Document Number: P1721R0
Date: 2019-06-16
Reply to: Marshall Clow
CppAlliance
mclow.lists@gmail.com

Mandating the Standard Library: Clause 29 - Input/output library

With the adoption of P0788R3, we have a new way of specifying requirements for the library clauses of the standard. This is one of a series of papers reformulating the requirements into the new format. This effort was strongly influenced by the informational paper P1369R0.

The changes in this series of papers fall into four broad categories.

— Change "participate in overload resolution" wording into "Constraints" elements

— Change "Requires" elements into either "Mandates" or "Expects", depending (mostly) on whether or not they can be checked at compile time.

— Drive-by fixes (hopefully very few)

This paper covers Clause 29 (Input/output), and is based on N4810.

The entire clause is reproduced here, but the changes are confined to a few sections:

— ios.init 29.5.3.1.6
— ios.base.storage 29.5.3.5
— ios.base.callback 29.5.3.6
— fpos.operations 29.5.4.2
— basic.ios.members 29.5.5.3
— streambuf.virt.get 29.6.3.4.3
— streambuf.virt.put 29.6.3.4.5
— istream.rvalue 29.7.4.5
— ostream.inserters.character 29.7.5.2.4
— ostream.rvalue 29.7.5.5
— ext.manip 29.7.7
— filebuf.members 29.9.2.3
— syncstream.syncbuf.cons 29.10.2.2
— syncstream.syncbuf.assign 29.10.2.3
— syncstream.osyncstream.overview 29.10.3.1
— fs.path.construct 29.11.7.4.1
— fs.path.modifiers 29.11.7.4.5
— fs.class.directory.iterator 29.11.12
— fs.dir.itr.members 29.11.12.1
— fs.rec.dir.itr.members 29.11.13.1
— fs.op.copy 29.11.14.3
— fs.op.copy.file 29.11.14.4
— fs.op.permissions 29.11.14.26

Drive-by fixes:

— Changed some random "shall"s to "meet"s.
— Re-ordered a couple of clauses in fs.path.modifiers (29.11.7.4.5).
— Struck a few clauses that read "Constructs an object of type XXXX".

Open questions:

— There's stuff (*Requires:*) in [streambuf.virt.get] and [streambuf.virt.put] that I haven't dealt with.
— There's also explanatory text in [fs.race.behavior]/2 that refers to "Requires:" elements.

Thanks to Daniel Krügler for his advice and reviews.

Help for the editors: The changes here can be viewed as latex sources with the following commands

```
git clone git@github.com:mclow/mandate.git
cd mandate
git diff master..chapter29 iostreams.tex
```

29 Input/output library [input.output]

29.1 General [input.output.general]

1 This Clause describes components that C++ programs may use to perform input/output operations.

2 The following subclauses describe requirements for stream parameters, and components for forward declarations of iostreams, predefined iostreams objects, base iostreams classes, stream buffering, stream formatting and manipulators, string streams, and file streams, as summarized in Table 103.

Table 103 — Input/output library summary

	Subclause	Header
29.2	Requirements	
29.3	Forward declarations	`<iosfwd>`
29.4	Standard iostream objects	`<iostream>`
29.5	Iostreams base classes	`<ios>`
29.6	Stream buffers	`<streambuf>`
29.7	Formatting and manipulators	`<iomanip>`, `<istream>`, `<ostream>`
29.8	String streams	`<sstream>`
29.9	File streams	`<fstream>`
29.10	Synchronized output streams	`<syncstream>`
29.11	File systems	`<filesystem>`
29.12	C library files	`<cstdio>`, `<cinttypes>`

3 Figure 1 illustrates relationships among various types described in this clause. A line from **A** to **B** indicates that **A** is an alias (e.g., a typedef) for **B** or that **A** is defined in terms of **B**.

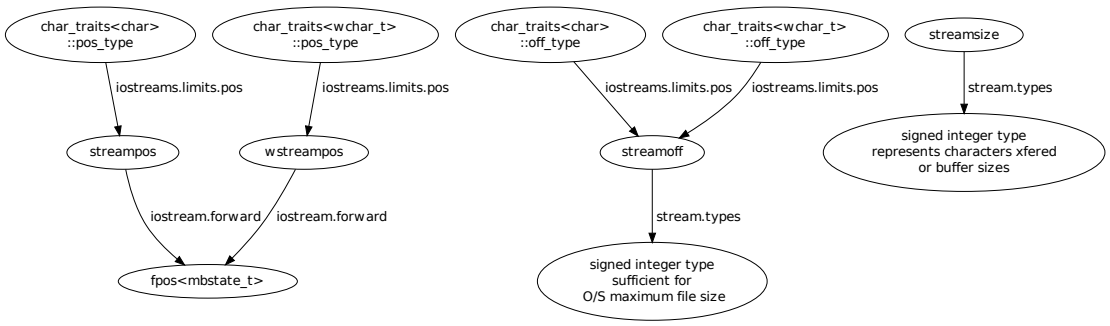

Figure 1 — Stream position, offset, and size types [non-normative]

29.2 Iostreams requirements [iostreams.requirements]

29.2.1 Imbue limitations [iostream.limits.imbue]

1 No function described in Clause 29 except for `ios_base::imbue` and `basic_filebuf::pubimbue` causes any instance of `basic_ios::imbue` or `basic_streambuf::imbue` to be called. If any user function called from a function declared in Clause 29 or as an overriding virtual function of any class declared in Clause 29 calls `imbue`, the behavior is undefined.

