
15.10. 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 capability 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 by using string streams.

Before the standardization of C++98, 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. 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.

15.10.1. String Stream Classes

The following stream classes — corresponding to the stream classes for files — are defined for strings:

- The class template `basic_istringstream<>` with the specializations `istringstream` and `wistringstream` for reading from strings ("input string stream")
- The class template `basic_ostringstream<>` with the specializations `ostringstream` and `wostringstream` for writing to strings ("output string stream")
- The class template `basic_stringstream<>` with the specializations `stringstream` and `wstringstream` for reading from and writing to strings
- The class template `basic_stringbuf<>` with the specializations `stringbuf` and `wstringbuf` , used by the other string stream classes to perform the 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 15.3.

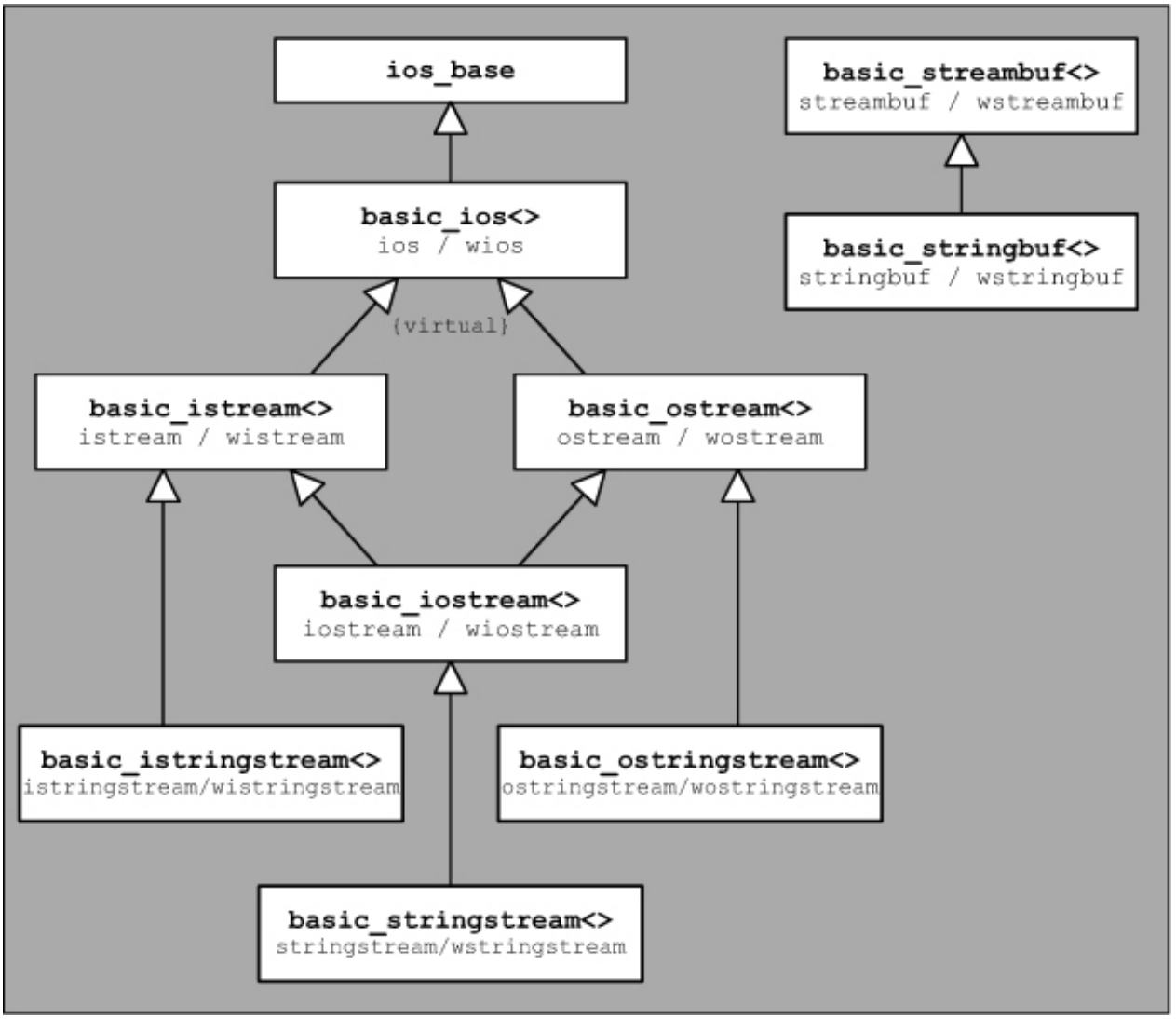

Figure 15.3. Class Hierarchy of the String Stream Classes

The classes are declared in the header file `<sstream>` like this:

[Click here to view code image]

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

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

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

    template <typename charT,
              typename traits = char_traits<charT>,
              typename 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()`, which is used to manipulate the buffer of the string stream classes (Table 15.38).

