and C++ output conventions).

`ios::unitbuf`

Flush after each output operation.

`ios::uppercase`

Use upper-case characters for the non-numeral elements in numeric displays; for instance, ‘0X7A’ rather than ‘0x7a’, or ‘3.14E+09’ rather than ‘3.14e+09’.

`fmtflags ios::flags (fmtflags value)`

Method

Set `value` as the complete collection of flags controlling the format state. The flag values are described under ‘ios::flags ()’.

Use `ios::setf` or `ios::unsetf` to change one property at a time.
fhftghf

fmtflags ios::setf (fmtflags flag)  
Method
Set one particular flag (of those described for `ios::flags ()'); return the complete collection of flags previously in effect. (Use `ios::unsetf to cancel.)

fmtflags ios::setf (fmtflags flag, fmtflags mask)  
Method
Clear the flag values indicated by mask, then set any of them that are also in flag. (Flag values are described for `ios::flags ()'). Return the complete collection of flags previously in effect. (See `ios::unsetf for another way of clearing flags.)

fmtflags ios::unsetf (fmtflags flag)  
Method
Make certain flag (a combination of flag values described for `ios::flags ()') is not set for this stream; converse of `ios::setf'. Returns the old values of those flags.

3.1.3 Changing stream properties using manipulators

For convenience, `manipulators provide a way to change certain properties of streams, or otherwise affect them, in the middle of expressions involving `<<' or `>>'. For example, you might write

```cpp
    cout << "|" << setfill('*') << setw(5) << 234 << "|";
```
to produce `'|**234|' as output.

ws  
Skip whitespace.  
Manipulator

flush  
Flush an output stream. For example, `cout << ... << flush;' has the same effect as `cout << ...; cout.flush();'.
Manipulator

endl  
Write an end of line character `\n', then flushes the output stream.  
Manipulator

ends  
Write `\0' (the string terminator character).  
Manipulator

setprecision (int signif)  
You can change the value of `ios::precision' in `<<' expressions with the manipulator `setprecision(signif)'; for example,
```cpp
    cout << setprecision(2) << 4.567;
```
prints `4.6'. Requires `#include <iomanip.h>'.
Manipulator

setw (int n)  
You can change the value of `ios::width' in `<<' expressions with the manipulator `setw(n)'; for example,
```cpp
    cout << setw(5) << 234;
```
prints `234' with two leading blanks. Requires `#include <iomanip.h>'.
Manipulator
**setbase (int base)**

Manipulator

Where `base` is one of 10 (decimal), 8 (octal), or 16 (hexadecimal), change the base value for numeric representations. Requires `#include <iomanip.h>`.

**dec**

Select decimal base; equivalent to `setbase(10)`.

**hex**

Select hexadecimal base; equivalent to `setbase(16)`.

**oct**

Select octal base; equivalent to `setbase(8)`.

**setfill (char padding)**

Set the padding character, in the same way as `ios::fill`. Requires `#include <iomanip.h>`.

### 3.1.4 Extended data fields

A related collection of methods allows you to extend this collection of flags and parameters for your own applications, without risk of conflict between them:

**static fmtflags ios::bitalloc ()**

Method

Reserve a bit (the single bit on in the result) to use as a flag. Using `bitalloc` guards against conflict between two packages that use `ios` objects for different purposes.

This method is available for upward compatibility, but is not in the ANSI working paper. The number of bits available is limited; a return value of 0 means no bit is available.

**static int ios::xalloc ()**

Method

Reserve space for a long integer or pointer parameter. The result is a unique nonnegative integer. You can use it as an index to `ios::iword` or `ios::pword`. Use `xalloc` to arrange for arbitrary special-purpose data in your `ios` objects, without risk of conflict between packages designed for different purposes.

**long& ios::iword (int index)**

Method

Return a reference to arbitrary data, of long integer type, stored in an `ios` instance. `index`, conventionally returned from `ios::xalloc`, identifies what particular data you need.

**long ios::iword (int index) const**

Method

Return the actual value of a long integer stored in an `ios`.

**void*& ios::pword (int index)**

Method

Return a reference to an arbitrary pointer, stored in an `ios` instance. `index`, originally returned from `ios::xalloc`, identifies what particular pointer you need.