29.2.2 Positioning type limitations [iostreams.limits.pos]

1 The classes of Clause 29 with template arguments `charT` and `traits` behave as described if `traits::pos_type` and `traits::off_type` are `streampos` and `streamoff` respectively. Except as noted explicitly below, their behavior when `traits::pos_type` and `traits::off_type` are other types is implementation-defined.

[2] In the classes of Clause 29, a template parameter with name `charT` represents a member of the set of types containing `char`, `wchar_t`, and any other implementation-defined character types that satisfy the requirements for a character on which any of the iostream components can be instantiated.

29.2.3 Thread safety [iostreams.threadsafety]

[1] Concurrent access to a stream object (29.8, 29.9), stream buffer object (29.6), or C Library stream (29.12) by multiple threads may result in a data race (**??**) unless otherwise specified (29.4). [*Note*: Data races result in undefined behavior (**??**). — *end note*]

[2] If one thread makes a library call a that writes a value to a stream and, as a result, another thread reads this value from the stream through a library call b such that this does not result in a data race, then a's write synchronizes with b's read.

29.3 Forward declarations [iostream.forward]

29.3.1 Header `<iosfwd>` synopsis [iosfwd.syn]

```
namespace std {
  template<class charT> struct char_traits;
  template<> struct char_traits<char>;
  template<> struct char_traits<char8_t>;
  template<> struct char_traits<char16_t>;
  template<> struct char_traits<char32_t>;
  template<> struct char_traits<wchar_t>;

  template<class T> class allocator;

  template<class charT, class traits = char_traits<charT>>
    class basic_ios;
  template<class charT, class traits = char_traits<charT>>
    class basic_streambuf;
  template<class charT, class traits = char_traits<charT>>
    class basic_istream;
  template<class charT, class traits = char_traits<charT>>
    class basic_ostream;
  template<class charT, class traits = char_traits<charT>>
    class basic_iostream;

  template<class charT, class traits = char_traits<charT>,
           class Allocator = allocator<charT>>
    class basic_stringbuf;
  template<class charT, class traits = char_traits<charT>,
           class Allocator = allocator<charT>>
    class basic_istringstream;
  template<class charT, class traits = char_traits<charT>,
           class Allocator = allocator<charT>>
    class basic_ostringstream;
  template<class charT, class traits = char_traits<charT>,
           class Allocator = allocator<charT>>
    class basic_stringstream;

  template<class charT, class traits = char_traits<charT>>
    class basic_filebuf;
  template<class charT, class traits = char_traits<charT>>
    class basic_ifstream;
  template<class charT, class traits = char_traits<charT>>
    class basic_ofstream;
  template<class charT, class traits = char_traits<charT>>
    class basic_fstream;

  template<class charT, class traits = char_traits<charT>,
           class Allocator = allocator<charT>>
    class basic_syncbuf;
```

```
  template<class charT, class traits = char_traits<charT>,
           class Allocator = allocator<charT>>
    class basic_osyncstream;

  template<class charT, class traits = char_traits<charT>>
    class istreambuf_iterator;
  template<class charT, class traits = char_traits<charT>>
    class ostreambuf_iterator;

  using ios  = basic_ios<char>;
  using wios = basic_ios<wchar_t>;

  using streambuf = basic_streambuf<char>;
  using istream   = basic_istream<char>;
  using ostream   = basic_ostream<char>;
  using iostream  = basic_iostream<char>;

  using stringbuf     = basic_stringbuf<char>;
  using istringstream = basic_istringstream<char>;
  using ostringstream = basic_ostringstream<char>;
  using stringstream  = basic_stringstream<char>;

  using filebuf  = basic_filebuf<char>;
  using ifstream = basic_ifstream<char>;
  using ofstream = basic_ofstream<char>;
  using fstream  = basic_fstream<char>;

  using syncbuf = basic_syncbuf<char>;
  using osyncstream = basic_osyncstream<char>;

  using wstreambuf = basic_streambuf<wchar_t>;
  using wistream   = basic_istream<wchar_t>;
  using wostream   = basic_ostream<wchar_t>;
  using wiostream  = basic_iostream<wchar_t>;

  using wstringbuf     = basic_stringbuf<wchar_t>;
  using wistringstream = basic_istringstream<wchar_t>;
  using wostringstream = basic_ostringstream<wchar_t>;
  using wstringstream  = basic_stringstream<wchar_t>;

  using wfilebuf  = basic_filebuf<wchar_t>;
  using wifstream = basic_ifstream<wchar_t>;
  using wofstream = basic_ofstream<wchar_t>;
  using wfstream  = basic_fstream<wchar_t>;

  using wsyncbuf = basic_syncbuf<wchar_t>;
  using wosyncstream = basic_osyncstream<wchar_t>;

  template<class state> class fpos;
  using streampos  = fpos<char_traits<char>::state_type>;
  using wstreampos = fpos<char_traits<wchar_t>::state_type>;
  using u8streampos = fpos<char_traits<char8_t>::state_type>;
  using u16streampos = fpos<char_traits<char16_t>::state_type>;
  using u32streampos = fpos<char_traits<char32_t>::state_type>;
}
```

1 Default template arguments are described as appearing both in `<iosfwd>` and in the synopsis of other headers but it is well-formed to include both `<iosfwd>` and one or more of the other headers.[287]

287) It is the implementation's responsibility to implement headers so that including `<iosfwd>` and other headers does not violate the rules about multiple occurrences of default arguments.

