
15.14. Performance Issues

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

One performance issue was mentioned in Section 15.2.3, page 752, already: You should include only those headers that are necessary to compile your code. In particular, you should avoid including `<iostream>` if the standard stream objects are not used.

15.14.1. Synchronization with C's Standard Streams

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

Depending on the implementation, this synchronization might imply some often unnecessary overhead. For example, implementing the standard C++ streams using the standard C files inhibits buffering in the corresponding stream buffers. However, the buffer in the stream buffers is necessary for some optimizations, especially during formatted reading (see Section 15.14.2, page 845). To allow switching to a better implementation, the static member function `sync_with_stdio()` is defined for the class `ios_base` (Table 15.47).

Table 15.47. Synchronizing Standard C++ and Standard C Streams

Static Function	Meaning
`sync_with_stdio()`	Returns whether the standard stream objects are synchronized with standard C streams and concurrency is supported
`sync_with_stdio(false)`	Disables the synchronization of C++ and C streams (has to be called before any I/O)

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

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

Note that you have to disable the synchronization before any other I/O operation. Calling this function after any I/O has occurred results in implementation-defined behavior.

The function returns the previous value with which the function was called. If not called before, it always returns `true` to reflect the default setup of the standard streams.

Note that since C++11, disabling the synchronization with the standard C streams also disables the concurrency support, which allows you to use the standard stream object by multiple threads, although interleaved characters are possible (see Section 4.5, page 56).

15.14.2. Buffering in Stream Buffers

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

Thus, all I/O is done using stream buffers, which implement a mechanism for buffering. However, it is not sufficient to rely solely on this buffering, because three aspects conflict with effective buffering:

1. It is often simpler to implement stream buffers without buffering. If the corresponding streams are not used frequently or are used only for output, buffering is probably not that important. (For output, the difference between stream buffer iterators and pointers is not as bad as for input; the main problem is comparing stream buffer iterators.) However, for stream buffers that are used extensively, buffering should definitely be implemented.
2. The flag `unitbuf` causes output streams to flush the stream after each output operation. Correspondingly, the manipulators `flush` and `endl` also flush the stream. For the best performance, all three should probably be avoided. However, when writing to the console, for example, it is probably still reasonable to flush the stream after writing complete lines. If you are stuck with a program that makes heavy use of `unitbuf`, `flush`, or `endl`, you might consider using a special stream buffer that does not use `sync()` to flush the stream buffer but uses another function that is called when appropriate.
3. Tieing streams with the `tie()` function (see Section 15.12.1, page 819) also results in additional flushing of streams. Thus,

streams should be tied only if it is really necessary.

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

15.14.3. Using Stream Buffers Directly

All member functions of the class `basic_istream` and `basic_ostream` that read or write characters operate according to the same schema: First, a corresponding `sentry` object is constructed, and then the operation is performed. The construction of the `sentry` object results in flushing of potentially tied objects, skipping of whitespace for input, and such implementation-specific operations as locking in multithreaded environments (see Section 15.5.4, page 772).

For unformatted I/O, most of the operations are normally useless anyway. Only locking operation might be useful if the streams are used in multithreaded environments. Thus, when doing unformatted I/O, it may be better to use stream buffers directly.

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

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

```
// io/copy1.cpp

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

  Here, `rdbuf()` yields the buffer of `cin` (see Section 15.12.2, page 820). Thus, the program copies all standard input to standard output.

- By passing a pointer to a stream buffer to operator `>>`, you can read directly into a stream buffer. For example, you could also copy all standard input to standard output in the following way:

Click here to view code image

```
// io/copy2.cpp

#include <iostream>

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

  Note that you have to clear the flag `skipws`. Otherwise, leading whitespace of the input is skipped (see Section 15.7.7, page 789).

Even for formatted I/O, it may be reasonable to use stream buffers directly. For example, if many numeric values are read in a loop, it is sufficient to construct just one `sentry` object that exists for the whole time the loop is executed. Then, within the loop, whitespace is skipped manually — using the `ws` manipulator would also construct a `sentry` object — and then the facet `num_get` (see Section 16.4.1, page 873) is used for reading the numeric values directly.

Note that a stream buffer has no error state of its own. It also has no knowledge of the input or output stream that might connect to it. So, calling

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

can't change the error state of `in` due to a failure or end-of-file.