
15.4. State of Streams

Streams maintain a state. The state identifies whether I/O was successful and, if not, the reason for the failure.

15.4.1. Constants for the State of Streams

For the general state of streams, several constants of type `iostate` are defined to be used as flags (Table 15.3). The type `iostate` is a member of the class `ios_base`. The exact type of the constants is an implementation detail; in other words, it is not defined whether `iostate` is an enumeration, a type definition for an integral type, or an instantiation of the class `bitset`.

Table 15.3. Constants of Type *iostate*

Constant	Meaning
goodbit	Everything is OK; none of the other bits is set
eofbit	End-of-file was encountered
failbit	Error; an I/O operation was not successful
badbit	Fatal error; undefined state

The constant `goodbit` is defined to have the value `0`. Thus, having `goodbit` set means that all other bits are cleared. The name `goodbit` may be somewhat confusing because it doesn't mean that any bit is set.

The difference between `failbit` and `badbit` is basically that `badbit` indicates a more fatal error:

- **`failbit`** is set if an operation was not processed correctly but the stream is generally OK. Normally, this flag is set as a result of a format error during reading. For example, this flag is set if an integer is to be read but the next character is a letter.
- **`badbit`** is set if the stream is somehow corrupted or if data is lost. For example, this flag is set when positioning a stream that refers to a file before the beginning of a file.

Note that `eofbit` normally happens with `failbit` because the end-of-file condition is checked and detected when an attempt is made to read beyond end-of-file. After reading the last character, the flag `eofbit` is not yet set. The next attempt to read a character sets `eofbit` *and* `failbit` because the read fails.

Some former implementations supported the flag `hardfail`. This flag is not supported in the standard.

These constants are not defined globally. Instead, they are defined within the class `ios_base`. Thus, you must always use them with the scope operator or with some object. For example:

```
std::ios_base::eofbit
```

Of course, it is also possible to use a class derived from `ios_base`. These constants were defined in the class `ios` in old implementations. Because `ios` is a type derived from `ios_base` and its use involves less typing, the use often looks like this:

```
std::ios::eofbit
```

These flags are maintained by the class `basic_ios` and are thus present in all objects of type `basic_istream` or `basic_ostream`. However, the stream buffers don't have state flags. One stream buffer can be shared by multiple stream objects, so the flags represent only the state of the stream as found in the last operation. Even this is the case only if `goodbit` was set prior to this operation. Otherwise, the flags may have been set by an earlier operation.

15.4.2. Member Functions Accessing the State of Streams

The current state of the flags can be determined by the member functions, as presented in Table 15.4.

Table 15.4. Member Functions for Stream States

Member Function	Meaning
`good()`	Returns `true` if the stream is OK (`goodbit` is "set")
`eof()`	Returns `true` if end-of-file was hit (`eofbit` is set)
`fail()`	Returns `true` if an error has occurred (`failbit` or `badbit` is set)
`bad()`	Returns `true` if a fatal error has occurred (`badbit` is set)
`rdstate()`	Returns the currently set flags
`clear()`	Clears all flags
`clear(`*state*`)`	Clears all and sets *state* flags
`setstate(`*state*`)`	Sets additional *state* flags

The first four member functions in Table 15.4 determine certain states and return a Boolean value. Note that `fail()` returns whether `failbit` or `badbit` is set. Although this is done mainly for historical reasons, it also has the advantage that one test suffices to determine whether an error has occurred.

In addition, the state of the flags can be determined and modified with the more general member functions. When `clear()` is called without parameters, all error flags, including `eofbit`, are cleared (this is the origin of the name *clear*):

```
// clear all error flags (including eofbit):
strm.clear();
```

If a parameter is given to `clear()`, the state of the stream is adjusted to be the state given by the parameter; that is, the flags set in the parameter are set for the stream, and the other flags are cleared. The only exception is that the `badbit` is always set if there is no stream buffer, which is the case if `rdbuf() == 0` (see Section 15.12.2, page 820, for details).

