
15.13. The Stream Buffer Classes

As mentioned in Section 15.2.1, page 749, reading and writing are not done by the streams directly but are delegated to stream buffers.

The general interface to deal with stream buffers is pretty simple (see Section 15.12.2, page 820):

- `rdbuf()` yields a pointer to the stream buffer of a stream.
- The constructor and `rdbuf()` of streams allow setting a stream buffer at construction time or changing the stream buffer while the stream exists. In both cases, you have to pass a pointer to the stream buffer, which is what `rdbuf()` returns.

This ability can be used to let streams write to the same output device (see Section 15.12.2, page 820, to redirect streams (see Section 15.12.3, page 822), read from and write to the same buffer (see Section 15.12.4, page 824), or use other character encodings, such as UTF-8 or UTF-16/UCS-2, as input and output format (see Section 16.4.4, page 903).

This section describes how the stream buffer classes operate. The discussion not only gives a deeper understanding of what is going on when I/O streams are used but also provides the basis to define new I/O channels. Before going into the details of stream buffer operation, the public interface is presented for readers interested only in using stream buffers.

15.13.1. The Stream Buffer Interfaces

To the user of a stream buffer, the class `basic_streambuf<>` is not much more than something that characters can be sent to or extracted from. Table 15.42 lists the public function for writing characters.

Table 15.42. Public Members for Writing Characters

Member Function	Meaning
`sputc(`*c*`)`	Sends the character *c* to the stream buffer
`sputn(`*s*`, `*n*`)`	Sends *n* characters from the sequence *s* to the stream buffer

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

The interface to reading characters from a stream buffer is a little bit more complex (Table 15.43) because for input, it is necessary to have a look at a character without consuming it. Also, it is desirable that characters can be put back into the stream buffer when parsing. Thus, the stream buffer classes provide corresponding functions.

Table 15.43. Public Members for Reading Characters

Member Function	Meaning
`in_avail()`	Returns a lower bound on the characters available
`sgetc()`	Returns the current character without consuming it
`sbumpc()`	Returns the current character and consumes it
`snextc()`	Consumes the current character and returns the next character
`sgetn(`*b*`, `*n*`)`	Reads *n* characters and stores them in the buffer *b*
`sputbackc(`*c*`)`	Returns the character *c* to the stream buffer
`sungetc()`	Moves one step back to the previous character

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

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

The two functions `sputbackc()` and `sungetc()` are used to move one step back, making the previous character the current

one. The function `sputbackc()` can be used to replace the previous character by another character. These two functions should be used only with care. Often, only one character can be put back.

Finally, there are functions to access the imbued locale object, to change the position, and to influence buffering. Table 15.44 lists these functions.

Table 15.44. Miscellaneous Public Stream Buffer Functions

Member Function	Meaning
`pubimbue(`*loc*`)`	Imbues the stream buffer with the locale *loc*
`getloc()`	Returns the current locale
`pubseekpos(`*pos*`)`	Repositions the current position to an absolute position
`pubseekpos(`*pos*`, `*which*`)`	Same with specifying the I/O direction
`pubseekoff(`*offset*`, `*rpos*`)`	Repositions the current position relative to another position
`pubseekoff(`*offset*`, `*rpos*`, `*which*`)`	Same with specifying the I/O direction
`pubsetbuf(`*buf*`, `*n*`)`	Influences buffering

Both `pubimbue()` and `getloc()` are used for internationalization (see Section 15.8, page 790): `pubimbue()` installs a new locale object in the stream buffer, returning the previously installed locale object; `getloc()` returns the currently installed locale object.

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

The functions `pubseekoff()` and `pubseekpos()` are used to manipulate the current position used for reading and/or writing. The position that is manipulated depends on the last argument, which is of type `ios_base::openmode` and which defaults to `ios_base::in|ios_base::out` if it is not specified. If `ios_base::in` is set, the read position is modified. Correspondingly, the write position is modified if `ios_base::out` is set. The function `pubseekpos()` moves the stream to an absolute position specified as the first argument, whereas the function `pubseekoff()` moves the stream relative to some other position. The offset is specified as the first argument. The position used as starting point is specified as the second argument and can be `ios_base::cur`, `ios_base::beg`, or `ios_base::end` (see Section 15.9.4, page 800, for details). Both functions return the position to which the stream was positioned or an invalid stream position. The invalid stream position can be detected by comparing the result with the object `pos_type(off_type(-1))` (`pos_type` and `off_type` are types for handling stream positions; see Section 15.9.4, page 799). The current position of a stream can be obtained by using `pubseekoff()`:

