
15.5. Standard Input/Output Functions

Instead of using the standard operators for streams (operator `<<` and operator `>>`), you can use the member functions presented in this section for reading and writing. These functions read or write "unformatted" data, unlike operators `>>` or `<<`, which read or write "formatted" data. When reading, the functions described in this section never skip leading whitespaces, which is different to operator `>>`, which, by default, skips leading whitespace. This is handled by a `sentry` object (see Section 15.5.4, page 772). Also, these functions handle exceptions differently from the formatted I/O operators: If an exception is thrown, either from a called function or as a result of setting a state flag (see Section 15.4.4, page 762), the `badbit` flag is set. The exception is then rethrown if the exception mask has `badbit` set.

The standard I/O functions use type `streamsize`, which is defined in `<ios>`, to specify counts:

```
namespace std {
    typedef ... streamsize;
    ...
}
```

The type `streamsize` usually is a signed version of `size_t`. It is signed because it is also used to specify negative values.

15.5.1. Member Functions for Input

In the following definitions, *istream* is a placeholder for the stream class used for reading. It can stand for `istream`, `wistream`, or other instantiation of the class template `basic_istream<>`. The type *char* is a placeholder for the corresponding character type, which is `char` for `istream` and `wchar_t` for `wistream`. Other types or values printed in italics depend on the exact definition of the character type or on the traits class associated with the stream.

For istreams, the C++ standard library provides several member functions to read character sequences. Table 15.7 compares their abilities (*s* refers to the character sequence the characters are read into).

Table 15.7. Abilities of Stream Operators Reading Character Sequences

Member Function	Reads Until	Number of Characters	Appends Terminator	Returns
`get(`*s*`,`*num*`)`	Excluding newline or end-of-file	Up to *num*−1	Yes	*istream*
`get(`*s*`,`*num*`,`*t*`)`	Excluding *t* or end-of-file	Up to *num*−1	Yes	*istream*
`getline(`*s*`,`*num*`)`	Including newline or end-of-file	Up to *num*−1	Yes	*istream*
`getline(`*s*`,`*num*`,`*t*`)`	Including `t` or end-of-file	Up to *num*−1	Yes	*istream*
`read(`*s*`,`*num*`)`	End-of-file	*num*	No	*istream*
`readsome(`*s*`,`*num*`)`	End-of-file	Up to *num*	No	Count

int **`istream::get`** `()`

- Reads the next character.
- Returns the read character or *EOF*.
- In general, the return type is `traits::int_type`, and *EOF* is the value returned by `traits::eof()`. For `istream`, the return type is `int`, and *EOF* is the constant `EOF`. Hence, for `istream`, this function corresponds to C's `getchar()` or `getc()`.
- Note that the returned value is not necessarily of the character type but can be of a type with a larger range of values. Otherwise, it would be impossible to distinguish *EOF* from characters with the corresponding value.

`istream&` **`istream::get`** `(`*char*`&` *c*`)`

- Assigns the next character to the passed argument *c*.
- Returns the stream. The stream's state tells whether the read was successful.

`istream&` **`istream::get`** `(`*char*`*` *str*`, streamsize` *count*`)`
`istream&` **`istream::get`** `(`*char*`*` *str*`, streamsize` *count*`,` *char delim*`)`

- Both forms read up to *count* `-1` characters into the character sequence pointed to by *str*.
- The first form terminates the reading if the next character to be read is the newline character of the corresponding character set. For `istream`, it is the character `'\n'`, and for `wistream`, it is `wchar_t('\n')` (see Section 16.1.5, page 857). In general, `widen('\n')` is used (see Section 15.8, page 790).
- The second form terminates the reading if the next character to be read is *delim*.
- Both forms return the stream. The stream's state tells whether the read was successful.
- The terminating character (*delim*) is not read.
- The read character sequence is terminated by a (terminating) null character.
- The caller must ensure that *str* is large enough for *count* characters.

istream& **istream**::**getline** (*char* str*, `streamsize` *count*)
istream& **istream**::**getline** (*char* str*, `streamsize` *count*, *char delim*)