29.3.2 Overview [iostream.forward.overview]

1 The class template specialization `basic_ios<charT, traits>` serves as a virtual base class for the class templates `basic_istream`, `basic_ostream`, and class templates derived from them. `basic_iostream` is a class template derived from both `basic_istream<charT, traits>` and `basic_ostream<charT, traits>`.

2 The class template specialization `basic_streambuf<charT, traits>` serves as a base class for class templates `basic_stringbuf`, `basic_filebuf`, and `basic_syncbuf`.

3 The class template specialization `basic_istream<charT, traits>` serves as a base class for class templates `basic_istringstream` and `basic_ifstream`.

4 The class template specialization `basic_ostream<charT, traits>` serves as a base class for class templates `basic_ostringstream`, `basic_ofstream`, and `basic_osyncstream`.

5 The class template specialization `basic_iostream<charT, traits>` serves as a base class for class templates `basic_stringstream` and `basic_fstream`.

6 [*Note*: For each of the class templates above, the program is ill-formed if `traits::char_type` is not the same type as `charT` (**??**). — *end note*]

7 Other *typedef-name*s define instances of class templates specialized for `char` or `wchar_t` types.

8 Specializations of the class template `fpos` are used for specifying file position information. [*Example*: The types `streampos` and `wstreampos` are used for positioning streams specialized on `char` and `wchar_t` respectively. — *end example*]

9 [*Note*: This synopsis suggests a circularity between `streampos` and `char_traits<char>`. An implementation can avoid this circularity by substituting equivalent types. — *end note*]

29.4 Standard iostream objects [iostream.objects]

29.4.1 Header <iostream> synopsis [iostream.syn]

```
#include <ios>          // see 29.5.1
#include <streambuf>    // see 29.6.1
#include <istream>      // see 29.7.1
#include <ostream>      // see 29.7.2

namespace std {
  extern istream cin;
  extern ostream cout;
  extern ostream cerr;
  extern ostream clog;

  extern wistream wcin;
  extern wostream wcout;
  extern wostream wcerr;
  extern wostream wclog;
}
```

29.4.2 Overview [iostream.objects.overview]

1 In this Clause, the type name `FILE` refers to the type `FILE` declared in `<cstdio>` (29.12.1).

2 The header `<iostream>` declares objects that associate objects with the standard C streams provided for by the functions declared in `<cstdio>` (29.12), and includes all the headers necessary to use these objects.

3 The objects are constructed and the associations are established at some time prior to or during the first time an object of class `ios_base::Init` is constructed, and in any case before the body of `main` (**??**) begins execution.[288] The objects are not destroyed during program execution.[289] The results of including `<iostream>` in a translation unit shall be as if `<iostream>` defined an instance of `ios_base::Init` with static storage duration.

4 Mixing operations on corresponding wide- and narrow-character streams follows the same semantics as mixing such operations on `FILE`s, as specified in the C standard library.

288) If it is possible for them to do so, implementations should initialize the objects earlier than required.
289) Constructors and destructors for static objects can access these objects to read input from `stdin` or write output to `stdout` or `stderr`.

5 Concurrent access to a synchronized (29.5.3.4) standard iostream object's formatted and unformatted input (29.7.4.1) and output (29.7.5.1) functions or a standard C stream by multiple threads shall not result in a data race (**??**). [*Note*: Users must still synchronize concurrent use of these objects and streams by multiple threads if they wish to avoid interleaved characters. — *end note*]

See also: ISO C 7.21.2

29.4.3 Narrow stream objects [narrow.stream.objects]

`istream cin;`

1 The object `cin` controls input from a stream buffer associated with the object `stdin`, declared in `<cstdio>` (29.12.1).

2 After the object `cin` is initialized, `cin.tie()` returns `&cout`. Its state is otherwise the same as required for `basic_ios<char>::init` (29.5.5.2).

`ostream cout;`

3 The object `cout` controls output to a stream buffer associated with the object `stdout`, declared in `<cstdio>` (29.12.1).

`ostream cerr;`

4 The object `cerr` controls output to a stream buffer associated with the object `stderr`, declared in `<cstdio>` (29.12.1).

5 After the object `cerr` is initialized, `cerr.flags() & unitbuf` is nonzero and `cerr.tie()` returns `&cout`. Its state is otherwise the same as required for `basic_ios<char>::init` (29.5.5.2).

`ostream clog;`

6 The object `clog` controls output to a stream buffer associated with the object `stderr`, declared in `<cstdio>` (29.12.1).

29.4.4 Wide stream objects [wide.stream.objects]

`wistream wcin;`

1 The object `wcin` controls input from a stream buffer associated with the object `stdin`, declared in `<cstdio>` (29.12.1).