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

15.13.2. Stream Buffer Iterators

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

The class templates `istreambuf_iterator<>` and `ostreambuf_iterator<>` are used to read or to write individual characters from or to objects of type `basic_streambuf<>`, respectively. The classes are defined in the header `<iterator>` like this:

Click here to view code image

```
namespace std {
    template <typename charT,
              typename traits = char_traits<charT> >
             class istreambuf_iterator;
    template <typename charT,
              typename traits = char_traits<charT> >
             class ostreambuf_iterator;
}
```

These iterators are special forms of stream iterators, which are described in Section 9.4.3, page 460. The only difference is that their elements are characters.

Output Stream Buffer Iterators

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

Click here to view code image

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

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

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

Table 15.45 lists all operations of output stream buffer iterators. The implementation is similar to ostream iterators (see Section 9.4.3, page 460). In addition, you can initialize the iterator with a buffer, and you can call `failed()` to query whether the iterator is able to write. If any prior writing of a character failed, `failed()` yields `true`. In this case, any writing with operator `=` has no effect.

Table 15.45. Operations of Output Stream Buffer Iterators

Expression	Effect
`ostreambuf_iterator<`*char*`>(`*ostream*`)`	Creates an output stream buffer iterator for *ostream*
`ostreambuf_iterator<`*char*`>(`*buffer_ptr*`)`	Creates an output stream buffer iterator for the buffer to which *buffer_ptr* refers
**iter*	No-op (returns *iter*)
iter `=` *c*	Writes character *c* to the buffer by calling `sputc(`*c*`)` for it
++iter	No-op (returns *iter*)
iter++	No-op (returns *iter*)
`failed()`	Returns whether the output stream iterator is not able to write anymore

Input Stream Buffer Iterators

Table 15.46 lists all operations of input stream buffer iterators. The implementation is similar to that for istream iterators (see Section 9.4.3, page 462). In addition, you can initialize the iterator with a buffer, and member function `equal()` is provided, which returns whether two input stream buffer iterators are equal. Two input stream buffer iterators are equal when they are both end-of-stream iterators or when neither is an end-of-stream iterator.

Table 15.46. Operations of Input Stream Buffer Iterators

Expression	Effect
`istreambuf_iterator<`*char*`>()`	Creates an end-of-stream iterator
`istreambuf_iterator<`*char*`>(`*istream*`)`	Creates an input stream buffer iterator for *istream* and might read the first character using `sgetc()`
`istreambuf_iterator<`*char*`>(`*buffer_ptr*`)`	Creates an input stream buffer iterator for the buffer to which *buffer_ptr* refers and might read the first character using `sgetc()`
**iter*	Returns the current character, read with `sgetc()` before (reads the first character if not done by the constructor)
++iter	Reads the next character with `sbumpc()` and returns its position
iter++	Reads the next character with `sbumpc()` but returns an iterator (proxy), where `*` yields the previous character
iter1`.equal(`*iter2*`)`	Returns whether both iterators are equal
iter1 `==` *iter2*	Tests *iter1* and *iter2* for equality
iter1 `!=` *iter2*	Tests *iter1* and *iter2* for inequality

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

1. A range from the current position in a stream to the end of the stream is defined by two iterators: `istreambuf_iterator<charT,traits>` (*stream*) for the current position and `istreambuf_iterator<charT,traits>()` for the end of the stream (*stream* is of type `basic_istream<charT,traits>` or `basic_streambuf<charT,traits>`).
2. It is not possible to create subranges using `istreambuf_iterator`s.

Example Use of Stream Buffer Iterators

The following example is the classic filter framework that simply writes all read characters with stream buffer iterators. It is a modified version of the example in Section 15.5.3, page 772:

Click here to view code image

```
// io/charcat2.cpp

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

int main()
{
    // input stream buffer iterator for cin
    istreambuf_iterator<char> inpos(cin);

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

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

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

You can also pass stream buffer iterators to algorithms to process all characters read from an input stream (see *io/countlines1.cpp* for a complete example):

Click here to view code image

