Chapter 13. Input/Output Using Stream Classes

The classes for I/O form an important part of the C++ standard library; a program without I/O is not of much use. Actually, the I/O classes from the C++ standard library are not restricted to files or to screen and keyboard. Instead, they form an extensible framework for the formatting of arbitrary data and access to arbitrary "external representations."

The IOStream library, as the classes for I/O are called, is the only part of the C++ standard library that was used widely prior to the standardization of C++. Early distributions of C++ systems came with a set of classes developed at AT&T that established a de facto standard for doing I/O. Although these classes have undergone several changes to fit consistently into the C++ standard library and to suit new needs, the basic principles of the IOStream library remain unchanged.

This chapter first presents a general overview of the most important components and techniques, and then demonstrates in detail how the IOStream library can be used in practice. Its use ranges from simple formatting to the integration of new external representations (a topic that is often addressed improperly).

This chapter does not attempt to discuss all aspects of the IOStream library in detail; to do that would take an entire book by itself. For details not found here, please consult one of the books that focus on the I/O stream library or the reference manual of the C++ standard library.

Many thanks to Dietmar Kühl, who is an expert on I/O and internationalization in the C++ standard library and gave very much feedback and wrote some parts of this chapter.

Recent Changes in the IOStream Library

For those already familiar with the "old-fashioned" IOStream library, this section outlines changes introduced during the standardization process. Although the basics of the I/O stream classes remained unchanged, some important features allowing additional customization were introduced. Here is a brief list of the major changes:

- I/O became internationalized.
- The string stream classes for character arrays of type char* were replaced with classes that use the string types of the C++ standard library. The former classes are still retained for backward compatibility, but their use is deprecated.[1]
- Exception handling was integrated into state and error handling.
- The IOStream library classes supporting assignment (those ending in _withassign) were replaced with a different approach available to all stream classes.
- The classes from the IOStream library were made templates to support different character representations. As a side effect, this renders simple forward declarations of stream classes illegal. A header was introduced to provide the appropriate declarations.

    [1] Deprecated means that a feature is not recommended because some superior feature exists. Also, deprecated features are likely to disappear from a future version of the standard.

- The classes from the IOStream library were made templates to support different character representations. As a side effect, this renders simple forward declarations of stream classes illegal. A header was introduced to provide the appropriate declarations.

So, instead of using

```
    class ostream; // wrong
```

this new header should be used:
#include <iosfwd> // OK

- Like the other parts of the C++ standard library, all symbols of the IOStream library are now declared in the namespace `std`.

## 13.1 Common Background of I/O Streams

Before going into details about stream classes, I briefly discuss the generally known aspects of streams to provide a common background. This section could be skipped by readers familiar with iostream basics.

### 13.1.1 Stream Objects

In C++, I/O is performed by using streams. A stream is a "stream of data" in which character sequences "flow." Following the principles of object orientation, a stream is an object with properties that are defined by a class. Output is interpreted as data flowing into a stream; input is interpreted as data flowing out of a stream. Global objects are predefined for the standard I/O channels.

### 13.1.2 Stream Classes

Just as there are different kinds of I/O (for example, input, output, and file access), there are different classes depending on the type of I/O. The following are the most important stream classes:

- **Class `istream`**
  - Defines input streams that can be used to read data.

- **Class `ostream`**
  - Defines output streams that can be used to write data.

Both classes are instantiations of template classes, namely of the classes `basic_istream<>` and `basic_ostream<>` using `char` as the character type. Actually, the whole IOStream library does not depend on a specific character type. Instead the character type used is a template argument for most of the classes in the IOStream library. This parameterization corresponds to the string classes and is used for internationalization (see also Section 14). This section concentrates on output to and output from "narrow streams"; that is, streams dealing with `char` as the character type. Later in this chapter the discussion is extended to streams that have other character types.

### 13.1.3 Global Stream Objects

The IOStream library defines several global objects of type `istream` and `ostream`. These objects correspond to the standard I/O channels:

- `cin`
\texttt{cin} (of class \texttt{istream}) is the standard input channel that is used for user input. This stream corresponds to C's \texttt{stdin}. Normally, this stream is connected to the keyboard by the operating system.

- \texttt{cout}

\texttt{cout} (of class \texttt{ostream}) is the standard output channel that is used for program output. This stream corresponds to C's \texttt{stdout}. Normally, this stream is connected to the monitor by the operating system.

- \texttt{cerr}

\texttt{cerr} (of class \texttt{ostream}) is the standard error channel that is used for all kinds of error messages. This stream corresponds to C's \texttt{stderr}. Normally, this stream is also connected to the monitor by the operating system. By default, \texttt{cerr} is not buffered.

- \texttt{clog}

\texttt{clog} (of class \texttt{ostream}) is the standard logging channel. There is no C equivalent for this stream. By default, this stream is connected to the same destination as \texttt{cerr}, with the difference that output to \texttt{clog} is buffered.

The separation of "normal" output and error messages makes it possible to treat these two kinds of output differently when executing a program. For example, the normal output of a program can be redirected into a file while the error messages are still appearing on the console. Of course, this requires that the operating system supports redirection of the standard I/O channels (most operating systems do). This separation of standard channels originates from the UNIX concept of I/O redirection.

### 13.1.4 Stream Operators

The shift operators \texttt{<<} for input and \texttt{>>} for output are overloaded for the corresponding stream classes. For this reason, in C++ the "shift operators" became the "I/O operators."\footnote{According to the fact that these operators insert characters into a stream or extract characters from a stream, some people also call the I/O operators \textit{inserters} and \textit{extractors}.} Using these operators, it is possible to chain multiple I/O operations.

For example, for each iteration, the following loop reads two integers from the standard input (as long as only integers are entered) and writes them to the standard output:

```cpp
int a, b;

// as long as input of a and b is successful
while (std::cin >> a >> b) {
    // output a and b
    std::cout << "a: " << a << " b: " << b << std::endl;
}
```

### 13.1.5 Manipulators
At the end of most output statements, a so-called manipulator is written:

```cpp
std::cout << std::endl
```

Manipulators are special objects that are used to, guess what, manipulate a stream. Often, manipulators only change the way input is interpreted or output is formatted, like the manipulators for the numeric bases `dec`, `hex`, and `oct`. Thus, manipulators for `ostreams` do not necessarily create output, and manipulators for `istreams` do not necessary consume input. But there are also manipulators that actually trigger some immediate action. For example, a manipulator can be used to flush the output buffer or to skip whitespace in the input buffer. The manipulator `endl` means "end line" and does two things:

1. Outputs a newline (that is, the character `'\n'`)
2. Flushes the output buffer (forces a write of all buffered data for the given stream using the stream method `flush()`)

The most important manipulators defined by the I/OStream library are provided in Table 13.1. Section 13.6, discusses manipulators in more detail, including those that are defined in the I/OStream library, and describes how to define your own manipulators.

<table>
<thead>
<tr>
<th>Manipulator</th>
<th>Class</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>endl</code></td>
<td>ostream</td>
<td>Outputs <code>'\n'</code> and flushes the output buffer</td>
</tr>
<tr>
<td><code>ends</code></td>
<td>ostream</td>
<td>Outputs <code>'\0'</code></td>
</tr>
<tr>
<td><code>flush</code></td>
<td>ostream</td>
<td>Flushes the output buffer</td>
</tr>
<tr>
<td><code>ws</code></td>
<td>istream</td>
<td>Reads and discards whitespaces</td>
</tr>
</tbody>
</table>

### 13.1.6 A Simple Example

The use of the stream classes is demonstrated by the following example. This program reads two floating-point values and outputs their product:

```cpp
// io/io1.cpp
#include <cstdlib>
#include <iostream>
using namespace std;

int main()
{
    double x, y;  // operands

    // print header string
    cout << "Multiplication of two floating point values" << endl;

    // read first operand
    cout << "first operand: ";
    if (! (cin >> x)) {
        /* input error
         * => error message and exit program with error status
         */
        cerr << "error while reading the first floating value"
    }
    cout << "second operand: ";
    if (! (cin >> y)) {
        /* input error
         * => error message and exit program with error status
         */
        cerr << "error while reading the second floating value"
    }
    cout << "Product = " << x * y << endl;
}
```
// read second operand
cout << "second operand: ";
if (! (cin >> y)) {
    /* input error
       * => error message and exit program with error status
       */
    cerr << "error while reading the second floating value"
         << endl;
    return EXIT_FAILURE;
}

// print operands and result
cout << x << " times " << y << " equals " << x * y << endl;

13.2 Fundamental Stream Classes and Objects

13.2.1 Classes and Class Hierarchy

The stream classes of the IOStream library form a hierarchy, as shown in Figure 13.1. For template classes, the upper row shows the name of the template class, and the lower row presents the names of the instantiations for the character types \texttt{char} and \texttt{wchar_t}.

Figure 13.1. Class Hierarchy of the Fundamental Stream Classes

The classes in this class hierarchy play the following roles:
The base class `ios_base` defines the properties of all stream classes independent of the character type and the corresponding character traits. Most of this class consists of components and functions for state and format flags.

The class template `basic_ios<>` is derived from `ios_base` and it defines the common properties of all stream classes that depend on the character types and the corresponding character traits. These properties include the definition of the buffer used by the stream. The buffer is an object of a class derived from the template class `basic_streambuf<>` with the corresponding template instantiation. It performs the actual reading and/or writing.

The class templates `basic_istream<>` and `basic_ostream<>` derive virtually from `basic_ios<>`, and define objects that can be used for reading or writing respectively. Like `basic_ios<>`, these classes are templates that are parameterized with a character type and its traits. When internationalization does not matter, the corresponding instantiations for the character type `char` (namely, `istream` and `ostream`) are used.

The class template `basic_iostream<>` derives from both `basic_istream<>` and `basic_ostream<>`. This class template defines objects that can be used for both reading and writing.

The class template `basic_streambuf<>` is the heart of the IOStream library. This class defines the interface to all representations that can be written to or read from by streams. It is used by the other stream classes to perform the actual reading and writing of characters. For access to some external representation, classes are derived from `basic_streambuf<>`. See the following subsection for details.

**Purpose of the Stream Buffer Classes**

The IOStream library is designed with a rigid separation of responsibilities. The classes derived from `basic_ios` "only" handle formatting of the data. The actual reading and writing of characters is performed by the stream buffers maintained by the `basic_ios` subobjects. The stream buffers supply character buffers for reading and writing. In addition, an abstraction from the external representation (for example files or strings) is formed by the stream buffers. Actually, they don't even do the formatting! The actual formatting is delegated to corresponding facets in the locale library. See Section 14.2.2 and Section 14.4 for details on facets.

Thus, stream buffers play an important role when performing I/O with new external representations (such as sockets or graphical user interface components), redirecting streams, or combining streams to form pipelines (for example, to compress output before writing to another stream). Also, the stream buffer synchronizes the I/O when doing simultaneous I/O on the same external representation. The details about these techniques are explained in Section 13.10.2. By using stream buffers it is quite easy to define access to a new "external representation" like a new storage device. All that has to be done is to derive a new stream buffer class from `basic_streambuf<>` (or an appropriate specialization) and define functions for reading and/or writing characters for this new external representation. All options for formatted I/O are available automatically if a stream object is initialized to use an object of the new stream buffer class. Section 13.13 explains how to define new stream buffers for access to special storage devices.

**Detailed Class Definitions**

Like all template classes in the IOStream library, the template class `basic_ios<>` is parameterized with two arguments and defined as

```cpp
namespace std {
    template <class charT,
```

---

**[3]** Actually, they don't even do the formatting! The actual formatting is delegated to corresponding facets in the locale library. See Section 14.2.2 and Section 14.4 for details on facets.
The template arguments are the character type used by the stream classes and a class describing the traits of the character type that are used by the stream classes. Examples of traits defined in the traits class are the value used to represent end-of-file and the instructions for how to copy or move a sequence of characters. Normally, the traits for a character type are coupled with the character type, thereby making it reasonable to define a template class that is specialized for specific character types. Hence, the traits class defaults to char_traits if charT is the character type argument. The C++ standard library provides specializations of the class char_traits for the character types char and wchar_t.

For more details about character traits, see Section 14.1.2.

[4] I use the term end-of-file for the "end of input data." This is according to the constant EOF in C.

There are two instantiations of the class basic_ios for the two character types used most often:

namespace std {
    typedef basic_ios<char> ios;
    typedef basic_ios<wchar_t> wios;
}

The type ios corresponds to the base class of the "old-fashioned" IOSStream library from AT&T and can be used for compatibility in older C++ programs. The stream buffer class used by basic_ios is defined similarly:

namespace std {
    template <class charT, class traits = char_traits<charT> >
    class basic_streambuf;
    typedef basic_streambuf<char> streambuf;
    typedef basic_streambuf<wchar_t> wstreambuf;
}

Of course, the class templates basic_istream, basic_ostream, and basic_iostream are also parameterized with the character type and a traits class:

namespace std {
    template <class charT, class traits = char_traits<charT> >
    class basic_istream;

    template <class charT, class traits = char_traits<charT> >
    class basic_ostream;

    template <class charT, class traits = char_traits<charT> >
    class basic_iostream;
}
As for the other classes, there are also type definitions for the instantiations of the two most important character types:

```cpp
namespace std {
    typedef basic_istream<char>    istream;
    typedef basic_istream<wchar_t> wistream;
    typedef basic_ostream<char>    ostream;
    typedef basic_ostream<wchar_t> wostream;
    typedef basic_iostream<char>    iostream;
    typedef basic_iostream<wchar_t> wiostream;
}
```

The types `istream` and `ostream` are the types normally used in the western hemisphere. They are mostly compatible with the "old-fashioned" stream classes of AT&T. The classes `istream_withassign`, `ostream_withassign`, and `iostream_withassign`, which are present in some older stream libraries (derived from `istream`, `ostream`, and `iostream` respectively) are not supported by the standard. The corresponding functionality is achieved differently (see Section 13.10.3).

There are additional classes for formatted I/O with files and strings. These classes are discussed in Section 13.9 and Section 13.11.

### 13.2.2 Global Stream Objects

Several global stream objects are defined for the stream classes. These objects are the objects for access to the standard I/O channels that are mentioned previously for streams with `char` as the character type and a set of corresponding objects for the streams using `wchar_t` as the character type (see Table 13.2).

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>istream</td>
<td>Cin</td>
<td>Reads input from the standard input channel</td>
</tr>
<tr>
<td>ostream</td>
<td>cout</td>
<td>Writes &quot;normal&quot; output to the standard output channel</td>
</tr>
<tr>
<td>ostream</td>
<td>cerr</td>
<td>Writes error messages to the standard error channel</td>
</tr>
<tr>
<td>ostream</td>
<td>clog</td>
<td>Writes log messages to the standard logging channel</td>
</tr>
<tr>
<td>wistream</td>
<td>wcin</td>
<td>Reads wide-character input from the standard input channel</td>
</tr>
<tr>
<td>wostream</td>
<td>wcout</td>
<td>Writes &quot;normal&quot; wide-character output to the standard output channel</td>
</tr>
<tr>
<td>wostream</td>
<td>wcerr</td>
<td>Writes wide-character error messages to the standard error channel</td>
</tr>
<tr>
<td>wostream</td>
<td>wclog</td>
<td>Writes wide-character log messages to the standard logging channel</td>
</tr>
</tbody>
</table>

By default, these standard streams are synchronized with the standard streams of C. That is, the C++ standard library ensures that the order of mixed output with C++ streams and C streams is preserved. Before any buffer of standard C++ streams writes data it flushes the buffer of the corresponding C streams and vice versa. Of course, this synchronization takes some time. If it isn't necessary you can turn it off by calling `sync_with_stdio(false)` before any input or output is done (see page 682).

### 13.2.3 Header Files

The definitions of the stream classes are scattered among several header files:
• `<iosfwd>`

Contains forward declarations for the stream classes. This header file is necessary because it is no longer permissible to use a simple forward declaration such as `class ostream`.

• `<streambuf>`

Contains the definitions for the stream buffer base class (basic_streambuf <>).

• `<istream>`

Contains the definitions for the classes that support input only (basic_istream<> ) and for the classes that support both input and output (basic_iostream<>).

At first, `<istream>` might not appear to be a logical choice for declaration of the classes for input and output. However, because there may be some initialization overhead at start-up for every translation unit that includes `<iostream>` (see the following paragraph for details), the declarations for input and output were put into `<istream>`.

• `<ostream>`

Contains the definitions for the output stream class (basic_ostream<>).

• `<iostream>`

Contains declarations of the global stream objects (such as `cin` and `cout`).

Most of the headers exist for the internal organization of the C++ standard library. For the application programmer it should be sufficient to include `<iosfwd>` for the declaration of the stream classes and `<istream>` or `<ostream>` when actually using the input or output functions respectively. The header `<iostream>` should only be included if the standard stream objects are to be used. For some implementations some code is executed at start-up for each translation unit including this header. The actual code being executed is not that expensive but it requires loading of the corresponding pages of the executable, which might be expensive. In general, only those headers defining necessary "stuff" should be included. In particular, header files should only include `<iosfwd>`, and the corresponding implementation files should then include the header with the complete definition.

For special stream features, such as parameterized manipulators, file streams, or string streams, there are additional headers (`<iomanip>`, `<fstream>`, `<sstream>`, and `<strstream>`). The details regarding these headers are provided in the sections that introduce these special features.

### 13.3 Standard Stream Operators `<<` and `>>`

In C and C++, operators `<<` and `>>` are used for shifting bits of an integer to the right or the left respectively. The classes basic_istream<> and basic_ostream<> overload operators `>>` and `<<` as the standard I/O operators. Thus, in C++ the "shift operators" became the "I/O operators."[6]


[5] At first, `<istream>` might not appear to be a logical choice for declaration of the classes for input and output. However, because there may be some initialization overhead at start-up for every translation unit that includes `<iostream>` (see the following paragraph for details), the declarations for input and output were put into `<istream>`. 
13.3.1 Output Operator <<

The class `basic_ostream` (and thus also the classes `ostream` and `wstream`) defines `<<` as an output operator. It overloads this operator for all fundamental types, including `char*`, `void*`, and `bool`.

The output operators for streams are defined to send their second argument to the corresponding stream. Thus, the data is sent in the direction of the arrow:

```cpp
int i = 7;
std::cout << i; // outputs: 7

float f = 4.5;
std::cout << f; // outputs: 4.5
```

The `<<` operator can be overloaded such that the second argument is an arbitrary data type. This allows the integration of your own data types into the I/O system. The compiler ensures that the correct function for outputting the second argument is called. Of course, this function should in fact transform the second argument into a sequence of characters sent to the stream.

The C++ standard library also uses this mechanism to provide output operators for strings (see page 524), bitsets (see page 468), and complex numbers (see page 539):

```cpp
std::string s("hello");
s += ", world";
std::cout << s; // outputs: hello, world

std::bitset<10> flags(7);
std::cout << flags; // outputs: 0000000011

std::complex<float> c(3.1,7.4);
std::cout << c; // outputs: (3.1,7.4)
```

The details about writing output operators for your own data types are explained in Section 13.12.

The fact that the output mechanism can be extended to incorporate your own data types is a significant improvement over C’s I/O mechanism that uses `printf()`: It is not necessary to specify the type of an object to be printed. Instead, the overloading of different types ensures that the correct function for printing is deduced automatically. The mechanism is not limited to standard types. Thus, the user has only one mechanism that works for all types.