The following example checks whether `failbit` is set and clears it if necessary:

```
// check whether failbit is set
if (strm.rdstate() & std::ios::failbit) {
    std::cout << "failbit was set" << std::endl;

    // clear only failbit
    strm.clear (strm.rdstate() & ~std::ios::failbit);
}
```

This example uses the bit operators `&` and `~`: Operator `~` returns the bitwise complement of its argument. Thus, the following expression returns a temporary value that has all bits set except `failbit`:

```
~ios::failbit
```

Operator `&` returns a bitwise "and" of its operands. Only the bits set in both operands remain set. Applying bitwise "and" to all currently set flags (`rdstate()`) and to all bits except `failbit` retains the value of all other bits while `failbit` is cleared.

Streams can be configured to throw exceptions if certain flags are set with `clear()` or `setstate()` (see Section 15.4.4, page 762). Such streams always throw an exception if the corresponding flag is set at the end of the method used to manipulate the flags.

Note that you always have to clear error bits explicitly. In C, it was possible to read characters after a format error. For example, if `scanf()` failed to read an integer, you could still read the remaining characters. Thus, the read operation failed, but the input stream was still in a good state. This is different in C++. If `failbit` is set, each following stream operation is a no-op until `failbit` is cleared explicitly.

In general, the set bits reflect only what happened sometime in the past: If a bit is set after an operation, this does not necessarily mean that this operation caused the flag to be set. Instead, the flag might have been set before the operation. Thus, if it not known whether an error bit is set, you should call `clear()` before an operation is executed to let the flags tell you what went wrong. Note however, that operations may have different effects after clearing the flags. For example, even if `eofbit` was set by an operation, this does not mean that after clearing `eofbit` the operation will set `eofbit` again. This can be the case, for example, if the accessed file grew between the two calls.

15.4.3. Stream State and Boolean Conditions

Two functions are defined for the use of streams in Boolean expressions (Table 15.5).

Table 15.5. Stream Operators for Boolean Expressions

Member Function	Meaning
`operator bool ()`	Returns whether the stream has not run into an error (corresponds to `!fail()`)
`operator ! ()`	Returns whether the stream has run into an error (corresponds to `fail()`)

With `operator bool()` ,[8] streams can be tested in control structures in a short and idiomatic way for their current state:

[8] Before C++11, the operator was declared as `operator void*()` , which could cause problems such as those described in Section 15.10.1, page 805.

```
// while the standard input stream is OK
while (std::cin) {
    ...
}
```

For the Boolean condition in a control structure, the type does not need a direct conversion to `bool` . Instead, a unique conversion to an integral type, such as `int` or `char` , or to a pointer type is sufficient. The conversion to `bool` is often used to read objects and test for success in the same expression:

```
if (std::cin >> x) {
    // reading x was successful
    ...
}
```

As discussed earlier, the following expression returns `cin` :

```
std::cin >> x
```

So, after `x` is read, the statement is

```
if (std::cin) {
    ...
}
```

Because `cin` is being used in the context of a condition, its operator `void*` is called, which returns whether the stream has run into an error.

A typical application of this technique is a loop that reads and processes objects:

```
// as long as obj can be read
while (std::cin >> obj) {
    // process obj (in this case, simply output it)
    std::cout << obj << std::endl;
}
```

This is C's classic filter framework for C++ objects. The loop is terminated if the `failbit` or `badbit` is set. This happens when an error occurred or at end-of-file (the attempt to read at end-of-file results in setting `eofbit` *and* `failbit` ; see Section 15.4.1, page 758). By default, operator `>>` skips leading whitespaces. This is normally exactly what is desired. However, if `obj` is of type `char` , whitespace is normally considered to be significant. In this case, you can use the `put()` and `get()` member functions of streams (see Section 15.5.3, page 772) or, even better, an `istreambuf_iterator` (see Section 15.13.2, page 831) to implement an I/O filter.