```
int countLines (std::istream& in)
{
    return std::count(std::istreambuf_iterator<char>(in),
                      std::istreambuf_iterator<char>(),
                      '\n');
}
```

See Section 14.6, page 732, for an example using all characters read from standard input to initialize a string.

15.13.3. User-Defined Stream Buffers

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

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

User-Defined Output Buffers

A buffer used to write characters is maintained with three pointers that can be accessed by the three functions `pbase()`,

`pptr()` , and `epptr()` (Figure 15.4). Here is what these pointers represent:

1. `pbase()` ("put base") is the beginning of the output buffer.
2. `pptr()` ("put pointer") is the current write position.
3. `epptr()` ("end put pointer") is the end of the output buffer. This means that `epptr()` points to one past the last character that can be buffered.

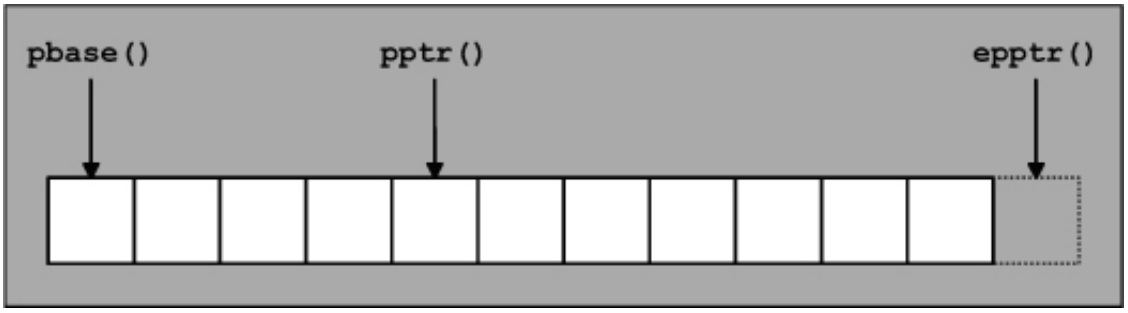

Figure 15.4. Interface to the Output Buffer

The characters in the range from `pbase()` to `pptr()` , not including the character pointed to by `pptr()` , are already written but not yet transported, or flushed, to the corresponding output channel.

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

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

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

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

```
// io/outbuf1.hpp

#include <streambuf>
#include <locale>
#include <cstdio>

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

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

In this case, each character sent to the stream buffer is written using the C function `putchar()` . However, before the character is written, it is turned into an uppercase character using `toupper()` (see Section 16.4.4, page 895). The function `getloc()` is used to get the locale object associated with the stream buffer (see also Section 15.8, page 790).

In this example, the output buffer is implemented specifically for the character type `char` (`streambuf` is the specialization of

`basic_streambuf<>` for the character type `char`). If other character types are used, you have to implement this function using character traits, which are introduced in Section 16.1.4, page 853. In this case, the comparison of `c` with end-of-file looks different: `traits::eof()` has to be returned instead of `EOF` and, if the argument `c` is EOF, the value `traits::not_eof(c)` should be returned, where `traits` is the second template argument to `basic_streambuf`. This might look as follows:

Click here to view code image

```
// io/outbuf1i18n.hpp

#include <streambuf>
#include <locale>
#include <cstdio>

template <typename charT,
          typename traits = std::char_traits<charT> >
class basic_outbuf : public std::basic_streambuf<charT,traits>
{
  protected:
    // central output function
    // - print characters in uppercase mode
    virtual typename traits::int_type
            overflow (typename traits::int_type c) {
        if (!traits::eq_int_type(c,traits::eof())) {
            // convert lowercase to uppercase
            c = std::toupper(c,this->getloc());

            // convert the character into a char (default: '?')
            char cc = std::use_facet<std::ctype<charT>>
                           (this->getloc()).narrow(c,'?');

            // and write the character to the standard output
            if (std::putchar(cc) == EOF) {
                return traits::eof();
            }
        }
        return traits::not_eof(c);
    }
};

typedef basic_outbuf<char>    outbuf;
typedef basic_outbuf<wchar_t> woutbuf;
```

Note that you have to qualify the call of `getloc()` by `this->` now because the base class depends on a template parameter. Also, we have to narrow the character before we pass it to `putchar()` because `putchar()` only accepts `char` only (see Section 16.4.4, page 891).