**void* ios::pword (int index) const**

Method

Return the actual value of a pointer stored in an `ios`. 
3.1.5 Synchronizing related streams

You can use these methods to synchronize related streams with one another:

```cpp
ostream* ios::tie () const
    Report on what output stream, if any, is to be flushed before accessing this one.
    A pointer value of 0 means no stream is tied.

ostream* ios::tie (ostream* assoc)
    Declare that output stream assoc must be flushed before accessing this stream.

int ios::sync_with_stdio ([int switch])
    Unless iostreams and C stdio are designed to work together, you may have
    to choose between efficient C++ streams output and output compatible with C stdio. Use ‘ios::sync_with_stdio()’ to select C compatibility.
    The argument switch is a GNU extension; use 0 as the argument to choose
    output that is not necessarily compatible with C stdio. The default value for
    switch is 1.
    If you install the stdio implementation that comes with GNU libio, there are
    compatible input/output facilities for both C and C++. In that situation, this
    method is unnecessary—but you may still want to write programs that call it,
    for portability.
```

3.1.6 Reaching the underlying streambuf

Finally, you can use this method to access the underlying object:

```cpp
streambuf* ios::rdbuf () const
    Return a pointer to the streambuf object that underlies this ios.
```

3.2 Managing output streams: class ostream

Objects of class ostream inherit the generic methods from ios, and in addition have the
following methods available. Declarations for this class come from ‘iostream.h’.

```cpp
ostream::ostream ()
    The simplest form of the constructor for an ostream simply allocates a new
    ios object.

ostream::ostream (streambuf* sb [, ostream tie])
    This alternative constructor requires a first argument sb of type streambuf*,
    to use an existing open stream for output. It also accepts an optional second
    argument tie, to specify a related ostream* as the initial value for ios::tie.
    If you give the ostream a streambuf explicitly, using this constructor, the sb
    is not destroyed (or deleted or closed) when the ostream is destroyed.
```
3.2.1 Writing on an ostream

These methods write on an ostream (you may also use the operator <<; see Chapter 2 [Operators and Default Streams], page 3).

`ostream& ostream::put (char c)`  
Write the single character `c`.

`ostream& ostream::write (string, int length)`  
Write `length` characters of a string to this `ostream`, beginning at the pointer `string`.  
`string` may have any of these types: `char*`, `unsigned char*`, `signed char*`.

`ostream& ostream::form (const char *format, ...)`  
Method  
A GNU extension, similar to `fprintf(file, format, ...)`.  
`format` is a printf-style format control string, which is used to format the (variable number of) arguments, printing the result on this `ostream`. See `ostream::vform` for a version that uses an argument list rather than a variable number of arguments.

`ostream& ostream::vform (const char *format, va_list args)`  
Method  
A GNU extension, similar to `vfprintf(file, format, args)`.  
`format` is a printf-style format control string, which is used to format the argument list `args`, printing the result on this `ostream`. See `ostream::form` for a version that uses a variable number of arguments rather than an argument list.

3.2.2 Repositioning an ostream

You can control the output position (on output streams that actually support positions, typically files) with these methods:

`streampos ostream::tellp ()`  
Method  
Return the current write position in the stream.

`ostream& ostream::seekp (streampos loc)`  
Method  
Reset the output position to `loc` (which is usually the result of a previous call to `ostream::tellp`). `loc` specifies an absolute position in the output stream.

`ostream& ostream::seekp (streamoff loc, rel)`  
Method  
Reset the output position to `loc`, relative to the beginning, end, or current output position in the stream, as indicated by `rel` (a value from the enumeration `ios::seekdir`):

- `beg` Interpret `loc` as an absolute offset from the beginning of the file.
- `cur` Interpret `loc` as an offset relative to the current output position.
- `end` Interpret `loc` as an offset from the current end of the output stream.
3.2.3 Miscellaneous ostream utilities

You may need to use these ostream methods for housekeeping:

**ostream\& flush ()**

Method

Deliver any pending buffered output for this ostream.

**int ostream::opfx ()**

Method

opfx is a prefix method for operations on ostream objects; it is designed to be called before any further processing. See ostream::osfx for the converse.

opfx tests that the stream is in state good, and if so flushes any stream tied to this one.