2 After the object `wcin` is initialized, `wcin.tie()` returns `&wcout`. Its state is otherwise the same as required for `basic_ios<wchar_t>::init` (29.5.5.2).

`wostream wcout;`

3 The object `wcout` controls output to a stream buffer associated with the object `stdout`, declared in `<cstdio>` (29.12.1).

`wostream wcerr;`

4 The object `wcerr` controls output to a stream buffer associated with the object `stderr`, declared in `<cstdio>` (29.12.1).

5 After the object `wcerr` is initialized, `wcerr.flags() & unitbuf` is nonzero and `wcerr.tie()` returns `&wcout`. Its state is otherwise the same as required for `basic_ios<wchar_t>::init` (29.5.5.2).

`wostream wclog;`

6 The object `wclog` controls output to a stream buffer associated with the object `stderr`, declared in `<cstdio>` (29.12.1).

29.5 Iostreams base classes [iostreams.base]

29.5.1 Header <ios> synopsis [ios.syn]

```
#include <iosfwd>   // see 29.3.1

namespace std {
  using streamoff  = implementation-defined;
  using streamsize = implementation-defined;
```

```
  template<class stateT> class fpos;

  class ios_base;
  template<class charT, class traits = char_traits<charT>>
    class basic_ios;

  // 29.5.6, manipulators
  ios_base& boolalpha  (ios_base& str);
  ios_base& noboolalpha(ios_base& str);

  ios_base& showbase   (ios_base& str);
  ios_base& noshowbase (ios_base& str);

  ios_base& showpoint  (ios_base& str);
  ios_base& noshowpoint(ios_base& str);

  ios_base& showpos    (ios_base& str);
  ios_base& noshowpos  (ios_base& str);

  ios_base& skipws     (ios_base& str);
  ios_base& noskipws   (ios_base& str);

  ios_base& uppercase  (ios_base& str);
  ios_base& nouppercase(ios_base& str);

  ios_base& unitbuf    (ios_base& str);
  ios_base& nounitbuf  (ios_base& str);

  // 29.5.6.2, adjustfield
  ios_base& internal   (ios_base& str);
  ios_base& left       (ios_base& str);
  ios_base& right      (ios_base& str);

  // 29.5.6.3, basefield
  ios_base& dec        (ios_base& str);
  ios_base& hex        (ios_base& str);
  ios_base& oct        (ios_base& str);

  // 29.5.6.4, floatfield
  ios_base& fixed      (ios_base& str);
  ios_base& scientific (ios_base& str);
  ios_base& hexfloat   (ios_base& str);
  ios_base& defaultfloat(ios_base& str);

  // 29.5.7, error reporting
  enum class io_errc {
    stream = 1
  };

  template<> struct is_error_code_enum<io_errc> : public true_type { };
  error_code make_error_code(io_errc e) noexcept;
  error_condition make_error_condition(io_errc e) noexcept;
  const error_category& iostream_category() noexcept;
}
```

29.5.2 Types [stream.types]

```
using streamoff = implementation-defined;
```

1 The type `streamoff` is a synonym for one of the signed basic integral types of sufficient size to represent the maximum possible file size for the operating system.[290]

290) Typically `long long`.

```
using streamsize = implementation-defined;
```

2 The type `streamsize` is a synonym for one of the signed basic integral types. It is used to represent the number of characters transferred in an I/O operation, or the size of I/O buffers.[291]

29.5.3 Class `ios_base` [ios.base]

```
namespace std {
  class ios_base {
  public:
    class failure; // see below

    // 29.5.3.1.2, fmtflags
    using fmtflags = T1;
    static constexpr fmtflags boolalpha = unspecified;
    static constexpr fmtflags dec = unspecified;
    static constexpr fmtflags fixed = unspecified;
    static constexpr fmtflags hex = unspecified;
    static constexpr fmtflags internal = unspecified;
    static constexpr fmtflags left = unspecified;
    static constexpr fmtflags oct = unspecified;
    static constexpr fmtflags right = unspecified;
    static constexpr fmtflags scientific = unspecified;
    static constexpr fmtflags showbase = unspecified;
    static constexpr fmtflags showpoint = unspecified;
    static constexpr fmtflags showpos = unspecified;
    static constexpr fmtflags skipws = unspecified;
    static constexpr fmtflags unitbuf = unspecified;
    static constexpr fmtflags uppercase = unspecified;
    static constexpr fmtflags adjustfield = see below;
    static constexpr fmtflags basefield = see below;
    static constexpr fmtflags floatfield = see below;

    // 29.5.3.1.3, iostate
    using iostate = T2;
    static constexpr iostate badbit = unspecified;
    static constexpr iostate eofbit = unspecified;
    static constexpr iostate failbit = unspecified;
    static constexpr iostate goodbit = see below;

    // 29.5.3.1.4, openmode
    using openmode = T3;
    static constexpr openmode app = unspecified;
    static constexpr openmode ate = unspecified;
    static constexpr openmode binary = unspecified;
    static constexpr openmode in = unspecified;
    static constexpr openmode out = unspecified;
    static constexpr openmode trunc = unspecified;

    // 29.5.3.1.5, seekdir
    using seekdir = T4;
    static constexpr seekdir beg = unspecified;
    static constexpr seekdir cur = unspecified;
    static constexpr seekdir end = unspecified;

    class Init;

    // 29.5.3.2, fmtflags state
    fmtflags flags() const;
    fmtflags flags(fmtflags fmtfl);
    fmtflags setf(fmtflags fmtfl);
    fmtflags setf(fmtflags fmtfl, fmtflags mask);
    void unsetf(fmtflags mask);
```

