STREAM INPUT/OUTPUT

Table of Contents

Section 1. Stream Input/Output

1.1 How to Read this Section

This section is an introduction to C++ stream input and output. Section 1.2 explains the iostreams facility, how it works in principle, and how it should be used. This section should be read by anyone who needs basic information on iostreams. For readers who require deeper understanding, the section also gives an overview of the iostream architecture, its components, and class hierarchy. It is not necessary to read this part in order to start using predefined iostreams as explained in Section 1.3.

Sections 1.3 to 1.6 explain the basic operation of iostreams. These sections can be read sequentially, or used as a reference for certain topics. Read sequentially, they provide you with basic knowledge needed for using iostreams. Used as a reference, they provide answers to questions like: How do I check for iostreams errors? and, How do I work with file streams?

Sections 1.7 to 1.9 explain more advanced usage of the iostreams.

Sections 1.10 to 1.13 explain how iostreams can be extended.

Section 1.14 describes the main differences between the Standard C++ Library iostreams and traditional iostreams.

Section 1.15 describes the main differences between the Standard C++ Library iostreams and the Rogue Wave's implementation of iostreams in its own*Standard C++ Library*. It points out features that are specific to the Rogue Wave implementation.

The Appendix describes standardization issues that are still open at the time of this writing and influence the content of this document.

1.1.1 Terminology

The Standard C++ Library consists mostly of class and function templates. Abbreviations for these templates are used throughout the *User'sGuide*. For example, `fstream` stands for `template <class charT, class traits> class basic_fstream`. A slightly more succinct notation for a class template is also frequently used: `basic_fstream <charT, traits>`.

You will also find certain contrived technical terms. For example, *file stream* stands for the abstract notion of the file stream class templates; `badbit` stands for the state flag `ios_base::badbit`.

1.1.2 Status of this document

Sections 1.2 through 1.6 are complete. The remaining sections will be completed for the final release of the *User's Guide*

1.2 The Architecture of Iostreams

This section will introduce you to iostreams: what they are, how they work, what kinds of problems they help solve, and how they are structured. Section 1.2.4 provides an overview of the class templates in iostreams. If you want to skip over the software architecture of iostreams, please go on to Section 1.3 on formatted input/output.

1.2.1 What are the Standard Iostreams?

The Standard C++ Library includes classes for text stream input/output. Before the current ANSI/ISO standard, most C++ compilers were delivered with a class library commonly known as the *iostreams* library. In this section, we refer to the C++ iostreams library as *traditional iostreams*, in contrast to the *standard iostreams* that are now part of the ANSI/ISO Standard C++ Library. The standard iostreams are to some extent compatible with the traditional iostreams, in that the overall architecture and the most commonly used interfaces are retained. Section 1.14 describes the incompatibilities in greater detail.

We can compare the standard iostreams not only with the traditional C++ iostreams library, but also with the I/O support in the Standard C Library. Many former C programmers still prefer the input/output functions offered by the C library, often referred to as *C Stdio*. Their familiarity with the C library is justification enough for using the C Stdio instead of C++ iostreams, but there are other reasons as well. For example, calls to the C functions `printf()` and `scanf()` are admittedly more concise with C Stdio. However, C Stdio has drawbacks, too, such as type insecurity and inability to extend consistently for user-defined classes. We'll discuss these in more detail in the following sections.

1.2.1.1 Type Safety

Let us compare a call to stdio functions with the use of standard iostreams. The stdio call reads as follows:

```
int i = 25;
char name[50] = "Janakiraman";
fprintf(stdout, "%d %s", i, name);
```

It correctly prints: `25 Janakiraman`.

But what if we inadvertently switch the arguments to `fprintf`? The error will be detected no sooner than run time. Anything can happen, from peculiar output to a system crash. This is not the case with the standard iostreams:

```
cout << i << ' ' << name << '\n';
```

Since there are overloaded versions of the shift operator `operator<<()`, the right operator will always be called. The function `cout << i` calls `operator<<(int)`, and `cout << name` calls `operator<<(char*)`. Hence, the standard iostreams are typesafe.

1.2.1.2 Extensibility to New Types

Another advantage of the standard iostreams is that user-defined types can be made to fit in seamlessly. Consider a type `Pair` that we want to print:

```
struct Pair { int x; string y; }
```

All we need to do is overload `operator<<()` for this new type `Pair`, and we can output pairs this way:

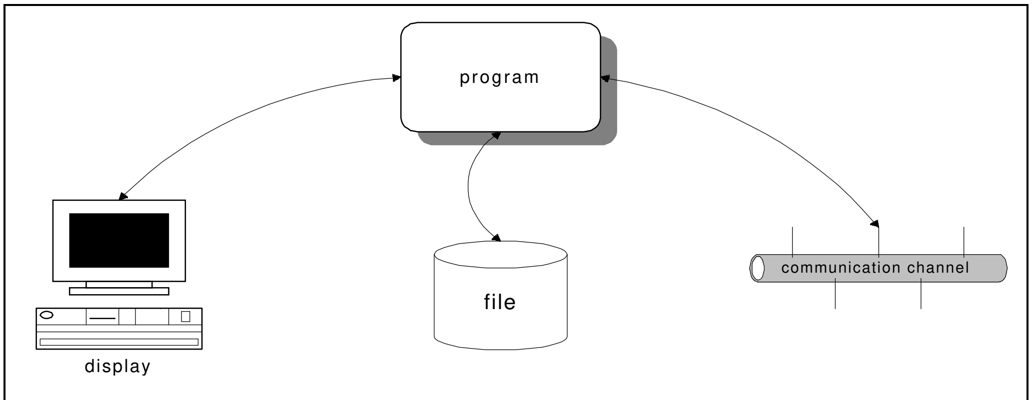
```
Pair p(5, "May");
cout << p;
```

The corresponding `operator<<()` can be implemented as:

```
basic_ostream<char>& operator<<(basic_ostream<char>& o, Pair& p)
{ return o << p.x << ' ' << p.y; }
```

1.2.2 How do the Standard Iostreams Work?

The main purpose of the standard iostreams is to serve as a tool for input and output of texts. Generally, input and output are the transfer of data between a program and any kind of external



IOStreams supports data transfer between a program and external devices.

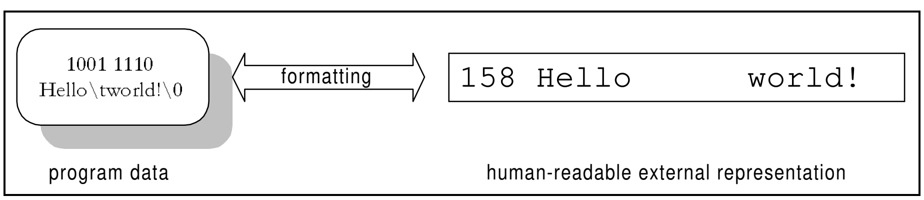
device, as illustrated in the figure below:

Figure 1: Data transfer supported by iostreams

The internal representation of such data is meant to be convenient for data processing in a program. On the other hand, the external representation can vary quite a bit: it might be a display in human-readable form, or a portable data exchange format. The intent of a representation, such as conserving space for storage, can also influence the representation.

T*ext I/O* involves the external representation of a sequence of characters. Every other case involves *binary I/O*. Traditionally, iostreams are used for text processing. Such text processing via iostreams involves two processes: *formatting* and *code conversion* .

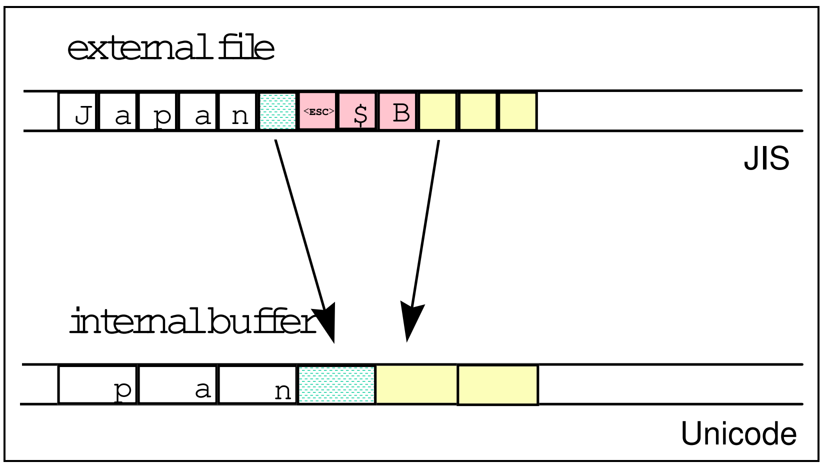
Formatting is the transformation from a byte sequence representing internal data into a human-readable character sequence; for example, from a floating point number, or an integer value held in a variable, into a sequence of digits. The figure below illustrates the



formatting process:

Figure 2: Formatting program data

Code conversion is the process of translating one character representation into another; for example, from wide characters held internally to a sequence of multibyte characters for external usage. Wide characters all have the same size, and thus are convenient for internal data processing. Multibyte characters have different sizes and are stored more compactly. They are typically used for data transfer, or for storage on external devices such as files. The figure



below illustrates the conversion process:

Figure 3: Code conversion between multibytes and wide characters

1.2.2.1 The Iostream Layers

The iostreams facility has two layers: one that handles formatting, and another that handles code conversion and transport of characters to and from the external device. The layers communicate via a buffer, as illustrated in the following figure:

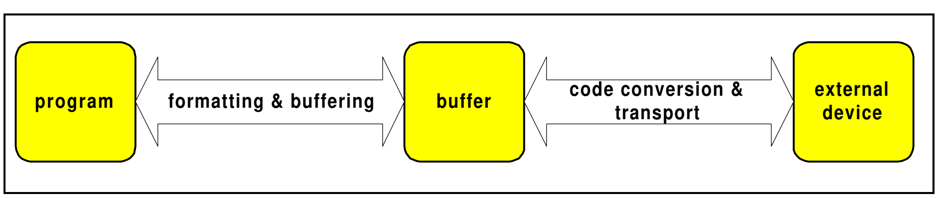


Figure 4: The iostreams layers

Let's take a look at the function of each layer in more detail:

- The Formatting Layer. Here the transformation between a program's internal data representation and a readable representation as a character sequence takes place. This formatting and parsing may involve, among other things:
  - Precision and notation of floating point numbers;
  - Hexadecimal, octal or decimal representation of integers;
  - Skipping of white spaces from input;
  - Field width for output;
  - Adapting of number formatting to local conventions.
- The Transport Layer. This layer is responsible for producing and consuming characters. It encapsulates knowledge about the properties of a specific external device. Among other things, this involves:
  - Block-wise output to files via system calls;
  - Code conversion to multibyte encodings.