The result is 1 when opfx succeeds; else (if the stream state is not good), the result is 0.

**void ostream::osfx ()**

Method

osfx is a suffix method for operations on ostream objects; it is designed to be called at the conclusion of any processing. All the ostream methods end by calling osfx. See ostream::opfx for the converse.

If the unitbuf flag is set for this stream, osfx flushes any buffered output for it.

If the stdio flag is set for this stream, osfx flushes any output buffered for the C output streams ‘stdout’ and ‘stderr’.

3.3 Managing input streams: class istream

Class istream objects are specialized for input; as for ostream, they are derived from ios, so you can use any of the general-purpose methods from that base class. Declarations for this class also come from ‘iostream.h’.

**istream::istream ()**

Constructor

When used without arguments, the istream constructor simply allocates a new ios object and initializes the input counter (the value reported by istream::gcount) to 0.

**istream::istream (streambuf \*sb [, ostream tie])**

Constructor

You can also call the constructor with one or two arguments. The first argument sb is a streambuf\*; if you supply this pointer, the constructor uses that streambuf for input. You can use the second optional argument tie to specify a related output stream as the initial value for ios::tie.

If you give the istream a streambuf explicitly, using this constructor, the sb is not destroyed (or deleted or closed) when the ostream is destroyed.

3.3.1 Reading one character

Use these methods to read a single character from the input stream:
int istream::get ()  
Method  
Read a single character (or EOF) from the input stream, returning it (coerced to an unsigned char) as the result.

istream& istream::get (char& c)  
Method  
Read a single character from the input stream, into &c.

int istream::peek ()  
Method  
Return the next available input character, but without changing the current input position.

3.3.2 Reading strings

Use these methods to read strings (for example, a line at a time) from the input stream:

istream& istream::get (char* c, int len [, char delim])  
Method  
Read a string from the input stream, into the array at c.

The remaining arguments limit how much to read: up to ‘len-1’ characters, or up to (but not including) the first occurrence in the input of a particular delimiter character delim—newline (\n) by default. (Naturally, if the stream reaches end of file first, that too will terminate reading.)

If delim was present in the input, it remains available as if unread; to discard it instead, see istream::getline.

get writes ‘\0’ at the end of the string, regardless of which condition terminates the read.

istream& istream::get (streambuf& sb [, char delim])  
Method  
Read characters from the input stream and copy them on the streambuf object sb. Copying ends either just before the next instance of the delimiter character delim (newline \n by default), or when either stream ends. If delim was present in the input, it remains available as if unread.

istream& istream::getline (charptr, int len [, char delim])  
Method  
Read a line from the input stream, into the array at charptr. charptr may be any of three kinds of pointer: char*, unsigned char*, or signed char*.

The remaining arguments limit how much to read: up to (but not including) the first occurrence in the input of a line delimiter character delim—newline (\n) by default, or up to ‘len-1’ characters (or to end of file, if that happens sooner).

If getline succeeds in reading a “full line”, it also discards the trailing delimiter character from the input stream. (To preserve it as available input, see the similar form of istream::get.)

If delim was not found before len characters or end of file, getline sets the ios::fail flag, as well as the ios::eof flag if appropriate.

getline writes a null character at the end of the string, regardless of which condition terminates the read.
**Chapter 3: Stream Classes**

**Method**

`istream::read (pointer, int len)`

Read `len` bytes into the location at `pointer`, unless the input ends first.

`pointer` may be of type `char*`, `void*`, `unsigned char*`, or `signed char*`.
If the `istream` ends before reading `len` bytes, `read` sets the `ios::fail` flag.

`istream::gets (char **s [, char delim])`

A GNU extension, to read an arbitrarily long string from the current input position to the next instance of the `delim` character (newline `\n` by default).
To permit reading a string of arbitrary length, `gets` allocates whatever memory is required. Notice that the first argument `s` is an address to record a character pointer, rather than the pointer itself.

`istream::scan (const char *format ...)`

A GNU extension, similar to `fscanf (file, format, ...)`. The `format` is a `scanf`-style format control string, which is used to read the variables in the remainder of the argument list from the `istream`.

`istream::vscan (const char *format, va_list args)`

Like `istream::scan`, but takes a single `va_list` argument.