291) `streamsize` is used in most places where ISO C would use `size_t`.

```
    streamsize precision() const;
    streamsize precision(streamsize prec);
    streamsize width() const;
    streamsize width(streamsize wide);

    // 29.5.3.3, locales
    locale imbue(const locale& loc);
    locale getloc() const;

    // 29.5.3.5, storage
    static int xalloc();
    long&  iword(int idx);
    void*& pword(int idx);

    // destructor
    virtual ~ios_base();

    // 29.5.3.6, callbacks
    enum event { erase_event, imbue_event, copyfmt_event };
    using event_callback = void (*)(event, ios_base&, int idx);
    void register_callback(event_callback fn, int idx);

    ios_base(const ios_base&) = delete;
    ios_base& operator=(const ios_base&) = delete;

    static bool sync_with_stdio(bool sync = true);

  protected:
    ios_base();

  private:
    static int index;  // exposition only
    long*  iarray;     // exposition only
    void** parray;     // exposition only
  };
}
```

1 `ios_base` defines several member types:

(1.1) — a type `failure`, defined as either a class derived from `system_error` or a synonym for a class derived from `system_error`;

(1.2) — a class `Init`;

(1.3) — three bitmask types, `fmtflags`, `iostate`, and `openmode`;

(1.4) — an enumerated type, `seekdir`.

2 It maintains several kinds of data:

(2.1) — state information that reflects the integrity of the stream buffer;

(2.2) — control information that influences how to interpret (format) input sequences and how to generate (format) output sequences;

(2.3) — additional information that is stored by the program for its private use.

3 [*Note*: For the sake of exposition, the maintained data is presented here as:

(3.1) — `static int index`, specifies the next available unique index for the integer or pointer arrays maintained for the private use of the program, initialized to an unspecified value;

(3.2) — `long* iarray`, points to the first element of an arbitrary-length `long` array maintained for the private use of the program;

(3.3) — `void** parray`, points to the first element of an arbitrary-length pointer array maintained for the private use of the program.

— *end note*]

29.5.3.1 Types [ios.types]

29.5.3.1.1 Class `ios_base::failure` [ios.failure]

```
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);
  };
}
```

1 An implementation is permitted to define `ios_base::failure` as a synonym for a class with equivalent functionality to class `ios_base::failure` shown in this subclause. [*Note*: When `ios_base::failure` is a synonym for another type it shall provide a nested type `failure`, to emulate the injected class name. — *end note*] The class `failure` defines the base class for the types of all objects thrown as exceptions, by functions in the iostreams library, to report errors detected during stream buffer operations.

2 When throwing `ios_base::failure` exceptions, implementations should provide values of `ec` that identify the specific reason for the failure. [*Note*: Errors arising from the operating system would typically be reported as `system_category()` errors with an error value of the error number reported by the operating system. Errors arising from within the stream library would typically be reported as `error_code(io_errc::stream, iostream_category())`. — *end note*]

```
explicit failure(const string& msg, const error_code& ec = io_errc::stream);
```

3 *Effects:* Constructs an object of class `failure` by constructing the base class with `msg` and `ec`.

```
explicit failure(const char* msg, const error_code& ec = io_errc::stream);
```

4 *Effects:* Constructs an object of class `failure` by constructing the base class with `msg` and `ec`.

29.5.3.1.2 Type `ios_base::fmtflags` [ios.fmtflags]

```
using fmtflags = T1;
```

1 The type `fmtflags` is a bitmask type (**??**). Setting its elements has the effects indicated in Table 104.

Table 104 — `fmtflags` effects

Element	Effect(s) if set
`boolalpha`	insert and extract `bool` type in alphabetic format
`dec`	converts integer input or generates integer output in decimal base
`fixed`	generate floating-point output in fixed-point notation
`hex`	converts integer input or generates integer output in hexadecimal base
`internal`	adds fill characters at a designated internal point in certain generated output, or identical to `right` if no such point is designated
`left`	adds fill characters on the right (final positions) of certain generated output
`oct`	converts integer input or generates integer output in octal base
`right`	adds fill characters on the left (initial positions) of certain generated output
`scientific`	generates floating-point output in scientific notation
`showbase`	generates a prefix indicating the numeric base of generated integer output
`showpoint`	generates a decimal-point character unconditionally in generated floating-point output
`showpos`	generates a `+` sign in non-negative generated numeric output
`skipws`	skips leading whitespace before certain input operations
`unitbuf`	flushes output after each output operation
`uppercase`	replaces certain lowercase letters with their uppercase equivalents in generated output

2 Type `fmtflags` also defines the constants indicated in Table 105.

Table 105 — `fmtflags` constants

Constant	Allowable values
`adjustfield`	`left | right | internal`
`basefield`	`dec | oct | hex`
`floatfield`	`scientific | fixed`

29.5.3.1.3 Type `ios_base::iostate` [ios.iostate]

```
using iostate = T2;
```

1 The type `iostate` is a bitmask type (**??**) that contains the elements indicated in Table 106.

Table 106 — `iostate` effects

Element	Effect(s) if set
`badbit`	indicates a loss of integrity in an input or output sequence (such as an irrecoverable read error from a file);
`eofbit`	indicates that an input operation reached the end of an input sequence;
`failbit`	indicates that an input operation failed to read the expected characters, or that an output operation failed to generate the desired characters.

