Revising the wording of stream input operations

Document #:	D1264R2
Date:	2022-11-08
Project:	Programming Language C++
Audience:	LWG

Reply-to: Louis Dionne <ldionne@apple.com>

1 Revision history

- R0 – Initial draft
- R1 –
  - Apply LWG small-group changes.
  - Rebase on top of N4778.
- R2 – Rebase on top of latest working draft

2 Abstract

The wording in **[istream]**, **[istream.formatted]** and **[istream.unformatted]** is very difficult to follow when it comes to exceptions. Some requirements are specified more than once in different locations, which makes it ambiguous how requirements should interact with each other.

This is problematic because implementations currently differ significantly on their handling of error flags and exceptions. For example, this libc++ bug report claims that libc++'s `operator>>(istream&, std::string&)` is not throwing exception when `failbit` is set and `failbit` exceptions are enabled. GCC and MSVC both behave as expected there. Unfortunately, the Standard seems to give reason to libc++, despite the behavior not making sense. Note that as of currently, libc++ has been fixed to be consistent with the wording in this paper.

[LWG2349] tries to solve this issue by applying a small patch to the current wording, but I think not all issues are solved this way. This wording-only proposal instead tries to overhaul the current wording to make it clearer, without changing the intended behavior.

3 Proposed wording

This wording is based on the latest working draft as of 2022-11-07.

3.1 Remove common wording

First, we clean up confusing wording that overlaps with wording in the formatted and unformatted input operations:

Remove in **[istream.general]/4**

> ~~If `rdbuf()->sbumpc()` or `rdbuf()->sgetc()` returns `traits::eof()`, then the input function, except as explicitly noted otherwise, completes its actions and does `setstate(eofbit)`, which may throw `ios_`base::failure ([iostate.flags]), before returning.~~

Remove in **[istream.general]/5**:

> ~~If one of these called functions throws an exception, then unless explicitly noted otherwise, the input function sets `badbit` in error state. If `badbit` is on in `exceptions()`, the input function rethrows the exception without completing its actions, otherwise it does not throw anything and proceeds as if the called function had returned a failure indication.~~

3.2 Revise wording for formatted input operations

We precise the execution of formatted input operations by introducing the notion of a *local error state*. In **[istream.formatted.reqmts]**:

> Each formatted input function begins execution by constructing an object of type `ios_`base::iostate, called the local error state, and initializing it to `ios_`base::goodbit. It then creates an object of class `sentry` with the `noskipws` (second) argument `false`. If the sentry object returns `true`, when converted to a value of type `bool`, the function endeavors to obtain the requested input. Otherwise, if the sentry constructor exits by throwing an exception or if the sentry object produces `false` when converted to a value of type `bool`, the function returns without attempting to obtain any input. If an extraction function ([istream.general]) returns `traits::eof()`, then `ios_`base::eofbit is set in the local error state and the input function stops trying to obtain the requested input. If an exception is thrown during input then `ios::badbit` is set in the local error state, `*this`'s error state is set to the local error state, and the exception is rethrown if `(exceptions() & badbit) != 0`. ~~If `(exceptions()&badbit) != 0` then the exception is rethrown.~~ After extraction is done, the input function calls `setstate`, which sets `*this`'s error state to the local error state, and may throw an exception. In any case, the formatted input function destroys the sentry object. If no exception has been thrown, it returns `*this`.

Then, we adjust the description of formatted input operations to take advantage of the local error state introduced above. In **[istream.formatted.arithmetic]**:

```
basic_istream& operator>>(unsigned short& val);
basic_istream& operator>>(unsigned int& val);
basic_istream& operator>>(long& val);
basic_istream& operator>>(unsigned long& val);
basic_istream& operator>>(long long& val);
basic_istream& operator>>(unsigned long long& val);
basic_istream& operator>>(float& val);
basic_istream& operator>>(double& val);
basic_istream& operator>>(long double& val);
basic_istream& operator>>(bool& val);
basic_istream& operator>>(void*& val);
```

As in the case of the inserters, these extractors depend on the locale's `num_get<>` object to perform parsing the input stream data. These extractors behave as formatted input functions (as described in [istream.formatted.reqmts]). After a `sentry` object is constructed, the conversion occurs as if performed by the following code fragment, where `state` represents the input function's local error state:

```
using numget = num_get<charT, istreambuf_iterator<charT, traits>>;
iostate err = iostate::goodbit;
use_facet<numget>(loc).get(*this, 0, *this, err state, val);
setstate(err);
```

In **[istream.formatted.arithmetic]/2**:

```
basic_istream& operator>>(short& val);
```