### 3.3.3 Repositioning an istream

Use these methods to control the current input position:

`streampos istream::tellg ()`

Return the current read position, so that you can save it and return to it later with `istream::seekg`.

`istream::seekg (streampos p)`

Reset the input pointer (if the input device permits it) to `p`, usually the result of an earlier call to `istream::tellg`.

`istream::seekg (streamoff offset, ios::seek_dir ref)`

Reset the input pointer (if the input device permits it) to `offset` characters from the beginning of the input, the current position, or the end of input. Specify how to interpret `offset` with one of these values for the second argument:

- `ios::beg` Interpret `loc` as an absolute offset from the beginning of the file.
- `ios::cur` Interpret `loc` as an offset relative to the current output position.
- `ios::end` Interpret `loc` as an offset from the current end of the output stream.

### 3.3.4 Miscellaneous istream utilities

Use these methods for housekeeping on `istream` objects:

`int istream::gcount ()`

Report how many characters were read from this `istream` in the last unformatted input operation.
int istream::ipfx (int keepwhite)  
Method
Ensure that the istream object is ready for reading; check for errors and end of file and flush any tied stream. ipfx skips whitespace if you specify 0 as the keepwhite argument, and ios::skipws is set for this stream.
To avoid skipping whitespace (regardless of the skipws setting on the stream), use 1 as the argument.
Call istream::ipfx to simplify writing your own methods for reading istream objects.

void istream::isfx ( )  
Method
A placeholder for compliance with the draft ANSI standard; this method does nothing whatever.
If you wish to write portable standard-conforming code on istream objects, call isfx after any operation that reads from an istream if istream::ipfx has any special effects that must be cancelled when done, istream::isfx will cancel them.

istream& istream::ignore ([int n [, int delim]])  
Method
Discard some number of characters pending input. The first optional argument n specifies how many characters to skip. The second optional argument delim specifies a “boundary” character: ignore returns immediately if this character appears in the input.
By default, delim is EOF; that is, if you do not specify a second argument, only the count n restricts how much to ignore (while input is still available).
If you do not specify how many characters to ignore, ignore returns after discarding only one character.

istream& istream::putback (char ch)  
Method
Attempts to back up one character, replacing the character backed-up over by ch. Returns EOF if this is not allowed. Putting back the most recently read character is always allowed. (This method corresponds to the C function ungetc.)

istream& istream::unget ( )  
Method
Attempt to back up one character.

3.4 Input and output together: class iostream

If you need to use the same stream for input and output, you can use an object of the class iostream, which is derived from both istream and ostream.
The constructors for iostream behave just like the constructors for istream.

istream::istream ( )  
Constructor
When used without arguments, the istream constructor simply allocates a new ios object, and initializes the input counter (the value reported by istream::gcount) to 0.
You can also call a constructor with one or two arguments. The first argument \textit{sb} is a \texttt{streambuf*}; if you supply this pointer, the constructor uses that \texttt{streambuf} for input and output.

You can use the optional second argument \textit{tie} (an \texttt{ostream*}) to specify a related output stream as the initial value for \texttt{ios::tie}.

As for \texttt{ostream} and \texttt{istream}, \texttt{iostream} simply uses the \texttt{ios} destructor. However, an \texttt{iostream} is not deleted by its destructor.

You can use all the \texttt{istream}, \texttt{ostream}, and \texttt{ios} methods with an \texttt{iostream} object.
4 Classes for Files and Strings

There are two very common special cases of input and output: using files, and using strings in memory.

libio defines four specialized classes for these cases:

`ifstream` Methods for reading files.
`ofstream` Methods for writing files.
`istrstream` Methods for reading strings from memory.
`ostrstream` Methods for writing strings in memory.

4.1 Reading and writing files

These methods are declared in `fstream.h`.

You can read data from class `ifstream` with any operation from class `istream`. There are also a few specialized facilities:

`ifstream::ifstream ()` Constructor
Make an `ifstream` associated with a new file for input. (If you use this version of the constructor, you need to call `ifstream::open` before actually reading anything)

`ifstream::ifstream (int fd)` Constructor
Make an `ifstream` for reading from a file that was already open, using file descriptor `fd`. (This constructor is compatible with other versions of iostreams for POSIX systems, but is not part of the ANSI working paper.)