  To reduce the number of accesses to the external device, a buffer is used. For output, the formatting layer sends sequences of characters to the transport layer, which stores them in a *stream buffer*. The actual transport to the external device happens only when the buffer is full. For input, the transport layer reads from the external device and fills the buffer. The formatting layer receives characters from the buffer. When the buffer is empty, the transport layer is responsible for refilling it.
- Locales. Both the formatting and the transport layers use the stream's locale. (See section 1.9 for details about locales.) The formatting layer delegates the handling of numeric entities to the locale's numeric facets. The transport layer uses the locale's code conversion facet for character-wise transformation between the buffer content and characters transported to and from the external device. The figure below shows how locales are used with iostreams:

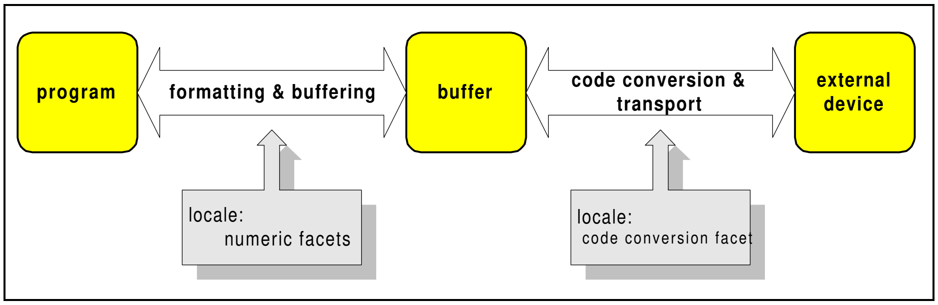


Figure 5: Usage of locales in iostreams

1.2.2.2 File and In-Memory I/O

Iostreams support two kinds of I/O: *file I/O* and *in-memory I/O* .

File I/O involves the transfer of data to and from an external device. The device need not necessarily be a file in the usual sense of the word. It could just as well be a communication channel, or another construct that conforms to the file abstraction.

In contrast, in-memory I/O involves no external device. Thus code conversion and transport are not necessary; only formatting is performed. The result of such formatting is maintained in memory, and can be retrieved in the form of a character string.

1.2.3 How Do the Standard Iostreams Help Solve Problems?

There are many situations in which iostreams are useful:

- File I/O. Iostreams can still be used for input and output to files, although file I/O has lost some of the importance it used to have. In the past, alpha-numeric user-interfaces were often built using file input/output to the standard input and output channels. Today almost all applications have graphical user interfaces.

  Nevertheless, iostreams are still useful for input and output to files other than the standard input and output channels, and for input and output to all other kinds of external media that fit into the file abstraction. For example, the Rogue Wave class library for network communications programming, *Net.h++*, uses iostreams for input and output to various kinds of communication streams like sockets and pipes.

- In-Memory I/O. Iostreams can perform in-memory formatting and parsing. Even with a graphical user interface, you will have to format the text you want to display. The standard iostreams offer internationalized in-memory I/O, which is a great help for text processing tasks like this. The formatting of numeric values, for example, depends on cultural conventions. The formatting layer uses a locale's numeric facets to adapt its formatting and parsing to cultural conventions.

- Internationalized Text Processing. Internationalized text processing is actively supported by iostreams.

  Iostreams use locales. As locales are extensible, any kind of facet can be carried by a locale, and thus used by a stream. By default, iostreams use only the numeric and the code conversion facets of a locale. However, there are date , time and monetary facets available in the Standard C++ Library. Other cultural dependencies can be encapsulated in unique facets and made accessible to a stream. You can easily internationalize iostreams to meet your needs.

- Binary I/O. The traditional iostreams used to suffer from a number of limitations. The biggest was its lack of conversion abilities: if you inserted a `double` into a stream, for example, you did not know what format would be used to represent this `double` on the external device. There was no portable way to insert it as binary.

  Standard iostreams are by far more flexible. The code conversion performed on transfer of internal data to external devices can be customized: the transport layer delegates the task of converting to a code conversion facet. To provide a stream with a suitable code conversion facet for binary output, you can insert a `double` into a file stream in a portable binary data exchange format. No such code conversion facets are provided by the Standard Library though, and implementing such a facet is not trivial. As an alternative, you might consider implementing an entire stream buffer layer that can handle binary I/O.

- Extending Iostreams. In a way, you can think of iostreams as a framework. There are many ways to extend and customize iostreams. You can add input and output operators for user-defined types. You can create your own formatting elements, the so-called manipulators. You can specialize entire streams, usually in conjunction with specialized stream buffers. You can provide different locales, which represent different cultural conventions, or contain special purpose facets. You can instantiate iostreams classes for new character types, other than `char` or `wchar_t`.

1.2.4 Internal Structure of the Iostreams Layers

As explained earlier, iostreams have two layers, one that handles formatting, and another that is responsible for code conversion and transport of characters to and from the external device. For convenience, let us repeat here the illustration of the iostreams

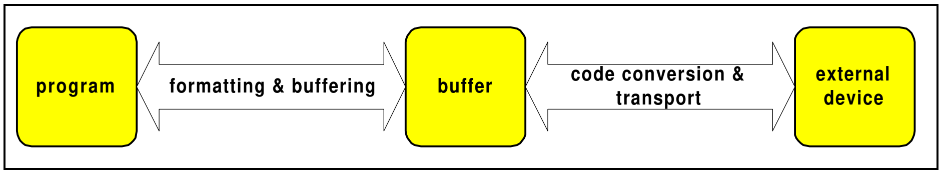


Figure 6. The iostreams layers

layers, Figure 4 from Section 1.2.2:

This section will give a more detailed description of the iostreams software architecture, including the classes and their inheritance relationship and respective responsibilities. If you would rather start using iostreams directly, skip this section and come back to it later, when you are curious to learn more about the details of iostreams.

1.2.4.1 The Internal Structure of the Formatting Layer

Classes that belong to the formatting layer are often referred to as the *stream classes*. The figure below illustrates the class hierarchy of all the stream classes:

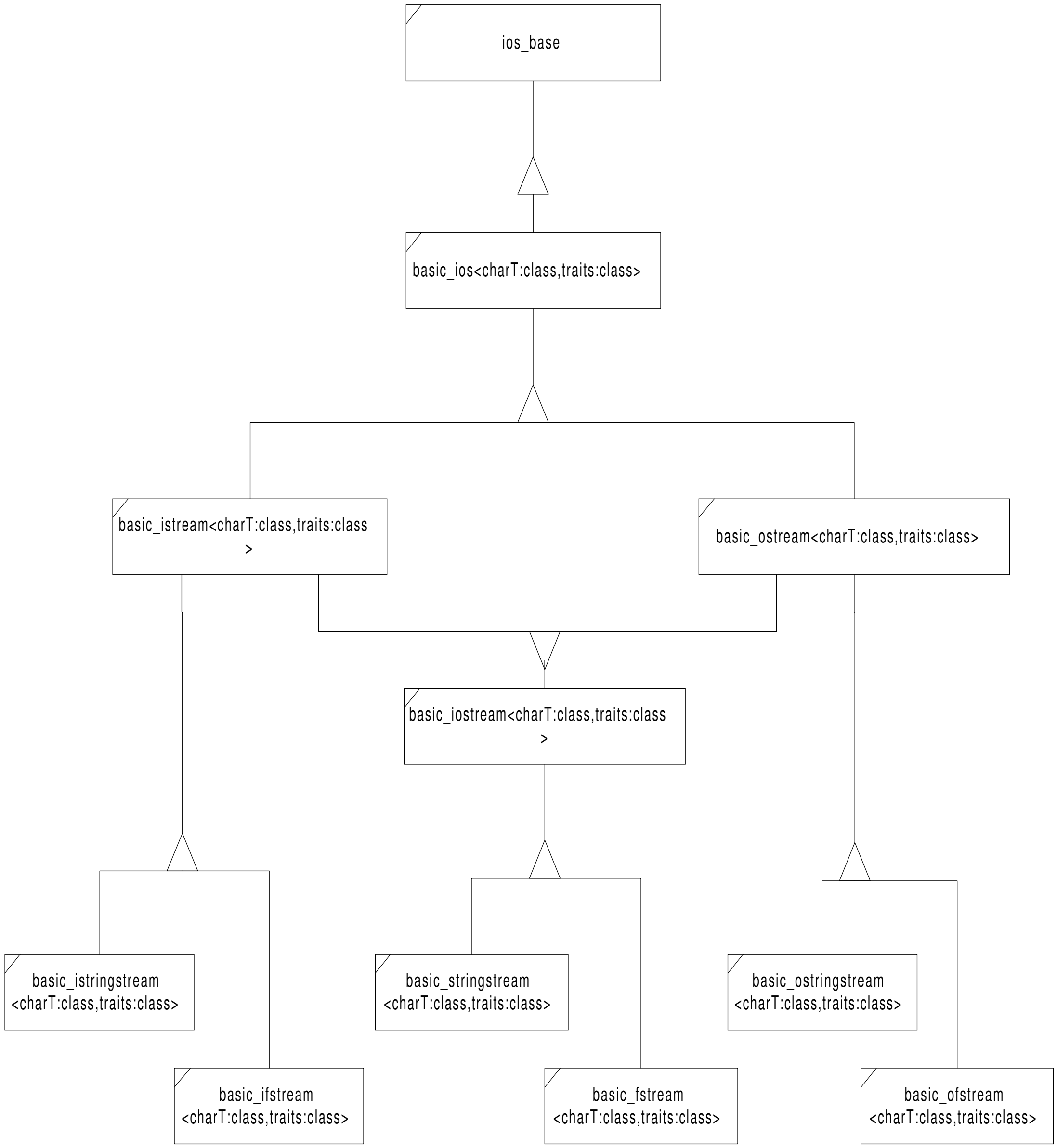


Figure 7: Internal class hierarchy of the formatting layer [1]