Table 15.38. Fundamental Operations for String Streams

Member Function	Meaning
`str()`	Returns the buffer as a string
`str(`*string*`)`	Sets the contents of the buffer to *string*

The following program demonstrates the use of string streams:

Click here to view code image

```
// io/sstream1.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. To remove the current contents from the stream, you can use the function `str()` to assign new contents to the buffer:

```
strm.str("");
```

The first lines written to `os` are each terminated with `endl`. This means that the string ends with a newline character. Because the string is printed followed by `endl`, two adjacent newline characters are written. This explains the empty lines in the output.

Before C++11, a typical programming error when dealing with string streams was to forget to extract the string with the function `str()` and to write to the stream directly instead. This was, from a compiler's point of view, a possible and reasonable thing to do because there was an implicit conversion to `void*` (see Section 15.4.3, page 760). As a result, instead of its value the state of the stream was written in the form of an address (see Section 15.3.3, page 756). With C++11, this conversion was replaced by an explicit conversion to `bool`, so passing a string stream to the output operator `<<` without calling `str()` is no longer possible.

A typical use for writing to an output string stream is to define output operators for user-defined types (see Section 15.11.1, page 810).

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 `x` with the value `3` and the floating-point `f` with the value `0.7` from the string `s`:

```
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 15.9.3, page 796) and/or an existing string. With the flag `ios::ate`, the characters written to a string stream can be appended to an existing string:[15]

[15] Whether the flag `ios::app` has the same effect is currently not clear; so using it here instead is not portable.

Click here to view code image

```
std::string s("value: ");
...
std::ostringstream os (s, std::ios::out|std::ios::ate);
os << 77 << " " << std::hex << 77 << std::endl;
std::cout << os.str();  //writes: value: 77 4d
std::cout << s;         //writes: value:
```

As you can see, the string returned from `str()` is a copy of the string `s`, with a decimal and a hexadecimal version of `77` appended. The string `s` itself is not modified.

15.10.2. Move Semantics for String Streams

Since C++11, string streams provide rvalue and move semantics. In fact, ostreams provide an output operator, and istreams provide an input iterator that accepts an rvalue reference for the stream. The effect is that you can use temporarily created stream objects. For example, you can insert into a temporarily created string stream:[16]

[16] Thanks to Daniel Krügler for providing this example.

Click here to view code image

```
// io/sstream2.cpp

#include <iostream>
#include <sstream>
#include <string>
#include <tuple>
#include <utility>
using namespace std;

tuple<string,string,string> parseName(string name)
{
    string s1, s2, s3;
    istringstream(name) >> s1 >> s2 >> s3;
    if (s3.empty()) {
        return tuple<string,string,string>(move(s1),"",move(s2));
    }
    else {
        return tuple<string,string,string>(move(s1),move(s2),move(s3));
    }
}

int main()
{
    auto t1 = parseName("Nicolai M. Josuttis");
    cout << "firstname: " << get<0>(t1) << endl;
    cout << "middle:    " << get<1>(t1) << endl;
    cout << "lastname:  " << get<2>(t1) << endl;
    auto t2 = parseName("Nico Josuttis");
    cout << "firstname: " << get<0>(t2) << endl;
    cout << "middle:    " << get<1>(t2) << endl;
    cout << "lastname:  " << get<2>(t2) << endl;
}
```

Before C++11, you had to implement

```
istringstream is(name);
is >> s1 >> s2 >> s3;
```

instead of

```
istringstream(name) >> s1 >> s2 >> s3;
```

In addition, string streams now have move and swap semantics, providing a move constructor, a move assignment operator, and `swap()`. So, you can pass a string stream as an argument or return a string stream from a function.

15.10.3. `char*` Stream Classes

The `char*` stream classes are retained only for backward compatibility. Their interface is error prone, and the classes are rarely used correctly. However, they are still in 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 *character sequence* will be used instead of *string* because the character sequence maintained by the `char*` stream classes is not always terminated with the (terminating) null character and thus is not really a string.

The `char*` stream classes are defined only for the character type `char`. They include

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

The `char*` stream classes are defined in the header file `<strstream>`.

An `istrstream` can be initialized with a character sequence (of type `char*`) that is either terminated with the (terminating) null 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:

```
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 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, the character sequence has to be released unless the stream was initialized with a buffer of fixed size. However, there is no guarantee how the memory was allocated,[17] so it is not always safe to release it by 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).

[17] There is 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 (terminating) null character (`'\0'`). This character has to be appended explicitly to the stream to terminate the character sequence. This can be done by using the `ends` manipulator. Some implementations append a string (terminating) null character automatically, but this behavior is not portable.

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

```
float x;
...
// create and fill char* stream
// - don't forget ends or '\0' !!!
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, mainly because the strings are copied and because the string class takes care of the used memory.