`ifstream::ifstream (const char* fname [, int mode [, int prot]])` Constructor
Open a file *fname* for this `ifstream` object.
By default, the file is opened for input (with `ios::in` as `mode`). If you use this constructor, the file will be closed when the `ifstream` is destroyed.
You can use the optional argument `mode` to specify how to open the file, by combining these enumerated values (with ‘|’ bitwise or). (These values are actually defined in class `ios`, so that all file-related streams may inherit them.) Only some of these modes are defined in the latest draft ANSI specification; if portability is important, you may wish to avoid the others.

`ios::in` Open for input. (Included in ANSI draft.)
`ios::out` Open for output. (Included in ANSI draft.)
`ios::ate` Set the initial input (or output) position to the end of the file.
`ios::app` Seek to end of file before each write. (Included in ANSI draft.)
ios::trunc
Guarantee a fresh file; discard any contents that were previously associated with it.

ios::nocreate
Guarantee an existing file; fail if the specified file did not already exist.

ios::noreplace
Guarantee a new file; fail if the specified file already existed.

ios::bin
Open as a binary file (on systems where binary and text files have different properties, typically how ‘\n’ is mapped; included in ANSI draft).

The last optional argument prot is specific to Unix-like systems; it specifies the file protection (by default ‘644’).

**Method**

```cpp
void ifstream::open (const char* fname [, int mode [, int prot]])
```

Open a file explicitly after the associated ifstream object already exists (for instance, after using the default constructor). The arguments, options and defaults all have the same meanings as in the fully specified ifstream constructor.

You can write data to class ofstream with any operation from class ostream. There are also a few specialized facilities:

**Constructor**

```cpp
ofstream::ofstream ()
```

Make an ofstream associated with a new file for output.

```cpp
ofstream::ofstream (int fd)
```

Make an ofstream for writing to a file that was already open, using file descriptor fd.

```cpp
ofstream::ofstream (const char* name [, int mode [, int prot]])
```

Open a file *name for this ofstream object. By default, the file is opened for output (with i os::out as mode). You can use the optional argument mode to specify how to open the file, just as described for ifstream::ifstream.

The last optional argument prot specifies the file protection (by default ‘644’).

**Destructor**

```cpp
ofstream::~ ofstream ()
```

The files associated with ofstream objects are closed when the corresponding object is destroyed.

**Method**

```cpp
void ofstream::open (const char* name [, int mode [, int prot]])
```

Open a file explicitly after the associated ofstream object already exists (for instance, after using the default constructor). The arguments, options and defaults all have the same meanings as in the fully specified ofstream constructor.
The class `fstream` combines the facilities of `ifstream` and `ofstream`, just as `iostream` combines `istream` and `ostream`.

The class `fstreambase` underlies both `ifstream` and `ofstream`. They both inherit this additional method:

```cpp
void fstreambase::close ()
Method
Close the file associated with this object, and set `ios::fail` in this object to mark the event.
```