---

[1] There are additional classes `strstream`, `istrstream` and `ostrstream`, which are not described in this users guide. You can find them in the class reference though. These classes are so-called *deprecated features* in the standard, i.e. they are provided solely for sake of compatibility with the tradition

Let us discuss in more detail the components and characteristics of the class hierarchy given in the figure:

- The Iostreams Base Class ***ios_base***. This class is the base class of all stream classes. Independent of character type, it encapsulates information that is needed by all streams. This information includes:
  - State information that reflects the integrity of the stream buffer;
  - Control information for parsing and formatting;
  - Additional information for the user's special needs (a way to extend iostreams, as we will see later on);
  - The locale imbued on the stream;
  - Additionally, ***ios_base*** defines several types that are used by all stream classes, such as format flags, status bits, open mode, exception class, etc.
- The Iostreams Character Type-Dependent Base Class. Here is the virtual base class for the stream classes:

  `basic_ios<class charT, class traits=char_traits<charT> >`

  The class holds a pointer to the stream buffer.

  Note that ***basic_ios<>*** is a class template taking two parameters, the type of character handled by the stream, and the *character traits*.

  The type of character can be type `char` for single-byte characters, or type `wchar_t` for wide characters, or any other user-defined character type. There are instantiations for `char` and `wchar_t` provided by the Standard C++ Library.

  For convenience, there are typedefs for these instantiations:

  `typedef basic_ios<char>` ios and `typedef basic_ios<wchar_t>` wios

  Also note that ios is not a class anymore, as it used to be in the traditional iostreams. If you have existing programs that use the old iostreams, they may no longer be compatible with the standard iostreams. (See list of incompatibilities in section 1.14)

---

iostreams, but they will not be supported in future versions of the standard iostreams.

- Character Traits. *Character traits* describe the properties of a character type. Many things change with the character type, such as:

  - The end-of-file value. For type `char`, the end-of file value is represented by an integral constant called `EOF`. For type `wchar_t`, there is a constant defined that is called `WEOF`. For an arbitrary user-defined character type, the associated character traits define what the end-of-file value for this particular character type is.

  - The type of the `EOF` value. This needs to be a type that can hold the `EOF` value. For example, for single-byte characters, this type is `int`, different from the actual character type `char`.

  - The equality of two characters. For an exotic user-defined character type, the equality of two characters might mean something different from just bit-wise equality. Here you can define it.

  And many more...

  There are specializations defined for type `char` and `wchar_t`. In general, this class template is not meant to be instantiated for a character type. You should always define class template specializations.

  Fortunately, the Standard C++ Library is designed to make the most common cases the easiest. The traits template parameter has a sensible default value, so usually you don't have to bother with character traits at all.

- The Input and Output Streams. The three stream classes for input and output are:

  ```
  basic_istream <class charT, class traits=char_traits<charT> >
  basic_ostream <class charT, class traits=char_traits<charT> >
  basic_iostream<class charT, class traits=char_traits<charT> >
  ```

  Class `istream` handles input, class `ostream` is for output. Class `iostream` deals with input *and* output; such a stream is called a *bidirectional* stream.

  These three classes define functions for parsing and formatting, which are overloaded versions of `operator>>()` for input, called *extractors*, and overloaded versions of `operator<<()` for output, called *inserters*

  Additionally, there are member functions for unformatted input and output, like `get()`, `put()`, etc.

- The File Streams. The file stream classes support input and output to and from files. They are:

```
basic_ifstream<class charT, class traits=char_traits<charT> >
basic_ofstream<class charT, class traits=char_traits<charT> >
basic_fstream<class charT, class traits=char_traits<charT> >
```

There are functions for opening and closing files, similar to the C functions `fopen()` and `fclose()`. Internally they use a special kind of stream buffer, called a *file buffer*, to control the transport of characters to/from the associated file. The function of the file streams is illustrated in the following figure:

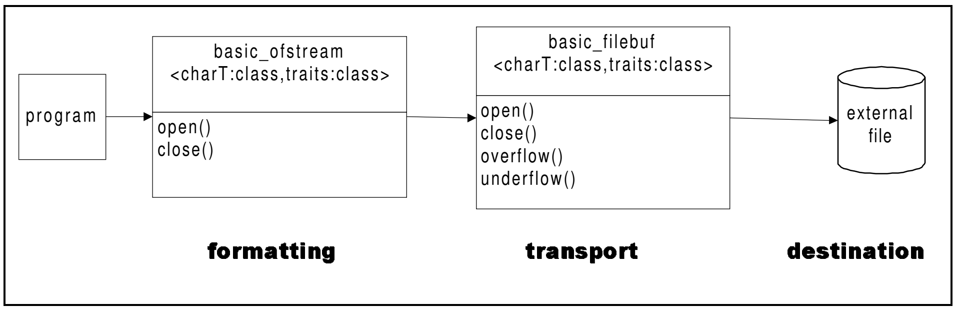


Figure 8: File I/O

- The String Streams. The string stream classes support in-memory I/O; that is, reading and writing to a string held in memory. They are:

```
basic_istringstream<class charT, class traits=char_traits<charT> >
basic_ostringstream<class charT, class traits=char_traits<charT> >
basic_stringstream<class charT, class traits=char_traits<charT> >
```

There are functions for getting and setting the string to be used as a buffer. Internally a specialized stream buffer is used. In this particular case, the buffer and the external device are the same. The figure below illustrates how the string stream classes work:

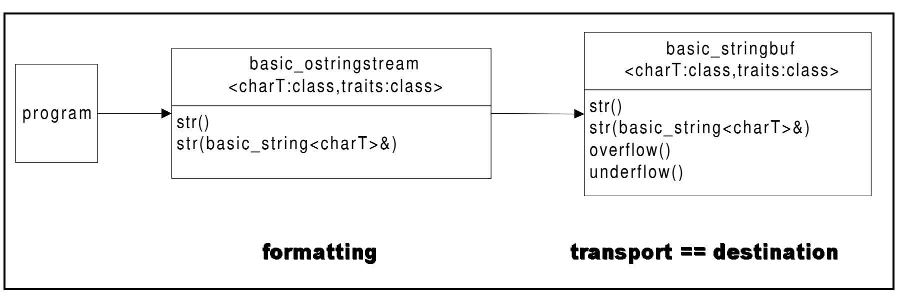
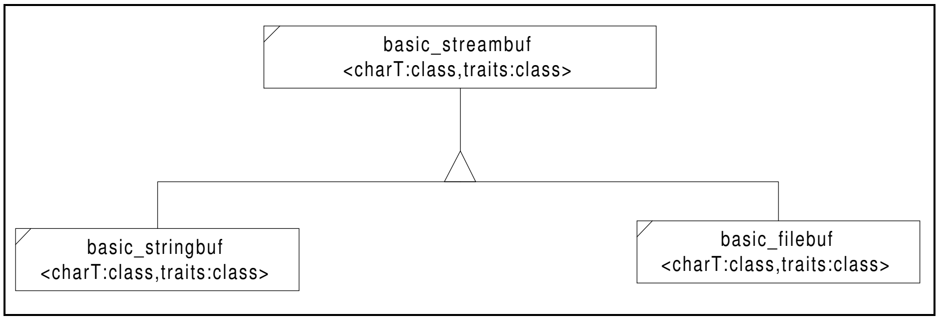


Figure 9: In-memory I/O

1.2.4.2 The Transport Layer's Internal Structure



Classes of the transport layer are often referred to as the stream buffer classes. Here is the class hierarchy of all stream buffer classes:

Figure 10: Hierarchy of the transport layer

The stream buffer classes are responsible for transfer of characters from and to external devices.

- The Stream Buffer. This class represents an abstract stream buffer:

  ```
  basic_streambuf<class charT, class traits=char_traits<charT>
  >
  ```

  It does not have any knowledge about the external device. Instead, it defines two virtual functions, `overflow()` and `underflow()`, to perform the actual transport. These two functions have knowledge of the peculiarities of the external device they are connected to. They have to be overwritten by all concrete stream buffer classes, like file and string buffers.

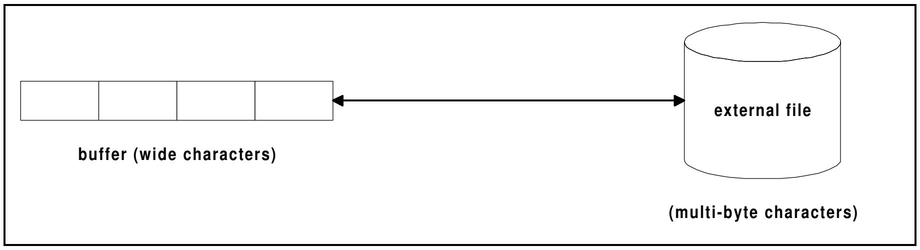
  The stream buffer class maintains two character sequences: the *get area*, which represents the input sequence read from an external device, and the *put area*, which is the output sequence to be written to the device. There are functions for providing the next character from the buffer, such as `sgetc()`, etc. They are typically called by the formatting layer in order to receive characters for parsing. Accordingly, there are also functions for placing the next character into the buffer, such as `sputc()`, etc.