The conversion occurs as if performed by the following code fragment (using the same notation as for the preceding code fragment):

```
using numget = num_get<charT, istreambuf_iterator<charT, traits>>;
iostate err = ios_base::goodbit;
long lval;
use_facet<numget>(loc).get(*this, 0, *this, err state, lval);
if (lval < numeric_limits<short>::min()) {
  err state |= ios_base::failbit;
  val = numeric_limits<short>::min();
} else if (numeric_limits<short>::max() < lval) {
  err state |= ios_base::failbit;
  val = numeric_limits<short>::max();
}  else {
  val = static_cast<short>(lval);
}
setstate(err);
```

In **[istream.formatted.arithmetic]/3**:

```
basic_istream& operator>>(int& val);
```

The conversion occurs as if performed by the following code fragment (using the same notation as for the preceding code fragment):

```
using numget = num_get<charT, istreambuf_iterator<charT, traits>>;
iostate err = ios_base::goodbit;
long lval;
use_facet<numget>(loc).get(*this, 0, *this, err state, lval);
if (lval < numeric_limits<int>::min()) {
  err state |= ios_base::failbit;
  val = numeric_limits<int>::min();
} else if (numeric_limits<int>::max() < lval) {
  err state |= ios_base::failbit;
  val = numeric_limits<int>::max();
}  else {
  val = static_cast<int>(lval);
}
```

~~setstate(err);~~

In **[istream.formatted.arithmetic]/3**:

```
basic_istream& operator>>(extended-floating-point-type& val);
```

[...] The conversion occurs as if performed by the following code fragment (using the same notation as for the preceding code fragment):

```
using numget = num_get<charT, istreambuf_iterator<charT, traits>>;
~~iostate err = ios_base::goodbit;~~
FP fval;
use_facet<numget>(loc).get(*this, 0, *this, ~~err~~ state, fval);
if (fval < -numeric_limits<extended-floating-point-type>::max()) {
  ~~err~~ state |= ios_base::failbit;
  val = -numeric_limits<extended-floating-point-type>::max();
} else if (numeric_limits<extended-floating-point-type>::max() < fval) {
  ~~err~~ state |= ios_base::failbit;
  val = numeric_limits<extended-floating-point-type>::max();
} else {
  val = static_cast<extended-floating-point-type>(fval);
}
~~setstate(err);~~
```

In **[istream.extractors]/10**:

```
template<class charT, class traits, size_t N>
  basic_istream<charT, traits>& operator>>(basic_istream<charT, traits>& in, charT (&s)[N]);
template<class traits, size_t N>
  basic_istream<char, traits>& operator>>(basic_istream<char, traits>& in, unsigned char (&s)[N]
template<class traits, size_t N>
  basic_istream<char, traits>& operator>>(basic_istream<char, traits>& in, signed char (&s)[N]);
```

[...] If the function extracted no characters, ~~it calls `setstate(failbit)`, which may throw `ios_base::failure` ([iostate.flags])~~ `ios_base::failbit` is set in the input function's local error state before `setstate` is called.

In **[istream.extractors]/12**:

```
template<class charT, class traits>
  basic_istream<charT, traits>& operator>>(basic_istream<charT, traits>& in, charT& c);
template<class traits>
  basic_istream<char, traits>& operator>>(basic_istream<char, traits>& in, unsigned char& c);
template<class traits>
  basic_istream<char, traits>& operator>>(basic_istream<char, traits>& in, signed char& c);
```

Effects: Behaves like a formatted input member (as described in [istream.formatted.reqmts]) of `in`. ~~After a `sentry` object is constructed a~~ A character is extracted from `in`, if one is available, and stored in `c`. Otherwise, ~~the function calls `in.setstate(failbit)`~~ `ios_base::failbit` is set in the input function's local error state before `setstate` is called.

In **[string.io]**:

```
template<class charT, class traits, class Allocator>
  basic_istream<charT, traits>&
    operator>>(basic_istream<charT, traits>& is, basic_string<charT, traits, Allocator>& str);
```

Effects: [...] After the last character (if any) is extracted, `is.width(0)` is called and the `sentry` object is destroyed. If the function extracts no characters, ~~it calls `is.setstate(ios::failbit)`, which may throw `ios`_base::failure ([iostate.flags])~~ `ios`_base::failbit is set in the input function's local error state before `setstate` is called.