### 4.2 Reading and writing in memory

The classes `istrstream`, `ostrstream`, and `strstream` provide some additional features for reading and writing strings in memory—both static strings, and dynamically allocated strings. The underlying class `strstreambase` provides some features common to all three; `strstreambuf` underlies that in turn.

```cpp
istrstream::istrstream (const char* str [, int size])
Constructor
Associate the new input string class `istrstream` with an existing static string starting at `str`, of size `size`. If you do not specify `size`, the string is treated as a NUL terminated string.
```

```cpp
ostrstream::ostrstream ()
Constructor
Create a new stream for output to a dynamically managed string, which will grow as needed.
```

```cpp
ostrstream::ostrstream (char* str [, int size [, int mode]])
Constructor
A new stream for output to a statically defined string of length `size`, starting at `str`. You may optionally specify one of the modes described for `ifstream::ifstream`; if you do not specify one, the new stream is simply open for output, with mode `ios::out`.
```

```cpp
int ostrstream::pcount ()
Method
Report the current length of the string associated with this `ostrstream`.
```

```cpp
char* ostrstream::str ()
Method
A pointer to the string managed by this `ostrstream`. Implies `ostrstream::freeze()`.
Note that if you want the string to be nul-terminated, you must do that yourself (perhaps by writing ends to the stream).
```

```cpp
void ostrstream::freeze ([int n])
Method
If `n` is nonzero (the default), declare that the string associated with this `ostrstream` is not to change dynamically; while frozen, it will not be reallocated if it needs more space, and it will not be deallocated when the `ostrstream` is destroyed. Use `freeze(1)` if you refer to the string as a pointer after creating it via `ostrstream` facilities.

`freeze(0)` cancels this declaration, allowing a dynamically allocated string to be freed when its `ostrstream` is destroyed.

If this `ostrstream` is already static—that is, if it was created to manage an existing statically allocated string—`freeze` is unnecessary, and has no effect.
int ostrstream::frozen ()
    Test whether freeze(1) is in effect for this string.

streambuf* strstreambase::rdbuf ()
    A pointer to the underlying strstreambuf.
5 Using the streambuf Layer

The istream and ostream classes are meant to handle conversion between objects in your program and their textual representation.

By contrast, the underlying streambuf class is for transferring raw bytes between your program, and input sources or output sinks. Different streambuf subclasses connect to different kinds of sources and sinks.

The GNU implementation of streambuf is still evolving; we describe only some of the highlights.

5.1 Areas of a streambuf

Streambuf buffer management is fairly sophisticated (this is a nice way to say “complicated”). The standard protocol has the following “areas”:

- The put area contains characters waiting for output.
- The get area contains characters available for reading.

The GNU streambuf design extends this, but the details are still evolving.

The following methods are used to manipulate these areas. These are all protected methods, which are intended to be used by virtual function in classes derived from streambuf. They are also all ANSI/ISO-standard, and the ugly names are traditional. (Note that if a pointer points to the ’end’ of an area, it means that it points to the character after the area.)

```cpp
char* streambuf::pbase () const  // Method
    Returns a pointer to the start of the put area.

char* streambuf::epptr () const  // Method
    Returns a pointer to the end of the put area.

char* streambuf::pptr () const  // Method
    If pptr() < epptr(), the pptr() returns a pointer to the current put position.
    (In that case, the next write will overwrite *pptr(), and increment pptr().)
    Otherwise, there is no put position available (and the next character written
    will cause streambuf::overflow to be called).

void streambuf::pbump (int N)  // Method
    Add N to the current put pointer. No error checking is done.

void streambuf::setp (char* P, char* E)  // Method
    Sets the start of the put area to P, the end of the put area to E, and the current
    put pointer to P (also).

char* streambuf::eback () const  // Method
    Returns a pointer to the start of the get area.

char* streambuf::egptr () const  // Method
    Returns a pointer to the end of the get area.
```
char* streambuf::gptr () const
Method
If gptr() < egptr(), then gptr() returns a pointer to the current get position.
(In that case the next read will read *gptr(), and possibly increment gptr().)
Otherwise, there is no read position available (and the next read will cause
streambuf::underflow to be called).

void streambuf::gbump (int N) Method
Add N to the current get pointer. No error checking is done.

void streambuf::setg (char* B, char* P, char* E) Method
Sets the start of the get area to B, the end of the get area to E, and the current
put pointer to P.

5.2 Simple output re-direction by redefining overflow

Suppose you have a function write_to_window that writes characters to a window object.
If you want to use the ostream function to write to it, here is one (portable) way to do it.
This depends on the default buffering (if any).
```cpp
#include <iostream.h>
/* Returns number of characters successfully written to win. */
extern int write_to_window (window* win, char* text, int length);

class windowbuf : public streambuf {
    window* win;

    public:
    windowbuf (window* w) { win = w; }
    int sync ();
    int overflow (int ch);
    // Defining xsputn is an optional optimization.
    // (streamsize was recently added to ANSI C++, not portable yet.)
    streamsize xsputn (char* text, streamsize n);
};

int windowbuf::sync ()
{ streamsize n = pptr () - pbase ();
    return (n && write_to_window (win, pbase (), n) != n) ? EOF : 0;
}