  A stream buffer also carries a locale object.

- The File Buffer. The file buffer classes associate the input and output sequences with a file. A file buffer takes the form:

`basic_filebuf<class charT, class traits=char_traits<charT> >`

The file buffer has functions like `open()` and `close()`. The file buffer class inherits a locale object from its stream buffer base class. It uses the locale's code conversion facet for transforming the external character encoding to the encoding used internally.



The figure below shows how the file buffer works:

Figure 11: Character code conversion performed by the file buffer

- The String Stream Buffer. These classes implement the in-memory I/O:

  `basic_stringbuf<class charT, class traits=char_traits<charT> >`

  With string buffers, the internal buffer and the external device are identical. The internal buffer is dynamic, in that it is extended if necessary to hold all the characters written to it. You can obtain copies of the internally held buffer, and you can provide a string to be used as internal buffer.

1.2.4.3 Collaboration of Streams and Stream Buffers

The base class ***basic_ios<>*** holds a pointer to a stream buffer. The derived stream classes, like file and string streams, contain a file or string buffer object. The stream buffer pointer of the base class refers to this embedded object. This operation is illustrated in the figure below:

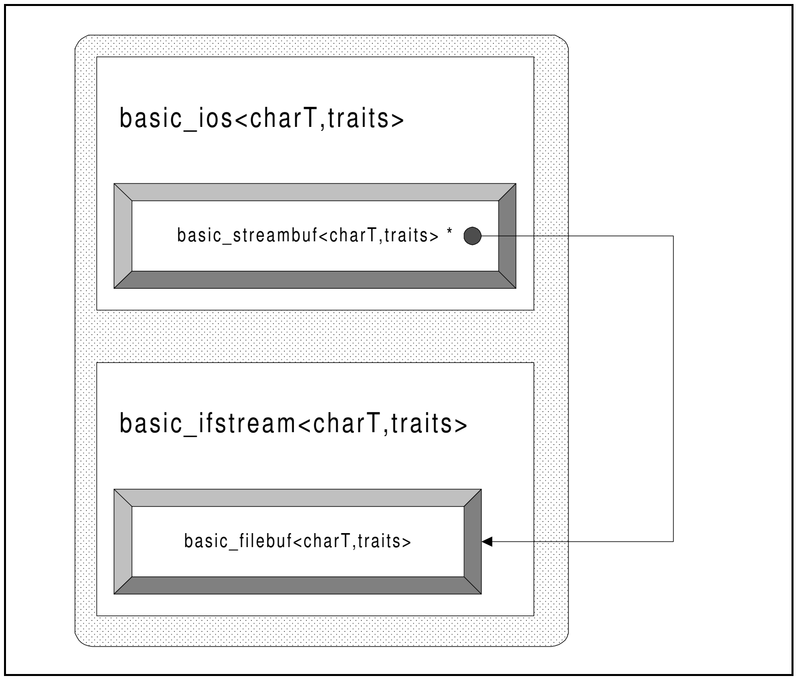


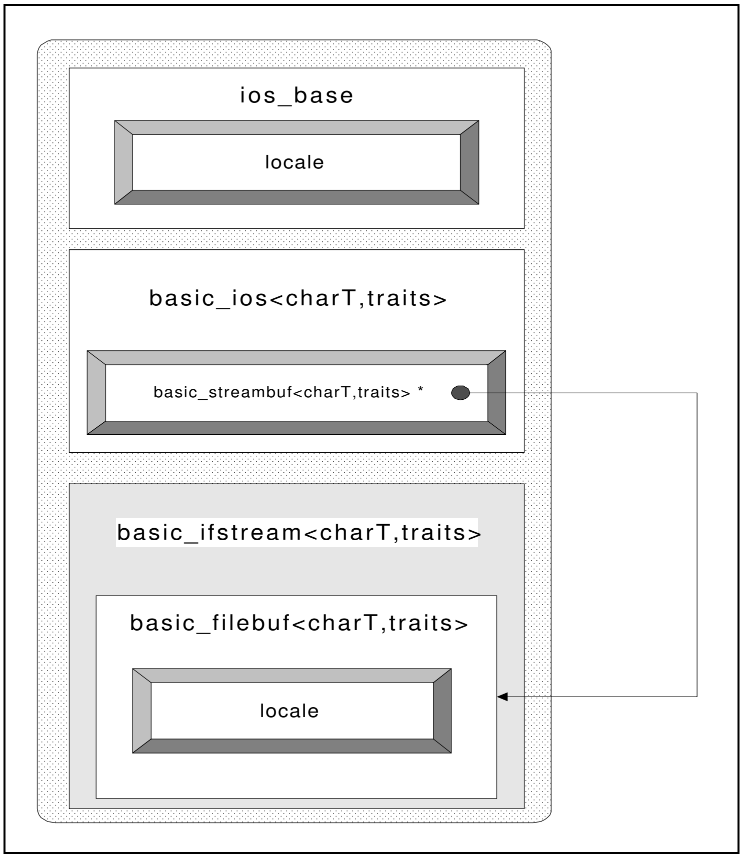
Figure 12: How an input file stream uses a file buffer

Stream buffers can be used independently of streams, as for unformatted I/O, for example. However, streams always need a stream buffer.

1.2.4.4 Collaboration of Locales and Iostreams

The base class ***ios_base*** contains a locale object. The formatting and parsing functions defined by the derived stream classes use the *numeric facets* of that locale.

The class ***basic_ios<charT>*** holds a pointer to the stream buffer. This stream buffer has a locale object, too, usually a copy of the same locale object used by the functions of the stream classes. The stream buffer's input and output functions use the *code conversion facet* of the attached locale. The figure below illustrates how the locales are



used:

Figure 13: How an input file stream uses locales

Now let's look at an example that illustrates how the facets are used by iostreams. Here the inserter for float values uses the `num_put` facet:

```
template<class charT>
basic_ostream<charT>& basic_ostream<charT>::operator<<(float val)
{ // ...
  use_facet<num_put<charT> >(getloc())
      .put(ostreambuf_iterator<charT>(*this),*this,fill(),val);
  // ...
}
```

1.3 Formatted Input/Output

This section describes the formatting facilities of iostreams. Here we begin using the predefined streams, and see how simple input and output are done. We will then explore in detail how parsing and formatting can be controlled.

1.3.1 The Predefined Streams

There are eight predefined standard streams that are automatically created and initialized at program start. These standard streams are associated with the C standard files `stdin`, `stdout`, and `stderr`, as shown in the table below:

Table 1: Predefined standard streams with their associated C standard files

Narrow character stream	Wide character stream	Associated C standard files
`cin`	`wcin`	`stdin`
`cout`	`wcout`	`stdout`
`cerr`	`wcerr`	`stderr`
`clog`	`wclog`	`stderr`

Like the C standard files, these streams are all by default associated with the terminal.

The difference between `clog` and `cerr` is that `clog` is fully buffered, whereas output to `cerr` is written to the external device after each formatting. With a fully buffered stream, output to the actual external device is written only when the buffer is full. Thus `clog` is more efficient for redirecting output to a file, while `cerr` is mainly useful for terminal I/O. Writing to the external device after every formatting, to the terminal in the case of `cerr`, serves the purpose of synchronizing output to and input from the terminal.

The standard streams are initialized in such a way that they can be used in constructors and destructors of static objects. The exact mechanism for this initialization is explained in Section 1.9.

1.3.2 Input and Output Operators

Now let's try to do some simple input and output to the predefined streams. The iostreams facility defines shift operators for formatted stream input and output. The output operator is the shift operator `operator<<()`, also called the *inserter* (defined in Section 1.2.4.1):

```
cout << "result: " << x << '\n';
```

Input is done via another shift operator `operator>>()`, often referred to as *extractor* (also defined in Section 1.2.4.1):

```
cin >> x >> y;
```

Both operators are overloaded for all built-in types in C++, as well as for some of the types defined in the Standard C++ Library; for example, there are inserters and extractors for `bool`, `char`, `int`, `long`, `float`, `double`, `string`, etc. When you insert or extract a value to or from a stream, the C++ function overload resolution chooses the correct extractor operator, based on the value's type. This is what makes C++ iostreams type-safe and better than C Stdio (see Section 1.2.1.1).

It is possible to print several units in one expression. For example:

```
cout << "result: " << x;
```

is equivalent to:

```
(cout.operator<<("result: ")).operator<<(x);
```

This is possible because each shift operator returns a reference to the respective stream. All shift operators for built-in types are member functions of their respective stream class. They are defined according to the following patterns:

```
template<class charT, class traits>
basic_istream<charT, traits>&
basic_istream<charT, traits>::operator>>(type& x)
{
  // read x
  return *this;
}
```

and:

```
template<class charT, class traits>
basic_ostream<charT, traits>&
basic_ostream<charT, traits>::operator<<(type x)
{
  // write x
  return *this;
}
```

Simple input and output of units as shown above is useful, yet not sufficient in many cases. For example, you may want to vary the way output is formatted, or input is parsed. Iostreams allow you to control the formatting features of its input and output operators in many ways. With iostreams, you can specify:

- The width of an output field and the adjustment of the output within this field;
- The decimal point, because it might vary according to cultural conventions;
- The precision and format of floating point numbers;
- Whether you want to skip white spaces when reading from an input stream;
- Whether integral values are displayed in decimal, octal or hexadecimal format,

and many more.

There are two mechanisms that have an impact on formatting:

- Formatting control via a stream's format state, and
- Localization via a stream's locale.