- Both forms are identical to their previous counterparts of `get()`, except as follows:
  - They terminate the reading *including* but not before the newline character or *delim*, respectively. Thus, the newline character or *delim* is read if it occurs within *count* `-1` characters, but it is *not* stored in *str*.
  - If they read lines with more than *count* `-1` characters, they set `failbit`.

istream& **istream**::**read** (*char* str*, `streamsize` *count*)

- Reads *count* characters into the string *str*.
- Returns the stream. The stream's state tells whether the read was successful.
- The string in *str* is *not* terminated automatically with a (terminating) null character.
- The caller must ensure that *str* has sufficient space to store *count* characters.
- Encountering end-of-file during reading is considered an error, and `failbit` is set in addition to `eofbit`.

`streamsize` **istream**::**readsome** (*char* str*, `streamsize` *count*)

- Reads up to *count* characters into the string *str*.
- Returns the number of characters read.
- The string in *str* is *not* terminated automatically with a (terminating) null character.
- The caller must ensure that *str* has sufficient space to store *count* characters.
- In contrast to `read()`, `readsome()` reads all available characters of the stream buffer, using the `in_avail()` member function of the buffer (see Section 15.13.1, page 827). This is useful when it is undesirable to wait for the input because it comes from the keyboard or other processes. Encountering end-of-file is not considered an error and sets neither `eofbit` nor `failbit`.

`streamsize` **istream**::**gcount** () `const`

- Returns the number of characters read by the last *unformatted* read operation.

istream& **istream**::**ignore** ()
istream& **istream**::**ignore** (`streamsize` *count*)
istream& **istream**::**ignore** (`streamsize` *count*, *int delim*)

- All forms extract and discard characters.
- The first form ignores one character.
- The second form ignores up to *count* characters.
- The third form ignores up to *count* characters until *delim* is extracted and discarded.
- If *count* is `std::numeric_limits<std::streamsize>::max()` (the largest value of type `std::streamsize`; see Section 5.3, page 115), all characters are discarded until either *delim* or end-of-file is reached.
- All forms return the stream.
- Examples:
  - The following call discards the rest of the line:

```
cin.ignore(numeric_limits<std::streamsize>::max(),'\n');
```

  - The following call discards the complete remainder of `cin`:

```
cin.ignore(numeric_limits<std::streamsize>::max());
```

int **istream**::**peek** ()

- Returns the next character to be read from the stream without extracting it. The next read will read this character (unless the read

position is modified).

- Returns *EOF* if no more characters can be read.
- *EOF* is the value returned from `traits::eof()`. For `istream`, this is the constant `EOF`.

`istream`& `istream::unget` ()
`istream`& `istream::putback` (*char c*)

- Both functions put the last character read back into the stream so that it is read again by the next read (unless the read position is modified).
- The difference between `unget()` and `putback()` is that for `putback()`, a check is made whether the character *c* passed is indeed the last character read.
- If the character cannot be put back or if the wrong character is put back with `putback()`, `badbit` is set, which may throw a corresponding exception (see Section 15.4.4, page 762).
- The maximum number of characters that can be put back with these functions is unspecified. Only one call of these functions between two reads is guaranteed to work by the standard and thus is portable.

When C-strings are read, it is safer to use the functions from this section than to use operator `>>`. The reason is that the maximum string size to be read must be passed explicitly as an argument. Although it is possible to limit the number of characters read when using operator `>>` (see Section 15.7.3, page 781), this is easily forgotten.

It is often better to use the stream buffer directly instead of using istream member functions. Stream buffers provide member functions that read single characters or character sequences efficiently, without overhead due to the construction of `sentry` objects (see Section 15.5.4, page 772, for more information on `sentry` objects). Section 15.13, page 826, explains the stream buffer interface in detail. Another alternative is to use the class template `istreambuf_iterator<>`, which provides an iterator interface to the stream buffer (see Section 15.13.2, page 828).

Two other functions for manipulating the read position are `tellg()` and `seekg()`, which are relevant mainly in conjunction with files. Their descriptions are deferred until Section 15.9.4, page 799.