int windowbuf::overflow (int ch)
{ streamsize n = pptr () - pbase ();
    if (n && sync ()
        return EOF;
    if (ch != EOF)
    {
        char cbuf[1];
        cbuf[0] = ch;
        if (write_to_window (win, cbuf, 1) != 1)
            return EOF;
    }
    pbump (-n); // Reset pptr().
    return 0;
}

streamsize windowbuf::xsputn (char* text, streamsize n)
{ return sync () == EOF ? 0 : write_to_window (win, text, n); }

int main (int argc, char**argv)
{
    window *win = ...;
    windowbuf wbuf(win);
    ostream wstr(&wbuf);
    wstr << "Hello world!\n";
}
```
5.3 C-style formatting for streambuf objects

The GNU streambuf class supports printf-like formatting and scanning.

```cpp
int streambuf::form (const char *format, ...) Method
Similar to fprintf(file, format, ...). The format is a printf-style format
control string, which is used to format the (variable number of) arguments,
printing the result on the this streambuf. The result is the number of characters
printed.
```

```cpp
int streambuf::vform (const char *format, va_list args) Method
Similar to vfprintf(file, format, args). The format is a printf-style format
control string, which is used to format the argument list args, printing the result
on the this streambuf. The result is the number of characters printed.
```

```cpp
int streambuf::scan (const char *format, ...) Method
Similar to fscanf(file, format, ...). The format is a scanf-style format con-
trol string, which is used to read the (variable number of) arguments from the
this streambuf. The result is the number of items assigned, or EOF in case of
input failure before any conversion.
```

```cpp
int streambuf::vscan (const char *format, va_list args) Method
Like streambuf::scan, but takes a single va_list argument.
```

5.4 Wrappers for C stdio

A stdiobuf is a streambuf object that points to a FILE object (as defined by stdio.h). All streambuf operations on the stdiobuf are forwarded to the FILE. Thus the stdiobuf object provides a wrapper around a FILE, allowing use of streambuf operations on a FILE. This can be useful when mixing C code with C++ code.

The pre-defined streams cin, cout, and cerr are normally implemented as stdiobuf objects that point to respectively stdin, stdout, and stderr. This is convenient, but it does cost some extra overhead.

If you set things up to use the implementation of stdio provided with this library, then cin, cout, and cerr will be set up to to use stdiobuf objects, since you get their benefits for free. See Chapter 6 [C Input and Output], page 31.

5.5 Reading/writing from/to a pipe

The procbuf class is a GNU extension. It is derived from streambuf. A procbuf can be closed (in which case it does nothing), or open (in which case it allows communicating through a pipe with some other program).