The stream's format state is the main means of format control, as we will demonstrate in the next section.

1.3.3 Format Control Using the Stream's Format State

1.3.3.1 Format Parameters

Associated with each stream are a number of *format state variables* that control the details of formatting and parsing. Format state variables are classes inherited from a stream's base classes, either ***ios_base*** or ***basic_ios<charT,traits>***. There are two kinds of format parameters:

- *Parameters that can have an arbitrary value.* The value is stored as a private data member in one of the base classes, and set and retrieved via public member functions inherited from that base class. There are three such parameters, described in the table below:

Table 2: Format parameters with arbitrary values

Access function	Defined in base class	Effect	Default
`width()`	***ios_base***	Minimal field width	0
`precision()`	***ios_base***	Precision of floating point	6

		values	
`fill()`	***basic_ios <charT,traits>***	Fill character for padding	the space character

- *Parameters that can have only a few different values, typically two or three.* They are represented by one or more bits in a data member of type `fmtflags` in class ***ios_base***. These are usually called *format flags*. You can set format flags using the `setf()` function in class ***ios_base***. You can retrieve them via the `flags()` function. You can clear them using `unsetf()`.

  Some format flags are grouped, because they are mutually exclusive; for example, the adjustment of the output within an output field. It can be adjusted to the left or to the right, or an internally specified adjustment can be chosen. One and only one of the corresponding three format flags, `left`, `right`, or `internal`, can be set. If you want to set one of these bits to 1, you had better set the other two to 0. To make this easier, there are *bit groups* defined that are mostly used to reset all bits in one group. The bit group for adjustment is `adjustfield`, defined as `left | right | internal`.

The table below gives an overview of all format flags and their effects on input and output operators. (See the *Class Reference* of ***ios_base*** for details on how the format flags affect input and output operations.) The first column below, *format flag*, lists the flag names; for example, `showpos` stands for `ios_base::showpos`. The *group* column lists the name of the group for flags that are mutually exclusive. The third column gives a brief description of the effect of setting the flag. The *stdio* column refers to format characters used by the C functions `scanf()` or `printf()` that have the same or similar effect. The last column, *default*, lists the setting that is used if you do not explicitly set the flag.

Table 3: Flags and their effects on operators

Format flag	Group	Effect	stdio	Default
	`adjustfield`	Adds fill characters to certain generated output for adjustment:		`right`
`left`		to the left	`-`	
`right`		to the right	`0`	

`internal`		adds fill characters at designated internal point.		

`dec`
`oct`
`hex`
	`basefield`	Converts integer input or generates integer output in:
		decimal base
		octal base
		hexadecimal base
			`%i`
			`%d,%u`
			`%o`
			`%x`
				`dec`

`fixed`
`scientific`
	`floatfield`	Generates floating point output:
		in fixed-point notation
		in scientific notation
			`%g,%G`
			`%f`
			`%e,%E`
				`fixed`

`boolalpha`		Inserts and extracts `bool` values in alphabetic format		0
`showpos`		Generates a + sign in non-negative generated numeric output	+	0
`showpoint`		Always generates a decimal-point in generated floating-point output	#	0
`showbase`		Generates a prefix indicating the numeric base of a generated integer output		0
`skipsw`		Skips leading white space before certain input operations		1
`unitbuf`		Flushes output after each formatting operation		0
`uppercase`		Replaces certain lowercase letters with	%X	0

		their uppercase equivalents in generated output	%E %G	

The effect of setting a format parameter is usually permanent; that is, the parameter setting is in effect until the setting is explicitly changed. The only exception to this rule is the field width. The width is automatically reset to its default value 0 after each input or output operation that uses the field width[2]. Here is an example:

```
int i; char* s[11];
cin >> setw(10) >> i >> s;                                    \\1
cout << setw(10) <<i << s;                                    \\2
```

//1 Extracting an integer is independent of the specified field width. The extractor for integers always reads as many digits as belong to the integer. As extraction of integers does not use the field width setting, the field width of 10 is still in effect when a character sequence is subsequently extracted. Only 10 characters will be extracted in this case. After the extraction, the field width is reset to 0.

//2 The inserter for integers uses the specified field width and fills the field with padding characters if necessary. After the insertion, it resets the field width to 0. Hence, the subsequent insertion of the string will not fill the field with padding characters for a string with less than 10 characters.

Please note: *With the exception of the field width, all format parameter settings are permanent. The field width parameter is reset after each use*.

The following code sample shows how you can control formatting by using some of the parameters:

```
#include <iostream>
using namespace ::std;
// …
ios_base::fmtflags original_flags = cout.flags();             \\1
cout<< 812<<'|';
cout.setf(ios_base::left,ios_base::adjustfield);              \\2
cout.width(10);                                               \\3
cout<< 813 << 815 << '\n';
cout.unsetf(ios_base::adjustfield);                           \\4
cout.precision(2);
cout.setf(ios_base::uppercase|ios_base::scientific);          \\5
```

[2] The details of exactly when width is reset to zero are not specified in the standard.

```
cout << 831.0 << ' ' << 8e2;
cout.flags(original_flags);                                      \\6
```

//1 Store the current format flag setting, in order to restore it later on.

//2 Change the adjustment from the default setting `right` to `left`.

//3 Set the field width from its default 0 to 10. The default 0 means that no padding characters are inserted.

//4 Clear the adjustment flags.

//5 Change the precision for floating-point values from its default 6 to 2, and set yet another couple of format flags that affect floating-point values.

//6 Restore the original flags.

The output is:

```
812|813       815
8.31E+02 8.00E+02$
```

1.3.3.2 Manipulators

Format control requires calling a stream's member functions. Each such call interrupts the respective shift expression. But what if you need to change formats within a shift expression? Indeed, this is possible in iostreams. Instead of writing:

```
cout<< 812 << '|';
cout.setf(ios_base::left,ios_base::adjustfield);
cout.width(10);
cout<< 813 << 815 << '\n';
```

you can write:

```
cout<< 812 << '|' << left << setw(10) << 813 << 815 << endl;
```

In this example, objects like `left`, `setw`, and `endl` are called *manipulators*. A manipulator is an object of a certain type; let us call the type `manip` for the time being. There are overloaded versions of `basic_istream <charT,traits>:: operator>>()` and `basic_ostream <charT,traits>:: operator<<()` for type `manip`. Hence a manipulator can be extracted from or inserted into a stream together with other objects that have the shift operators defined. (Section1.8 explains in greater detail how manipulators work and how you can implement your own manipulators.)

The effect of a manipulator need not be an actual input to or output from the stream. Most manipulators set just one of the above described format flags, or do some other kind of stream manipulation. For example, an expression like:

```
cout << left;
```

is equivalent to:

```
cout.setf (ios_base::left, ios_base::adjustfield);.
```

Nothing is inserted into the stream. The only effect is that the format flag for adjusting the output to the left is set.

On the other hand, the manipulator `endl` inserts the newline character to the stream, and flushes to the underlying stream buffer. The expression:

```
cout << endl;
```

is equivalent to:

```
cout << '\n'; cout.flush();
```

Some manipulators take arguments, like `setw(int)`. The `setw` manipulator sets the field width. The expression:

```
cout << setw(10);
```

is equivalent to:

```
cout.width(10);
```

In general, you can think of a manipulator as an object you can insert into or extract from a stream, the effect of which is some kind of manipulation of that stream.

Some manipulators can be applied only to output streams, others only to input streams. Most manipulators change format bits only in one of the stream base classes, ***ios_base*** or ***basic_ios<charT,traits>***. These can be applied to input and output streams.

The following table gives an overview of all manipulators defined by iostreams. The first column, *manipulator*, lists its name. All manipulators are classes defined in the namespace `::std`. The second column indicates whether the manipulator is intended to be used with istreams (i), ostreams (o), or both (io). The third column summarizes the effect of the manipulator. The last column, *equivalent*, lists the corresponding call to the stream's member function.

Note that the second column only indicates the *intended* use of a manipulator. In many cases it is possible to apply an output manipulator to an input stream, and vice versa. Generally, this kind of non-intended manipulation is harmless; it does not have any effect. For instance, if you apply the `showpoint` manipulator, which is an output manipulator, to an input stream, the manipulation will simply be ignored. However, if you manipulate a bidirectional stream during input with an output manipulator, the manipulation will have no effect on input operations, but it will have an impact on subsequent output operations.