Operator `<<` can also be used to print multiple objects in one statement. By convention, the output operators return their first argument. Thus, the result of an output operator is the output stream. This allows you to chain calls to output operators like this:

```cpp
std::cout << x << " times " << y << " is " << x * y << std::endl;
```

Operator `<<` is evaluated from left to right. Thus:

```cpp
std::cout << x
```

is executed first. Note that the evaluative order of the operator does not imply any specific order in which the arguments are evaluated; only the order in which the operators are executed is defined. This expression returns its first operand— `std::cout`. So,
std::cout << " times "

is executed next. The object y, the string literal " is ", and the result of x * y are printed accordingly. Note that the multiplication operator has a higher priority than operator <<, so you need no parentheses around x * y. However, there are operators that have less priority, such as all logical operators. In this example, if x and y are floating-point numbers with the values 2.4 and 5.1, the following is printed:

2.4 times 5.1 is 12.24

13.3.2 Input Operator `>>`

The class basic_istream (and thus also the classes istream and wistream) defines `>>` as an input operator. Similar to basic_ostream, this operator is overloaded for all fundamental types including, char*, void*, and bool. The input operators for streams are defined to store the value read in their second argument. As with operator `<<`, the data is sent in the direction of the arrow:

```cpp
int i;
std::cin >> i; // reads an int from standard input and stores it in i

float f;
std::cin >> f; // reads a float from standard input and stores it in f
```

Note that the second argument is modified. To make this possible, the second argument is passed by nonconstant reference. Like output operator `<<` it is also possible to overload the input operator for arbitrary data types and to chain the calls:

```cpp
float f;
std::complex c;
std::cin >> f >> c;
```

To make this possible, leading whitespace is skipped by default. However, this automatic skipping of whitespace can be turned off (see page 625).

13.3.3 Input/Output of Special Types

The standard I/O operators are also defined for types bool, char*, and void*. In addition, you can extend it for your own types.

Type bool
By default, Boolean values are printed and read numerically: `false` is converted from and to 0, and `true` is converted from and to 1. When reading, values different from 0 and 1 are considered to be an error. In this case the `ios::failbit` is set, which might throw a corresponding exception (see page 602).

It is also possible to set up the formatting options of the stream to use character strings for the I/O of Boolean values (see page 617). This touches on the topic of internationalization: Unless a special locale object is used, the strings "true" and "false" are used. In other locale objects, different strings might be used. For example, a German locale object would use the strings "wahr" and "falsch." See Chapter 14 especially for more details.

**Types `char` and `wchar_t`**

When a `char` or `wchar_t` is being read with operator `>`, leading whitespace is skipped by default. To read any character (whether or not it is whitespace) you can either clear the flag `skipws` (see page 625) or use the member function `get()` (see page 608).

**Type `char*`**

A C-string (that is, a `char*`) is read word wise. That is, when a C-string is being read, leading whitespace is skipped by default and the string is read until another whitespace character or end-of-file is encountered. Whether leading whitespace is skipped automatically can be controlled with the flag `skipws` (see Section 13.7.7).

Note that this behavior means that the string you read can become arbitrarily long. It is already a common error in C programs to assume that a string can be a maximum of 80 characters long. There is no such restriction. Thus, you must arrange for a premature termination of the input when the string is too long. To do this, you should always set the maximum length of the string to be read. This normally looks something like this:

```cpp
c char buffer [81]; // 80 characters and '\0'
std::cin >> std::setw(81) >> buffer;
```

The manipulator `setw()` and the corresponding stream parameter are described in detail in Section 13.7.3.

The type string from the C++ standard library (see Chapter 11) grows as needed to accommodate a lengthy string. It is much easier and safer to use the string class instead of `char*`. In addition, it provides a convenient function for reading line-by-line (see page 493). So, whenever you can avoid the use of C-strings and use strings.

**Type `void*`**

Operators `<<` and `>>` also provide the possibility of printing a pointer and reading it back in again. An address is printed in an implementation-dependent format if a parameter of type `void*` is passed to the output operator. For example, the following statement prints the contents of a C-string and its address:

```cpp
c char* cstring = "hello";
std::cout << "string \n" << cstring << "\n is located at address: "
<< static_cast<void*>(cstring) << std::endl;
```

The result of this statement might appear as follows:
string "hello" is located at address: 0x10000018

It is even possible to read an address again with the input operator. However, note that addresses are normally transient. The same object can get a different address in a newly started program. A possible application of printing and reading addresses may be programs that exchange addresses for object identification or programs that share memory.

**Stream Buffers**

You can use operators `>>` and `<<` to read directly into a stream buffer and to write directly out of a stream buffer respectively. This is probably the fastest way to copy files by using C++ I/O streams. See page 683 for examples.

**User-Defined Types**

In principle it is very easy to extend this technique to your own types. However, to be able to pay attention to all possible formatting data and error conditions, this takes more effort than you might think. See Section 13.12, for a detailed discussion about extending the standard I/O mechanism for your own types.

**13.4 State of Streams**

Streams maintain a state. The state identifies whether I/O was successful and, if not, the reason for the failure.

**13.4.1 Constants for the State of Streams**

For the general state of streams, several constants of type `iostate` are defined to be used as flags (Table 13.3). The type `iostate` is a member of the class `ios_base`. The exact type of the constants is an implementation detail (in other words, it is not defined whether `iostate` is an enumeration, a type definition for an integral type, or an instantiation of the class `bitset`).

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>goodbit</code></td>
<td>Everything is OK; none of the other bits is set</td>
</tr>
<tr>
<td><code>eofbit</code></td>
<td>End-of-file was encountered</td>
</tr>
<tr>
<td><code>failbit</code></td>
<td>Error; an I/O operation was not successful</td>
</tr>
<tr>
<td><code>badbit</code></td>
<td>Fatal error; undefined state</td>
</tr>
</tbody>
</table>

`goodbit` is defined to have the value 0. Thus, having `goodbit` set actually means that all other bits are cleared. The name `goodbit` may be somewhat confusing because it doesn't mean that one bit is set; it means that all bits are cleared.

The difference between `failbit` and `badbit` is basically that `badbit` indicates a more fatal error:

- `failbit` is set if an operation was not processed correctly but the stream is generally OK. Normally this flag is set as a result of a format error during reading. For example, this flag is set if an integer is to be read but the next character is a letter.
- `badbit` is set if the stream is somehow corrupted or if data is lost. For example, this flag is set when positioning a stream that refers to a file before the beginning of a file.
Note that `eofbit` normally happens with `failbit` because the end-of-file condition is checked and detected when an attempt is made to read beyond end-of-file. After reading the last character, the flag `eofbit` is not yet set. The next attempt to read a character sets `eofbit` and `failbit`, because the read fails.

Some former implementations supported the flag `hardfail`. This flag is not supported in the standard.

These constants are not defined globally. Instead, they are defined within the class `ios_base`. Thus, you must always use them with the scope operator or with some object. For example:

```cpp
std::ios_base::eofbit
```

Of course, it is also possible to use a class derived from `ios_base`. These constants were defined in the class `ios` in old implementations. Because `ios` is a type derived from `ios_base` and its use involves less typing, the use often looks like this:

```cpp
std::ios::eofbit
```

These flags are maintained by the class `basic_ios` and are thus present in all objects of type `basic_istream` or `basic_ostream`. However, the stream buffers don’t have state flags. One stream buffer can be shared by multiple stream objects, so the flags only represent the state of the stream as found in the last operation. Even this is only the case if `goodbit` was set prior to this operation. Otherwise the flags may have been set by some earlier operation.

### 13.4.2 Member Functions Accessing the State of Streams

The current state of the flags can be determined by the member functions, as presented in Table 13.4.

<table>
<thead>
<tr>
<th>Member Function</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>good()</code></td>
<td>Returns <code>true</code> if the stream is OK (<code>goodbit</code> is &quot;set&quot;)</td>
</tr>
<tr>
<td><code>eof()</code></td>
<td>Returns <code>true</code> if end-of-file was hit (<code>eofbit</code> is set)</td>
</tr>
<tr>
<td><code>fail()</code></td>
<td>Returns <code>true</code> if an error has occurred (<code>failbit</code> or <code>badbit</code> is set)</td>
</tr>
<tr>
<td><code>bad()</code></td>
<td>Returns <code>true</code> if a fatal error has occurred (<code>badbit</code> is set)</td>
</tr>
<tr>
<td><code>rdstate()</code></td>
<td>Returns the currently set flags</td>
</tr>
<tr>
<td><code>clear()</code></td>
<td>Clears all flags</td>
</tr>
<tr>
<td><code>clear(state)</code></td>
<td>Clears all and sets <code>state</code> flags</td>
</tr>
<tr>
<td><code>setstate(state)</code></td>
<td>Sets additional <code>state</code> flags</td>
</tr>
</tbody>
</table>

The first four member functions in Table 13.4 determine certain states and return a Boolean value. Note that `fail()` returns whether `failbit` or `badbit` is set. Although this is done mainly for historical reasons, it also has the advantage that one test suffices to determine whether an error has occurred.

In addition, the state of the flags can be determined and modified with the more general member functions. When `clear()` is called without parameters, all error flags (including `eofbit`) are cleared (this is the origin of the name `clear`):

```cpp
// clear all error flags (including eofbit):
strm.clear();
```
If a parameter is given to `clear()`, the state of the stream is adjusted to be the state given by the parameter; that is, the flags set in the parameter are set for the stream, while the other flags are cleared. The only exception is that the `badbit` is always set if there is no stream buffer (this is the case if `rdbuf()` == 0; see Section 13.10.2, for details).

The following example checks whether `failbit` is set and clears it if necessary:

```cpp
// check whether failbit is set
if (strm.rdstate() & std::ios::failbit) {
    std::cout << "failbit was set" << std::endl;

    // clear only failbit
    strm.clear (strm.rdstate() & ~std::ios::failbit);
}
```

This example uses the bit operators `&` and `~`: Operator `~` returns the bitwise complement of its argument. Thus, the expression

```
~ios::failbit
```

returns a temporary value that has all bits except `failbit` set. Operator `&` returns a bitwise "and" of its operands. Only the bits set in both operands remain set. Applying bitwise "and" to all currently set flags (`rdstate()` and to all bits except `failbit` retains the value of all other bits while `failbit` is cleared.

Streams can be configured to throw exceptions if certain flags are set with `clear()` or `setstate()` (see Section 13.4.4). Such streams always throw an exception if the corresponding flag is set at the end of the method used to manipulate the flags.

Note that you always have to clear error bits explicitly. In C it was possible to read characters after a format error. For example, if `scanf()` failed to read an integer, you could still read the remaining characters. Thus, the read operation failed, but the input stream was still in a good state. This is different in C++. If `failbit` is set, each following stream operation is a no-op until `failbit` is cleared explicitly.

In general, it has to be mentioned that the set bits reflect only what happened sometime in the past: If a bit is set after some operation this does not necessarily mean that this operation caused the flag to be set. Instead, the flag might have been set before the operation. Thus, `goodbit` should be set (if it is not known to be set) before an operation is executed if the flags arc then used to tell you what went wrong. Also, after clearing the flags the operations may yield different results. For example, even if `eofbit` was set by an operation, this does not mean that after clearing `eofbit` (and any other bits set) the operation will set `eofbit` again. This can be the case, for example, if the accessed file grew between the two calls.

### 13.4.3 Stream State and Boolean Conditions

Two functions are defined for the use of streams in Boolean expressions (Table 13.5).

<table>
<thead>
<tr>
<th>Member Function</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void*()</code></td>
<td>Returns whether the stream has not run into an error (corresponds to <code>!fail()</code>).</td>
</tr>
<tr>
<td><code>operator !()</code></td>
<td>Returns whether the stream has run into an error (corresponds to <code>fail()</code>).</td>
</tr>
</tbody>
</table>

With operator `void*()`, streams can be tested in control structures in a short and idiomatic way for their current state.
For the Boolean condition in a control structure, the type does not need a direct conversion to bool. Instead, a unique conversion to an integral type (such as int or char) or to a pointer type is sufficient. The conversion to void* is often used to read objects and test for success in the same expression:

```cpp
if (std::cin >> x) {
    // reading x was successful
    ...
}
```

As discussed earlier, the expression

```cpp
std::cin >> x
```

returns cin. So after x is read, the statement is

```cpp
if (std::cin) {
    ...
}
```

Because cin is being used in the context of a condition, its operator void* is called, which returns whether the stream has run into an error. A typical application of this technique is a loop that reads and processes objects:

```cpp
// as long as obj can be read
while (std::cin >> obj) {
    // process obj (in this case, simply output it)
    std::cout << obj << std::endl;
}
```

This is C's classic filter framework for C++ objects. The loop is terminated if the failbit or badbit is set. This happens when an error occurred or at end-of-file (the attempt to read at end-of-file results in setting eofbit and failbit; see page 598). By default, operator >> skips leading whitespaces. This is normally exactly what is desired. However, if obj is of type char, whitespace is normally considered to be significant. In this case you can use the put() and get() member functions of streams (see page 611) or, even better, an istreambuf_iterator (see page 667) to implement an I/O filter.

With operator !, the inverse test can be performed. The operator is defined to return whether a stream has run into an error; that is, it returns true if failbit or badbit is set. It can be used like this:

```cpp
if (! std::cin) {
    // the stream cin is not OK
    ...
}
```
Like the implicit conversion to a Boolean value, this operator is often used to test for success in the same expression in which an object was read:

```cpp
if (! (std::cin >> x)) {
    // the read failed
    ...
}
```

Here, the expression

```cpp
std::cin >> x
```

returns `cin`, to which operator `!` is applied. The expression after `!` must be placed within parentheses. This is due to the operator precedence rules: without the parentheses, operator `!` would be evaluated first. In other words, the expression

```cpp
! std::cin >> x
```

is equivalent to the expression

```cpp
(!std::cin) >> x
```

This is probably not what is intended. Although these operators are very convenient in Boolean expressions, one oddity has to be noted: Double "negation" does not yield the original object:

- `cin` is a stream object of class `istream`.
- `!! cin` is a Boolean value describing the state of `cin`.

As with other features of C++, it can be argued whether the use of the conversions to a Boolean value is good style. The use of member functions such as `fail()` normally yields a more readable program:

```cpp
std::cin >> x;
if (std::cin.fail()) {
    ...
}
```

### 13.4.4 Stream State and Exceptions

Exception handling was introduced to C++ for the handling of errors and exceptions (see page 15). However, this was done after streams were already in wide use. To stay backward compatible, by default, streams throw no exceptions. However, for the standardized streams, it is possible to define, for every state flag, whether setting that flag will trigger an exception. This definition is done by the `exceptions()` member function (Table 13.6).

| Table 13.6. Stream Member Functions for Exceptions |
**Member Function** | **Meaning**
--- | ---
exceptions(flags) | Sets flags that trigger exceptions
exceptions() | Returns the flags that trigger exceptions

**Calling exceptions() without an argument yields the current flags for which exceptions are triggered.** No exceptions are thrown if the function returns goodbit. This is the default, to maintain backward compatibility. When exceptions() is called with an argument, exceptions are thrown as soon as the corresponding state flags are set. If a state flag is already set when exceptions() is called with an argument, an exceptions is thrown if the corresponding flag is set in the argument.

The following example configures the stream so that, for all flags, an exception is thrown:

```cpp
// throw exceptions for all "errors"
strm.exceptions (std::ios::eofbit | std::ios::failbit | std::ios::badbit);
```

If 0 or goodbit is passed as an argument, no exceptions are generated:

```cpp
// do not generate exceptions
strm.exceptions (std::ios::goodbit);
```

Exceptions are thrown when the corresponding state flags are set after calling clear() or setstate(). An exception is even thrown if the flag was already set and not cleared:

```cpp
// this call throws an exception if failbit is set on entry
strm.exceptions (std::ios::failbit);
...
// throw an exception (even if failbit was already set)
strm.setstate (std::ios::failbit);
```

The exceptions thrown are objects of the class std::ios_base::failure, which is derived from class exception (see Section 3.3.1):

```cpp
namespace std {
    class ios_base::failure : public exception {
    public:
        // constructor
        explicit failure (const string& msg);
        // destructor
        virtual ~failure();
        // return information about the exception
        virtual const char* what() const;
    };
}
```

Unfortunately, the standard does not require that the exception object includes any information about the erroneous stream or the kind of error. The only portable method that can be used to get information about the error is the error message returned from what(). But note, only calling what() is portable; the string it returns is not. If additional information is necessary, the programmer must arrange to get the required information.
This behavior shows that exception handling is intended to be used more for unexpected situations. It is called exception handling rather than error handling. Expected errors, such as format errors during input from the user, are considered to be "normal" and are usually better handled using the state flags.

The major area in which stream exceptions are useful is reading preformatted data such as automatically written files. But even then, problems arise if exception handling is used. For example, if it is desired to read data until end-of-file, you can't get exceptions for errors without getting an exception for end-of-file. This is because the detection of end-of-file also sets the failbit (meaning that reading an object was not successful). To distinguish end-of-file from an input error you have to check the state of the stream.

The next example demonstrates how this might look. It shows a function that reads floating-point values from a stream until end-of-file is reached. Then it returns the sum of the floating-point values read:

```cpp
// io/sum1a.cpp
#include <iostream>

namespace MyLib {
    double readAndProcessSum (std::istream& strm)
    {
        using std::ios;
        double value, sum;

        // save current state of exception flags
        ios::iostate oldExceptions = strm.exceptions();
        /*let failbit and badbit throw exceptions
         *-NOTE: failbit is also set at end-of-file
         */
        strm.exceptions (ios::failbit | ios::badbit);

        try {
            /*while stream is OK
             *- read value and add it to sum
             */
            sum = 0;
            while (strm >> value) {
                sum += value;
            }
        }
        catch (...) {
            /*if exception not caused by end-of-file
             *- restore old state of exception flags
             *- rethrow exception
             */
            if (!strm.eof()) { // restore exception flags
                strm.exceptions (oldExceptions); // restore
                throw; // rethrow
            }
        }

        // restore old state of exception flags
        strm.exceptions (oldExceptions);
    }
}
```
First the function stores the set stream exceptions in `oldExceptions` to restore them later. Then the stream is configured to throw an exception on certain conditions. In a loop, all values are read and added as long as the stream is OK. If end-of-file is reached, the stream is no longer OK, and a corresponding exception is thrown even though no exception is thrown for setting `eofbit`. This happens because end-of-file is detected on an unsuccessful attempt to read more data, which also sets the `failbit`. To avoid the behavior that end-of-file throws an exception, the exception is caught locally to check the state of the stream by using `eof()`. The exception is propagated only if `eof()` yields `false`.

Note that restoring the original exception flags may cause exceptions, `exceptions()` throws an exception if a corresponding flag is set in the stream already. Thus, if the state did throw exceptions for `eofbit`, `failbit`, or `badbit` on function entry, these exceptions are propagated to the caller.

This function can be called in the simplest case from the following main function:

```cpp
#include <iostream>
#include <cstdlib>

double MyLib::readAndProcessSum (std::istream&);

int main()
{
    using namespace std;
    double sum;

    try {
        sum = MyLib::readAndProcessSum(cin);
    }
    catch (const ios::failure& error) {
        cerr << "I/O exception: " << error.what() << endl;
        return EXIT_FAILURE;
    }
    catch (const exception& error) {
        cerr << "standard exception: " << error.what() << endl;
        return EXIT_FAILURE;
    }
    catch (...) {
        cerr << "unknown exception" << endl;
        return EXIT_FAILURE;
    }

    // print sum
    cout << "sum: " << sum << endl;
}
```

The question arises whether this is worth the effort. It is also possible to work with streams not throwing an exception. In this case, an exception is thrown if an error is detected. This has the additional advantage that user-defined error messages and error classes can be used:
// io/sum2a.cpp
#include <istream>

namespace MyLib {
    double readAndProcessSum (std::istream& strm) {
        double value, sum;
        /*while stream is OK
         *- read value and add it to sum
         */
        sum = 0;
        while (strm >> value) {
            sum += value;
        }
        if (!strm.eof()) {
            throw std::ios::failure
                ("input error in readAndProcessSum()");
        }
        // return sum
        return sum;
    }
}

This looks somewhat simpler, doesn't it? This version of the function needs the header <string> because the constructor of the class failure takes a reference to a constant string as an argument. To construct an object of this type, the definition is needed but the header <istream> is only required to provide a declaration.