2 Type `iostate` also defines the constant:

(2.1) — `goodbit`, the value zero.

29.5.3.1.4 Type `ios_base::openmode` [ios.openmode]

```
using openmode = T3;
```

1 The type `openmode` is a bitmask type (**??**). It contains the elements indicated in Table 107.

Table 107 — `openmode` effects

Element	Effect(s) if set
`app`	seek to end before each write
`ate`	open and seek to end immediately after opening
`binary`	perform input and output in binary mode (as opposed to text mode)
`in`	open for input
`out`	open for output
`trunc`	truncate an existing stream when opening

29.5.3.1.5 Type `ios_base::seekdir` [ios.seekdir]

```
using seekdir = T4;
```

1 The type `seekdir` is an enumerated type (**??**) that contains the elements indicated in Table 108.

Table 108 — `seekdir` effects

Element	Meaning
`beg`	request a seek (for subsequent input or output) relative to the beginning of the stream
`cur`	request a seek relative to the current position within the sequence
`end`	request a seek relative to the current end of the sequence

29.5.3.1.6 Class `ios_base::Init` [ios.init]

```
namespace std {
  class ios_base::Init {
  public:
    Init();
    Init(const Init&) = default;
    ~Init();
    Init& operator=(const Init&) = default;
  private:
    static int init_cnt; // exposition only
  };
}
```

1 The class `Init` describes an object whose construction ensures the construction of the eight objects declared in `<iostream>` (29.4) that associate file stream buffers with the standard C streams provided for by the functions declared in `<cstdio>` (29.12.1).

2 For the sake of exposition, the maintained data is presented here as:

(2.1) — `static int init_cnt`, counts the number of constructor and destructor calls for class `Init`, initialized to zero.

`Init();`

3 *Effects:* ~~Constructs an object of class `Init`.~~ Constructs and initializes the objects `cin`, `cout`, `cerr`, `clog`, `wcin`, `wcout`, `wcerr`, and `wclog` if they have not already been constructed and initialized.

`~Init();`

4 *Effects:* Destroys an object of class `Init`. If there are no other instances of the class still in existence, calls `cout.flush()`, `cerr.flush()`, `clog.flush()`, `wcout.flush()`, `wcerr.flush()`, `wclog.flush()`.

29.5.3.2 State functions [fmtflags.state]

`fmtflags flags() const;`

1 *Returns:* The format control information for both input and output.

`fmtflags flags(fmtflags fmtfl);`

2 *Ensures:* `fmtfl == flags()`.

3 *Returns:* The previous value of `flags()`.

`fmtflags setf(fmtflags fmtfl);`

4 *Effects:* Sets `fmtfl` in `flags()`.

5 *Returns:* The previous value of `flags()`.

`fmtflags setf(fmtflags fmtfl, fmtflags mask);`

6 *Effects:* Clears `mask` in `flags()`, sets `fmtfl & mask` in `flags()`.

7 *Returns:* The previous value of `flags()`.

`void unsetf(fmtflags mask);`

8 *Effects:* Clears `mask` in `flags()`.

`streamsize precision() const;`

9 *Returns:* The precision to generate on certain output conversions.

`streamsize precision(streamsize prec);`

10 *Ensures:* `prec == precision()`.

11 *Returns:* The previous value of `precision()`.

`streamsize width() const;`

12 *Returns:* The minimum field width (number of characters) to generate on certain output conversions.

```
streamsize width(streamsize wide);
```

13 *Ensures:* `wide == width()`.

14 *Returns:* The previous value of `width()`.

29.5.3.3 Functions [ios.base.locales]

```
locale imbue(const locale& loc);
```

1 *Effects:* Calls each registered callback pair `(fn, idx)` (29.5.3.6) as `(*fn)(imbue_event, *this, idx)` at such a time that a call to `ios_base::getloc()` from within `fn` returns the new locale value `loc`.

2 *Returns:* The previous value of `getloc()`.

3 *Ensures:* `loc == getloc()`.

```
locale getloc() const;
```

4 *Returns:* If no locale has been imbued, a copy of the global C++ locale, `locale()`, in effect at the time of construction. Otherwise, returns the imbued locale, to be used to perform locale-dependent input and output operations.

29.5.3.4 Static members [ios.members.static]

```
bool sync_with_stdio(bool sync = true);
```

1 *Returns:* `true` if the previous state of the standard iostream objects (29.4) was synchronized and otherwise returns `false`. The first time it is called, the function returns `true`.

2 *Effects:* If any input or output operation has occurred using the standard streams prior to the call, the effect is implementation-defined. Otherwise, called with a `false` argument, it allows the standard streams to operate independently of the standard C streams.