Table 4: Manipulators

Manipulator		Effect	Equivalent
`flush`	o	Flushes stream buffer	`o.flush()`
`endl`	o	Inserts newline and flushes buffer	`o.put(traits::newline());` `o.flush();`
`ends`	o	Inserts end of string character	`o.put(traits::eos())`
`ws`	i	Skips white spaces	
`boolalpha`	io	Puts `bool` values in alphabetic format	`io.setf(ios_base::boolalpha)`
`noboolalpha`	io	Resets the above	`io.unsetf(ios_base::boolalpha)`
`showbase`	o	Generates a prefix indicating the numeric base of an integer	`o.setf(ios_base::showbase)`
`noshowbase`	o	Resets the above	`o.unsetf (ios_base::showbase)`
`showpoint`	o	Always generates a decimal-point for floating-point values	`o.setf(ios_base::showpoint)`
`noshowpoint`	o	Resets the above	`o.unsetf (ios_base::showpoint)`
`showpos`	o	Generates a `+` sign for non-negative numeric values	`o.setf(ios_base::showpos)`
`noshowpos`	o	Resets the above	`o.unsetf (ios_base::showpos)`
`skipws`	i	Skips leading white space	`i.setf(ios_base::skipws)`
`noskipws`	i	Resets the above	`i.unsetf(ios_base::skipws)`
`uppercase`	o	Replaces certain lowercase letters with their uppercase equivalents	`o.setf(ios_base::uppercase)`

nouppercase		Resets the above	o.unsetf (ios_base::uppercase)
unitbuf	o	Flushes output after each formatting operation	o.setf(ios_base::unitbuf)
nounitbuf	o	Resets the above	o.unsetf(ios_base::unitbuf)
internal	o	Adds fill characters at designated internal point	o.setf(ios_base::internal, ios_base::adjustfield)
left	o	Adds fill characters for adjustment to the left	o.setf(ios_base::left, ios_base::adjustfield)
right	o	Adds fill characters for adjustment to the right	o.setf(ios_base::right, ios_base::adjustfield)
dec	io	Converts integers to/from decimal notation	io.setf(ios_base::dec, ios_base::basefield)
hex	io	Converts integers to/from hexadecimal notation	io.setf(ios_base::hex, ios_base::basefield)
oct	io	Converts to/from octal notation	io.setf(ios_base::oct, ios_base::basefield)
fixed	o	Puts floating point values in fixed-point notation	o.setf(ios_base::fixed, ios_base::floatfield)
scientific		Puts floating point values in scientific notation	o.setf(ios_base::scientific, ios_base::floatfield)
setiosflags (ios_base:: fmtflags mask)	io	Sets ***ios*** flags	io.setf(mask)

resetiosflags (ios_base:: fmtflags mask)	io	Clear ***ios*** flags	`io.setf((ios_base::fmtflags)0, mask)`
setbase (int base)	io	set base for integer notation (base = 8, 10, 16)	`io.setf (base == 8?ios_base::oct: base == 10 ? ios_base::dec : base == 16 ? ios_base::hex : ios_base::fmtflags(0), ios_base::basefield)`
setfill(charT c)	io	set fill character for padding	`io.fill(c)`
setprecision (int n)	io	set precision of floating point values	`io.precision(n)`
setw(int n)	io	set minimal field width	`io.width(n)`

1.3.4 Localization Using the Stream's Locale

Associated with each stream is a locale that has an impact on the parsing and formatting of numeric values. This is how localization of software takes place. As discussed in the section on locale, the representation of numbers often depends on cultural conventions. In particular, the *decimal point* need not be a period, as in the following example:

```
cout.imbue(locale("De_DE"));
cout << 1000000.50 << endl;
```

The output will be:

```
1000000,50
```

Other cultural conventions, like the grouping of digits, are irrelevant. There is no formatting of numeric values that involves grouping.

1.3.5 Formatted Input

In principle, input and output operators behave symmetrically. There is only one important difference: for output you control the precise format of the inserted character sequence, while with input the format of an extracted character sequence is never exactly described.

This is for practical reasons. Suppose you want to extract the next floating point value from a stream, but you do not want to anticipate its exact format. You want to extract it anyway, no matter whether it is signed or not, or in exponential notation with a small or capital E for the exponent, etc. Hence, extractors in general accept an item in any format that is permitted for its type.

Formatted input is done the following way:

1. Extractors automatically ignore all white spaces (blanks, tabulators, newlines [3]) that precede the item to be extracted.
2. When the first relevant character is found, they extract characters from the input stream until they find a separator, that is, a character that does not belong to the item. White spaces in particular are separators.
3. The separator remains in the input stream and becomes the first character extracted in a subsequent extraction.

Several format parameters, which control insertion, are irrelevant for extraction. The format parameters fill character, `fill()`, and the adjustment flags, `left`, `right`, and `internal`, have no effect on extraction. The field width is relevant only for extraction of strings, and ignored otherwise.

1.3.5.1 Skipping Characters

You can use the manipulator `noskipsw` to switch off the automatic skipping of white spaces. Extracting the white space characters can be necessary, for example, if you expect that the input has a certain format, and you need to check for violations of the format requirements. This procedure is shown in the following code:

```
cin >> noskipsw;
char c;
do
{ float fl;
  c = ' '; cin >> fl >> c;  // extract number and separator
if (c == ',' || c == '\n')  // next char is ',' or newline ?
    process(fl);            // yes: use the number
}
while (c == ',');
if (c != '\n') error();     // no: error!
```

[3] The classification of a character as a *white space character* depends on the character set that is used. The extractor takes the respective information from the locale's ctype facet.

If you have to skip a sequence of characters other than white spaces, you can use the istream's member function `ignore()`. The call:

```
basic_ifstream<myChar,myTraits> InputStream("file-name");
InputStream.ignore(numeric_limits<streamsize>::max()
                    ,myChar('\n'));
```

or, for ordinary tiny characters of type `char`:

```
ifstream InputStream("file-name");
InputStream.ignore(INT_MAX,'\n');
```

ignores all characters until the end of the line. This example uses a file stream that is not predefined. File streams are described in section 1.5.3.

1.3.5.2 Input of Strings

When you extract strings from an input stream, characters will be read until:

- a white space character is found, or
- the end of the input is reached,
- or a certain number (width()-1, if width() != 0) characters are extracted.

An end-of-string character will be added to the extracted character sequence. Note that the field width will be reset to 0 after the extraction of a string.

There are subtle differences between the extraction of a character sequences into a character array or a string object, i.e.

```
char buf[SZ];
cin >> buf;
```

is different from

```
string s;
cin >> s;
```

Extraction into a string is safe, because strings automatically extend their capacity if necessary. You can extract as many characters as you want; the string will always adjust its size accordingly. Character arrays, on the other hand, have a fixed size and cannot dynamically extend their capacity. If you extract more characters than the character array can take, then the extractor writes beyond the end of the array. In order to prevent the extractor from doing this, it is necessary to set the field width each time you extract characters into a character array:

```
char buf[SZ];
```

```
cin >> width(SZ) >> buf;
```

1.4 Error State of Streams

It probably comes as no surprise that streams have an error state. Our examples have avoided it, so we'll deal with it now. When an error occurs, flags are set in the state according to the general category of the error. Flags and their error categories are summarized in the table below:

Table 5: Flags and corresponding error categories

iostate flag	Error category
`ios_base::goodbit`	Everything's fine
`ios_base::eofbit`	An input operation reached the end of an input sequence
`ios_base::failbit`	An input operation failed to read the expected character, or An output operation failed to generate the desired characters
`ios_base::badbit`	Indicates the loss of integrity of the underlying input or output sequence

Note that the flag `ios_base::goodbit` is not really a flag; its value, zero, indicates the absence of any error flag. It means the stream is okay. By convention, all input and output operations have no effect once the stream state is different from zero.

There are several situations when both `eofbit` and `failbit` are set. However, the two have different meanings and do not always occur in conjunction. The flag `ios_base::eofbit` is set on an attempt to read past the end of the input sequence. This occurs in the following two typical examples:

1. Assume the extraction happens character-wise. Once the last character is read, the stream is still in good state; `eofbit` is not yet set. Any subsequent extraction, however, will be an attempt to read past the end of the input sequence. Thus, `eofbit` will be set.
2. If you do not read character-wise, but extract an integer or a string, for example, you will always read past the end of the input sequence. This is because the input operators read characters until they find a separator, or hit the end of the input

sequence. Consequently, if the input contains the sequence `…. 912749<eof>` and an integer is extracted, `eofbit` will be set.

The flag `ios_base::failbit` is set as the result of a read (or write) operation that fails. For example, if you try to extract an integer, but the input sequence contains only white spaces, then the extraction of an integer will fail, and the `failbit` will be set. Let's go back to our two examples:

1. After reading the last available character, the extraction not only reads past the end of the input sequence; it also fails to extract the requested character. Hence, `failbit` will be set in addition to `eofbit`.
2. Here it is different. Although the end of the input sequence will be reached by extracting the integer, the input operation does not fail: the desired integer will indeed be read. Hence, in this situation only the `eofbit` will be set.

In addition to these input and output operations, there are other situations that can trigger failure. For example, file streams set `failbit` if the associated file cannot be opened (see section 1.5).

The flag `ios_base::badbit` indicates problems with the underlying stream buffer. These problems can be:

- *Memory shortage*. There is no memory available to create the buffer, or the buffer has size zero for other reasons[4], or the stream cannot allocate memory for its own internal data[5], for example, `iword` and `pword`.
- *The underlying stream buffer throws an exception*. The stream buffer might lose its integrity, as in memory shortage, or when the code conversion fails, or an unrecoverable read error from the external device occurs. The stream buffer can indicate this loss of integrity by throwing an exception, which will be caught by the stream and result in setting the `badbit` in the stream's state.

---

[4] The stream buffer can be created as the stream's responsibility, or the buffer can be provided from outside the stream, so inadvertently the buffer could have zero size.

[5] The standard does not yet specify whether the inability to allocate memory for the stream's internal data will result in a `badbit` set or a `bad_alloc` or `ios_failure` thrown.

Generally, you should keep in mind that `badbit` indicates an error situation that is likely to be unrecoverable, whereas `failbit` indicates a situation that might allow you to retry the failed operation. The flag `eofbit` just indicates that the end of the input sequence was reached.

What can you do to check for such errors? You have two possibilities for detecting stream errors:

- You can declare that you want to have an exception raised once an error occurs in any input or output operation, or
- You can actively check the stream state after each input or output operation.

We will explore these possibilities in the next two sections.

1.4.1 Checking the Stream State

Let's first examine how you can check for errors using the stream state. A stream has several member functions for this purpose, which are summarized with their effects in the following table:

Table 6: Stream member functions for error checking

ios_base member function	Effect
`bool good()`	True if no error flag is set
`bool eof()`	True if `eofbit` is set
`bool fail()`	True if `failbit` or `badbit` is set
`bool bad()`	True if `badbit` is set
`bool operator!()`	As `fail()`
`operator void*()`	0 if `fail()` and 1 otherwise
`iostate rdstate()`	Value of stream state

It is a good idea to check the stream state in some central place, for example:

```
if (!cout) error();
```

The state of cout is examined with operator!(), which will return true if the stream state indicates an error has occurred.