In **[bitset.operators]**:

```
template<class charT, class traits, size_t N>
  basic_istream<charT, traits>&
    operator>>(basic_istream<charT, traits>& is, bitset<N>& x);
```

[...] If no characters are stored in `str`, ~~calls `is.setstate(ios`_base::failbit) (which may throw `ios`_base::failure ([iostate.flags]))~~ `ios`_base::failbit is set in the input function's local error state before `setstate` is called.

3.3 Revise wording for unformatted input operations

In **[istream.unformatted]/1**:

Each unformatted input function begins execution by constructing an object of type `ios`_base::iostate, called the local error state, and initializing it to `ios`_base::goodbit. It then creates an object of class `sentry` with the default argument `noskipws` (second) argument `true`. If the `sentry` object returns `true`, when converted to a value of type `bool`, the function endeavors to obtain the requested input. Otherwise, if the `sentry` constructor exits by throwing an exception or if the `sentry` object ~~returns~~ produces `false`, when converted to a value of type `bool`, the function returns without attempting to obtain any input. In either case the number of extracted characters is set to 0; unformatted input functions taking a character array of nonzero size as an argument shall also store a null character (using `charT()`) in the first location of the array. If an extraction function ([istream.general]) returns `traits::eof()`, then `ios`_base::eofbit is set in the local error state and the input function stops trying to obtain the requested input. If an exception is thrown during input then `ios_base::badbit` is turned on in ~~`*this`'s error state~~ the local error state, `*this`'s error state is set to the local error state, and the exception is rethrown if `(exceptions() & badbit) != 0`. ~~(Exceptions thrown from `basic_ios<>`::clear() are not caught or rethrown.) If `(exceptions()&badbit) != 0` then the exception is rethrown. It also counts the number of characters extracted.~~ If no exception has been thrown it ~~ends by storing~~ stores the ~~count~~ number of characters extracted in a member object ~~and returning the value specified~~. After extraction is done, the input function calls `setstate`, which sets `*this`'s error state to the local error state, and may throw an exception. In any event the `sentry` object is destroyed before leaving the unformatted input function.

In **[istream.unformatted]/4**:

```
int_type get();
```

Effects: Behaves as an unformatted input function (as described above). After constructing a `sentry` object, extracts a character `c`, if one is available. Otherwise, ~~the function calls setstate(failbit), which may throw ios_base::failure ([iostate.flags])~~ `ios_base::failbit` is set in the input function's local error state before `setstate` is called.

In **[istream.unformatted]/6**:

```
basic_istream<charT, traits>& get(char_type& c);
```

Effects: Behaves as an unformatted input function (as described above). After constructing a `sentry` object, extracts a character, if one is available, and assigns it to `c`. Otherwise, ~~the function calls setstate(failbit) (which may throw ios_base::failure ([iostate.flags]))~~ `ios_base::failbit` is set in the input function's local error state before `setstate` is called.

In **[istream.unformatted]/8**:

```
basic_istream<charT, traits>& get(char_type* s, streamsize n, char_type delim);
```

Effects: [...]

- `n` is less than one or `n - 1` characters are stored;
- end-of-file occurs on the input sequence ~~(in which case the function calls setstate(eofbit))~~;
- `traits::eq(c, delim)` for the next available input character `c` (in which case `c` is not extracted).

If the function stores no characters, ~~it calls `setstate(failbit)` (which may throw ios_base::failure ([iostate.flags]))~~ `ios_base::failbit` is set in the input function's local error state before `setstate` is called. In any case, if `n` is greater than zero it then stores a null character into the next successive location of the array.

In **[istream.unformatted]/13**:

```
basic_istream<charT, traits>& get(basic_streambuf<char_type, traits>& sb, char_type delim);
```

Effects: [...]

- end-of-file occurs on the input sequence;
- inserting in the output sequence fails (in which case the character to be inserted is not extracted);
- `traits::eq(c, delim)` for the next available input character `c` (in which case `c` is not extracted);
- an exception occurs (in which case, the exception is caught but not rethrown).

If the function inserts no characters, ~~it calls `setstate(failbit)`, which may throw ios_base::failure ([iostate.flags])~~ `ios_base::failbit` is set in the input function's local error state before `setstate` is called.

In **[istream.unformatted]/18**:

```
basic_istream<charT, traits>& getline(char_type* s, streamsize n, char_type delim);
```

Effects: [...]

1. end-of-file occurs on the input sequence ~~(in which case the function calls `setstate(eofbit)`)~~;
2. `traits::eq(c, delim)` for the next available input character `c` (in which case the input character is extracted but not stored);
3. `n` is less than one or `n - 1` characters are stored (in which case the function calls `setstate(failbit)`).