3 When a standard iostream object `str` is *synchronized* with a standard stdio stream `f`, the effect of inserting a character `c` by

```
fputc(f, c);
```

is the same as the effect of

```
str.rdbuf()->sputc(c);
```

for any sequences of characters; the effect of extracting a character `c` by

```
c = fgetc(f);
```

is the same as the effect of

```
c = str.rdbuf()->sbumpc();
```

for any sequences of characters; and the effect of pushing back a character `c` by

```
ungetc(c, f);
```

is the same as the effect of

```
str.rdbuf()->sputbackc(c);
```

for any sequence of characters.[292]

29.5.3.5 Storage functions [ios.base.storage]

```
static int xalloc();
```

1 *Returns:* `index ++`.

2 *Remarks:* Concurrent access to this function by multiple threads shall not result in a data race (**??**).

```
long& iword(int idx);
```

3 ~~*Requires:*~~ *Expects:* `idx` is a value obtained by a call to `xalloc`.

4 *Effects:* If `iarray` is a null pointer, allocates an array of `long` of unspecified size and stores a pointer to its first element in `iarray`. The function then extends the array pointed at by `iarray` as necessary

292) This implies that operations on a standard iostream object can be mixed arbitrarily with operations on the corresponding stdio stream. In practical terms, synchronization usually means that a standard iostream object and a standard stdio object share a buffer.

to include the element `iarray[idx]`. Each newly allocated element of the array is initialized to zero. The reference returned is invalid after any other operations on the object.[293] However, the value of the storage referred to is retained, so that until the next call to `copyfmt`, calling `iword` with the same index yields another reference to the same value. If the function fails[294] and `*this` is a base class subobject of a `basic_ios<>` object or subobject, the effect is equivalent to calling `basic_ios<>::setstate(badbit)` on the derived object (which may throw `failure`).

5 *Returns:* On success `iarray[idx]`. On failure, a valid `long&` initialized to 0.

```
void*& pword(int idx);
```

6 ~~*Requires:*~~ *Expects:* `idx` is a value obtained by a call to `xalloc`.

7 *Effects:* If `parray` is a null pointer, allocates an array of pointers to `void` of unspecified size and stores a pointer to its first element in `parray`. The function then extends the array pointed at by `parray` as necessary to include the element `parray[idx]`. Each newly allocated element of the array is initialized to a null pointer. The reference returned is invalid after any other operations on the object. However, the value of the storage referred to is retained, so that until the next call to `copyfmt`, calling `pword` with the same index yields another reference to the same value. If the function fails[295] and `*this` is a base class subobject of a `basic_ios<>` object or subobject, the effect is equivalent to calling `basic_ios<>::setstate(badbit)` on the derived object (which may throw `failure`).

8 *Returns:* On success `parray[idx]`. On failure a valid `void*&` initialized to 0.

9 *Remarks:* After a subsequent call to `pword(int)` for the same object, the earlier return value may no longer be valid.

29.5.3.6 Callbacks [ios.base.callback]

```
void register_callback(event_callback fn, int idx);
```

1 *Effects:* Registers the pair `(fn, idx)` such that during calls to `imbue()` (29.5.3.3), `copyfmt()`, or `~ios_base()` (29.5.3.7), the function `fn` is called with argument `idx`. Functions registered are called when an event occurs, in opposite order of registration. Functions registered while a callback function is active are not called until the next event.

2 ~~*Requires:*~~ *Expects:* The function `fn` ~~shall~~does not throw exceptions.

3 *Remarks:* Identical pairs are not merged. A function registered twice will be called twice.

29.5.3.7 Constructors and destructor [ios.base.cons]

```
ios_base();
```

1 *Effects:* Each `ios_base` member has an indeterminate value after construction. The object's members shall be initialized by calling `basic_ios::init` before the object's first use or before it is destroyed, whichever comes first; otherwise the behavior is undefined.

```
~ios_base();
```

2 *Effects:* Destroys an object of class `ios_base`. Calls each registered callback pair `(fn, idx)` (29.5.3.6) as `(*fn)(erase_event, *this, idx)` at such time that any `ios_base` member function called from within `fn` has well-defined results.

29.5.4 Class template `fpos` [fpos]

```
namespace std {
  template<class stateT> class fpos {
  public:
    // 29.5.4.1, members
    stateT state() const;
    void state(stateT);
  private;
    stateT st; // exposition only
```

293) An implementation is free to implement both the integer array pointed at by `iarray` and the pointer array pointed at by `parray` as sparse data structures, possibly with a one-element cache for each.
294) For example, because it cannot allocate space.
295) For example, because it cannot allocate space.

```
  };
}
```

29.5.4.1 Members [fpos.members]

```
void state(stateT s);
```

1 *Effects:* Assigns `s` to `st`.

```
stateT state() const;
```

2 *Returns:* Current value of `st`.