13.5 Standard Input/Output Functions

Instead of using the standard operators for streams (operator << and operator >>), you can use several other member functions for reading and writing, which are presented in this section. The functions in this section read or write "unformatted" data (unlike operators >> or <<, which read or write "formatted" data). When reading, they never skip leading whitespaces (unlike the operators that are, by default, configured to skip leading whitespace). Also, they handle exceptions differently than the formatted I/O functions: If an exception is thrown, either from a called function or as a result of setting a state flag (see Section 13.4.4), the badbit flag is set. The exception is then rethrown if the exception mask has badbit set. However, the unformatted functions create a sentry object like the formatted functions do (see Section 13.12.4). These functions use type streamsize to specify counts, which is defined in <ios>:

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.

### 13.5.1 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 some other instantiation of the template class `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.

The C++ standard library provides several member functions to read character sequences. Table 13.7 compares their abilities.

<table>
<thead>
<tr>
<th>Member Function</th>
<th>Reads Until</th>
<th>Number of Characters</th>
<th>Appends Termin.</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get (s, num)</code></td>
<td>Excluding newline or end-of-file</td>
<td>Up to num-1</td>
<td>Yes</td>
<td><code>istream</code></td>
</tr>
<tr>
<td><code>get(s, num, t)</code></td>
<td>Excluding t or end-of-file</td>
<td>Up to num-1</td>
<td>Yes</td>
<td><code>istream</code></td>
</tr>
<tr>
<td><code>getline(s, num)</code></td>
<td>Including newline or end-of-file</td>
<td>Up to num-1</td>
<td>Yes</td>
<td><code>istream</code></td>
</tr>
<tr>
<td><code>getline(s, num, t)</code></td>
<td>Including t or end-of-file</td>
<td>Up to num-1</td>
<td>Yes</td>
<td><code>istream</code></td>
</tr>
<tr>
<td><code>read(s, num)</code></td>
<td>end-of-file</td>
<td>num</td>
<td>No</td>
<td><code>istream</code></td>
</tr>
<tr>
<td><code>readsome(s, num)</code></td>
<td>end-of-file</td>
<td>Up to num</td>
<td>No</td>
<td><code>count</code></td>
</tr>
</tbody>
</table>

### `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` characters in 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 page 691). In general, `widen('\n')` is used (see page 626).
- 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 (\texttt{delim}) is not read.
The read character sequence is terminated by a string termination character.
The caller must ensure that \texttt{str} is large enough for \texttt{count} characters.

\begin{verbatim}
\texttt{istream& istream::getline (char* str, streamsize count)}
\texttt{istream& istream::getline (char* str, streamsize count, char delim)}
\end{verbatim}

- Both forms are identical to their previous counterparts of \texttt{get()} except that they terminate the reading \texttt{including} but not before the newline character or \texttt{delim} respectively.
- Thus, the newline character or \texttt{delim} is read if it occurs within \texttt{count} characters, but it is not stored in \texttt{str}.

\begin{verbatim}
\texttt{istream& istream::read (char* str, streamsize count)}
\end{verbatim}

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

\begin{verbatim}
\texttt{streamsize istream::readsome (char* str, streamsize count)}
\end{verbatim}

- Reads up to \texttt{count} characters in the string \texttt{str}.
- Returns the number of characters read.
- The string in \texttt{str} is \texttt{not} terminated automatically with the string termination character.
- The caller must ensure that \texttt{str} has sufficient space to store \texttt{count} characters.
- In contrast to \texttt{read()}, \texttt{readsome()} reads all available characters of the stream buffer (using the \texttt{in_avail()} member function of the buffer). 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 \texttt{eofbit} nor \texttt{failbit}.

\begin{verbatim}
\texttt{streamsize istream::gcount () const}
\end{verbatim}

- Returns the number of characters read by the last \textit{unformatted} read operation.

\begin{verbatim}
\texttt{istream& istream::ignore ()}
\texttt{istream& istream::ignore (streamsize count)}
\texttt{istream& istream::ignore (streamsize count, int delim)}
\end{verbatim}

- All forms extract and discard characters.
- The first form ignores one character.
- The second form ignores up to \texttt{count} characters.
- The third form ignores up to \texttt{count} characters until \texttt{delim} is extracted and discarded.
- If \texttt{count} is \texttt{numeric_limits<int>::max()} (the largest int, see Section 4.3), all characters are discarded until either \texttt{delim} or end-of-file is reached.
- All forms return the stream.
- Examples:
  - The following call discards the rest of the line:
The following call discards the complete remainder of cin:

```cpp
cin.ignore(numeric_limits<int>::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 `ungetc()` 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 13.4.4).
- The maximum number of characters that can be put back with these functions is implementation defined. 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 `>>`. This is because 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 page 618), 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 13.12.4, for more information on `sentry` objects). Section 13.13, explains the stream buffer interface in detail. Another alternative is to use the template class `istreambuf_iterator`, which provides an iterator interface to the stream buffer (see Section 13.13.2).

Two other functions for manipulating the read position are `tellg()` and `seekg()`. These are relevant mainly in conjunction with files, so their descriptions are deferred until Section 13.9.2.

### 13.5.2 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 some other instantiation of the template class `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 string termination character does not terminate the write and will be written.
- The caller must ensure that str really 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(). These functions are relevant mainly in conjunction with files, so their descriptions are deferred until Section 13.9.2. Like the input functions, it may also be reasonable to use the stream buffer directly or to use the template class ostreambuf_iterator for unformatted writing. There is actually no point in using the unformatted output functions, except that these functions might handle some locks in multithreaded environments using sentry objects. See Section 13.14.3, for details.

### 13.5.3 Example Uses

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

```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);
}
}
```

In C, it is necessary to use an object of type int for character processing to tell whether end of file was reached. In this version, the read character is accessed simply by using the dereference operator.

To perform better, you can operate directly on stream buffers. See page 667 for a version of this example that uses stream buffer iterators for I/O and page 683 for a version that copies the whole input in one statement.

### 13.6 Manipulators

Manipulators for streams were introduced in Section 13.1.5. They are objects that modify a stream when applied with the standard I/O operators. This does not necessarily mean that
something is read or written. The basic manipulators defined in `<istream>` or `<ostream>` are presented in Table 13.8.

<table>
<thead>
<tr>
<th>Manipulator</th>
<th>Class</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>flush</code></td>
<td>basic_ostream</td>
<td>Flushes the output buffer to its device</td>
</tr>
<tr>
<td><code>endl</code></td>
<td>basic_ostream</td>
<td>Inserts a newline character into the buffer and flushes the output buffer to its device</td>
</tr>
<tr>
<td><code>ends</code></td>
<td>basic_ostream</td>
<td>Inserts a string termination character into the buffer</td>
</tr>
<tr>
<td><code>ws</code></td>
<td>basic_istream</td>
<td>Reads and ignores whitespaces</td>
</tr>
</tbody>
</table>

Table 13.8. Manipulators Defined in `<istream>` or `<ostream>`

There are additional manipulators, for example, to change I/O formats. These manipulators are introduced in Section 13.7, about formatting.

### 13.6.1 How Manipulators Work

Manipulators are implemented using a very simple trick. This trick not only enables the convenient manipulation of streams, it also demonstrates the power provided by function overloading. Manipulators are nothing more than functions that are passed to the I/O operators as arguments. The functions are then called by the operator. For example, the output operator for class `ostream` is basically overloaded like this:

```c
ostream& ostream::operator << ( ostream& (*op) (ostream&))
{
    // call the function passed as parameter with this stream as the argument
    return (*op) (*this);
}
```

The real implementation looks a little bit more complicated because it has to construct a `sentry` object and because it is actually a function template.

```c
std::ostream& std::endl (std::ostream& strm)
{
    // write newline
    strm.put('\n');

    // flush the output buffer
    strm.flush();

    // return strm to allow chaining
    return strm;
}
```
You can use this manipulator in an expression such as the following:

```cpp
std::cout << std::endl
```

Here, operator `<<` is called for stream `cout` with the `endl()` function as the second operand. The implementation of operator `<<` transforms this call into a call of the passed function with the stream as the argument:

```cpp
std::endl(std::cout)
```

The same effect as "writing" the manipulator can also be achieved by calling this expression directly. There is actually an advantage in using the function notation: It is not necessary to provide the namespace:

```cpp
endl(std::cout)
```

This is because functions are looked up in the namespaces where their arguments are defined if they are not found otherwise (see page 17). Because the stream classes are actually template classes parameterized with the character type, the real implementation of `endl()` looks like this:

```cpp
template<class charT, class traits>
std::basic_ostream<charT,traits>&
std::endl (std::basic_ostream<charT,traits>& strm)
{
    strm.put(strm.widen(\'\n\'));
    strm.flush();
    return strm;
}
```

The member function `widen()` is used to convert the newline character into the character set currently used by the stream. See Section 13.8, for more details.

The C++ standard library also contains manipulators with arguments. How these manipulators work exactly is implementation dependent, and there is no standard way to implement user-defined manipulators with arguments.

The standard manipulators with arguments are defined in the header file `<iomanip>`, which must be included to work with the standard manipulators taking arguments:

```cpp
#include <iomanip>
```

The standard manipulators taking arguments are all concerned with details of formatting, so they are described when formatting options are described.

### 13.6.2 User-Defined Manipulators

You can define your own manipulators. All you need to do is to write a function such as `endl()`. For example, the following function defines a manipulator that ignores all characters until end-of-line:

```cpp
// io/ignore.hpp
```
```cpp
#include <istream>
#include <limits>

template <class charT, class traits>
inline
std::basic_istream<charT,traits>&
ignoreLine (std::basic_istream<charT,traits>& strm)
{
    // skip until end-of-line
    strm.ignore(std::numeric_limits<int>::max(),strm.widen('\n'));

    // return stream for concatenation
    return strm;
}
```

The manipulator simply delegates the work to the function `ignore()`, which in this case discards all characters until end-of-line (`ignore()` was introduced on page 609). The application of the manipulator is very simple:

```cpp
// ignore the rest of the line
std::cin >> ignoreLine;
```

Applying this manipulator multiple times enables you to ignore multiple lines:

```cpp
// ignore two lines
std::cin >> ignoreLine >> ignoreLine;
```

This works because a call to the function `ignore (max, c)` ignores all characters until the `c` is found in the input stream (or `max` characters are read or the end of the stream was reached). However, this character is discarded, too, before the function returns.

## 13.7 Formatting

Two concepts influence the definition of I/O formats: Most obviously, there are format flags that define, for example, numeric precision, the fill character, or the numeric base. Apart from this, there exists the possibility of adjusting the formats to meet special national conventions. This section introduces the format flags. Section 13.8, and Chapter 14 describe the aspects of internationalized formatting.

### 13.7.1 Format Flags

The class `ios_base` has several members that are used for the definition of various I/O formats. For example, it has members that store the minimum field width, the precision of floating-point numbers, or the fill character. A member of type `ios::fmtflags` stores configuration flags defining, for example, whether positive numbers should be preceded by a positive sign or whether Boolean values should be printed numerically or as words. Some of the format flags form groups. For example, the flags for octal, decimal, and hexadecimal formats of integer numbers form a group. Special masks are defined to make dealing with such groups easier.

### Table 13.9. Member Function to Access Format Flags
Several member functions can be used to handle all of the format definitions of a stream. These
are presented in Table 13.9. The functions setf() and unsetf() set or clear, respectively,
one or more flags. You can manipulate multiple flags at once by combining them using the "binary
or" operator; that is, operator |. The function setf() can take a mask as the second argument
to clear all flags in a group before setting the flags of the first argument, which are also limited to
a group. This does not happen with the version of setf() that takes only one argument. For
example:

```
// set flags showpos and uppercase
std::cout.setf (std::ios::showpos | std::ios::uppercase);

// set only the flag hex in the group basefield
std::cout.setf (std::ios::hex, std::ios::basefield);

// clear the flag uppercase
std::cout.unsetf (std::ios::uppercase);
```

Using flags() you can manipulate all format flags at once. Calling flags() without an
argument returns the current format flags. Calling flags() with an argument takes this argument
as the new state of all format flags and returns the old state. Thus, flags() with an argument
clears all flags and sets the flags that were passed. Using flags() is useful, for example, for
saving the current state of the flags to restore the original state later. The following statements
demonstrate an example:

```
using std::ios, std::cout;

// save actual format flags
ios::fmtflags oldFlags = cout.flags();

// do some changes
cout.setf(ios::showpos | ios::showbase | ios::uppercase);
cout.setf(ios::internal, ios::adjustfield);
cout << std::hex << x << std::endl;

// restore saved format flag
cout.flags(oldFlags);
```
By using `copyfmt()` you can copy all the format information from one stream to another. See page 653 for an example.

You can also use manipulators to set and clear format flags. These are presented in Table 13.10.

<table>
<thead>
<tr>
<th>Manipulator</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>setiosflags(flags)</code></td>
<td>Sets flags as format flags (calls <code>setf(flags)</code> for the stream)</td>
</tr>
<tr>
<td><code>resetiosflags(mask)</code></td>
<td>Clears all flags of the group identified by <code>mask</code> (calls <code>setf(0,mask)</code> for the stream)</td>
</tr>
</tbody>
</table>

The manipulators `setiosflags()` and `resetiosflags()` provide the possibility of setting or clearing, respectively, one or more flags in a write or read statement with operator `<<` or `>>` respectively. To use one of these manipulators, you must include the header file `<iomanip>`.

For example:

```cpp
#include <iostream>
#include <iomanip>
...
std::cout << resetiosflags(std::ios::adjustfield) // clear adjustm. flags
          << setiosflags(std::ios::left); // left-adjust values
```

Some flag manipulations are performed by specialized manipulators. These manipulators are used often because they are more convenient and more readable. They are discussed in the following subsections.

### 13.7.2 Input/Output Format of Boolean Values

The `boolalpha` flag defines the format used to read or to write Boolean values. It defines whether a numeric or a textual representation is used for Boolean values (Table 13.11).

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolalpha</td>
<td>It set, specifies the use of textual representation; if not set, specifies the use of numeric representation</td>
</tr>
</tbody>
</table>

If the flag is not set (this is the default), Boolean values are represented using numeric strings. In this case, the value 0 is always used for `false` and the value 1 is always used for `true`. When reading a Boolean value as a numeric string it is considered to be an error (setting `failbit` for the stream) if the value is different from 0 or 1.

If the flag is set, Boolean values are written using a textual representation. When a Boolean value is read, the string has to match the textual representation of either `true` or `false`. The stream's locale object is used to + the strings used to represent `true` and `false` (see page 626 and page 698). The standard "C" locale object uses the strings "true" and "false" as representations of the Boolean values.

Special manipulators are defined for the convenient manipulation of this flag (Table 13.12).

<table>
<thead>
<tr>
<th>Manipulator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolalpha</td>
<td>Forces textual representation (sets the flag <code>ios::boolalpha</code>)</td>
</tr>
<tr>
<td>noboolalpha</td>
<td>Forces numeric representation (clears the flag <code>ios::boolalpha</code>)</td>
</tr>
</tbody>
</table>

For example, the following statements print `b` first in numeric representation and then in textual representation:
bool b;
...
cout << noboolalpha << b << " == " << boolalpha << b << endl;

### 13.7.3 Field Width, Fill Character, and Adjustment

Two member functions are used to define the field width and the fill character: `width()` and `fill()` (Table 13.13).

<table>
<thead>
<tr>
<th>Member Function</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>width()</td>
<td>Returns the actual field width</td>
</tr>
<tr>
<td>width(val)</td>
<td>Sets the field width to <code>val</code> and returns the previous field width</td>
</tr>
<tr>
<td>fill()</td>
<td>Returns the actual fill character</td>
</tr>
<tr>
<td>fill(c)</td>
<td>Defines <code>c</code> as the fill character and returns the previous fill character</td>
</tr>
</tbody>
</table>

#### Using Field Width, Fill Character, and Adjustment for Output

For the output `width()` defines a minimum field. This definition applies only to the next formatted field written. Calling `width()` without arguments returns the current field width. Calling `width()` with an integral argument changes the width and returns the former value. The default value for the minimum field width is 0, which means that the field may have any length. This is also the value to which the field width is set after a value was written. Note that the field width is never used to truncate output. Thus, you can't specify a maximum field width. Instead, you have to program it. For example, you could write to a string and output only a certain number of characters.

`fill()` defines the fill character that is used to fill the difference between the formatted representation of a value and the minimum field width. The default fill character is a space. To adjust values within a field, three flags are defined, as shown in Table 13.14. These flags are defined in the class `ios_base` together with the corresponding mask.

<table>
<thead>
<tr>
<th>Mask</th>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>adjustfield</td>
<td>left</td>
<td>Left-adjusts the value</td>
</tr>
<tr>
<td></td>
<td>right</td>
<td>Right-adjusts the value</td>
</tr>
<tr>
<td></td>
<td>internal</td>
<td>Left-adjusts the sign and right-adjusts the value</td>
</tr>
<tr>
<td></td>
<td>None</td>
<td>Right-adjusts the value (the default)</td>
</tr>
</tbody>
</table>

After any formatted I/O operation is performed, the default field width is restored. The values of the fill character and the adjustment remain unchanged until they are modified explicitly. `Table 13.15` presents the effect of the functions and the flags used for different values. The underscore is used as the fill character.

<table>
<thead>
<tr>
<th>Adjustment</th>
<th>width()</th>
<th>-42</th>
<th>0.12</th>
<th>&quot;Q&quot;</th>
<th>'Q'</th>
</tr>
</thead>
<tbody>
<tr>
<td>left</td>
<td>6</td>
<td>-42_</td>
<td>0.12_</td>
<td>Q_</td>
<td>Q_</td>
</tr>
<tr>
<td>right</td>
<td>6</td>
<td>-_42</td>
<td>0.12_</td>
<td>_Q</td>
<td>_Q</td>
</tr>
<tr>
<td>internal</td>
<td>6</td>
<td>-_42</td>
<td>0.12_</td>
<td>_Q</td>
<td>_Q</td>
</tr>
</tbody>
</table>

Note that the adjustment for single characters has changed during the standardization. Before standardization, the field width was ignored if single characters were written. It was used for the
next formatted output that was not a single character. This bug was fixed. However, for programs
that used this bug as a feature, the fix breaks backward compatibility.
Several manipulators are defined to handle the field width, the fill character, and the adjustment
(Table 13.16).

<table>
<thead>
<tr>
<th>Manipulator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>setw(val)</td>
<td>Sets the field width for input and output to val (corresponds to width())</td>
</tr>
<tr>
<td>setfill(c)</td>
<td>Defines c as the fill character (corresponds to fill())</td>
</tr>
<tr>
<td>left</td>
<td>Left-adjusts the value</td>
</tr>
<tr>
<td>right</td>
<td>Right-adjusts the value</td>
</tr>
<tr>
<td>internal</td>
<td>Left-adjusts the sign and right-adjusts the value</td>
</tr>
</tbody>
</table>

The manipulators `setw()` and `setfill()` use an argument, so you must include the header file
`<iomanip>` to use them. For example, the statements

```cpp
#include <iostream>
#include <iomanip>
...
std::cout << std::setw(8) << std::setfill('_') << -3.14 << ' ' << 42 << std::endl;
std::cout << std::setw(8) << "sum: " << std::setw(8) << 42 << std::endl;
```

produce this output:

```
...-3.14 42
...sum: ...42
```

### Using Field Width for Input

You can use the field width also to define the maximum number of characters read when
character sequences of type `char*` are read. If the value of `width()` is not 0, then at most
`width()-1` characters are read.

Because of the fact that ordinary C-strings can't grow while values are read, `width()` or `setw()`
should always be used when reading them with operator `>>`. For example:

```cpp
char buffer [81];

// read, at most, 80 characters:
cin >> setw (sizeof (buffer)) >> buffer;
```

This reads, at most, 80 characters, although `sizeof (buffer)` is 81 because one character is
used for the string termination character (which is appended automatically). Note that the
following code is a common error:

```cpp
char* s;
cin >> setw (sizeof (s)) >> s;  //RUNTIME ERROR
```
This is because `s` is only declared as a pointer without any storage for characters, and 
`sizeof(s)` is the size of the pointer instead of the size of the storage to which it points. This is a 
typical example of the problems you encounter if you use C-strings. By using strings, you won’t 
run into these problems:

```cpp
string buffer;
cin >> buffer;  //OK
```

### 13.7.4 Positive Sign and Uppercase Letters

Two format flags are defined to influence the general appearance of numeric values: `showpos` 
and `uppercase` (Table 13.17).

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>showpos</td>
<td>Writes a positive sign on positive numbers</td>
</tr>
<tr>
<td>uppercase</td>
<td>Uses uppercase letters</td>
</tr>
</tbody>
</table>

`ios::showpos` dictates that a positive sign for positive numeric values be written. If the flag is 
not set, only negative values are written with a sign, `ios::uppercase` dictates that letters in 
numeric values be written using uppercase letters. This flag applies to integers using 
hexadecimal format and to floating-point numbers using scientific notation. By default, letters are 
written as lowercase and no positive sign is written. For example, the statements

```cpp
std::cout << 12345678.9 << std::endl;
std::cout.setf (std::ios::showpos | std::ios::uppercase);
std::cout << 12345678.9 << std::endl;
```

produce this output:

```
1.23457e+07
+1.23457E+07
```

Both flags can be set or cleared using the manipulators presented in Table 13.18.

<table>
<thead>
<tr>
<th>Manipulator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>showpos</td>
<td>Forces to write a positive sign on positive numbers (sets the flag <code>ios::showpos</code>)</td>
</tr>
<tr>
<td>noshowpos</td>
<td>Forces not to write a positive sign (clears the flag <code>ios::showpos</code>)</td>
</tr>
<tr>
<td>uppercase</td>
<td>Forces uppercase letters (sets the flag <code>ios::upper case</code>)</td>
</tr>
<tr>
<td>nouppercase</td>
<td>Forces lowercase letters (clears the flag <code>ios::upper case</code>)</td>
</tr>
</tbody>
</table>

### 13.7.5 Numeric Base

A group of three flags defines which base is used for I/O of integer values. The flags are defined 
in the class `ios_base` with the corresponding mask (Table 13.19).

<table>
<thead>
<tr>
<th>Mask</th>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The C++ Standard Library

The C++ Standard Library

<table>
<thead>
<tr>
<th>basefield</th>
<th>oct</th>
<th>Writes and reads octal</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>dec</td>
<td>Writes and reads decimal (default)</td>
</tr>
<tr>
<td></td>
<td>hex</td>
<td>Writes and reads hexadecimal</td>
</tr>
<tr>
<td></td>
<td>None</td>
<td>Writes decimal and reads according to the leading characters of the integral value</td>
</tr>
</tbody>
</table>

A change in base applies to the processing of all integer numbers until the flags are reset. By default, decimal format is used. There is no support for binary notation. However, you can read and write integral values in binary by using class `bitset`. See Section 10.4.1, for details.

If none of the base flags is set, output uses a decimal base. If more than one flag is set, decimal is used as the base.

The flags for the numeric base also affect input. If one of the flags for the numeric base is set, all numbers are read using this base. If no flag for the base is set when numbers are read the base is determined by the leading characters: A number starting with `0x` or `0X` is read as a hexadecimal number. A number starting with `0` is read as an octal number. In all other cases, the number is read as a decimal value.

There are basically two ways to switch these flags:

1. Clear one flag and set another:

2. 

3. `std::cout.unsetf(std::ios::dec);`

4. `std::cout.setf(std::ios::hex);`

5. 

6. Set one flag and clear all other flags in the group automatically:

7. `std::cout.setf(std::ios::hex, std::ios::basefield);`

8. 

In addition, manipulators are defined that make the handling of these flags significantly simpler (Table 13.20).

<table>
<thead>
<tr>
<th>Table 13.20. Manipulators Defining the Base of Integral Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manipulator</td>
</tr>
<tr>
<td>------------</td>
</tr>
<tr>
<td>oct</td>
</tr>
<tr>
<td>dec</td>
</tr>
<tr>
<td>hex</td>
</tr>
</tbody>
</table>

For example, the following statements write `x` and `y` in hexadecimal, and `z` in decimal:

```cpp
int x, y, z;
...
std::cout << std::ios::hex << x << std::endl;
std::cout << y << ' ' << std::ios::dec << z << std::endl;
```

An additional flag, `showbase`, lets you write numbers according to the usual C/C++ convention for indicating numeric bases of literal values (Table 13.21).

<table>
<thead>
<tr>
<th>Table 13.21. Flags to Indicate the Numeric Base</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flag</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>showbase</td>
</tr>
</tbody>
</table>

If `ios::showbase` is set, octal numbers are preceded by a `0` and hexadecimal numbers are preceded by `0x` (or, if `ios::uppercase` is set, by `0X`). For example, the statements
```cpp
std::cout << 127 << ' ' << 255 << std::endl;
std::cout << std::hex << 127 << ' ' << 255 << std::endl;
std::cout.setf(std::ios::showbase);
std::cout << 127 << ' ' << 255 << std::endl;
std::cout.setf(std::ios::uppercase);
std::cout << 127 << ' ' << 255 << std::endl;
```

produce this output:

```
127 255
7f ff
0x7f 0xff
0X7F 0XFF
```

`ios::showbase` can also be manipulated using the manipulators presented in Table 13.22.

<table>
<thead>
<tr>
<th>Manipulator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>showbase</td>
<td>Indicates numeric base (sets the flag <code>ios::showbase</code>)</td>
</tr>
<tr>
<td>noshowbase</td>
<td>Does not indicate numeric base (clears the flag <code>ios::showbase</code>)</td>
</tr>
</tbody>
</table>

### 13.7.6 Floating-Point Notation

Several flags and members control the output of floating-point values. The flags, presented in Table 13.23, define whether output is written using decimal or scientific notation. These flags are defined in the class `ios_base` together with the corresponding mask. If `ios::fixed` is set, floating-point values are printed using decimal notation. If `ios::scientific` is set scientific (that is, exponential) notation is used.

<table>
<thead>
<tr>
<th>Mask</th>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>floatfield</td>
<td>fixed</td>
<td>Uses decimal notation</td>
</tr>
<tr>
<td></td>
<td>scientific</td>
<td>Uses scientific notation</td>
</tr>
<tr>
<td>None</td>
<td></td>
<td>Uses the “best” of these two notations (default)</td>
</tr>
</tbody>
</table>

To define the precision, the member function `precision()` is provided (see Table 13.24).

<table>
<thead>
<tr>
<th>Member Function</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>precision()</td>
<td>Returns the actual precision of floating-point values</td>
</tr>
<tr>
<td>precision(val)</td>
<td>Sets <code>val</code> as the new precision of floating-point values and returns the old</td>
</tr>
</tbody>
</table>

If scientific notation is used, `precision()` defines the number of decimal places in the fractional part. In all cases, the remainder is not cut off but rounded. Calling `precision()` without arguments returns the current precision. Calling it with an argument sets the precision to that value and returns the previous precision. The default precision is six decimal places.

By default, neither `ios::fixed` nor `ios::scientific` is set. In this case, the notation used depends on the value written. All meaningful but, at most, `precision()` decimal places are written as follows: A leading zero before the decimal point and/or all trailing zeros, and potentially even the decimal point, are removed. If `precision()` places are sufficient, decimal notation is used; otherwise, scientific notation is used.
Using the flag `showpoint`, you can force the stream to write a decimal point and trailing zeros until `precision()` places are written (Table 13.25).

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>showpoint</code></td>
<td>Always writes a decimal point</td>
</tr>
</tbody>
</table>

Table 13.26 shows the somewhat complicated dependencies between flags and precision, using two concrete values as an example.

<table>
<thead>
<tr>
<th>precision()</th>
<th>421.0</th>
<th>0.0123456789</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>2</td>
<td>4.2e+02</td>
</tr>
<tr>
<td>With <code>showpoint</code></td>
<td>2</td>
<td>4.2e+02</td>
</tr>
<tr>
<td>fixed</td>
<td>2</td>
<td>421.00</td>
</tr>
<tr>
<td>scientific</td>
<td>2</td>
<td>421.000000</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>421.0000000</td>
</tr>
</tbody>
</table>

As for integral values, `ios::showpos` can be used to write a positive sign, `ios::uppercase` can be used to dictate whether the scientific notation should use an uppercase E or a lowercase e. The flag `ios::showpoint`, the notation, and the precision can be configured using the manipulators presented in Table 13.27.

For example, the statement

```cpp
std::cout << std::scientific << std::showpoint
    << std::setprecision(8) << 0.123456789 << std::endl;
```

produces this output:

```plaintext
1.23456789e-001
```

<table>
<thead>
<tr>
<th>Manipulator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>showpoint</code></td>
<td>Always writes a decimal point (sets the flag <code>ios::showpoint</code>)</td>
</tr>
<tr>
<td><code>noshowpoint</code></td>
<td>Does not require a decimal point (clears the flag <code>showpoint</code>)</td>
</tr>
<tr>
<td><code>setprecision(val)</code></td>
<td>Sets <code>val</code> as the new value for the precision</td>
</tr>
<tr>
<td><code>fixed</code></td>
<td>Uses decimal notation</td>
</tr>
<tr>
<td><code>scientific</code></td>
<td>Uses scientific notation</td>
</tr>
</tbody>
</table>

`setprecision()` is a manipulator with an argument, so you must include the header file `<iomanip>` to use it.

13.7.7 General Formatting Definitions

Two more format flags complete the list of formatting flags: `skipws` and `unitbuf` (Table 13.28).
Table 13.28. Other Formatting Flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>skipws</td>
<td>Skips leading whitespaces automatically when reading a value with operator &gt;&gt;</td>
</tr>
<tr>
<td>unitbuf</td>
<td>Flushes the output buffer after each write operation</td>
</tr>
</tbody>
</table>

*ios::skipws* is set by default, meaning that by default leading whitespaces are skipped by certain read operations. Normally, it is useful to have this flag set. For example, with it set, reading the separating spaces between numbers explicitly is not necessary. However, this implies reading space characters using operator >> is not possible because leading whitespaces are always skipped.

*ios::unitbuf* controls the buffering of the output. With *ios::unitbuf* set, output is basically unbuffered. The output buffer is flushed after each write operation. By default, this flag is not set. However, for the streams *cerr* and *wcerr* this flag is set initially.

Both flags can be manipulated using the manipulators presented in Table 13.29.

### 13.8 Internationalization

You can adapt I/O formats to national conventions. The class *ios_base* defines for this purpose the member functions presented in Table 13.30.

Each stream uses an associated locale object. The initial default locale object is a copy of the global locale object at the construction time of the stream. The locale object defines, for example, details about numeric formatting, such as the character used as the decimal point or the strings used for the textual representation of Boolean values.

#### Table 13.29. Manipulators for Other Formatting Flags

<table>
<thead>
<tr>
<th>Manipulator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>skipws</td>
<td>Skips leading whitespaces with operator &gt;&gt; (sets the flag <em>ios::skipws</em>)</td>
</tr>
<tr>
<td>noskipws</td>
<td>Does not skip leading whitespaces with operator &gt;&gt; (clears the flag <em>ios::skipws</em>)</td>
</tr>
<tr>
<td>unitbuf</td>
<td>Flushes the output buffer after each write operation (sets the flag <em>ios::unitbuf</em>)</td>
</tr>
<tr>
<td>nounitbuf</td>
<td>Does not flush the output buffer after each write operation (clears the flag <em>ios::unitbuf</em>)</td>
</tr>
</tbody>
</table>

#### Table 13.30. Member Functions for Internationalization

<table>
<thead>
<tr>
<th>Member Function</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>imbue(loc)</td>
<td>Sets the locale object</td>
</tr>
<tr>
<td>getloc()</td>
<td>Returns the current locale object</td>
</tr>
</tbody>
</table>

In contrast to the C localization facilities, you can configure each stream individually with a specific locale object. This capability can be used, for example, to read floating-point values according to American format and to write them using German format (in German, a comma is used as the "decimal point.") Section 14.2.1, presents an example and discusses the details. Several characters, mainly special characters, are often needed in the character set of the stream. For this reason, some conversion functions are provided by streams (Table 13.31).

#### Table 13.31. Stream Functions for the Internationalization of Characters

<table>
<thead>
<tr>
<th>Member Function</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>widen(c)</td>
<td>Converts the char character c to a character of the stream's character set</td>
</tr>
<tr>
<td>narrow(c, def)</td>
<td>Converts character c from the stream's character set to a char (if there is no such char, <em>def</em> is returned)</td>
</tr>
</tbody>
</table>

For example, to get the newline character from the character set of the stream *strm*, you can use a statement like

```
strm.widen('\n')
```
For additional details on locales and on internationalization in general, see Chapter 14.

13.9 File Access

Streams can be used to access files. The C++ standard library provides three class templates for which the following standard specializations are predefined:

1. The template class `basic_ifstream<>` with the specializations `ifstream` and `wifstream` is for read access to files ("input file stream").
2. The template class `basic_ofstream<>` with the specializations `ofstream` and `wofstream` is for write access to files ("output file stream").
3. The template class `basic_fstream<>` with the specializations `fstream` and `wifstream` is for access to files that should be both read and written.
4. The template class `basic_filebuf<>` with the specializations `filebuf` and `wfilebuf` is used by the other file stream classes to perform the actual reading and writing of characters.

The classes are related to the stream base classes, as depicted in Figure 13.2.

**Figure 13.2. Class Hierarchy of the File Stream Classes**

These classes are declared in the header file `<fstream>` as follows:
namespace std {
  template <class charT,
    class traits = char_traits<charT> >
    class basic_ifstream;
  typedef basic_ifstream<char> ifstream;
  typedef basic_ifstream<wchar_t> wifstream;

  template <class charT,
    class traits = char_traits<charT> >
    class basic_ofstream;
  typedef basic_ofstream<char> ofstream;
  typedef basic_ofstream<wchar_t> wofstream;

  template <class charT,
    class traits = char_traits<charT> >
    class basic_fstream;
  typedef basic_fstream<char> fstream;
  typedef basic_fstream<wchar_t> wfstream;

  template <class charT,
    class traits = char_traits<charT> >
    class basic_filebuf;
  typedef basic_filebuf<char> filebuf;
  typedef basic_filebuf<wchar_t> wfilebuf;
}

Compared with the mechanism of C, a major advantage of the file stream classes for file access
is the automatic management of files. The files are automatically opened at construction time and
closed at destruction time. This is possible, of course, through appropriate definitions of
corresponding constructors and destructors.
It is important to note for streams that are both read and written that it is not possible to switch
arbitrarily between reading and writing. Once you started to read or to write a file you have to
perform a seek operation, potentially to the current position, to switch from reading to writing or
vice versa. The only exception to this rule is if you have read until end-of-file. In this case you can
continue with writing characters immediately. Violating this rule can lead to all kinds of strange
effects.

(6) This is a restriction inherited from C. However, it is likely that implementations of the standard C++ library
make use of this restriction.

If a file stream object is constructed with a C-string (type char*) as an argument, opening the file
for reading and/or writing is attempted automatically. Whether this attempt was successful is
reflected in the stream's state. Thus, the state should be examined after construction.
The following program opens the file charset.out and writes the current character set (all
characters for the values between 32 and 255) into this file:

```
// io/charset.cpp
#include <string>     // for strings
#include <iostream>   // for I/O
#include <fstream>    // for file I/O
#include <iomanip>    // for setw()
#include <cstdlib>    // for exit()
using namespace std;

// forward declarations
```
void writeCharsetToFile (const string& filename);
void outputFile (const string& filename);

int main()
{
    writeCharsetToFile("charset.out");
    outputFile("charset.out");
}

void writeCharsetToFile (const string& filename)
{
    // open output file
    ofstream file (filename.c_str());

    // file opened?
    if (! file) {
        // NO, abort program
        cerr << "can't open output file " << filename << "\n"
             << endl;
        exit (EXIT_FAILURE);
    }

    // write character set
    for (int i=32; i<256; i++) {
        file << "value: " << setw(3) << i << " char: " << static_cast<char> (i) << endl;
    }

} // closes file automatically

void outputFile (const string& filename)
{
    // open input file
    ifstream file (filename.c_str());

    // file opened?
    if (! file) {
        // NO, abort program
        cerr << "can't open input file " << filename << "\n"
             << endl;
        exit(EXIT_FAILURE);
    }

    // copy file contents to cout
    char c;
    while (file.get(c)) { cout.put(c); }

} // closes file automatically

In writeCharsetToFile(), the constructor of the class of stream takes care of opening the file named by the given file name:

    std::ofstream file(filename.c_str());
The file name is a string, so `c_str()` is used to convert it to `const char*` (see page 484 for details about `c_str()`). Unfortunately, there is no constructor for the file stream classes that takes `string` as the argument type. After this, it is determined whether the stream is in a good state:

```cpp
if (! file) {
    ...
}
```

If opening the stream was not successful, this test will fail. After this check, a loop prints the values 32 to 255 together with the corresponding characters.

In the function `outputFile()`, the constructor of the class `ifstream` opens the file. Then the contents of the file are written characterwise.

At the end of both functions the file opened locally is closed automatically when the corresponding stream goes out of scope. The destructors of the classes `ifstream` and `ofstream` take care of closing the file if it is still open at destruction time.

If a file should be used longer than the scope in which it was created, you can allocate the file object on the heap and delete it later when it is no longer needed:

```cpp
std::ofstream* filePtr = new std::ofstream("xyz");
...
delete filePtr;
```

In this case, some smart pointer class, such as `CountedPtr` (see Section 6.8) or `auto_ptr` (see Section 4.2), should be used.

Instead of copying the file contents character-by-character, you could also output the whole contents in one statement by passing a pointer to the stream buffer of the file as an argument to `operator <<`:

```cpp
// copy file contents to cout
std::cout << file.rdbuf();
```

See page 683 for details.

### 13.9.1 File Flags

For precise control over the processing mode of a file, a set of flags is defined in the class `ios_base` (Table 13.32). These flags are of type `openmode`, which is a bit mask type similar to `fmtflags`.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>Opens for reading (default for <code>ifstream</code>)</td>
</tr>
<tr>
<td>out</td>
<td>Opens for writing (default for <code>ofstream</code>)</td>
</tr>
<tr>
<td>app</td>
<td>Always appends at the end when writing</td>
</tr>
<tr>
<td>ate</td>
<td>Positions at the end of the file after opening (&quot;at end&quot;)</td>
</tr>
<tr>
<td>trunc</td>
<td>Removes the former file contents</td>
</tr>
<tr>
<td>binary</td>
<td>Does not replace special characters</td>
</tr>
</tbody>
</table>
binary configures the stream to suppress conversion of special characters or character sequences such as end-of-line or end-of-file. In operating systems, such as MS-DOS or OS/2, a line end in text files is represented by two characters (CR and LF). In normal text mode (binary is not set), newline characters are replaced by the two-character sequence, and vice versa, when reading or writing to avoid special processing. In binary mode (binary is set), none of these conversions take place.

binary should always be used if the contents of a file do not consist of a character sequence but are processed as binary data. An example is the copying of files by reading the file to be copied character-by-character and writing those characters without modifying them. If the file is processed as text, the flag should not be set because special handling of newlines would be required. For example, a newline would still consist of two characters.

Some implementations provide additional flags such as nocr (the file must exist when it is opened) and noreplace (the file must not exist). However, these flags are not standard and thus are not portable.

The flags can be combined by using operator |. The resulting openmode can be passed as an optional second argument to the constructor. For example, the following statement opens a file for appending text at the end:

```cpp
std::ofstream file("xyz.out", std::ios::out|std::ios::app);
```

Table 13.33 correlates the various combinations of flags with the strings used in the interface of C's function for opening files: fopen(). The combinations with the binary and the ate flags set are not listed. A set binary corresponds to strings with b appended, and a set ate corresponds to a seek to the end of the file immediately after opening. Other combinations not listed in the table, such as trunc | app, are not allowed.

<table>
<thead>
<tr>
<th><code>ios_base</code> Flags</th>
<th>Meaning</th>
<th>CMode</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>Reads (file must exist)</td>
<td>&quot;r&quot;</td>
</tr>
<tr>
<td>out</td>
<td>Empties and writes (creates if necessary)</td>
<td>&quot;w&quot;</td>
</tr>
<tr>
<td>out</td>
<td>trunc</td>
<td>Empties and writes (creates if necessary)</td>
</tr>
<tr>
<td>out</td>
<td>app</td>
<td>Appends (creates if necessary)</td>
</tr>
<tr>
<td>in</td>
<td>out</td>
<td>Reads and writes; initial position is the start (file must exist)</td>
</tr>
<tr>
<td>in</td>
<td>out</td>
<td>trunc</td>
</tr>
</tbody>
</table>

Whether a file is opened for reading and/or for writing is independent of the corresponding stream object's class. The class only determines the default open mode if no second argument is used. This means that files used only by the class if stream or the class of stream can be opened for reading and writing. The open mode is passed to the corresponding stream buffer class, which opens the file. However, the operations possible for the object are determined by the stream's class.

The file owned by a file stream can also be opened or closed explicitly. For this, three member functions are defined (Table 13.34).

These functions are useful mainly if a file stream is created without being initialized. The following example demonstrates their use. It opens all files with names that are given as arguments to the program, and writes their contents (this corresponds to the UNIX program cat).

<table>
<thead>
<tr>
<th>Table 13.34. Member Functions to Open and Close Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member Function</td>
</tr>
<tr>
<td>open(name)</td>
</tr>
<tr>
<td>open (name, flags)</td>
</tr>
<tr>
<td>close()</td>
</tr>
<tr>
<td>is_open()</td>
</tr>
</tbody>
</table>
// io/cat1. cpp
// header files for file I/O
#include <fstream>
#include <iostream>
using namespace std;

/* for all file names passed as command-line arguments
 * - open, print contents, and close file
 */
int main (int argc, char* argv[]) {
    ifstream file;
    // for all command-line arguments
    for (int i=1; i<argc; ++i) {
        // open file
        file.open(argv[i]);

        // write file contents to cout
        char c;
        while (file.get(c)) {
            cout.put(c);
        }

        // clear eofbit and failbit set due to end-of-file
        file.clear();

        // close file
        file.close();
    }
}

Note that after the processing of a file, clear() must be called to clear the state flags that are set at end-of-file. This is required because the stream object is used for multiple files. The member function open() does not clear the state flags. open() open() never clears any state flags. Thus, if a stream was not in a good state, after closing and reopening it you still have to call clear() to get to a good state. This is also the case, if you open a different file. Instead of processing character-by-character, you could also print the entire contents of the file in one statement by passing a pointer to the stream buffer of the file as an argument to operator <<:

    // write file contents to cout
    std::cout << file.rdbuf();

See page 683 for details.

13.9.2 Random Access

Table 13.35 lists the member function defined for positioning within C++ streams.

| Table 13.35. Member Functions for Stream Positions | 547 |
Class | Member Function | Meaning
---|---|---
`basic_istream<>` | `tellg()` | Returns the read position
| `seekg(pos)` | Sets the read position as an absolute value
| `seekg(offset, rpos)` | Sets the read position as a relative value

`basic_ostream<>` | `tellp()` | Returns the write position
| `seekp(pos)` | Sets the write position as an absolute value
| `seekp(offset, rpos)` | Sets the write position as a relative value

These functions distinguish between read and write position (g stands for get and p stands for put). Read position functions are defined in `basic_istream`, and write position functions are defined in `basic_ostream`. However, not all stream classes support positioning. For example, positioning the streams `cin`, `cout`, and `cerr` is not defined. The positioning of files is defined in the base classes because, usually, references to objects of type `istream` and `ostream` are passed around.

The functions `seekg()` and `seekp()` can be called with absolute or relative positions. To handle absolute positions, you must use `tellg()` and `tellp()`. They return an absolute position as the value of type `pos_type`. This value is not an integral value or simply the position of the character as an index. This is because the logical position and the real position can differ. For example, in MS-DOS text files, newline characters are represented by two characters in the file even though it is logically only one character. Things are even worse if the file uses some multibyte representation for the characters.

The exact definition of `pos_type` is a bit complicated: The C++ standard library defines a global template class `fpos<>` for file positions. Class `fpos<>` is used to define types `streampos` for `char` and `wstreampos` for `wchar_t` streams. These types are used to define the `pos_type` of the corresponding character traits (see Section 14.1.2). And the `pos_type` member of the traits is used to define `pos_type` of the corresponding stream classes. Thus, you could also use `streampos` as the type for the stream positions. However, using `long` or `unsigned long` is wrong because `streampos` is not an integral type (anymore).

For example:

```cpp
// save current file position
std::ios::pos_type pos = file.tellg();
...
// seek to file position saved in pos
file.seekg(pos);
```

Instead of

```cpp
std::ios::pos_type pos;
```

you could also write:

```cpp
std::streampos pos;
```

For relative values, the offset can be relative to three positions, for which corresponding constants are defined (Table 13.36). The constants are defined in class `ios_base` and are of type `seekdir`.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
</table>

(Formerly, `streampos` was used for stream positions, and it was simply defined as `unsigned long`.)

For example:

```cpp
// save current file position
std::ios::pos_type pos = file.tellg();
...
// seek to file position saved in pos
file.seekg(pos);
```
The type for the offset is `off_type`, which is an indirect definition of `streamoff`. Similar to `pos_type`, `streamoff` is used to define `off_type` of the traits (see page 689) and the stream classes. However, `streamoff` is a signed integral type, so you can use integral values as stream offsets. For example:

```cpp
// seek to the beginning of the file
file.seekg (0, std::ios::beg);
...
// seek 20 character forward
file.seekg (20, std::ios::cur);
...
// seek 10 characters before the end
file.seekg (-10, std::ios::end);
```

In all cases, care must be taken to position only within a file. If a position ends up before the beginning of a file or beyond the end, the behavior is undefined. The following example demonstrates the use of `seekg()`. It uses a function that writes the contents of a file twice:

```cpp
// io/cat2.cpp

// header files for file I/O
#include <iostream>
#include <fstream>

void printFileTwice (const char* filename)
{
    // open file
    std::ifstream file(filename);

    // print contents the first time
    std::cout << file.rdbuf();

    // clear eofbit and failbit set due to end-of-file
    file.clear();

    // seek to the beginning
    file.seekg(0);

    // print contents the second time
    std::cout << file.rdbuf();
}

int main (int argc, char* argv[])
{
    // print all files passed as a command-line argument twice
    for (int i=1; i<argc; ++i) {
        printFileTwice (argv[i]);
    }
}
```
Note that `ios::eofbit` and `ios::failbit` are set when end-of-file is reached. Hence, the stream is no longer in a good state. It has to be restored to a good state via `clear()` before it can be manipulated in any way (including changes of the read position). Different functions are provided for the manipulation of the read and the write positions; but for the standard streams, the same position is maintained for the read and write positions in the same stream buffer. This is important if multiple streams use the same stream buffer. It is explained in more detail in Section 13.10.2.

### 13.9.3 Using File Descriptors

Some implementations provide the possibility of attaching a stream to an already opened I/O channel. To do this, you initialize the file stream with a file descriptor.

File descriptors are integers that identify an open I/O channel. In UNIX-like systems, file descriptors are used in the low-level interface to the I/O functions of the operating system. Three file descriptors are predefined:

1. 0 for the standard input channel
2. 1 for the standard output channel
3. 2 for the standard error channel

These channels may be connected to files, the console, other processes, or some other I/O facility.

The C++ standard library unfortunately does not provide this possibility of attaching a stream to an I/O channel using file descriptors. This is because the language is supposed to be independent of any operating system. In practice, though, the possibility probably still exists. The only drawback is that using it is not portable to all systems. What is missing at this point is a corresponding specification in a standard of operating system interfaces such as POSIX or X/OPEN. However, such a standard is not yet planned. However, it is possible to initialize a stream by a file descriptor. See Section 13.13.3, for a description and implementation of a possible solution.

### 13.10 Connecting Input and Output Streams

Often you need to connect two streams. For example, you may want to ensure that text asking for input is written on the screen before the input is read. Another example is reading from and writing to the same stream. This is mainly of interest regarding files. A third example is the need to manipulate the same stream using different formats. This section discusses all of these techniques.

#### 13.10.1 Loose Coupling Using `tie()`

You can tie a stream to an output stream. This means the buffers of both streams are synchronized in a way that the buffer of the output stream is flushed before each input or output of the other stream. That is, for the output stream, the function `flush()` is called. Table 13.37 lists the member functions defined in `basic_ios` for tying one stream to another. Calling the function `tie()` without any argument returns a pointer to the output stream that is currently tied to a stream. To tie a new output stream to a stream, a pointer to that output stream must be passed as the argument to `tie()`. The argument is a pointer because you can also pass 0 or `NULL` as an argument. This argument means "no tie," and unties any tied output stream. 0 is also returned by `tie()` if no output stream is tied. For each stream, you can only
have one output stream that is tied to this stream. However, you can tie an output stream to different streams.

<table>
<thead>
<tr>
<th>Table 13.37. Tieing One Stream to Another</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Member Function</strong></td>
</tr>
<tr>
<td>tie()</td>
</tr>
<tr>
<td>tie(ostream* strm)</td>
</tr>
</tbody>
</table>

By default, the standard input is connected to the standard output using this mechanism:

```cpp
// predefined connections:
std::cin.tie (&std::cout);
std::wcin.tie (&std::wcout);
```

This ensures that a message asking for input is flushed before requesting the input. For example, during the statements

```cpp
std::cout << "Please enter x: ";
std::cin >> x;
```

the function `flush()` is called implicitly for `cout` before reading `x`.

To remove the connection between two streams, you pass 0 or `NULL` to `tie()`. For example:

```cpp
// decouple cin from any output stream
std::cin.tie (static_cast<std::ostream*>(0));
```

This might improve the performance of a program because it avoids unnecessary additional flushing of streams (see Section 3, page 683, for a discussion of stream performance). You can also tie one output stream to another output stream. For example, the following statement arranges that before something is written to the error stream, the normal output is flushed:

```cpp
// tieing cout to cerr
cerr.tie (&cout);
```

### 13.10.2 Tight Coupling Using Stream Buffers

Using the function `rdbuf()`, you can couple streams tightly by using a common stream buffer (Table 13.38). These functions suit several purposes, which are discussed in this and the following subsections.

`rdbuf()` allows several stream objects to read from the same input channel or to write to the same output channel without garbling the order of the I/O. The use of multiple stream buffers does not work smoothly because the I/O operations are buffered. Thus, when using different streams with different buffers for the same I/O channel means that I/O may pass other I/O. An additional constructor of `basic_istream` and `basic_ostream` is used to initialize the stream with a stream buffer passed as the argument. For example:

<table>
<thead>
<tr>
<th>Table 13.38. Stream Buffer Access</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Member</strong></td>
</tr>
</tbody>
</table>

```cpp
// predefined connections:
std::cin.tie (&std::cout);
std::wcin.tie (&std::wcout);
```
Function  
---
rdbuf()  | Returns a pointer to the stream buffer  
rdbuf(streambuf*)  | Installs the stream buffer pointed to by the argument and returns a pointer to the previously used stream buffer

```cpp
// io/rdbuf1.cpp
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    // stream for hexadecimal standard output
    ostream hexout(cout.rdbuf());
    hexout.setf (ios::hex, ios::basefield);
    hexout.setf (ios::showbase);

    // switch between decimal and hexadecimal output
    hexout << "hexout: " << 177 << " ";
    cout   << "cout: "   << 177 << " ";
    hexout << "hexout: " << -49 << " ";
    cout   << "cout: "   << -49 " ";
    hexout << endl;
}
```

Note that the destructor of the classes basic_istream and basic_ostream does not delete the corresponding stream buffer (it was not opened by these classes anyway). Thus, you can pass a stream device by using a pointer to the stream buffer instead of a stream reference:

```cpp
// io/rdbuf2.cpp
#include <iostream>
#include <fstream>

void hexMultiplicationTable (std::streambuf* buffer, int num)
{
    std::ostream hexout(buffer);
    hexout << std::hex << std::showbase;

    for (int i=1; i<=num; ++i) {
        for (int j=1; j<=10; ++j) {
            hexout << i*j << ' ';
        }
        hexout << std::endl;
    }

} // does NOT close buffer

int main()
{
    using namespace std;
    int num = 5;
}
cout << "We print " << num
<< " lines hexadecimal" << endl;

hexMultiplicationTable(cout.rdbuf(),num);

cout << "That was the output of " << num
<< " hexadecimal lines " << endl;
}

The advantage of this approach is that the format does not need to be restored to its original state
after it is modified because the format applies to the stream object, not to the stream buffer. Thus,
the corresponding output of the program is as follows:

We print 5 lines hexadecimal
0x1 0x2 0x3 0x4 0x5 0x6 0x7 0x8 0x9 0xa
0x2 0x4 0x6 0x8 0xa 0xc 0xe 0x10 0x12 0x14
0x3 0x6 0x9 0xc 0xf 0x12 0x15 0x18 0x1b 0x1e
0x4 0x8 0xc 0x10 0x14 0x18 0x1c 0x20 0x24 0x28
0x5 0xa 0xf 0x14 0x19 0x1e 0x23 0x28 0x2d 0x32
That was the output of 5 hexadecimal lines

However, this has the disadvantage that construction and destruction of a stream object involves
more overhead than just setting and restoring some format flags. Also note that the destruction of
a stream object does not flush the buffer. To make sure that an output buffer is flushed, it has to be
flushed manually.
The fact that the stream buffer is not destroyed applies only to basic_istream and
basic_ostream. The other stream classes destroy the stream buffers they allocated originally,
but they do not destroy stream buffers set with rdbuf() (for more details see the next
subsection).

13.10.3 Redirecting Standard Streams

In the old implementation of the IOSTream library, the global streams cin, cout, cerr, and
clog were objects of the classes istream_withassign and ostream_withassign. It was therefore possible to redirect the streams by assigning streams to other streams. This possibility was removed from the C++ standard library. However, the possibility to redirect streams was retained and extended to apply to all streams. A stream can be redirected by setting a stream buffer.
The setting of stream buffers means the redirection of I/O streams controlled by the program
without help from the operating system. For example, the following statements set things up such
that output written to cout is not sent to the standard output channel but rather to the file
cout.txt:

std::ofstream file ("cout.txt");
std::cout.rdbuf (file.rdbuf());

The function copyfmt() can be used to assign all format information of a given stream to
another stream object:

std::ofstream file ("cout.txt");
file.copyfmt (std::cout);
std::cout.rdbuf (file.rdbuf());
Caution! The object file is local and is destroyed at the end of the block. This also destroys the corresponding stream buffer. This differs from the "normal" streams because file streams allocate their stream buffer objects at construction time and destroy them on destruction. Thus, in this example, cout can no longer be used for writing. Actually, it cannot even be destroyed safely at program termination. Thus, the old buffer should always be saved and restored later! The following example does this in the function redirect():

```
// io/redirect.cpp

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

void redirect(ostream&);

int main()
{
    cout << "the first row" << endl;
    redirect (cout);
    cout << "the last row" << endl;
}

void redirect (ostream& strm)
{
    ofstream file("redirect.txt");
    // save output buffer of the stream
    streambuf* strm_buffer = strm.rdbuf();
    // redirect output into the file
    strm.rdbuf (file.rdbuf());
    file << "one row for the file" << endl;
    strm << "one row for the stream" << endl;
    // restore old output buffer
    strm.rdbuf (strm_buffer);
} // closes file AND its buffer automatically
```

The output of the program is this

```
the first row
the last row
```

and the contents of the file redirect.txt are

```
one row for the file
one row for the stream
```
As you can see, the output written in `redirect()` to `cout` (using the parameter name `strm`) is sent to the file. The output written after the execution of `redirect()` in `main()` is sent to the restored output channel.

### 13.10.4 Streams for Reading and Writing

A final example of the connection between streams is the use of the same stream for reading and writing. Normally, a file can be opened for reading and writing using the class `fstream`:

```cpp
std::fstream file ("example.txt", std::ios::in | std::ios::out);
```

It is also possible to use two different stream objects, one for reading and one for writing. This can be done, for example, with the following declarations:

```cpp
std::ofstream out ("example.txt", ios::in | ios::out);
std::istream in (out.rdbuf());
```

The declaration of `out` opens the file. The declaration of `in` uses the stream buffer of `out` to read from it. Note that `out` must be opened for both reading and writing. If it is only opened for writing, reading from the stream will result in undefined behavior. Also note that `in` is not of type `ifstream`. The file is already opened and there is a corresponding stream buffer. All that is needed is a second stream object. As in previous examples, the file is closed when the file stream object is destroyed.

It is also possible to create a file stream buffer and install it in both stream objects. The code looks like this:

```cpp
std::filebuf buffer;
std::ostream out (&buffer);
std::istream in (&buffer);
buffer.open("example.txt", std::ios::in | std::ios::out);
```

`filebuf` is the usual specialization of the class `basic.filebuf<>` for the character type `char`. This class defines the stream buffer class used by file streams. The following program is a complete example. In a loop, four lines are written to a file. After each writing of a line, the whole contents of the file are written to standard output:

```cpp
// io/rw1.cpp

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

int main()
{
    // open file "example.dat" for reading and writing
    filebuf buffer;
    ostream output(&buffer);
    istream input(&buffer);
    buffer.open ("example.dat", ios::in | ios::out | ios::trunc);
    for (int i=1; i<=4; i++) {
```
// write one line
output << i << ". line" << endl;

// print all file contents
input.seekg(0); // seek to the beginning
char c;
while (input.get(c)) {
    cout.put(c);
}
cout << endl;
input.clear(); // clear eofbit and failbit
}

The output of the program is as follows:

1. line
1. line
2. line
1. line
2. line
3. line
1. line
2. line
3. line
4. line

Although two different stream objects are used for reading and writing, the read and write positions are tightly coupled. seekg() and seekp() call the same member function of the stream buffer. Thus, the read position must always be set to the beginning of the file in order for the complete contents of the file to be written. After the whole contents of the file are written, the read/write position is again at the end of the file so that new lines are appended to the file.

Actually, this function can distinguish whether the read position, the write position, or both positions are to be modified. Only the standard stream buffers maintain one position for reading and writing.

It is important to perform a seek between read and write operations to the same file unless you have reached the end of the file while reading. Without this seek you are likely to end up with a garbled file or with even more fatal errors.

As mentioned before, instead of processing character-by-character, you could also print the entire contents in one statement by passing a pointer to the stream buffer of the file as an argument to operator << (see page 683 for details):

```cpp
std::cout << input.rdbuf();
```

### 13.11 Stream Classes for Strings
The mechanisms of stream classes can also be used to read from strings or to write to strings. String streams provide a buffer but don't have an I/O channel. This buffer/string can be manipulated with special functions. A major use of this is the processing of I/O independent of the actual I/O. For example, text for output can be formatted in a string and then sent to an output channel sometime later. Another use is reading input line-by-line and processing each line using string streams.