Using this stream buffer in the following program:

Click here to view code image

```
// io/outbuf1.cpp

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

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

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

produces the following output:

```
31 HEXADECIMAL: 1F
```

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

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

addition, a class derived from `ostream` is defined that maintains such a stream buffer to which the file descriptor is passed:

Click here to view code image

```
// io/outbuf2.hpp

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

// for write():
#ifdef _MSC_VER
#include <io.h>
#else
#include <unistd.h>
#endif

class fdoutbuf : public std::streambuf {
  protected:
    int fd;     // file descriptor

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

class fdostream : public std::ostream {
  protected:
    fdoutbuf buf;
  public:
    fdostream (int fd) : std::ostream(0), buf(fd) {
        rdbuf(&buf);
    }
};
```

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

Click here to view code image

```
// io/outbuf2.cpp

#include <iostream>
#include "outbuf2.hpp"

int main()
{
    fdostream out(1);      // stream with buffer writing to file descriptor 1

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

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

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

Click here to view code image

```
// io/outbuf3.hpp

#include <cstdio>
#include <streambuf>

// for write():
#ifdef _MSC_VER
# include <io.h>
#else
# include <unistd.h>
#endif

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

  public:
    // constructor
    // - initialize data buffer
    // - one character less to let the bufferSizeth character cause a call of
overflow()
    outbuf() {
        setp (buffer, buffer+(bufferSize-1));
    }

    // destructor
    // - flush data buffer
    virtual ~outbuf() {
        sync();
    }

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

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

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

The constructor initializes the write buffer with `setp()`:

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

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

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

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

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

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

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

User-Defined Input Buffers

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

A stream buffer maintains a read buffer with three pointers that can be accessed through the member functions `eback()`, `gptr()`, and `egptr()` (Figure 15.5):

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

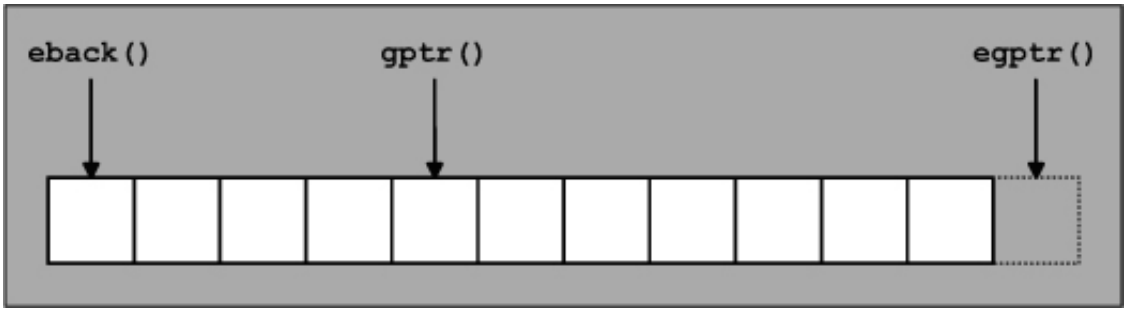

Figure 15.5. Interface for Reading from Stream Buffers

The characters between the read position and the end position have been transported from the external representation to the program's memory, but they still await processing by the program.

Single characters can be read using the function `sgetc()` or `sbumpc()`. These two functions differ in that the read pointer is incremented by `sbumpc()` but not by `sgetc()`. If the buffer is read completely (`gptr() == egptr()`), no character is available, and the buffer has to be refilled by a call of the virtual function `underflow()`, which is responsible for the reading of data. If no characters are available, the function `sbumpc()` calls the virtual function `uflow()` instead. The default implementation of `uflow()` is to call `underflow()` and then increment the read pointer. The default implementation of `underflow()` in the base class `basic_streambuf` is to return EOF. This means that it is impossible to read characters with the default implementation.

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

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

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

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

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

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

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

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

Click here to view code image

```
// io/inbuf1.hpp

#include <cstdio>
#include <cstring>
#include <streambuf>

// for read():
#ifdef _MSC_VER
# include <io.h>
#else
# include <unistd.h>
#endif

class inbuf : public std::streambuf {

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

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

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

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

```
            }

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

            // read new characters
            int num;
            num = read (0, buffer+4, bufferSize-4);
            if (num <= 0) {
                // ERROR or EOF
                return EOF;
            }

            // reset buffer pointers
            setg (buffer+(4-numPutback),    // beginning of putback area
                  buffer+4,                 // read position
                  buffer+4+num);            // end of buffer

            // return next character
            return traits_type::to_int_type(*gptr());
        }
    };
```

The constructor initializes all pointers so that the buffer is completely empty (Figure 15.6). If a character is read from this stream buffer, the function `underflow()` is called. This function, always used by this stream buffer to read the next characters, starts by checking for read characters in the input buffer. If characters are present, they are moved to the putback area by using the function `memcpy()`. These are, at most, the last four characters of the input buffer. Then POSIX's low-level I/O function `read()` is used to read the next character from the standard input channel. After the buffer is adjusted to the new situation, the first character read is returned.