15.5.2. Member Functions for Output

In the following definitions, *ostream* is a placeholder for the stream class used for writing. It can stand for `ostream`, `wostream`, or other instantiation of the class template `basic_ostream<>`. The type *char* is a placeholder for the corresponding character type, which is `char` for `ostream` and `wchar_t` for `wostream`. Other types or values printed in italics depend on the exact definition of the character type or on the traits class associated with the stream.

`ostream`& `ostream::put` (*char c*)

- Writes the argument *c* to the stream.
- Returns the stream. The stream's state tells whether the write was successful.

`ostream`& `ostream::write` (`const` *char* str*, `streamsize` *count*)

- Writes *count* characters of the string *str* to the stream.
- Returns the stream. The stream's state tells whether the write was successful.
- The (terminating) null character does *not* terminate the write and will be written.
- The caller must ensure that *str* contains at least *count* characters; otherwise, the behavior is undefined.

`ostream`& `ostream::flush` ()

- Flushes the buffers of the stream: forces a write of all buffered data to the device or I/O channel to which it belongs.

Two other functions modify the write position: `tellp()` and `seekp()`, which are relevant mainly in conjunction with files. Their descriptions are deferred until Section 15.9.4, page 799.

As with the input functions, it may be reasonable to use the stream buffer directly (see Section 15.14.3, page 846) or to use the class template `ostreambuf_iterator<>` for unformatted writing (see Section 15.13.2, page 828). In fact, there is no point in using the unformatted output functions except that they use `sentry` objects (see Section 15.5.4, page 772), which, for example, synchronize tied output streams (see Section 15.12.1, page 819).

15.5.3. Example Uses

The classic C/UNIX filter framework that simply writes all read characters looks like this in C++:

```
// io/charcat1.cpp

#include <iostream>
using namespace std;
```

```
int main()
{
    char c;

    // while it is possible to read a character
    while (cin.get(c)) {
        // print it
        cout.put(c);
    }
}
```

With each call of the following expression, the next character is simply assigned to `c`, which is passed by reference:

```
cin.get(c)
```

The return value of `get()` is the stream; thus, `while` tests whether `cin` is still in a good state.[10]

[10] Note that this interface is better than the usual C interface for filters. In C, you have to use `getchar()` or `getc()`, which return both the next character or whether end-of-file was reached. This causes the problem that you have to process the return value as `int` to distinguish any `char` value from the value for end-of-file.

For a better performance, you can operate directly on stream buffers. See Section 15.13.2, page 831, for a version of this example that uses stream buffer iterators for I/O and Section 15.14.3, page 846, for a version that copies the whole input in one statement.

15.5.4. `sentry` Objects

The I/O stream operators and functions use a common scheme for providing their functionality: First, some preprocessing prepares the stream for I/O. Then the actual I/O is done, followed by some postprocessing.

To implement this scheme, classes `basic_istream` and `basic_ostream` each define an auxiliary class `sentry`. The constructor of these classes does the preprocessing, and the destructor does the corresponding postprocessing.[11] Thus, all formatted and unformatted I/O operators and functions use a `sentry` object before they perform their actual processing and operate as follows:

[11] These classes replace the member functions that were used in former implementations of the IOStream library (`ipfx()`, `isfx()`, `opfx()`, and `osfx()`). Using the new classes ensures that the postprocessing is invoked even if the I/O is aborted with an exception.

```
sentry se(strm);     // indirect pre- and postprocessing
if (se) {
    ...              // the actual processing
}
```

The `sentry` object takes as the constructor argument the stream `strm`, on which the pre- and postprocessing should be done. The remaining processing then depends on the state of this object, which indicates whether the stream is OK. This state can be checked using the conversion of the `sentry` object to `bool`. For input streams, the `sentry` object can be constructed with an optional Boolean value that indicates whether skipping of whitespace should be avoided even though the flag `skipws` is set:

```
sentry se(strm,true);    // don't skip whitespaces during the additional processing
```

The pre- and postprocessing perform all general tasks of I/O using streams. These tasks include synchronizing several streams, checking whether the stream is OK, and skipping whitespaces, as well as possibly implementation-specific tasks. For example, in a multithreaded environment, the additional processing might be used for corresponding locking.

If an I/O operator operates directly on the stream buffer, a corresponding `sentry` object should be constructed first.