The original stream classes for strings are replaced by a set of new ones in the C++ standard library. Formerly, the string stream classes used type `char*` to represent a string. Now, type `string` (or in general `basic_string<>`) is used. The old string stream classes are also part of the C++ standard library, but they are deprecated. They are retained for backward compatibility, but they might be removed in future versions of the standard. Thus, they should not be used in new code and should be replaced in legacy code. Still, a brief description of these classes is found at the end of this section.

### 13.11.1 String Stream Classes

The following stream classes are defined for strings (they correspond to the stream classes for files):

- The class `basic_istringstream` with the specializations `istringstream` and `wistringstream` for reading from strings ("input string stream")
- The class `basic_ostringstream` with the specializations `ostringstream` and `wostringstream` for writing to strings ("output string stream")
- The class `basic_stringstream` with the specializations `stringstream` and `wstringstream` for reading from and writing to strings
- The template class `basic_stringbuf<>` with the specializations `stringbuf` and `wstringbuf` is used by the other string stream classes to perform the actual reading and writing of characters.

These classes have a similar relationship to the stream base classes, as do the file stream classes. The class hierarchy is depicted in Figure 13.3.
The classes are declared in the header file `<sstream>` like this:

```cpp
namespace std {
  template <class charT,
            class traits = char_traits<charT>,
            class Allocator = allocator<charT> >
    class basic_istringstream;
  typedef basic_istringstream<char>    istringstream;
  typedef basic_istringstream<wchar_t> wistringstream;

  template <class charT,
            class traits = char_traits<charT>,
            class Allocator = allocator<charT> >
    class basic_ostringstream;
  typedef basic_ostringstream<char>    ostringstream;
  typedef basic_ostringstream<wchar_t> wostringstream;

  template <class charT,
            class traits = char_traits<charT>,
            class Allocator = allocator<charT> >
    class basic_stringstream;
  typedef basic_stringstream<char>    stringstream;
  typedef basic_stringstream<wchar_t> wstringstream;

  template <class charT,
            class traits = char_traits<charT>,
            class Allocator = allocator<charT> >
    class basic_stringbuf;
```
typedef basic_stringbuf<char>    stringbuf;
typedef basic_stringbuf<wchar_t> wstringbuf;

The major function in the interface of the string stream classes is the member function `str()`. This function is used to manipulate the buffer of the string stream classes (Table 13.39).

<table>
<thead>
<tr>
<th>Member Function</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>str()</code></td>
<td>Returns the buffer as a string</td>
</tr>
<tr>
<td><code>str(string)</code></td>
<td>Sets the contents of the buffer to <code>string</code></td>
</tr>
</tbody>
</table>

The following program demonstrates the use of string streams:

```cpp
#include <iostream>
#include <sstream>
#include <bitset>
using namespace std;

int main()
{
    ostringstream os;

    // decimal and hexadecimal value
    os << "dec: " << 15 << hex << " hex: " << 15 << endl;
    cout << os.str() << endl;

    // append floating value and bitset
    bitset<15> b(5789);
    os << "float: " << 4.67 << " bitset: " << b << endl;

    // overwrite with octal value
    os.seekp(0);
    os << "oct: " << oct << 15;
    cout << os.str() << endl;
}
```

The output of this program is as follows:

```
dec: 15 hex: f
oct: 17 hex: f
float: 4.67 bitset: 001011010011101
```

First a decimal and a hexadecimal value are written to `os`. Next a floating-point value and a bitset (written in binary) are appended. Using `seekp()`, the write position is moved to the beginning of the stream. So, the following call of operator `<<` writes at the beginning of the string, thus overwriting the beginning of the existing string stream. However, the characters that are not overwritten remain valid. If you want to remove the current contents from the stream, you can use the function `str()` to assign new contents to the buffer:

```cpp
strm.str(" ");
```
The first lines written to os are each terminated with \texttt{endl}. This means that the string ends with a newline. Because the string is printed followed by \texttt{endl}, two adjacent newlines are written. This explains the empty lines in the output.

A typical programming error when dealing with string streams is to forget to extract the string with the function \texttt{str()}, and instead to write to the stream directly. This is, from a compiler’s point of view, a possible and reasonable thing to do in that there is a conversion to \texttt{void*}. As a result, the state of the stream is written in the form of an address (see page 596). A typical use for writing to an output string stream is to define output operators for user-defined types (see Section 13.12.1).

Input string streams are used mainly for formatted reading from existing strings. For example, it is often easier to read data line-by-line and then analyze each line individually. The following lines read the integer \texttt{x} with the value 3 and the floating-point \texttt{f} with the value 0.7 from the string \texttt{s}:

```cpp
int x;
float f;
std::string s = "3.7";

std::istringstream is(s);
is >> x >> f;
```

A string stream can be created with the flags for the file open modes (see Section 13.9.1,) and/or an existing string. With the flag \texttt{ios::app} or \texttt{ios::ate}, the characters written to a string stream can be appended to an existing string:

```cpp
std::string s;
...
std::ostringstream os (s, ios::out|ios::app);
os << 77 << std::hex << 77;
```

However, this means that the string returned from \texttt{str()} is a copy of the string \texttt{s}, with a decimal and a hexadecimal version of 77 appended. The string \texttt{s} itself is not modified.

### 13.11.2 \texttt{char*} Stream Classes

The \texttt{char*} stream classes are retained only for backward compatibility. Their interface is error prone and they are rarely used correctly. However, they are still in heavy use and thus are described briefly here. Note that the standard version described here has slightly modified the old interface.

In this subsection, the term \textit{character sequence} will be used instead of \textit{string}. This is because the character sequence maintained by the \texttt{char*} stream classes is not always terminated with the string termination character (and thus it is not really a string).

The \texttt{char*} stream classes are defined only for the character type \texttt{char}. They include:

- The class \texttt{istrstream} for reading from character sequences (input string stream)
- The class \texttt{ostrstream} for writing to character sequences (output string stream)
- The class \texttt{strstream} for reading from and writing to character sequences
- The class \texttt{strstreambuf} used as a stream buffer for \texttt{char*} streams

The \texttt{char*} stream classes are defined in the header file \texttt{<strstream>}. 
An `istrstream` can be initialized with a character sequence (of type `char*`) that is either terminated with the string termination character `\0` or for which the number of characters is passed as the argument. A typical use is the reading and processing of whole lines:

```cpp
char buffer [1000]; // buffer for at most 999 characters

// read line
std::cin.get(buffer, sizeof(buffer));

// read/process line as stream
std::istrstream input(buffer);
...
input >> x;
```

A `char*` stream for writing can either maintain a character sequence that grows as needed or it can be initialized with a buffer of fixed size. Using the flag `ios::app` or `ios::ate`, you can append the characters written to a character sequence that is already stored in the buffer. Care must be taken when using `char*` stream as a string. In contrast to string streams, `char*` streams are not always responsible for the memory used to store the character sequence. With the member function `str()`, the character sequence is made available to the caller together with the responsibility for the corresponding memory. Unless the stream is initialized with a buffer of fixed size (for which the stream is never responsible), the following three rules have to be obeyed:

1. Because ownership of the memory is transferred to the caller, unless the stream was initialized with a buffer of fixed size, the character sequence has to be released. However, there is no guarantee how the memory was allocated, thus it is not always safe to release it using `delete[]`. Your best bet is to return the memory to the stream by calling the member function `freeze()` with the argument `false` (the following paragraphs present an example).

   [11] There is actually a constructor that takes two function pointers as an argument: a function to allocate memory and a function to release memory.

2. With the call to `str()`, the stream is no longer allowed to modify the character sequence. It calls the member function `freeze()` implicitly, which freezes the character sequence. The reason for this is to avoid complications if the allocated buffer is not sufficiently large and new memory has to be allocated.

3. The member function `str()` does not append a string termination character (`'\0'`). This character has to be appended explicitly to the stream to terminate the character sequence. This can be done using the `ends` manipulator. Some implementations append a string termination character automatically, but this behavior is not portable.

The following example demonstrates the use of a `char*` stream:

```cpp
float x;
...
/* create and fill char* stream
   */
std::ostrstream buffer; // dynamic stream buffer
buffer << "float x: " << x << std::ends;

// pass resulting C-string to foo() and return memory to buffer
```
char* s = buffer.str();
foo(s);
buffer.freeze(false);

A frozen char* stream can be restored to its normal state for additional manipulation. To do so, the member function freeze() has to be called with the argument false. With this operation, ownership of the character sequence is returned to the stream object. This is the only safe way to release the memory for the character sequence. The next example demonstrates this:

float x;
...
std::ostrstream buffer; // dynamic char* stream

// fill char* stream
buffer << "float x: " << x << std::ends;

/* pass resulting C-string to foo() */
* - freezes the char* stream
*/
foo(buffer.str());

// unfreeze the char* stream
buffer.freeze(false);

// seek writing position to the beginning
buffer.seekp (0, ios::beg);

// refill char* stream
buffer << "once more float x: " << x << std::ends;

/* pass resulting C-string to foo() again */
* - freezes the char* stream
*/
foo(buffer.str());

// return memory to buffer
buffer.freeze(false);

The problems related to freezing the stream are removed from the string stream classes. This is mainly because the strings are copied and because the string class takes care of the used memory.

13.12 Input/Output Operators for User-Defined Types

As mentioned earlier in this chapter, a major advantage of streams over the old I/O mechanism of C is the possibility that the stream mechanism can be extended to user-defined types. To do this, you must overload operators << and >>. This is demonstrated using a class for fractions in the following subsection.

13.12.1 Implementing Output Operators

In an expression with the output operator, the left operand is a stream and the right operand is the object to be written:
stream << object

According to language rules this can be interpreted in two ways:

1. As stream. operator<<(object)
2. As operator<<(stream,object)

The first way is used for built-in types. For user-defined types you have to use the second way because the stream classes are closed for extensions. All you have to do is implement global operator << for your user-defined type. This is rather easy, unless access to private members of the objects is necessary (which I cover later).

For example, to print an object of class Fraction with the format numerator/denominator, you can write the following function:

```
// io/frac1out.hpp
#include <iostream>

inline std::ostream& operator << (std::ostream& strm, const Fraction& f)
{  
    strm << f.numerator() << '/' << f.denominator();
    return strm;
}
```

The function writes the numerator and the denominator, separated by the character '/', to the stream that is passed as the argument. The stream can be a file stream, a string stream, or some other stream. To support the chaining of write operations or the access to the streams state in the same statement, the stream is returned by the function.

This simple form has two drawbacks:

1. Because ostream is used in the signature, the function applies only to streams with the character type char. If the function is intended only for use in Western Europe or in North America, this is no problem. On the other hand, a more general version requires only a little extra work, so it should at least be considered.
2. Another problem arises if a field width is set. In this case, the result is probably not what might be expected. The field width applies to the immediately following write; in this case, to the numerator. Thus, the statements

3. Fraction vat(16,100); // I'm German and we have a uniform VAT of 16%...
4. std::cout << "VAT: " << std::left << std::setw(8) << vat << std::endl;
5. vat << ' ' << std::endl;

result in this output:

```
VAT: "16   /100"
```
The next version solves both of these problems:

```cpp
// io/frac2out.hpp
#include <iostream>
#include <sstream>

template <class charT, class traits>
inline
std::basic_ostream<charT,traits>&
operator << (std::basic_ostream<charT,traits>& strm,
             const Fraction& f)
{
    /* string stream
     * - with same format
     * - without special field width
     */
    std::basic_ostringstream<charT,traits> s;
    s.copyfmt(strm);
    s.width(0);

    // fill string stream
    s << f.numerator() << '/' << f.denominator();

    // print string stream
    strm << s.str();

    return strm;
}
```

The operator has become a template function that is parameterized to suit all kinds of streams. The problem with the field width is addressed by writing the fraction first to a string stream without setting any specific width. The constructed string is then sent to the stream passed as the argument. This results in the characters representing the fraction being written with only one write operation, to which the field width is applied. Thus, the statements

```cpp
Fraction vat (16,100); // I'm German...
std::cout << "VAT: \"" << std::left << std::setw(8)
           << vat << '\"' << std::endl;
```

now produce the following output:

```
VAT: "15/100 
```

### 13.12.2 Implementing Input Operators

Input operators are implemented according to the same principle as output operators (described in the previous subsection). However, input incurs the likely problem of read failures. Input functions normally need special handling of cases in which reading might fail.
When implementing a read function you can choose between simple or flexible approaches. For example, the following function uses a simple approach. It reads a fraction without checking for error situations:

```cpp
// io/frac1in.hpp
#include <iostream>

inline
std::istream& operator >> (std::istream& strm, Fraction& f)
{
  int n, d;
  strm >> n;   // read value of the numerator
  strm.ignore();  // skip '/'
  strm >> d;  // read value of the denominator
  f = Fraction(n, d);  // assign the whole fraction
  return strm;
}
```

This implementation has the problem that it can be used only for streams with the character type `char`. In addition, whether the character between the two numbers is indeed the character '/' is not checked.

Another problem arises when undefined values are read. When reading a zero for the denominator, the value of the read fraction is not well-defined. This problem is detected in the constructor of the class `Fraction` that is invoked by the expression `Fraction(n, d)`. However, handling inside class `Fraction` means that a format error automatically results in an error handling of the class `Fraction`. Because it is common practice to record format errors in the stream, it might be better to set `ios_base::failbit` in this case.

Lastly, the fraction passed by reference might be modified even if the read operation is not successful. This can happen, for example, when the read of the numerator succeeds, but the read of the denominator fails. This behavior contradicts common conventions established by the predefined input operators, and thus is best avoided. A read operation should be successful or have no effect.

The following implementation is improved to avoid these problems. It is also more flexible because it is parameterized to be applicable to all stream types:

```cpp
// io/frac2in.hpp
#include <iostream>
template <class charT, class traits>
inline
std::basic_istream<charT,traits>&
operator >> (std::basic_istream<charT,traits>& strm, Fraction& f)
{
  int n, d;
  // read value of numerator
  strm >> n;
  /* if available
```
Here the denominator is read only if the first number is followed by the character '/', otherwise, a denominator of one is assumed and the integer read is interpreted as the whole fraction. Hence, the denominator is optional.

This implementation also tests whether a denominator with value 0 was read. In this case, the `ios_base::failbit` is set, which might trigger a corresponding exception (see Section 13.4.4). Of course, the behavior can be implemented differently if the denominator is zero. For example, an exception could be thrown directly, or the check could be skipped so that the fraction is initialized with zero, which would throw the appropriate exception by class `Fraction`. Lastly, the state of the stream is checked and the new value is assigned to the fraction only if no input error occurred. This final check should always be done to make sure that the value of an object is changed only if the read was successful.

Of course, it can be argued whether it is reasonable to read integers as fractions. In addition, there are other subtleties that may be improved. For example, the numerator must be followed by the character '/' without separating whitespaces. But the denominator may be preceded by arbitrary whitespaces because normally these are skipped. This hints at the complexity involved in reading nontrivial data structures.

### 13.12.3 Input/Output Using Auxiliary Functions

If the implementation of an I/O operator requires access to the private data of an object, the standard operators should delegate the actual work to auxiliary member functions. This technique also allows polymorphic read and write functions. This might look as follows:

```cpp
class Fraction {
...
public:
```
virtual void printOn (std::ostream& strm) const; // output
virtual void scanFrom (std::istream& strm); // input
...
};

std::ostream& operator << (std::ostream& strm, const Fraction& f)
{
    f.printOn (strm);
    return strm;
}

std::istream& operator >> (std::istream& strm, Fraction& f)
{
    f.scanFrom (strm);
    return strm;
}

A typical example is the direct access to the numerator and denominator of a fraction during input:

void Fraction::scanFrom (std::istream& strm)
{
    ...
    // assign values directly to the components
    num = n;
    denom = d;
}

If a class is not intended to be used as a base class, the I/O operators can be made friends of the class. However, note that this approach reduces the possibilities significantly when inheritance is used. Friend functions cannot be virtual; so as a result, the wrong function might be called. For example, if a reference to a base class actually refers to an object of a derived class and is used as an argument for the input operator, the operator for the base class is called. To avoid this problem, derived classes should not implement their own I/O operators. Thus, the implementation sketched previously is more general than the use of friend functions. It should be used as a standard approach, although most examples use friend functions instead.

13.12.4 User-Defined Operators Using Unformatted Functions

The I/O operators implemented in the previous subsections delegate most of the work to some predefined operators for formatted I/O. That is, operators << and >> are implemented in terms of the corresponding operators for more basic types. The I/O operators defined in the C++ standard library are defined differently. The common scheme used for these operators is as follows: First, with some preprocessing the stream is prepared for actual I/O. Then the actual I/O is done, followed by some postprocessing. This scheme should be used for your own I/O operators, too, to provide consistency for I/O operators. The 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. 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.
If an I/O operator uses a function for unformatted I/O or operates directly on the stream buffer, the first thing to be done should be the construction of a corresponding `sentry` object. The remaining processing should then depend 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`. Thus, I/O operators generally look like this:

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

The `sentry` object takes the stream `strm`, on which the preprocessing and postprocessing should be done, as the constructor argument.

The additional processing is used to arrange general tasks of the I/O operators. 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 can be used for corresponding locking.

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:

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

The following examples demonstrate this for class `Row`, which is used to represent the lines in a text processor or editor:

- The output operator writes a line by using the stream buffer's member function `sputn()`:

```cpp
std::ostream& operator<< (std::ostream& strm, const Row& row) {
    // ensure pre- and postprocessing
    std::ostream::sentry se(strm);
    if (se) {
        // perform the output
        strm.write(row.c_str(),row.len());
    }
    return strm;
}
```

- The input operator reads a line character-by-character in a loop. The argument `true` is passed to the constructor of the `sentry` object to avoid the skipping of whitespaces:

```cpp
std::istream& operator>> (std::istream& strm, Row& row) {
    /* ensure pre- and postprocessing
     * - true: Yes, don't ignore leading whitespaces
     */
    ...
}
std::istream::sentry se(strm,true);
if (se) {
    // perform the input
    char c;
    row.clear();
    while (strm.get(c) && c != '\n') {
        row.append(c);
    }
}

return strm;
}

Of course, it is also possible to use this framework even if functions do not use unformatted functions for their implementation but use I/O operators instead. However, using
basic_istream or basic_ostream members for reading or writing characters within code guarded by sentry objects is unnecessarily expensive. Whenever possible, the corresponding basic_streambuf should be used instead.

13.12.5 User-Defined Format Flags

When user-defined I/O operators are being written, it is often desirable to have formatting flags specific to these operators, probably set by using a corresponding manipulator. For example, it would be nice if the output operator for fractions, shown previously, could be configured to place spaces around the slash that separates numerator and denominator.
The stream objects support this by providing a mechanism to associate data with a stream. This mechanism can be used to associate corresponding data (for example, using a manipulator), and later retrieve the data. The class ios_base defines the two functions iword() and pword(),
each taking an int argument as the index, to access a specific long& or void* & respectively. The idea is that iword() and pword() access long or void* objects in an array of arbitrary size stored with a stream object. Formatting flags to be stored for a stream are then placed at the same index for all streams. The static member function xalloc() of the class ios_base is used to obtain an index that is not yet used for this purpose.
Initially, the objects accessed with iword() or pword() are set to 0. This value can be used to represent the default formatting or to indicate that the corresponding data was not yet accessed. Here is an example:

```cpp
// get index for new ostream data
static const int iword_index = std::ios_base::xalloc();

// define manipulator that sets this data
std::ostream& fraction_spaces (std::ostream& strm)
{
    strm.iword(iword_index) = true;
    return strm;
}

std::ostream& operator<< (std::ostream& strm, const Fraction& f)
{
    /* query the ostream data
     * - if true, use spaces between numerator and denominator
     * - if false, use no spaces between numerator and denominator
     */
    strm.iword(iword_index) = false;
    return strm;
}
```
This example uses a simple approach to the implementation of the output operator because the main feature to be exposed is the use of the function iword(). The format flag is considered to be a Boolean value that defines whether spaces between numerator and denominator should be written.

In the first line, the function ios_base::xalloc() is used to obtain an index that can be used to store the format flag. The result of this call is stored in a constant because it is never modified. The function fraction_spaces() is a manipulator that sets the int value that is stored at the index iword_index in the integer array associated with the stream strm to true. The output operator retrieves that value and writes the fraction according the value stored. If the value is false, the default formatting using no spaces is used. Otherwise, spaces are placed around the slash.

When iword() and pword() are used, references to int or void* objects are returned. These references stay valid only until the next call of iword() or pword() for the corresponding stream object or until the stream object is destroyed. Normally, the results from iword() and pword() should not be saved. It is assumed that the access is fast, although it is not required that the data is really represented by using an array.

The function copyfmt() copies all format information (see page 615). This includes the arrays accessed with iword() and pword(). This may pose a problem for the objects stored with a stream using pword(). For example, if a value is the address of an object, the address is copied instead of the object. If you copy only the address, it may happen that if the format of one stream is changed, the format of other streams would be affected. In addition, it may be desirable that an object associated with a stream using pword() is destroyed when the stream is destroyed. So, a deep copy rather than a shallow copy may be necessary for such an object.

A callback mechanism is defined by ios_base to support behavior, such as making a deep copy if necessary or deleting an object when destroying a stream. The function register_callback() can be used to register a function that is called if certain operations are performed on the ios_base object. It is declared as follows:

```cpp
namespace std {
  class ios_base { public:
    // kinds of callback events
denum event { erase_event, imbue_event, copyfmt_event }; // type of callbacks
dtypedef void (*event_callback) (event e, ios_base& strm,
in arg);
d// function to register callbacks
dvoid register_callback (event_callback cb, int arg);
...  
  