These conditions are tested in the order shown.

If the function extracts no characters, ~~it calls `setstate(failbit)` (which may throw `ios_base::failure` ([iostate.flags]))~~ `ios_base::failbit` is set in the input function's local error state before `setstate` is called.

In any case, if `n` is greater than zero, it then stores a null character (using `charT()`) into the next successive location of the array.

In **[istream.extractors]/14**:

```
basic_istream<charT, traits>& operator>>(basic_streambuf<charT, traits>* sb);
```

Effects: [...] If the function inserts no characters, ~~it calls `setstate(failbit)`, which may throw `ios_base::failure` ([iostate.flags])~~ `ios_base::failbit` is set in the input function's local error state before `setstate` is called. ~~If it inserted no characters because it caught an exception thrown while extracting characters from `*this` and `failbit` is on in `exceptions()` ([iostate.flags]), then the caught exception is rethrown.~~

In **[string.io]/6**:

```
template<class charT, class traits, class Allocator>
basic_istream<charT, traits>&
  getline(basic_istream<charT, traits>& is,
          basic_string<charT, traits, Allocator>& str,
          charT delim);
template<class charT, class traits, class Allocator>
basic_istream<charT, traits>&
  getline(basic_istream<charT, traits>&& is,
          basic_string<charT, traits, Allocator>& str,
          charT delim);
```

Effects: [...]

- end-of-file occurs on the input sequence;~~(in which case, the `getline` function calls `is.setstate(ios_base::eofbit)`).~~
- `traits::eq(c, delim)` for the next available input character `c` (in which case, `c` is extracted but not appended) ~~([iostate.flags])~~ ;
- `str.max_size()` characters are stored (in which case, ~~the function calls `is.setstate(ios_base::failbit)` ([iostate.flags])~~ `ios_base::failbit` is set in the input function's local error state)

The conditions are tested in the order shown. In any case, after the last character is extracted, the sentry object is destroyed.

If the function extracts no characters, ~~it calls `is.setstate(ios_base::failbit)` which may throw `ios_base::failure` ([iostate.flags])~~ `ios_base::failbit` is turned on in the input function's local error state before `setstate` is called.

4 Appendix: a few test cases

This section contains test cases that were handled in different ways by the implementations. They are provided as a proof that we need to solve the problem, and for the implementer's reference if they deem it useful.

First, let's introduce a few definitions from the Standard so we can refer to them below.

(A) **[istream]/3** (applies to both formatted and unformatted input functions):

> If `rdbuf()->sbumpc()` or `rdbuf()->sgetc()` returns `traits::eof()`, then the input function, except as explicitly noted otherwise, completes its actions and does `setstate(eofbit)`, which may throw `ios_base::failure` ([iostate.flags]), before returning.

(B) **[istream]/4** (applies to both formatted and unformatted input functions):

> If one of these called functions throws an exception, then unless explicitly noted otherwise, the input function sets `badbit` in error state. If `badbit` is on in `exceptions()`, the input function rethrows the exception without completing its actions, otherwise it does not throw anything and proceeds as if the called function had returned a failure indication.

(C) **[istream.formatted.reqmts]** **[istream.formatted.reqmts]/1** (applies only to formatted input functions):

> Each formatted input function begins execution by constructing an object of class `sentry` with the `noskipws` (second) argument `false`. If the sentry object returns `true`, when converted to a value of type `bool`, the function endeavors to obtain the requested input. If an exception is thrown during input then `ios::badbit` is set in `*this`'s error state. If `(exceptions()&badbit) != 0` then the exception is rethrown. In any case, the formatted input function destroys the sentry object. If no exception has been thrown, it returns `*this`.

(D) **[istream.unformatted]/1** (applies only to unformatted input functions):

> [...] If an exception is thrown during input then `ios::badbit` is turned on in `*this`'s error state. (Exceptions thrown from `basic_ios<>::clear()` are not caught or rethrown.) If `(exceptions()&badbit) != 0` then the exception is rethrown. It also counts the number of characters extracted. If no exception has been thrown it ends by storing the count in a member object and returning the value specified. In any event the sentry object is destroyed before leaving the unformatted input function.