```cpp
procbuf::procbuf () Constructor
Creates a procbuf in a closed state.
```
Chapter 5: Using the streambuf Layer

**procbuf**

**procbuf::open** (const char *command, int mode)  
Method
Uses the shell (`/bin/sh`) to run a program specified by `command`.

If `mode` is `ios::in`, standard output from the program is sent to a pipe; you can read from the pipe by reading from the procbuf. (This is similar to `popen(command, "r")`.)

If `mode` is `ios::out`, output written written to the procbuf is written to a pipe; the program is set up to read its standard input from (the other end of) the pipe. (This is similar to `popen(command, "w")`.)

The procbuf must start out in the closed state. Returns `*this` on success, and `NULL` on failure.

**procbuf::procbuf** (const char *command, int mode)  
Constructor
Calls `procbuf::open(command, mode)`.

**procbuf**

**procbuf::close** ()  
Method
Waits for the program to finish executing, and then cleans up the resources used. Returns `*this` on success, and `NULL` on failure.

**procbuf::procbuf** ()  
Destructor
Calls `procbuf::close`.

### 5.6 Backing up

The GNU iostream library allows you to ask a streambuf to remember the current position. This allows you to go back to this position later, after reading further. You can back up arbitrary amounts, even on unbuffered files or multiple buffers’ worth, as long as you tell the library in advance. This unbounded backup is very useful for scanning and parsing applications. This example shows a typical scenario:
// Read either "dog", "hound", or "hounddog".
// If "dog" is found, return 1.
// If "hound" is found, return 2.
// If "hounddog" is found, return 3.
// If none of these are found, return -1.
int my_scan(streambuf* sb)
{
    streammarker fence(sb);
    char buffer[20];
    // Try reading "hounddog":
    if (sb->sgetn(buffer, 8) == 8
        && strncmp(buffer, "hounddog", 8) == 0)
        return 3;
    // No, no "hounddog": Back up to 'fence'
    sb->seekmark(fence); //
    // ... and try reading "dog":
    if (sb->sgetn(buffer, 3) == 3
        && strncmp(buffer, "dog", 3) == 0)
        return 1;
    // No, no "dog" either: Back up to 'fence'
    sb->seekmark(fence); //
    // ... and try reading "hound":
    if (sb->sgetn(buffer, 5) == 5
        && strncmp(buffer, "hound", 5) == 0)
        return 2;
    // No, no "hound" either: Back up and signal failure.
    sb->seekmark(fence); // Backup to 'fence'
    return -1;
}

streammarker::streammarker (streambuf* sbuf) Constructor
    Create a streammarker associated with sbuf that remembers the current position of the get pointer.

int streammarker::delta (streammarker& mark) Method
    Return the difference between the get positions corresponding to *this and mark (which must point into the same streambuf as this).

int streambuf::seekmark (streammarker& mark) Method
    Move the get pointer to where it (logically) was when mark was constructed.

5.7 Forwarding I/O activity

An indirectbuf is one that forwards all of its I/O requests to another streambuf.

An indirectbuf can be used to implement Common Lisp synonym-streams and two-way-streams:
class synonymbuf : public indirectbuf {
    Symbol *sym;
    synonymbuf(Symbol *s) { sym = s; }
    virtual streambuf *lookup_stream(int mode) {
        return coerce_to_streambuf(lookup_value(sym));
    }
};
6 C Input and Output

libio is distributed with a complete implementation of the ANSI C stdio facility. It is implemented using streambuf objects. See Section 5.4 [Wrappers for C stdio], page 26.

The stdio package is intended as a replacement for the whatever stdio is in your C library. Since stdio works best when you build libc to contain it, and that may be inconvenient, it is not installed by default.

Extensions beyond ANSI:

- A stdio FILE is identical to a streambuf. Hence there is no need to worry about synchronizing C and C++ input/output—they are by definition always synchronized.
- If you create a new streambuf sub-class (in C++), you can use it as a FILE from C. Thus the system is extensible using the standard streambuf protocol.
- You can arbitrarily mix reading and writing, without having to seek in between.
- Unbounded ungetc() buffer.
Index

(Index is nonexistent)
# Table of Contents

1 Introduction ............................................. 1
   1.1 Licensing terms for libio ........................... 1
   1.2 Acknowledgements .................................. 1

2 Operators and Default Streams ................. 3

3 Stream Classes .............................. 5
   3.1 Shared properties: class ios ....................... 5
       3.1.1 Checking the state of a stream ................ 5
       3.1.2 Choices in formatting .......................... 7
       3.1.3 Changing stream properties using manipulators .... 9
       3.1.4 Extended data fields ............................ 10
       3.1.5 Synchronizing related streams ................ 11
       3.1.6 Reaching the underlying streambuf .......... 11
   3.2 Managing output streams: class ostream .......... 11
       3.2.1 Writing on an ostream .......................... 12
       3.2.2 Repositioning an ostream ........................ 12
       3.2.3 Miscellaneous ostream utilities ............... 13
   3.3 Managing input streams: class istream .......... 13
       3.3.1 Reading one character ........................... 13
       3.3.2 Reading strings .................................... 14
       3.3.3 Repositioning an istream ......................... 15
       3.3.4 Miscellaneous istream utilities ............... 15
   3.4 Input and output together: class iostream ....... 16

4 Classes for Files and Strings ............... 19
   4.1 Reading and writing files ......................... 19
   4.2 Reading and writing in memory ................. 21

5 Using the streambuf Layer .................. 23
   5.1 Areas of a streambuf ............................. 23
   5.2 Simple output re-direction by redefining overflow .... 24
   5.3 C-style formatting for streambuf objects .......... 26
   5.4 Wrappers for C stdio ............................. 26
   5.5 Reading/writing from/to a pipe .................... 26
   5.6 Backing up ........................................... 27
   5.7 Forwarding I/O activity ........................... 28

6 C Input and Output ......................... 31

Index ............................................. 33