  
```
register_callback() takes a function pointer as the first argument and an int argument as the second. The int argument is passed as the third argument when a registered function is called. It can, for example, be used to identify an index for pword() to signal which member of the array has to be processed. The argument strm that is passed to the callback function is the ios_base object that caused the call to the callback function. The argument e identifies the reason why the callback function was called. The reasons for calling the callback functions are listed in Table 13.40.

<table>
<thead>
<tr>
<th>Event</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>ios_base::imbue_event</td>
<td>A locale is set with imbue()</td>
</tr>
<tr>
<td>ios_base::erase_event</td>
<td>The stream is destroyed or copyfmt() is used</td>
</tr>
<tr>
<td>ios_base::copy_event</td>
<td>copyfmt() is used</td>
</tr>
</tbody>
</table>

If copyfmt() is used, the callbacks are called twice for the object on which copyfmt() is called. First, before anything is copied, the callbacks are invoked with the argument erase_event to do all the cleanup necessary (for example, deleting objects stored in the pword() array). The callbacks called are those registered for the object. After the format flags are copied, which includes the list of callbacks from the argument stream, the callbacks are called again, this time with the argument copy_event. This pass can, for example, be used to arrange for deep copying of objects stored in the pword() array. Note that the callbacks are also copied and the original list of callbacks is removed. Thus, the callbacks invoked for the second pass are the callbacks just copied.

The callback mechanism is very primitive. It does not allow callback functions to be unregistered, except by using copyfmt() with an argument that has no callbacks registered. Also, registering a callback function twice, even with the same argument, results in calling the callback function twice. It is, however, guaranteed that the callbacks are called in the opposite order of registration. This has the effect that a callback function registered from within some other callback function is not called before the next time the callback functions are invoked.

### 13.12.6 Conventions for User-Defined Input/Output Operators

Several conventions that should be obeyed by the implementations of your own I/O operators have been presented. They correspond to the behavior that is typical for the predefined I/O operators. To summarize, these conventions are the following:

- The output format should allow an input operator that can read the data without loss of information. Especially for strings, this is close to impossible because a problem with spaces arises. A space character in the string cannot be distinguished from a space character between two strings.
- The current formatting specification of the stream should be taken into account when doing I/O. This applies especially to the width for writing.
- If an error occurs, an appropriate state flag should be set.
- The objects should not be modified in case of an error. If multiple data is read, the data should first be stored in auxiliary objects before the value of the object passed to the read operator is set.
- Output should not be terminated with a newline, mainly because it is otherwise impossible to write other objects on the same line.
- Even values that are too large should be read completely. After the read, a corresponding error flag should be set, and the value returned should be some meaningful value, such as the maximum value.
- If a format error is detected, no character should be read, if possible.
13.13 The Stream Buffer Classes

As mentioned in Section 13.2.1, the actual reading and writing is not done by the streams directly, but is delegated to stream buffers. This section describes how these classes operate. The discussion not only gives a deeper understanding of what is going on when I/O streams are used, but also provides the basis to define new I/O channels. Before going into the details of stream buffer operation, the public interface is presented for those only interested in using stream buffers.

13.13.1 User’s View of Stream Buffers

To the user of a stream buffer the class basic_streambuf is not much more than something that characters can be sent to or extracted from. Table 13.41 lists the public function for writing characters.

<table>
<thead>
<tr>
<th>Member Function</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>sputc(c)</td>
<td>Sends the character c to the stream buffer</td>
</tr>
<tr>
<td>sputn(s, n)</td>
<td>Sends n character from the sequence s to the stream buffer</td>
</tr>
</tbody>
</table>

The function sputc() returns traits_type::eof() in case of an error, where traits_type is a type definition in the class basic_streambuf. The function sputn() writes the number of characters specified by the second argument unless the stream buffer cannot consume them. It does not care about string termination characters. This function returns the number of characters written.

The interface to reading characters from a stream buffer is a little bit more complex (Table 13.42). This is because for input it is necessary to have a look at a character without consuming it. Also, it is desirable that characters can be put back into the stream buffer when parsing. Thus, the stream buffer classes provide corresponding functions.

<table>
<thead>
<tr>
<th>Member Function</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>in_avail()</td>
<td>Returns a lower bound on the characters available</td>
</tr>
<tr>
<td>sgetc()</td>
<td>Returns the current character without consuming it</td>
</tr>
<tr>
<td>sbumpc()</td>
<td>Returns the current character and consumes it</td>
</tr>
<tr>
<td>snextc()</td>
<td>Consumes the current character and returns the next character</td>
</tr>
<tr>
<td>sgetn(b, n)</td>
<td>Reads n characters and stores them in the buffer b</td>
</tr>
<tr>
<td>sputbackc(c)</td>
<td>Returns the character c to the stream buffer</td>
</tr>
<tr>
<td>sungetc()</td>
<td>Steps one step back to the previous character</td>
</tr>
</tbody>
</table>

The function in_avail() can be used to determine how many characters are at least available. This can be used, for example, to make sure that reading does not block when reading from the keyboard. However, there can be more characters available.

Until the stream buffer has reached the end of the stream, there is a current character. The function sgetc() is used to get the current character without moving on to the next character. The function sbumpc() reads the current character and moves on to next character, making this the new current character. The last function reading a single character, snextc() makes the next character the current one and then reads this character. All three functions return traits_type::eof() to indicate failure. The function sgetn() reads a sequence of characters into a buffer. The maximum number of characters to be read is passed as an argument. The function returns the number of characters read.

The two functions sputbackc() and sungetc() are used to move one step back, making the previous character the current one. The function sputbackc() can be used to replace the
previous character by some other character. These two functions should only be used with care. Often it is only possible to put back just one character. Finally, there are functions to access the imbeded locale object, to change the position, and to influence buffering. Table 13.43 lists these functions.

<table>
<thead>
<tr>
<th>Member Function</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>pubimbue(loc)</td>
<td>Imbues the stream buffer with the locale loc</td>
</tr>
<tr>
<td>getloc()</td>
<td>Returns the current locale</td>
</tr>
<tr>
<td>Pubseekpos(pos)</td>
<td>Repositions the current position to an absolute position</td>
</tr>
<tr>
<td>pubseekpos(pos, which)</td>
<td>Same with specifying the I/O direction</td>
</tr>
<tr>
<td>pubseekoff(offset, rpos)</td>
<td>Repositions the current position relative to another position</td>
</tr>
<tr>
<td>pubseekoff(offset, rpos, which)</td>
<td>Same with specifying the I/O direction</td>
</tr>
<tr>
<td>pubsetbuf(b, n)</td>
<td>Influences buffering</td>
</tr>
</tbody>
</table>

pubimbue() and getloc() are used for internationalization (see page 625). pubimbue() installs a new locale object in the stream buffer returning the previously installed locale object. getloc() returns the currently installed locale object.

The function pubsetbuf() is intended to provide some control over the buffering strategy of stream buffers. However, whether it is honored depends on the concrete stream buffer class. For example, it makes no sense to use pubsetbuf() for string stream buffers. Even for file stream buffers the use of this function is only portable if it is called before the first I/O operation is performed and if it is called as pubsetbuf(0,0) (that is, no buffer is to be used). This function returns 0 on failure and the stream buffer otherwise.

The functions pubseekoff() and pubseekpos() are used to manipulate the current position used for reading and/or writing. Which position is manipulated depends on the last argument, which is of type ios_base::openmode and which defaults to ios_base::in|ios_base::out if it is not specified. If ios_base::in is set, the read position is modified. Correspondingly, the write position is modified if ios_base::out is set.

The function pubseekpos() moves the stream to an absolute position specified as the first argument whereas the function pubseekoff() moves the stream relative to some other position. The offset is specified as the first argument. The position used as starting point is specified as the second argument and can be either ios_base::cur, ios_base::beg, or ios_base::end (see page 635 for details). Both functions return the position to which the stream was positioned or an invalid stream position. The invalid stream position can be detected by comparing the result with the object pos_type(off_type(-1))(pos_type and off_type are types for handling stream positions; see page 634). The current position of a stream can be obtained using pubseekoff():

sbuf.pubseekoff(0, std::ios::cur)

13.13.2 Stream Buffer Iterators

An alternative way to use a member function for unformatted I/O is to use the stream buffer iterator classes. These classes provide iterators that conform to input iterator or output iterator requirements and read or write individual characters from stream buffers. This fits character-level I/O into the algorithm library of the C++ standard library.

The template classes istreambuf_iterator and ostreambuf_iterator are used to read or to write individual characters from or to objects of type basic_streambuf. The classes are defined in the header <iterator> like this:
namespace std {
    template <class charT,
              class traits = char_traits<charT> >
    istreambuf_iterator;
    template <class charT,
              class traits = char_traits<charT> >
    ostreambuf_iterator;
}

These iterators are special forms of stream iterators, which are described in Section 7.4.3. The only difference is that their elements are characters.

Output Stream Buffer Iterators

Here is how a string can be written to a stream buffer using an ostreambuf_iterator:

```cpp
// create iterator for buffer of output stream cout
std::ostreambuf_iterator<char> bufWriter(std::cout);

std::string hello("hello, world\n");
std::copy(hello.begin(), hello.end(),
          // source: string
          bufWriter);
```

The first line of this example constructs an output iterator of type ostreambuf_iterator from the object cout. Instead of passing the output stream you could also pass a pointer to the stream buffer directly. The remainder constructs a string object and copies the characters in this object to the constructed output iterator.

Table 13.44 lists all operations of output stream buffer iterators. The implementation is similar to ostream iterators (see page 278). In addition, you can initialize the iterator with a buffer and you can call failed() to query whether the iterator is able to write. If any prior writing of a character failed, failed() yields true. In this case, any writing with operator = has no effect.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>ostreambuf_iterator&lt;char&gt;(ostream)</td>
<td>Creates an output stream buffer iterator for ostream</td>
</tr>
<tr>
<td>ostreambuf_iterator&lt;char&gt;(buffer_ptr)</td>
<td>Creates an output stream buffer iterator for the buffer to which buffer_ptr refers</td>
</tr>
<tr>
<td>*iter</td>
<td>No-op (returns iter)</td>
</tr>
<tr>
<td>iter = c</td>
<td>Writes character c to the buffer by calling sputc(c) for it</td>
</tr>
<tr>
<td>++iter</td>
<td>No-op (returns iter)</td>
</tr>
<tr>
<td>iter++</td>
<td>No-op (returns iter)</td>
</tr>
<tr>
<td>failed()</td>
<td>Returns whether the output stream iterator is not able to write anymore</td>
</tr>
</tbody>
</table>

Input Stream Buffer Iterators

Table 13.45 lists all operations of input stream buffer iterators. The implementation is similar to istream iterators (see page 280). In addition, you can initialize the iterator with a buffer, and a member function, equal(), is provided, which returns whether two input stream buffer iterators
are equal. Two input stream buffer iterators are equal when they are both end-of-stream iterators or when neither is an end-of-stream iterator.

What is somewhat obscure is what it means for two objects of type `istreambuf_iterator` to be equivalent: Two `istreambuf_iterator` objects are equivalent if both iterators are end-of-stream iterators or if neither of them is an end-of-stream iterator (whether the output buffer is the same doesn't matter). One possibility to get an end-of-stream iterator is to construct an iterator with the default constructor. In addition, an `istreambuf_iterator` becomes an end-of-stream iterator when an attempt is made to advance the iterator past the end of the stream (in other words, if `sbumpc()` returns `traits_type::eof()`). This behavior has two major implications:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>istreambuf_iterator&lt;char&gt;()</code></td>
<td>Creates an end-of-stream iterator</td>
</tr>
<tr>
<td><code>istreambuf_iterator&lt;char&gt;(istream)</code></td>
<td>Creates an input stream buffer iterator for <code>istream</code> and might read the first character using <code>sgetc()</code></td>
</tr>
<tr>
<td><code>istreambuf_iterator&lt;char&gt;(buffer_ptr)</code></td>
<td>Creates an input stream buffer iterator for the buffer to which <code>buffer_ptr</code> refers and might read the first character using <code>sgetc()</code></td>
</tr>
<tr>
<td><code>*iter</code></td>
<td>Returns the actual character, read with <code>sgetc()</code> before (reads the first character if not done by the constructor)</td>
</tr>
<tr>
<td><code>++iter</code></td>
<td>Reads the next character with <code>sbumpc()</code> and returns its position</td>
</tr>
<tr>
<td><code>iter++</code></td>
<td>Reads the next character with <code>sbumpc()</code> but returns an iterator for the previous character</td>
</tr>
<tr>
<td><code>iter1.equal(iter2)</code></td>
<td>Returns whether both iterators are equal</td>
</tr>
<tr>
<td><code>iter1 != iter2</code></td>
<td>Tests <code>iter1</code> and <code>iter2</code> for inequality</td>
</tr>
</tbody>
</table>

1. A range from the current position in a stream to the end of the stream is defined by the two iterators `istreambuf_iterator<charT,traits>(stream)` (for the current position) and `istreambuf_iterator<charT,traits>()` (for the end of the stream), where `stream` is of type `basic_istream<charT,traits>` or `basic_streambuf<charT,traits>`.

2. It is not possible to create subranges using `istreambuf_iterators`.

Example Use of Stream Buffer Iterators

The following example is the classic filter framework that simply writes all read characters with stream buffer iterators. It is a modified version of the example on page 611:

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

int main()
{
    // input stream buffer iterator for cin
    istreambuf_iterator<char> inpos(cin);
    // io/charcat2.cpp
    #include <iostream>
    #include <iterator>
    using namespace std;
    int main()
    {
```

```cpp
```
// end-of-stream iterator
istreambuf_iterator<char> endpos;

// output stream buffer iterator for cout
ostreambuf_iterator<char> outpos(cout);

// while input iterator is valid
while (inpos != endpos) {
    *outpos = *inpos; // assign its value to the output iterator
    ++inpos;
    ++outpos;
}

13.13.3 User-Defined Stream Buffers

Stream buffers are buffers for I/O. Their interface is defined by class \texttt{basic\_streambuf<>}. For the character types \texttt{char} and \texttt{wchar\_t}, the specializations \texttt{streambuf} and \texttt{wstreambuf}, respectively, are predefined. These classes are used as base classes when implementing the communication over special I/O channels. However, doing this requires an understanding of the stream buffer's operation.

The central interface to the buffers is formed by three pointers for each of the two buffers. The pointers returned from the functions \texttt{eback()}, \texttt{gptr()}, and \texttt{egptr()} form the interface to the read buffer. The pointers returned from the functions \texttt{pbase()}, \texttt{pptr()}, and \texttt{epptr()} form the interface to the write buffer. These pointers are manipulated by the read and write operations, which may result in corresponding reactions in the corresponding read or write channel. The exact operation is examined separately for reading and writing.

User-Defined Output Buffers

A buffer used to write characters is maintained with three pointers that can be accessed by the three functions \texttt{pbase()}, \texttt{pptr()}, and \texttt{epptr()} (Figure 13.4). Here is what these pointers represent:

\begin{figure}
\centering
\includegraphics[width=0.8\textwidth]{figure.png}
\caption{The Interface to the Output Buffer}
\end{figure}

1. \texttt{pbase()} ("put base") is the beginning of the output buffer.
2. \texttt{pptr()} ("put pointer") is the current write position.
3. \texttt{epptr()} ("end put pointer") is the end of the output buffer. This means that \texttt{epptr()} points to one past the last character that can be buffered.
The characters in the range from `pbase()` to `pptr()` (not including the character pointed to by `pptr()`) are already written but not yet transported (flushed) to the corresponding output channel.

A character is written using the member function `sputc()`. This character is copied to the current write position if there is a spare write position. Then the pointer to the current write position is incremented. If the buffer is full (`pptr() == epptr()`), the contents of the output buffer are sent to the corresponding output channel. This is done by calling the virtual function `overflow()`. This function is effectively responsible for the actual sending of the characters to some “external representation” (which may actually be internal, as in the case of string streams). The implementation of `overflow()` in the base class `basic_streambuf` only returns end-of-file, which indicates that no more characters could be written.

The member function `sputn()` can be used to write multiple characters at once. This function delegates the work to the virtual function `xsputn()`, which can be implemented for more efficient writing of multiple characters. The implementation of `xsputn()` in class `basic_streambuf` basically calls `sputc()` for each character. Thus, overriding `xsputn()` is not necessary. However, often, writing multiple characters can be implemented more efficiently than writing characters one at a time. Thus, this function can be used to optimize the processing of character sequences.

Writing to a stream buffer does not necessarily involve using the buffer. Instead, the characters can be written as soon as they are received. In this case, the value 0 or NULL has to be assigned to the pointers that maintain the write buffer. The default constructor does this automatically.