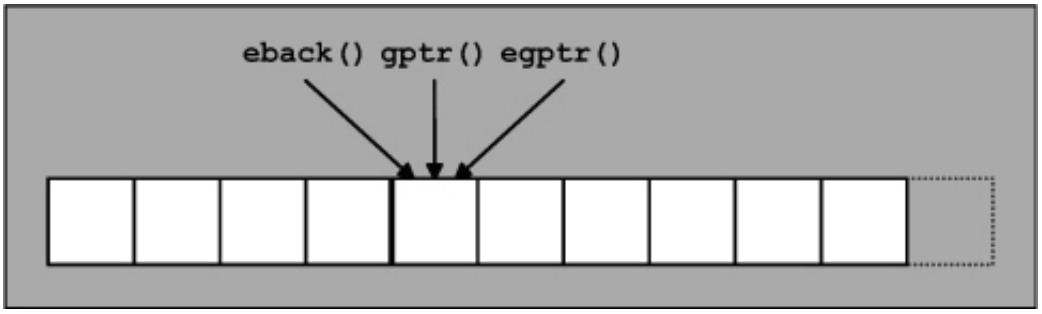

Figure 15.6. Get Buffer after Initialization

For example, if the characters `'H'`, `'a'`, `'l'`, `'l'`, `'o'`, and `'w'` are read by the first call to `read()`, the state of the input buffer changes, as shown in Figure 15.7. The putback area is empty because the buffer was filled for the first time, and there are no characters yet that can be put back.

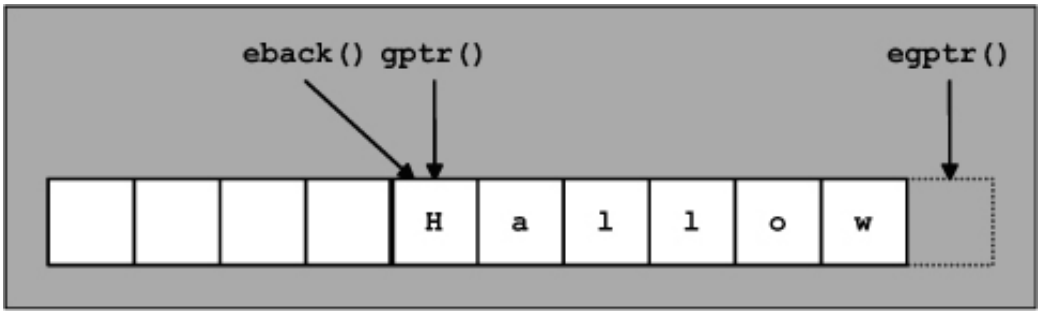

Figure 15.7. Get Buffer after Reading *Hallow*

After these characters are extracted, the last four characters are moved into the putback area, and new characters are read. For example, if the characters `'e'`, `'e'`, `'n'`, and `'\n'` are read by the next call of `read()`, the result is as shown in Figure 15.8.

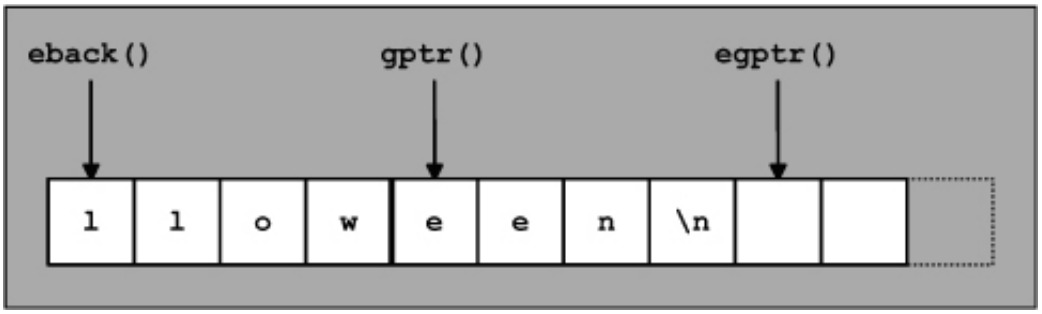

Figure 15.8. Get Buffer after Reading Four More Characters

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

Click here to view code image

```
// io/inbuf1.cpp

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

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

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

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

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

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