With operator `!` , the inverse test can be performed. The operator is defined to return whether a stream has run into an error; that is, the operator returns `true` if `failbit` or `badbit` is set. The operator can be used like this:

```
if (! std::cin) {
    // the stream cin is not OK
    ...
}
```

Like the implicit conversion to a Boolean value, this operator is often used to test for success in the same expression in which an object was read:

```
if (! (std::cin >> x)) {
    // the read failed
    ...
}
```

Here, the following expression returns `cin`, to which operator `!` is applied:

```
std::cin >> x
```

The expression after `!` must be placed within parentheses because of operator precedence rules: Without the parentheses, operator `!` would be evaluated first. In other words, the expression

```
!std::cin >> x
```

is equivalent to the expression

```
(!std::cin) >> x
```

This is probably not what is intended.

Although these operators are very convenient in Boolean expressions, one oddity has to be noted: Double "negation" does *not* yield the original object:

- `cin` is a stream object of class `istream`.
- `!!cin` is a Boolean value describing the state of `cin`.

As with other features of C++, it can be argued whether the use of the conversions to a Boolean value is good style. The use of member functions, such as `fail()`, normally yields a more readable program:

```
std::cin >> x;
if (std::cin.fail()) {
    ...
}
```

15.4.4. Stream State and Exceptions

Exception handling was introduced to C++ for the handling of errors and exceptions (see Section 4.3, page 41). However, this was done after streams were already in wide use. To stay backward compatible, by default, streams throw no exceptions. However, for the standardized streams, it is possible to define, for every state flag, whether setting that flag will trigger an exception. This definition is done by the `exceptions()` member function (Table 15.6).

Table 15.6. Stream Member Functions for Exceptions

Member Function	Meaning
`exceptions(`*flags*`)`	Sets flags that trigger exceptions
`exceptions()`	Returns the flags that trigger exceptions

Calling `exceptions()` without an argument yields the current flags for which exceptions are triggered. No exceptions are thrown if the function returns `goodbit`. This is the default, to maintain backward compatibility. When `exceptions()` is called with an argument, exceptions are thrown as soon as the corresponding state flags are set. If a state flag is already set when `exceptions()` is called with an argument, an exception is thrown if the corresponding flag is set in the argument.

The following example configures the stream so that, for all flags, an exception is thrown:

Click here to view code image

```
// throw exceptions for all "errors"
strm.exceptions (std::ios::eofbit | std::ios::failbit |
                 std::ios::badbit);
```

If `0` or `goodbit` is passed as an argument, no exceptions are generated:

```
// do not generate exceptions
strm.exceptions (std::ios::goodbit);
```

Exceptions are thrown when the corresponding state flags are set after calling `clear()` or `setstate()`. An exception is even thrown if the flag was already set and not cleared:

```
// this call throws an exception if failbit is set on entry
strm.exceptions (std::ios::failbit);
...
// throw an exception (even if failbit was already set)
strm.setstate (std::ios::failbit);
```

The exceptions thrown are objects of the class `std::ios_base::failure`. Since C++11, this class is derived from `std::system_error` (see Section 4.3.1, page 44).[9]

[9] Before C++11, class `std::ios_base::failure` was directly derived from class `std::exception`.

Click here to view code image

```
namespace std {
    class ios_base::failure : public system_error {
      public:
        explicit failure (const string& msg,
                          const error_code& ec = io_errc::stream);
        explicit failure (const char* msg,
                          const error_code& ec = io_errc::stream);
    };
}
```

Implementations are requested to provide an `error_code` object that provides the specific reason for the failure. In fact, an error caused by the operating system should have the category() `"system"` and the `value()` that was reported by the operating system. An error arising from within the I/O stream library should have the category() `"iostream"` and the `value()` `std::io_errc::stream`. See Section 4.3.2, page 45, for details about class `error_code` and how to deal with it.

Not throwing exceptions as default demonstrates that exception handling is intended to be used more for unexpected situations. It is called *exception handling* rather than *error handling*. Expected errors, such as format errors during input from the user, are considered to be "normal" and are usually better handled using the state flags.

The major area in which stream exceptions are useful is reading preformatted data, such as automatically written files. But even then, problems arise if exception handling is used. For example, if it is desired to read data until end-of-file, you can't get exceptions for errors without getting an exception for end-of-file. The reason is that the detection of end-of-file also sets the `failbit`, meaning that reading an object was not successful. To distinguish end-of-file from an input error, you have to check the state of the stream.