With this information, the following example of a simple stream buffer can be implemented. This stream buffer does not use a buffer. Thus, the function `overflow()` is called for each character. Implementing this function is all that is necessary:

```cpp
// io/outbuf1.hpp
#include <streambuf>
#include <locale>
#include <cstdio>

class outbuf : public std::streambuf
{
  protected:
    /* central output function
     * - print characters in uppercase mode
     */
    virtual int_type overflow (int_type c) {
      if (c != EOF) {
        // convert lowercase to uppercase
        c = std::toupper(c, getloc());

        // and write the character to the standard output
        if (putchar(c) == EOF) {
          return EOF;
        }
      }
      return c;
    }
};
```

In this case, each character sent to the stream buffer is written using the C function `putchar()`. However, before the character is written it is turned into an uppercase character using `toupper()`.
toupper() (see page 718). The function getloc() is used to get the locale object that is associated with the stream buffer (see also page 626). In this example, the output buffer is implemented specifically for the character type char (streambuf is the specialization of basic_streambuf<> for the character type char). If other character types are used, you have to implement this function using character traits, which are introduced in Section 14.1.2. In this case, the comparison of c with end-of-file looks different. traits::eof() has to be returned instead of EOF, and if the argument c is EOF, the value traits::not_eof (c) should be returned (where traits is the second template argument to basic_streambuf). This might look as follows:

```cpp
#include <iostream>
#include "outbuf1.hpp"

int main()
{
  outbuf ob; //create special output buffer
  std::ostream out (&ob); // initialize output stream with that output buffer

  out << "31 hexadecimal: " << std::hex << 31 << std::endl;
}
```

Using this stream buffer in the following program:

```cpp
#include <iostream>
#include "outbuf1.hpp"

int main()
{
  outbuf ob; //create special output buffer
  std::ostream out (&ob); // initialize output stream with that output buffer

  out << "31 hexadecimal: " << std::hex << 31 << std::endl;
}
```
produces the following output:

31 HEXADECIMAL: 1F

The same approach can be used to write to other arbitrary destinations. For example, the
constructor of a stream buffer may take a file descriptor, the name of a socket connection, or two
other stream buffers used for simultaneous writing to initialize the object. Writing to the
corresponding destination requires only that \texttt{overflow()} be implemented. In addition, the
function \texttt{xspuon()} should also be implemented to make writing to the stream buffer more
efficient.

For convenient construction of the stream buffer, it is also reasonable to implement a special
stream class that mainly passes the constructor argument to the corresponding stream buffer.
The next example demonstrates this. It defines a stream buffer class initialized with a file
descriptor, to which characters are written with the function \texttt{write()} (a low-level I/O function
used on UNIX- like operating systems). In addition, a class derived from \texttt{ostream} is defined that
maintains such a stream buffer, to which the file descriptor is passed:

```cpp
// io/outbuf2.hpp

#include <iostream>
#include <streambuf>
#include <cstdio>

extern "C" {
  int write (int fd, const char* buf, int num);
}

class fdoutbuf : public std::streambuf {
  protected:
    int fd;    // file descriptor
  public:
    // constructor
    fdoutbuf (int_fd) : fd(_fd) {
    }
  protected:
    // write one character
    virtual int_type overflow (int_type c) {
      if (c != EOF) {
        char z = c;
        if (write (fd, &z, 1) != 1) {
          return EOF;
        }
      }
      return c;
    }
    // write multiple characters
    virtual
    std::streamsize xsputn (const char* s,
                            std::streamsize num) {
      return write(fd,s,num);
    }
  };
  class fdostream : public std::ostream {
```
protected:
    fdoutbuf buf;
public:
    fdostream (int fd) : buf(fd), std::ostream(&buf) {
    }
};

This stream buffer also implements the function `xsputn()` to avoid calling `overflow()` for each character if a character sequence is sent to this stream buffer. This function writes the whole character sequence with one call to the file identified by the file descriptor `fd`. The function `xsputn()` returns the number of characters written successfully. Here is a sample application:

```cpp
// io/outbuf2.cpp
#include <iostream>
#include "outbuf2.hpp"

int main()
{
    fdostream out(1); // stream with buffer writing to file descriptor 1
    out << "31 hexadecimal: " << std::hex << 31 << std::endl;
}
```

This program creates a output stream that is initialized with the file descriptor 1. This file descriptor, by convention, identifies the standard output channel. Thus, in this example the characters are simply printed. If some other file descriptor is available (for example, for a file or a socket), it can also be used as the constructor argument.

To implement a stream buffer that really buffers, the write buffer has to be initialized using the function `setp()`. This is demonstrated by the next example:

```cpp
// io/outbuf3.hpp
#include <cstdio>
#include <streambuf>
extern "C" {
    int write (int fd, const char* buf, int num);
}

class outbuf : public std::streambuf {
protected:
    static const int bufferSize = 10;  // size of data buffer
    char buffer [bufferSize];         // data buffer

public:
    /* constructor
     * - initialize data buffer
     * - one character less to let the bufferSize th character
     * cause a call of overflow()
     */
    outbuf() {
        setp (buffer, buffer+(bufferSize-1));
    }
```
/* destructor
 * - flush data buffer
 */
virtual ~outbuf() {
 sync();
}

protected:
// flush the characters in the buffer
int flushBuffer() {
    int num = pptr()-pbase();
    if (write (1, buffer, num) != num) {
        return EOF;
    }
    pbump (-num);     // reset put pointer accordingly
    return num;
}

/* buffer full
 * - write c and all previous characters
 */
virtual int_type overflow (int_type c) {
    if (c != EOF) {
        // insert character into the buffer
        *pptr() = c;
        pbump(1);
    }
    // flush the buffer
    if (flushBuffer() == EOF) {
        // ERROR
        return EOF;
    }
    return c;
}

/* synchronize data with file/destination
 * - flush the data in the buffer
 */
virtual int sync() {
    if (flushBuffer() == EOF) {
        // ERROR
        return -1;
    }
    return 0;
}

The constructor initializes the write buffer with setp():

setp (buffer, buffer+(size-1));

The write buffer is set up such that overflow() is already called when there is still room for one character. If overflow() is not called with EOF as the argument, the corresponding character can be written to the write position because the pointer to the write position is not increased
beyond the end pointer. After the argument to `overflow()` is placed in the write position, the whole buffer can be emptied.

The member function `flushBuffer()` does exactly this. It writes the characters to the standard output channel (file descriptor 1) using the function `write()`. The stream buffer's member function `pbump()` is used to move the write position back to the beginning of the buffer.

The function `overflow()` inserts the character that caused the call of `overflow()` into the buffer if it is not EOF. Then, `pbump()` is used to advance the write position to reflect the new end of the buffered characters. This moves the write position beyond the end position (`epptr()`) temporarily.

This class also features the virtual function `sync()` that is used to synchronize the current state of the stream buffer with the corresponding storage medium. Normally, all that needs to be done is to flush the buffer. For the unbuffered versions of the stream buffer, overriding this function was not necessary because there was no buffer to be flushed.

The virtual destructor ensures that data is written that is still buffered when the stream buffer is destroyed.

These are the functions that are overridden for most stream buffers. If the external representation has some special structure, overriding additional functions may be useful. For example, the functions `seekoff()` and `seekpos()` may be overridden to allow manipulation of the write position.

**User-Defined Input Buffers**

The input mechanism works basically the same as the output mechanism. However, for input there is also the possibility of undoing the last read. The functions `sungetc()` (called by `unget()` of the input stream) or `sputbackc()` (called by `putback()` of the input stream) can be used to restore the stream buffer to its state before the last read. It is also possible to read the next character without moving the read position beyond this character. Thus, you must override more functions to implement reading from a stream buffer than is necessary to implement writing to a stream buffer.

A stream buffer maintains a read buffer with three pointers that can be accessed through the member function `eback()`, `gptr()` and `egptr()` (Figure 13.5):

**Figure 13.5. The Interface for Reading from Stream Buffers**

1. eback() ("end back") is the beginning of the input buffer, or (this is where the name comes from) the end of the putback area. The character can only be put back up to this position without taking special action.
2. gptr() ("get pointer") is the current read position.
3. egptr() ("end get pointer") is the end of the input buffer.

The characters between the read position and the end position have been transported from the external representation to the program's memory, but they still await processing by the program.
Single characters can be read using the function \texttt{sgetc()} or \texttt{sbumpc()}. These two functions differ in that the read pointer is incremented by \texttt{sbumpc()}, but not by \texttt{sgetc()}. If the buffer is read completely (\texttt{gptr()} == \texttt{egptr()}), there is no character available and the buffer has to be refilled. This is done by a call of the virtual function \texttt{underflow()}. This function is responsible for the reading of data. The function \texttt{sbumpc()} calls the virtual function \texttt{uflow()} instead, if no characters are available. The default implementation of \texttt{uflow()} is to call \texttt{underflow()} and then increment the read pointer. The default implementation of \texttt{underflow()} in the base class \texttt{basic_streambuf} is to return EOF. This means it is impossible to read characters with the default implementation.

The function \texttt{sgetn()} is used for reading multiple characters at once. This function delegates the processing to the virtual function \texttt{xsgetn()}. The default implementation of \texttt{xsgetn()} simply extracts multiple characters by calling \texttt{sbumpc()} for each character. Like the function \texttt{xsputn()} for writing, \texttt{xsgetn()} can be implemented to optimize the reading of multiple characters.

For input it is not sufficient just to override one function as it is the case of output. Either a buffer has to be set up, or at the very least \texttt{underflow()} and \texttt{uflow()} have to implemented. This is because \texttt{underflow()} does not move past the current character, but \texttt{underflow()} may be called from \texttt{sgetc()}. Moving on to the next character has to be done using buffer manipulation or using a call to \texttt{uflow()}. In any case, \texttt{underflow()} has to be implemented for any stream buffer capable of reading characters. If both \texttt{underflow()} and \texttt{uflow()} are implemented, there is no need to set up a buffer.

A read buffer is set up with the member function \texttt{setg()}, which takes three arguments in this order:

1. A pointer to the beginning of the buffer (\texttt{eback()})
2. A pointer to the current read position (\texttt{gptr()})
3. A pointer to the end of the buffer (\texttt{egptr()})

Unlike \texttt{setp()}, \texttt{setg()} takes three arguments. This is necessary to be able to define the room for storing characters that are put back into the stream. Thus, when the pointers to the read buffer are being set up, it is reasonable to have some characters (at least one) that are already read but still stored in the buffer.

As mentioned, characters can be put back into the read buffer using the functions \texttt{sputbackc()} and \texttt{sungetc()}. \texttt{sputbackc()} gets the character to be put back as its argument and ensures that this character was indeed the character read. Both functions decrement the read pointer, if possible. Of course, this only works as long as the read pointer is not at the beginning of the read buffer. If you attempt to put a character back after the beginning of the buffer is reached, the virtual function \texttt{pbackfail()} is called. By overriding this function you can implement a mechanism to restore the old read position even in this case. In the base class \texttt{basic_streambuf}, no corresponding behavior is defined. Thus, in practice, it is not possible to go back an arbitrary number of characters. For streams that do not use a buffer, the function \texttt{pbackfail()} should be implemented because it is generally assumed that at least one character can be put back into the stream.

If a new buffer was just read, another problem arises: Not even one character can be put back if the old data is not saved in the buffer. Thus, the implementation of \texttt{underflow()} often moves the last few characters (for example, four characters) of the current buffer to the beginning of the buffer and appends the newly read characters thereafter. This allows some characters to be moved back before \texttt{pbackfail()} is called.

The following example demonstrates how such an implementation might look. In the class \texttt{inbuf}, an input buffer with ten characters is implemented. This buffer is split into a maximum of four characters for the putback area and six characters for the "normal" input buffer:

```cpp
// io/inbuf1.hpp
```
```cpp
#include <cstdio>
#include <cstring>
#include <streambuf>

extern "C" {
    int read (int fd, char* buf, int num);
}

class inbuf : public std::streambuf {
protected:
    /* data buffer:
     * - at most, four characters in putback area plus
     * - at most, six characters in ordinary read buffer
     */
    static const int bufferSize = 10; // size of the data buffer
    char buffer[bufferSize]; // data buffer

public:
    /* constructor
     * - initialize empty data buffer
     * - no putback area
     * => force underflow()
     */
    inbuf() {
        setg (buffer+4, // beginning of putback area
              buffer+4, // read position
              buffer+4); // end position
    }

protected:
    // insert new characters into the buffer
    virtual int_type underflow() {
        // is read position before end of buffer?
        if (gptr() < egptr()) {
            return *gptr();
        }

        /* process size of putback area
         * - use number of characters read
         * - but at most four
         */
        int numPutback;
        numPutback = gptr() - eback();
        if (numPutback > 4) {
            numPutback = 4;
        }

        /* copy up to four characters previously read into
         * the putback buffer (area of first four characters)
         */
        std::memcpy (buffer+(4-numPutback), gptr()-numPutback,
                     numPutback);

        // read new characters
        int num;
        num = read (0, buffer+4, bufferSize-4);
    }
```

if (num <= 0) {
    // ERROR or EOF
    return EOF;
}

// reset buffer pointers
setg (buffer+(4-numPutback), // beginning of putback

area
    buffer+4, // read position
    buffer+4+num); // end of buffer

// return next character
return *gptr();
};

The constructor initializes all pointers so that the buffer is completely empty (Figure 13.6). If a character is read from this stream buffer, the function `underflow()` is called. This function is always used by this stream buffer to read the next characters. It starts by checking for read characters in the input buffer. If characters are present, they are moved to the putback area using the function `memcpy()`. These are, at most, the last four characters of the input buffer. Then POSIX's low-level I/O function `read()` is used to read the next character from the standard input channel. After the buffer is adjusted to the new situation, the first character read is returned.

Figure 13.6. Get Buffer After Initialization

For example, if the characters 'H', 'a', 'l', 'l', 'o', and 'w' are read by the first call to `read()`, the state of the input buffer changes, as shown in Figure 13.7. The putback area is empty because the buffer was filled for the first time, and there are no characters yet that can be put back.

Figure 13.7. Get Buffer After Reading H a l l o w
After these characters are extracted, the last four characters are moved into the putback area and new characters are read. For example, if the characters 'e', 'e', 'n', and '\n' are read by the next call of `read()` the result is as shown in Figure 13.8.

**Figure 13.8. Get Buffer After Reading Four More Characters**

Here is an example of the use of this stream buffer:

```cpp
#include <iostream>
#include "inbuf1.hpp"

int main()
{
    inbuf ib; // create special stream buffer
    std::istream in(&ib); // initialize input stream with that buffer

    char c;
    for (int i=1; i<=20; i++) {
        // read next character (out of the buffer)
        in.get(c);

        // print that character (and flush)
        std::cout << c << std::flush;

        // after eight characters, put two characters back into the stream
        if (i == 8) {
            in.unget();
        }
    }
    std::cout << std::endl;
}
```

The program reads characters in a loop and writes them out. After the eighth character is read, two characters are put back. Thus, the seventh and eighth characters are printed twice.

### 13.14 Performance Issues
This section specifically addresses issues that focus on performance. In general the stream classes should be pretty efficient, but performance can be improved further in applications in which I/O is performance critical.

One performance issue was mentioned in Section 13.2.3, already: You should only include those headers that are necessary to compile your code. In particular, you should avoid including `<iostream>` if the standard stream objects are not used.

### 13.14.1 Synchronization with C’s Standard Streams

By default, the eight C++ standard streams (the four narrow character streams `cin`, `cout`, `cerr`, and `clog`, and their wide-character counterpart) are synchronized with the corresponding files from the C standard library (`stdin`, `stdout`, and `stderr`). By default `clog` and `wclog` use the same stream buffer as `cerr` and `wcerr` respectively. Thus, they are also synchronized with `stderr` by default, although there is no direct counterpart in the C standard library.

Depending on the implementation, this synchronization might imply some often unnecessary overhead. For example, if the standard C++ streams are implemented using the standard C files, this basically inhibits buffering in the corresponding stream buffers. However, the buffer in the stream buffers is necessary for some optimizations especially during formatted reading (see Section 13.14.2). To allow switching to a better implementation, the static member function `sync_with_stdio()` is defined for the class `ios_base` (Table 13.46).

<table>
<thead>
<tr>
<th>Static Function</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sync_with_stdio()</code></td>
<td>Returns whether the standard stream objects are synchronized with standard C streams</td>
</tr>
<tr>
<td><code>sync_with_stdio(false)</code></td>
<td>Disables the synchronization of C++ and C streams provided it is called before any I/O</td>
</tr>
</tbody>
</table>

The function `sync_with_stdio()` takes an optional Boolean value as argument that determines whether the synchronization with the standard C streams should be turned on. Thus, to turn the synchronization off you have to pass false as the argument:

```cpp
std::ios::sync_with_stdio(false);  // disable synchronization
```

Note that you have to disable the synchronization before any other I/O operation. Calling this function after any I/O has occurred results in implementation-defined behavior. The function returns the previous value with which the function was called. If not called before, it always returns true to reflect the default setup of the standard streams.

### 13.14.2 Buffering in Stream Buffers

Buffering I/O is important for efficiency. One reason for this is that system calls are, in general, relatively expensive and it pays to avoid them if possible. There is, however, another more subtle reason in C++ for doing buffering in stream buffers, at least for input: The functions for formatted I/O use stream buffer iterators to access the streams, and operating on stream buffer iterators is slower than operating on pointers. The difference is not that big, but it is sufficient to justify improved implementations for frequently used operations like formatted reading of numeric values. However, for such improvements it is essential that stream buffers are buffered.

Thus, all I/O is done using stream buffers, which implement a mechanism for buffering. However, it is not sufficient to rely solely on this buffering because there are three aspects that conflict with effective buffering:
1. It is often simpler to implement stream buffers without buffering. If the corresponding streams are not used frequently or are only used for output (for output the difference between stream buffer iterators and pointers is not as bad as for input; the main problem is comparing stream buffer iterators), buffering is probably not that important. However, for stream buffers that are used extensively, buffering should definitely be implemented.

2. The flag `unitbuf` causes output streams to flush the stream after each output operation. Correspondingly, the manipulators `flush` and `endl` also flush the stream. For the best performance all three should probably be avoided. However, when writing to the console, for example, it is probably still reasonable to flush the stream after writing complete lines. If you are stuck with a program that makes heavy use of `unitbuf`, `flush`, or `endl`, you might consider using a special stream buffer that does not use `sync()` to flush the stream buffer but uses some other function that is called when appropriate.

3. Tying streams with the `tie()` function (see Section 13.10.1) also results in additional flushing of streams. Thus, streams should only be tied if it is really necessary.

When implementing new stream buffers, it may be reasonable to implement them without buffering first. Then, if the stream buffer is identified as a bottleneck, it is still possible to implement buffering without affecting anything in the remainder of the application.

### 13.14.3 Using Stream Buffers Directly

All member functions of the class `basic_istream` and `basic_ostream` that read or write characters operate according to the same schema: First, a corresponding `sentry` object is constructed, then the actual operation is performed. The construction of the `sentry` object results in flushing of potentially tied objects, skipping of whitespace (for input only), and implementation-specific operations like locking in multithreaded environments (see Section 13.12.4).

For unformatted I/O, most of the operations are normally useless anyway. Only the locking operation might be useful if the streams are used in multithreaded environments (note that the C++ standard does not address multithreading). Thus, when doing unformatted I/O it is normally much better to use stream buffers directly.

To support this behavior, you can use operators `<<` and `>>` with stream buffers as follows:

- By passing a pointer to a stream buffer to operator `<<`, you can output all input of its device. This is probably the fastest way to copy files by using C++ I/O streams. For example:

```
#include <iostream>

int main()
{
    // copy all standard input to standard output
    std::cout << std::cin.rdbuf();
}
```

Here, `rdbuf()` yields the buffer of `cin` (see page 638). Thus, the program copies all standard input to standard output.
By passing a pointer to a stream buffer to operator `>>,` you can read directly into a stream buffer.

For example, you could also copy all standard input to standard output in the following way:

```cpp
// io/copy2.cpp
#include <iostream>

int main()
{
  // copy all standard input to standard output
  std::cin >> std::cout.rdbuf();
}
```

Note that you have to clear the flag `skipws`. Otherwise, leading whitespace of the input is skipped.

Even for formatted I/O it may be reasonable to use stream buffers directly. For example, if lots of numeric values are read in a loop, it is sufficient to construct just one `sentry` object that exists for the whole time the loop is executed. Then, within the loop, whitespace is skipped manually (using the `ws` manipulator would also construct a `sentry` object) and then the facet `num_get` (see Section 14.4.1) is used for reading the numeric values directly. Note that a stream buffer has no error state of its own. It also has no knowledge of the input or output stream that might connect to it. So, inside of:

```cpp
//copy contents of in to out
out « in.rdbuf();
```

there is no way to change the error state of `in` due to a failure of end-of-file.