With all this laid out, here's a couple of test cases:

1. Formatted input operation which fails to extract from a non-empty stream

```cpp
#include <iostream>
#include <sstream>
int main () {
    std::stringbuf buf("not empty");
    std::istream is(&buf);
    is.exceptions(std::ios::failbit);

    bool threw = false;
    try {
        unsigned int tmp{};
        is >> tmp;
    } catch (std::ios::failure const&) {
        threw = true;
    }

    std::cout << "bad = " << is.bad() << std::endl;
    std::cout << "fail = " << is.fail() << std::endl;
    std::cout << "eof = " << is.eof() << std::endl;
    std::cout << "threw = " << threw << std::endl;
}
```

   The current behavior is the following:

	libstdc++	MSVC	libc++
bad	0	0	1
fail	1	1	1
eof	0	0	0
threw	1	1	0

   My interpretation is that per the definition of `operator>>(unsigned int&)` in **[istream.formatted.arithmet** we try to extract an `unsigned int` from the stream:

```cpp
using numget = num_get<charT, istreambuf_iterator<charT, traits>>;
iostate err = iostate::goodbit;
use_facet<numget>(loc).get(*this, 0, *this, err, val);
setstate(err);
```

   This `num_get::get` fails because the format is wrong and reports that by setting `err` to `std::ios_base::failbit`, which results in `setstate(err)` throwing because `failbit` had been set in the exceptions. I don't think (B) applies here because the exception is not being thrown as part of `rdbuf()->sbumpc()` or `rdbuf()->sgetc()`. However, (C) seems to apply, which means that we catch the exception and set badbit on the stream, but we do not rethrow the exception because `badbit` is not set in `exceptions()`. Hence, libc++'s behavior seems correct to me, despite being useless.

2. Formatted input operation which fails to extract from an empty stream

```cpp
#include <iostream>
#include <sstream>
int main () {
    std::stringbuf buf; // empty
    std::istream is(&buf);
    is.exceptions(std::ios::failbit);

    bool threw = false;
    try {
        unsigned int tmp{};
        is >> tmp;
    } catch (std::ios::failure const&) {
        threw = true;
    }

    std::cout << "bad = " << is.bad() << std::endl;
    std::cout << "fail = " << is.fail() << std::endl;
    std::cout << "eof = " << is.eof() << std::endl;
    std::cout << "threw = " << threw << std::endl;
}
```

The current behavior is the following:

	libstdc++	MSVC	libc++
bad	0	0	1
fail	1	1	1
eof	1	1	1
threw	1	1	0

My interpretation is that per (C), we create a sentry object which attempts to skip whitespace and fails because we're at the end of file. The sentry calls `setstate(failbit | eofbit)`, which throws an exception because failbit is set in the exceptions. We then set `badbit` on the stream and do not rethrow the exception, because `badbit` is not in the exceptions. Also note that I don't think (B) applies here, because we never make it to the operations specified in (A), which I think is what (B) is referring to. Hence, libc++ is correct again here.

3. Unformatted input operation which hits EOF

```cpp
#include <iostream>
#include <sstream>
int main() {
    std::stringbuf sb("rrrrrrrrr");
    std::istream is(&sb);
    is >> std::noskipws;
    is.exceptions(std::ios::eofbit);
```

```
    bool threw = false;
    try {
        while (true) {
            is.get();
            if (is.eof())
                break;
        }
    } catch (std::ios::failure const&) {
        threw = true;
    }

    std::cout << "bad = " << is.bad() << std::endl;
    std::cout << "fail = " << is.fail() << std::endl;
    std::cout << "eof = " << is.eof() << std::endl;
    std::cout << "threw = " << threw << std::endl;
}
```

The current behavior is the following:

	libstdc++	MSVC	libc++
bad	0	0	1
fail	1	1	1
eof	1	1	1
threw	1	1	0

My interpretation is that we create the sentry, which doesn't do much because we're not trying to skip whitespace. We then try to extract a character and fail because we hit the end of file. Per the definition of `basic_istream::get()` in **[istream.unformatted]/4**, we call `setstate(failbit)`, which throws an exception. Per (A), we're also somehow required to call `setstate(eofbit)`. Finally, per (D), we also set `badbit` on the stream, and we don't rethrow any exception because `badbit` is not in the exceptions. I think this makes libc++ right again.

Actually, I don't think this specification can be implemented as-is because of [LWG61], which added the part "(Exceptions thrown from `basic_ios<>::clear()` are not caught or rethrown.)". This would make it effectively impossible to call both `setstate(failbit)` and `setstate(eofbit)`, and also to set the `badbit` on the stream. Unless I'm missing a clever implementation trick, you basically have to catch and rethrow.

5 References

[LWG2349] Zhihao Yuan, *Clarify input/output function rethrow behavior*
https://cplusplus.github.io/LWG/issue2349

[LWG61] Matt Austern, *Ambiguity in iostreams exception policy*
https://cplusplus.github.io/LWG/issue61