The next example demonstrates how this might look. It shows a function that reads floating-point values from a stream until end-of-file is reached and returns the sum of the floating-point values read:

Click here to view code image

```
// io/sum1a.cpp

#include <istream>

namespace MyLib {
    double readAndProcessSum (std::istream& strm)
    {
        using std::ios;
        double value, sum;

        // save current state of exception flags
        ios::iostate oldExceptions = strm.exceptions();

        // let failbit and badbit throw exceptions
        // - NOTE: failbit is also set at end-of-file
        strm.exceptions (ios::failbit | ios::badbit);

        try {
            // while stream is OK
            // - read value and add it to sum
            sum = 0;
            while (strm >> value) {
                sum += value;
            }
        }
        catch (...) {
            // if exception not caused by end-of-file
            // - restore old state of exception flags
            // - rethrow exception
            if (!strm.eof()) {
                strm.exceptions(oldExceptions);     // restore exception flags
                throw;                              // rethrow
            }
        }

        // restore old state of exception flags
        strm.exceptions (oldExceptions);

        // return sum
        return sum;
    }
}
```

First, the function stores the set stream exceptions in `oldExceptions` to restore them later. Then the stream is configured to throw an exception on certain conditions. In a loop, all values are read and added as long as the stream is OK. If end-of-file is reached, the stream is no longer OK, and a corresponding exception is thrown even though no exception is thrown for setting `eofbit`. This happens because end-of-file is detected on an unsuccessful attempt to read more data, which also sets the `failbit`. To avoid the behavior that end-of-file throws an exception, the exception is caught locally to check the state of the stream by using `eof()`. The exception is propagated only if `eof()` yields `false`.

Note that restoring the original exception flags may cause exceptions: `exceptions()` throws an exception if a corresponding flag is set in the stream already. Thus, if the state did throw exceptions for `eofbit`, `failbit`, or `badbit` on function entry, these exceptions are propagated to the caller.

This function can be called in the simplest case from the following main function:

Click here to view code image

```
// io/summain.cpp

#include <iostream>
#include <exception>
#include <cstdlib>

namespace MyLib {
    double readAndProcessSum (std::istream&);
}

int main()
{
    using namespace std;
    double sum;

    try {
        sum = MyLib::readAndProcessSum(cin);
    }
    catch (const ios::failure& error) {
        cerr << "I/O exception: " << error.what() << endl;
        return EXIT_FAILURE;
    }
    catch (const exception& error) {
        cerr << "standard exception: " << error.what() << endl;
        return EXIT_FAILURE;
    }

    catch (...) {
        cerr << "unknown exception" << endl;
        return EXIT_FAILURE;
    }

    // print sum
    cout << "sum: " << sum << endl;
}
```

The question arises whether this is worth the effort. It is also possible to work with streams not throwing an exception. In this case, an exception is thrown if an error is detected. This has the additional advantage that user-defined error messages and error classes can be used:

Click here to view code image

```
// io/sum2a.cpp

#include <istream>

namespace MyLib {
    double readAndProcessSum (std::istream& strm)
    {
        double value, sum;

        // while stream is OK
        // - read value and add it to sum
        sum = 0;
        while (strm >> value) {
            sum += value;
        }

        if (!strm.eof()) {
            throw std::ios::failure
                    ("input error in readAndProcessSum()");
        }

        // return sum
```

```
            return sum;
        }
    }
```

This looks somewhat simpler, doesn't it?

I/O Exceptions before C++11

Before C++11, class `std::ios_base::failure` was directly derived from class `std::exception` and had only a constructor taking a `std::string` argument:

```
namespace std {
    class ios_base::failure : public exception {
      public:
        explicit failure (const string& msg);
        ...
    };
}
```

This caused the following limitations:

- For the generated exception object, it was possible to call `what()` only to get an implementation-specific string for the reason of the failure. No support for an exception category or value was provided.
- Because the constructor did take only a `std::string`, you had to include `<string>` when passing a string literal. (To enable the conversion to `std::string`, you need the declaration of the corresponding string constructor.)