
15.12. Connecting Input and Output Streams

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

15.12.1. Loose Coupling Using `tie()`

You can *tie* a stream to an output stream. This means that the buffers of both streams are synchronized in a way that the buffer of the output stream is flushed before each input or output of the other stream. That is, for the output stream, the function `flush()` is called. Table 15.40 lists the member functions defined in `basic_ios` to tie one stream to another.

Table 15.40. Tieing One Stream to Another

Member Function	Meaning
`tie()`	Returns a pointer to the output stream that is tied to the stream
`tie(`*ostream* strm*`)`	Ties the output stream to which the argument refers to the stream and returns a pointer to the previous output stream that was tied to the stream, if any

Calling the function `tie()` without any argument returns a pointer to the output stream that is currently tied to a stream. To tie a new output stream to a stream, a pointer to that output stream must be passed as the argument to `tie()`. The argument is defined to be a pointer because you can also pass `nullptr` (or `0` or `NULL`) as an argument. This argument means "no tie" and unties any tied output stream. If no output stream is tied, `tie()` returns `nullptr` or `0`. For each stream, you can have only one output stream that is tied to this stream. However, you can tie an output stream to different streams.

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

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

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

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

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

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

```
// decouple cin from any output stream
std::cin.tie (nullptr);
```

This might improve the performance of a program by avoiding unnecessary additional flushing of streams (see Section 15.14.2, page 846, for a discussion of stream performance).

You can also tie one output stream to another output stream. For example, with the following statement, the normal output is flushed before something is written to the error stream:

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

15.12.2. Tight Coupling Using Stream Buffers

Using the function `rdbuf()`, you can couple streams tightly by using a common stream buffer (Table 15.41). These functions suit several purposes, which are discussed in this and the following subsections.

Table 15.41. Stream Buffer Access

Member Function	Meaning
`rdbuf()`	Returns a pointer to the stream buffer
`rdbuf(`*streambuf**`)`	Installs the stream buffer pointed to by the argument and returns a pointer to the previously used stream buffer

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

```
// io/streambuffer1.cpp

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

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

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

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

Click here to view code image

```
// io/streambuffer2.cpp

#include <iostream>
#include <fstream>

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

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

}   // does NOT close buffer

int main()
{
    using namespace std;
    int num = 5;

    cout << "We print " << num
         << " lines hexadecimal" << endl;

    hexMultiplicationTable(cout.rdbuf(),num);

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

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

```
We print 5 lines hexadecimal
0x1 0x2 0x3 0x4 0x5 0x6 0x7 0x8 0x9 0xa
```

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

However, the disadvantage of this approach is that construction and destruction of a stream object involve more overhead than just setting and restoring some format flags. Also, note that the destruction of a stream object does not flush the buffer. To make sure that an output buffer is flushed, it has to be flushed manually.

The fact that the stream buffer is not destroyed applies only to `basic_istream` and `basic_ostream`. The other stream classes destroy the stream buffers they allocated originally, but they do not destroy stream buffers set with `rdbuf()`.

15.12.3. Redirecting Standard Streams

In the old implementation of the IOStream library, the global streams `cin`, `cout`, `cerr`, and `clog` were objects of the classes `istream_withassign` and `ostream_withassign`. It was therefore possible to redirect the streams by assigning streams to other streams. This possibility was removed from the C++ standard library. However, the possibility to redirect streams was retained and extended to apply to all streams. A stream can be redirected by setting a stream buffer.

The setting of stream buffers means the redirection of I/O streams controlled by the program without help from the operating system. For example, the following statements set things up such that output written to `cout` is not sent to the standard output channel but to the file `cout.txt`:

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

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

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

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

Click here to view code image

```
// io/streamredirect1.cpp

#include <iostream>
#include <fstream>
#include <memory>

using namespace std;

void redirect(ostream&);

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

    redirect(cout);

    cout << "the last row" << endl;
}

void redirect (ostream& strm)
{
    // save output buffer of the stream
    // - use unique pointer with deleter that ensures to restore
    //   the original output buffer at the end of the function
    auto del = [&](streambuf* p) {
                   strm.rdbuf(p);
               };
    unique_ptr<streambuf,decltype(del)> origBuffer(strm.rdbuf(),del);

    // redirect ouput into the file redirect.txt
    ofstream file("redirect.txt");
    strm.rdbuf (file.rdbuf());

    file << "one row for the file" << endl;
```

```
    strm << "one row for the stream" << endl;
} // closes file AND its buffer automatically
```

By using a unique pointer (see Section 5.2.5, page 98), we can ensure that, even when `resize()` is left due to an exception, the original output buffer stored in `origBuffer` gets restored.[19]

[19] Thanks to Daniel Krügler for pointing this out.

The output of the program is this as follows:

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

The contents of the file `redirect.txt` are afterward:

```
one row for the file
one row for the stream
```

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

15.12.4. Streams for Reading and Writing

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

Click here to view code image

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

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

Click here to view code image

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

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

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

Click here to view code image

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

`filebuf` is the usual specialization of the class `basic_filebuf<>` for the character type `char`. This class defines the stream buffer class used by file streams.

The following program is a complete example. In a loop, four lines are written to a file. After each writing of a line, the contents of the file are written to standard output:

Click here to view code image

```
// io/streamreadwrite1.cpp

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

int main()
{
    // open file "example.dat" for reading and writing
    filebuf buffer;
    ostream output(&buffer);
    istream input(&buffer);
    buffer.open ("example.dat", ios::in | ios::out | ios::trunc);

    for (int i=1; i<=4; i++) {
```

```
            // write one line
            output << i << ". line" << endl;

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

The output of the program is as follows:

```
1. line

1. line
2. line

1. line
2. line
3. line

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

Although two different stream objects are used for reading and writing, the read and write positions are tightly coupled. `seekg()` and `seekp()` call the same member function of the stream buffer.[20] Thus, the read position must always be set to the beginning of the file in order for the complete contents of the file to be written, after which the read/write position is again at the end of the file so that new lines written are appended.

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

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

As mentioned before, instead of processing character by character, you could also print the entire contents in one statement by passing a pointer to the stream buffer of the file as an argument to operator `<<` (see Section 15.14.3, page 846, for details):

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