An ostream can also appear in a boolean position to be tested as follows:

```
if (cout << x) // okay!
```

The magic here is the operator void*() that returns a non-zero value when the stream state is non-zero.

Finally, the explicit member functions can also be used:

```
if (cout << x, cout.good()) // okay!;
```

Note that the difference between good() and operator!(): good() takes all flags into account; operator !() and fail() ignore eofbit.

1.4.2 Catching Exceptions

By default a stream does not throw any exceptions.[6] You have to explicitly activate an exception because a stream contains *an exception mask*. Each flag in this mask corresponds to one of the error flags. For example, once the badbit flag is set in the exception mask, an exception will be thrown each time the badbit flag gets set in the stream state. The following code demonstrates how to activate an exception:

```
try {
   InStr.exceptions(ios_base::badbit | ios_base::failbit);    \\1
   in >> x;
   // do lots of other stream i/o
```

[6] This is only partly true. The standard does not yet specify how memory allocation errors are to be handled in iostreams. The two models basically are:

- A bad_alloc exception is thrown, regardless of whether or not any bits in the exception mask are set.

- The streams layer catches bad_alloc exceptions thrown during allocation of its internal resources, that is, iword and pword. It would then set badbit or failbit. An exception would be thrown only if the respective bit in the exception mask asks for it. It has to be specified whether the exception thrown in such a case would be ios_failure or the original bad_alloc.

Moreover, the streams layer has to catch all exceptions thrown by the stream buffer layer. It sets badbit and rethrows the original exception if the exception mask permits it.

```
}
catch(ios_base::failure exc)                                  \\2
{   cerr << exc.what() << endl;
    rethrow;
}
```

//1 In calling the `exceptions()` function, you indicate what flags in the stream's state shall cause an exception to be thrown.

//2 Objects thrown by the stream's operations are of types derived from `ios_base::failure`. Hence this catch clause will, in principle, catch all stream exceptions.

Note that each change of either the stream state or the exception mask can result in an exception thrown.

1.5 File Input/Output

The file streams allow input and output to files. Compared to the C stdio functions for file I/O, they have the advantage of following the idiom "Resource acquisition is initialization"[7]. In other words, you can open a file on construction of a stream, and the file will be closed automatically on destruction of the stream. Consider the following code:

```
void use_file(const char* fileName)
{
   FILE* f = fopen("fileName", "w");
   // use file
   fclose(f);
}
```

If an exception is thrown while the file is in use, the file will never be closed. With a file stream, however, it is assured that the file will be closed whenever the file stream goes out of scope, as in the following example:

```
void use_file(const char* fileName)
{
   ofstream f("fileName);
   // use file
}
```

Here the file will be closed even if an exception occurs during use of the open file.

There are three class templates that implement file streams: ***basic_ifstream <charT,traits>***, ***basic_ofstream <charT,traits>***, and ***basic_fstream <charT,traits>***. These templates are derived from the

---

[7] See Bjarne Stroustrup, *The C++ Programming Language*, p.308ff.

stream base class ***basic_ios <charT, traits>***. Therefore, they inherit all the functions for formatted input and output described in Section 1.3, as well as the stream state. They also have functions for opening and closing files, and a constructor that allows opening a file and connecting it to the stream. For convenience, there are the regular typedefs `ifstream`, `ofstream`, and `fstream`, with `wifstream`, `wofstream`, and `wfstream` for the respective tiny and wide character file streams.

The buffering is done via a specialized stream buffer class, ***basic_filebuf <charT,traits>***.

1.5.1 Difference between Predefined File Streams (cin, cout, cerr, and clog) and File Streams

The main differences between a predefined standard stream and a file stream are:

- A file stream needs to be connected to a file before it can be used. The predefined streams can be used right away, even in static constructors that are executed before the `main()` function is called.
- You can reposition a file stream to arbitrary file positions. This usually does not make any sense with the predefined streams, as they are connected to the terminal by default.

1.5.2 Code Conversion in Wide Character Streams

In a large character set environment, a file is assumed to contain multibyte characters. To provide the contents of a such a file as a wide character sequence for internal processing, `wifstream` and `wofstream` perform corresponding conversions. The actual conversion is delegated to the file buffer, which relays the task to the imbued locale's code conversion facet.

1.5.3 Creating File Streams

There are two ways to create a file stream[8]: you can create an empty file stream, open a file, and connect it to the stream later on;

[8] The traditional iostreams supported a constructor, taking a file descriptor that allowed connection of a file stream to an already open file. This is no longer available in the standard iostreams. However, Rogue Wave's implementation of the standard

or you can open the file and connect it to a stream at construction time. These two procedures are demonstrated, respectively, in the two following examples:

```
ifstream file;                                              \\1
…;
file.open(argv[1]);                                         \\2
if (!file) // error: unable to open file for input
```

or:

```
ifstream source("src.cpp");.................................\\3
if (!source) // error: unable to open src.cpp for input
```

//1 A file stream is created that is not connected to any file. Any operation on the file stream will fail.

//2 Here a file is opened and connected to the file stream. If the file cannot be opened, `ios_base::failbit` will be set; otherwise, the file stream is now ready for use.

//3 Here the file gets both opened and connected to the stream.

You can explicitly close the connected file. The file stream is then empty, until it is reconnected to another file:

```
ifstream f;                                                 \\1
for (int i=1; i<argc; ++i)
{
    f.open(argv[i]);                                        \\2
    if (f)                                                  \\3
    {
        process(f);                                         \\4
        close(f);                                           \\5
    }
    else
        cerr << "file " << argv[i] << " cannot be opened.\n";
}
```

//1 An empty file stream is created.

//2 A file is opened and connected to the file stream.

//3 Here we check whether the file was successfully opened. If the file could not be opened, the `failbit` would be set.

//4 Now the file stream is usable, and the file's content can be read and processed.

//5 Close the file again. The file stream is empty again.

---

iostreams provides an according extensions (see section1.15.1 for reference).

1.5.4 The Open Mode

Sometimes a program will want to modify the way in which a file is opened or used. For example, in some cases it is desirable that writes append to the end of a file rather than overwriting the existing values. The file stream constructor takes a second argument, the *open mode* , which allows such variations to be specified. Here is an example:

```
ofstream oStr("out.txt",ios_base::out|ios_base::app);
```

The open mode argument is of type `ios_base::openmode`, which is a bitmask type like the format flags and the stream state. The following bits are defined:

Table 7: Flag names and effects

Flag Names	Effects
`ios_base::in`	Open file for reading
`ios_base::out`	Open file for writing
`ios_base::ate`	Start position is at file end
`ios_base::app`	Append file, i.e., start writing at file end
`ios_base::trunc`	Truncate file, i.e., delete file content
`ios_base::binary`	Binary mode

Each file maintains a *file position* that indicates the position in the file where the next byte will be read or written. When a file is opened, the initial file position is usually at the beginning of the file. The open modes `ate` (= at end) and `app` (= append) change this default to the end of the file.

There is a subtle difference between `ate` and `app` mode. If the file is opened in append mode, all output to the file will be done at the current end of the file, regardless of intervening repositioning. Even if you modify the file position to a position before the file's end, you cannot write there. With at-end mode, you can navigate to a position before the end of file and write to it.

If you open an already existing file for writing, you usually want to overwrite the content of the output file. The open mode `trunc` (= truncate) has the effect of discarding the file content, in which case the initial file length is set to zero. Therefore, if you want to replace an output file's content rather than extend the file, you have to open the file ineither the `out` or `out|trunc` mode; open mode `out` alone will not suffice. Note that in this release a bug prevents you from opening it in out mode alone. The file position will be at the beginning of the file in this case, which is exactly what you expect for overwriting an output file.

If you want to extend an output file, you open it in at-end or append mode. In this case, the file content is retained because the `trunc` flag is not set, and the initial file position is at the file's end. However, you may additionally set the `trunc` flag; the file content will be discarded and the output will be done at the end of an empty file.

Input mode only works for file that already exist. Otherwise, the stream construction will fail, as indicated by `failbit` set in the stream state. Files that are opened for writing will be created if they do not yet exist. The constructor only fails if the file cannot be created.

The `binary` open mode is explained in section 1.5.5.

The effect of combining these open modes is similar to the mode argument of the C library function `fopen(name,mode)`. The following table gives an overview of the most frequently used combinations of open modes for text files and their counterparts in C stdio:

Table 8: Open modes and their C stdio counterparts

Open Mode	C stdio Equivalent	Effect
`in`	`"r"`	Open text file for reading only
`out|trunc`	`"w"`	Truncate to zero length, if existent, or create text file for writing only
`out|app`	`"a"`	Append; open or create text file only for writing at end of file
`in|out`	`"r+"`	Open text file for update (reading and writing)
`in|out|trunc`	`"w+"`	Truncate to zero length, if existent, or create text file for update
`in|out|app`	`"a+"`	Append; open or create text file for update, writing at end of file

The default open modes are listed below. Note that abbreviations are used, e.g., `ifstream` stands for `basic_ifstream<charT,traits>`.

Table 9: Default open modes

File Stream	Default Open Mode
`ifstream`	in
`ofstream`	out|trunc
`fstream`	in|out|trunc

1.5.5 Binary and Text Mode

The representation of text files varies among operating systems. For example, the end of a line in an UNIX environment is represented by the *linefeed* character `'\n'`. On PC-based systems, the end of the line consists of two characters,*carriage return* `'\r'` and *linefeed* `'\n'`. The end of the file differs as well on these two operating systems. Peculiarities on other operating systems are also conceivable.

To make programs more portable among operating systems, an automatic conversion can be done on input and output. The carriage return / linefeed sequence, for example, can be converted to a single `'\n'` character on input; the `'\n'` can be expanded to `"\r\n"` on output. This conversion mode is called*text mode* , as opposed to *binary mode* .. In binary mode, no such conversions are performed.

The mode flag `ios_base::binary` has the effect of opening a file in binary mode to be used for reading text files. In this case, the automatic conversion will be suppressed; however, it must be set for reading binary files.

The effect of the binary open mode is frequently misunderstood. It does *not* put the inserters and extractors into a binary mode, and hence suppress the formatting they usually perform. Binary input and output is done solely by `basic_istream <charT> ::read()` and `basic_ostream <charT> ::write()`.

1.5.6 File Positioning

1.6 In-Memory Input/Output

Iostreams supports not only input and output to external devices like files. It also allows in-memory parsing and formatting. Source and sink of the characters read or written becomes a string held somewhere in memory. You use in-memory I/O if the information to be read is already available in the form of a string, or if the formatted result will be processed as a string. For example, to interpret the contents of the string argv[1] as an integer value, the code might look like this:

```
int i;
if (istringstream(argv[1]) >> i)                          \\1
   // use the value of i
```

//1 The argument of the input string stream constructor is a string. Here the character array argv[1] is implicitly converted to a string, and the integer value extracted from the string argv[1].

The inverse operation, taking a value and converting it to characters that are stored in a string, might look like this:

```
struct date {
  int day,month,year;
} today = {8,4,1996};
ostringstream ostr;                                           \\1
ostr << today.month << '-' << today.day <<'-' << today.year;  \\2
if (ostr)
   display(ostr.str());                                       \\3
```

//1 An output string stream is allocated.

//2 Values are inserted into the output string stream. Remember that the manipulator ends inserts an end-of-string.

//3 The result of the formatting can be retrieved in the form of a string, which is returned by str().

As with file streams, there are three class templates that implement string streams: ***basic_istringstream <charT,traits>***, ***basic_ostringstream <charT,traits>***, and ***basic_stringstream <charT,traits>***. They are derived from the stream base classes ***basic_istream <charT, traits>*** and ***basic_ostream <charT, traits>***. Therefore, they inherit all the functions for formatted input and output described in Section 1.3, as well as the stream state. They also have functions for setting and retrieving the string that serves as source or sink, and constructors that allow you to set the string before construction time. For convenience, there are the regular typedefs istringstream, ostringstream, and stringstream, with wistringstream, wostringstream, and wstringstream for the respective tiny and wide character string streams.

The buffering is done via a specialized stream buffer class, ***basic_stringbuf <charT,traits>***.

1.6.1 The Internal Buffer

String streams can take a string, provided either as an argument to the constructor, or set later on via the `str(const basic_string<charT>&)` function. This string is copied into an internal buffer, and serves as source or sink of characters to subsequent insertions or extractions. Each time the string is retrieved via the `str()` function, a copy of the internal buffer is created and returned.[9]

Output string streams are *dynamic*.[10] The internal buffer is allocated once an output string stream is constructed. The buffer is automatically extended during insertion each time the internal buffer is full.

Input string streams are always *static*. You can extract as many items as are available in the string you provided the string stream with.

1.6.2 The Open Modes

The only open modes that have an effect on string streams are `in`, `out`, `ate`, and `app`. They have more or less the same meaning that they have with file streams (see section 1.5.4). The only difference for `ate` and `app` is that they write at the end-of-string, instead of the end-of-file; that is, they overwrite the end-of-string character.

The `binary` open mode is irrelevant, because there is no conversion to and from the dependent file format of the operating system. The `trunc` open mode is simply ignored.

---

[9] Note that you cannot provide or retrieve strings of any other string type than `basic_string<charT>`, where `charT` is the character type that was used to instantiate the string stream itself. The string template class with 3 template parameters is: `basic_string <charT, traits, Allocator>`. Strings with traits or an allocator that are different from the default traits or allocator cannot be used with string streams. At this printing, this is still an open issues in iostreams.

[10] This was different in the old iostreams, where you could have dynamic and static output streams. See section 1.14.4 for further details.

1.7 Input/Output of User-Defined Types

1.8 Manipulators

1.9 Locales

1. attaching locales

1.10 Extending Streams

1. xalloc & pword
2. derivation of new stream classes
3. initialization of statics, i.e. the predefined streams

1.11 Stream Buffers

1. class streambuf - the sequence abstraction
2. deriving new stream buffer classes
3. connecting iostream and streambuf objects

1.12 Defining A Code Conversion Facet

1. requirements imposed on the code conversion facet by the stream buffer
2. defining a tiny character code conversion facet (ASCII <-> EBCDIC)
3. defining a wide character code conversion (JIS <-> Unicode)

1.13 User-Defined Character Types

1. defining the right traits

1.14 Differences from the Traditional Iostreams[11]

The standard iostreams differs substantially from the traditional iostreams. This section briefly describes the main differences.

---

[11]This is a preliminary release of the users guide. The list in this section is not meant to be complete.

1.14.1 The Character Type

You may already have used iostreams in the past — the traditional iostreams. The iostreams included in the Standard C++ Library are mostly compatible, yet slightly different from what you know. The most apparent change is that the new iostream classes are templates, taking the type of character as a template parameter.

The traditional iostreams were of limited use: they could handle only byte streams, i.e., read files byte per byte, and work internally with a buffer of bytes. They had problems with languages that have alphabets containing thousands of characters. These are encoded as multibytes for storage on external devices like files, and represented as wide characters internally. They require a code conversion with each input and output.

The new templatized iostreams can handle this. They can be instantiated for one-byte skinny characters of type `char`, and for wide characters of type `wchar_t`. You can even instantiate iostream classes for any user-defined character type. Section 1.13 describes in detail how this can be done.

1.14.2 Internationalization

Another new feature is the internationalization of iostreams. The traditional iostreams were incapable of adjusting themselves to local conventions. Output of numerical items was always done following the US English conventions for number formatting. The new iostreams are internationalized. They use the standard locales described in the section on locales.

1.14.3 File Streams

1.14.3.1 Connecting Files and Streams

The traditional iostreams supported a file stream constructor, taking a file descriptor that allowed connection of a file stream to an already open file. This is no longer available in the Standard iostreams.

The functions `attach()` and `detach()` do not exist anymore.

1.14.3.2 Open Modes

In the traditional iostreams, if you open a file for writing only, i.e., open mode is `out`, then the initial file length will be set to zero. In other words, the file's content will be lost. If you open the file in at-end or append mode, the file content will be retained, and you will extend the file rather than overwriting it.

The truncate open mode has the effect of discarding the file content. This behavior is implied when a file is opened for writing only, i.e., open mode `out` is equivalent to `out|trunc`.

1.14.3.3 The File Buffer

Due to changes in the iostream architecture, the file buffer is now contained as a data member in the file stream classes. In some old implementations, the buffer was inherited from a base class called `fstreambase`.

The old file streams had a destructor; the new file streams don't need one. Flushing the buffer and closing the file is now done by the file buffer's destructor.

1.14.4 String Streams

Output string streams are always dynamic. The `str()` function does not have the functionality of freezing the string stream anymore. Instead, the string provided via `str()` is copied into the internal buffer; it is not used as the internal buffer. Accordingly, the string returned via `str()` is always a copy of the internal buffer.

If you need to influence a string stream's internal buffering, you have to do it via `pubsetbuf()`.

The classes `strstream`, `istrstream`, `ostrstream` and `strstreambuf` are deprecated features in the standard iostreams, i.e. the are still provided by this implementation of the standard iostream and will be omitted in the future.

1.15 Differences from the Standard IOStreams

This section describes differences from the ISO/ANSI standard C++ library specification. Whenever you use one of the features described in this section you should keep in mind that this usage will impair the portability of your program. It will not conform to the standard.

1.15.1 Extensions

Rogue Wave's implementation of the standard iostreams has several extensions

1.15.1.1 **File Descriptors**

The traditional iostreams allowed to connect a file stream to a file using *a file descriptor*. File descriptors are used by functions like `open()`, `close()`, `read()` and `write()`, that are part of most C libraries, especially on Unix based platforms. However, the ISO/ANSI standard for the programming language C and its library does not include these functions, nor does it mention file descriptors. In this sense the use of file descriptors introduces platform and operating system dependencies into your program. This exactly is the reason why the standard iostreams does not use file descriptors.

However, you might already have programs that use the file descriptor features of the traditional iostreams. You might as well need to access system specific files like pipes, which are only accessible via file descriptors. In order to address these concerns, Rogue Wave's implementation offers additional constructors and member functions in the file stream and file buffer classes that enable you to work with file descriptors.

The main additions are:

- There are constructors that take a file descriptor rather than a file name.
- Several constructors and the open() member functions have an additional third parameter that allows to specify the file access rights, which is not possible with the standard interface. This parameter has a default, so that you usually need not care about the file protection.

1.15.2 Restrictions

Rogue Wave's implementation of the standard iostreams has several restrictions, most of which are due to current compilers' limited capabilities of handling standard C++.

- Member Templates
- Explicit Template Argument Specification (use_facet, has_facet in locale)

1.15.3 Deprecated Features

- the `strstream` classes

Appendix: Open Issues in the Standard

1. `setbuf()` is a virtual function that is overwritten in filebuf and stringbuf. Neither of them is specified.
2. The memory allocation strategy of `operator<<()` for strings is not specified.
3. The behavior of `pubsetbuf(buf,size)` in the case of a NULL buffer is not specified.
4. It is not specified whether streams throw exceptions in case of memory shortage. For example, if the allocation of memory for iword and pword fails, will an exception be thrown? Which exception will be thrown: `bad_alloc` or `ios_failure`? Which bit will be set: `badbit` or `failbit`?
5. String streams only take ***basic_string<charT>*** ,where `charT` is the string stream's character type. You cannot provide or retrieve strings that have traits or an allocator default from the default traits or default allocator for strings.