29.5.4.2 Requirements [fpos.operations]

1 An `fpos` type specifies file position information. It holds a state object whose type is equal to the template parameter `stateT`. Type `stateT` shall meet the *Cpp17DefaultConstructible* (Table **??**), *Cpp17CopyConstructible* (Table **??**), *Cpp17CopyAssignable* (Table **??**), and *Cpp17Destructible* (Table **??**) requirements. If `is_trivially_copy_constructible_v<stateT>` is `true`, then `fpos<stateT>` has a trivial copy constructor. If `is_trivially_copy_assignable<stateT>` is `true`, then `fpos<stateT>` has a trivial copy assignment operator. If `is_trivially_destructible_v<stateT>` is `true`, then `fpos<stateT>` has a trivial destructor. All specializations of `fpos` ~~satisfy~~meet the *Cpp17DefaultConstructible*, *Cpp17CopyConstructible*, *Cpp17CopyAssignable*, *Cpp17Destructible*, and *Cpp17EqualityComparable* (Table **??**) requirements. In addition, the expressions shown in Table 109 are valid and have the indicated semantics. In that table,

(1.1) — `P` refers to an instance of `fpos`,

(1.2) — `p` and `q` refer to values of type `P` or `const P`,

(1.3) — `pl` and `ql` refer to modifiable lvalues of type `P`,

(1.4) — `O` refers to type `streamoff`, and

(1.5) — `o` refers to a value of type `streamoff` or `const streamoff`.

Table 109 — Position type requirements

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
`P(o)`	`P`	converts from `offset`	*Effects:* Value-initializes the state object.
`P p(o);` `P p = o;`			*Effects:* Value-initializes the state object. *Ensures:* `p == P(o)`
`P()`	`P`	`P(0)`	
`P p;`		`P p(0);`	
`O(p)`	`streamoff`	converts to `offset`	`P(O(p)) == p`
`p != q`	convertible to `bool`	`!(p == q)`	
`p + o`	`P`	`+` offset	*Remarks:* With `ql = p + o;`, then: `ql - o == p`
`pl += o`	`P&`	`+=` offset	*Remarks:* With `ql = pl;` before the `+=`, then: `pl - o == ql`
`p - o`	`P`	`-` offset	*Remarks:* With `ql = p - o;`, then: `ql + o == p`
`pl -= o`	`P&`	`-=` offset	*Remarks:* With `ql = pl;` before the `-=`, then: `pl + o == ql`
`o + p`	convertible to `P`	`p + o`	`P(o + p) == p + o`
`p - q`	`streamoff`	distance	`p == q + (p - q)`

2 Stream operations that return a value of type `traits::pos_type` return `P(O(-1))` as an invalid value to signal an error. If this value is used as an argument to any `istream`, `ostream`, or `streambuf` member that accepts a value of type `traits::pos_type` then the behavior of that function is undefined.

29.5.5 Class template `basic_ios` [ios]

29.5.5.1 Overview [ios.overview]

```
namespace std {
  template<class charT, class traits = char_traits<charT>>
  class basic_ios : public ios_base {
  public:
    using char_type   = charT;
    using int_type    = typename traits::int_type;
    using pos_type    = typename traits::pos_type;
    using off_type    = typename traits::off_type;
    using traits_type = traits;

    // 29.5.5.4, flags functions
    explicit operator bool() const;
    bool operator!() const;
    iostate rdstate() const;
    void clear(iostate state = goodbit);
    void setstate(iostate state);
    bool good() const;
    bool eof()  const;
    bool fail() const;
    bool bad()  const;

    iostate exceptions() const;
    void exceptions(iostate except);

    // 29.5.5.2, constructor/destructor
    explicit basic_ios(basic_streambuf<charT, traits>* sb);
    virtual ~basic_ios();

    // 29.5.5.3, members
    basic_ostream<charT, traits>* tie() const;
    basic_ostream<charT, traits>* tie(basic_ostream<charT, traits>* tiestr);

    basic_streambuf<charT, traits>* rdbuf() const;
    basic_streambuf<charT, traits>* rdbuf(basic_streambuf<charT, traits>* sb);

    basic_ios& copyfmt(const basic_ios& rhs);

    char_type fill() const;
    char_type fill(char_type ch);

    locale imbue(const locale& loc);

    char      narrow(char_type c, char dfault) const;
    char_type widen(char c) const;

    basic_ios(const basic_ios&) = delete;
    basic_ios& operator=(const basic_ios&) = delete;

  protected:
    basic_ios();
    void init(basic_streambuf<charT, traits>* sb);
    void move(basic_ios& rhs);
    void move(basic_ios&& rhs);
    void swap(basic_ios& rhs) noexcept;
    void set_rdbuf(basic_streambuf<charT, traits>* sb);

  };
}
```

29.5.5.2 Constructors [basic.ios.cons]

```
explicit basic_ios(basic_streambuf<charT, traits>* sb);
```

1 *Effects:* Constructs an object of class `basic_ios`, assigning initial values to its member objects by calling `init(sb)`.

```
basic_ios();
```

2 *Effects:* Constructs an object of class `basic_ios` (29.5.3.7) leaving its member objects uninitialized. The object shall be initialized by calling `basic_ios::init` before its first use or before it is destroyed, whichever comes first; otherwise the behavior is undefined.

```
~basic_ios();
```

3 *Remarks:* The destructor does not destroy `rdbuf()`.

```
void init(basic_streambuf<charT, traits>* sb);
```

4 *Ensures:* The postconditions of this function are indicated in Table 110.

Table 110 — `basic_ios::init()` effects

Element	Value
`rdbuf()`	`sb`
`tie()`	`0`
`rdstate()`	`goodbit` if `sb` is not a null pointer, otherwise `badbit`.
`exceptions()`	`goodbit`
`flags()`	`skipws | dec`
`width()`	`0`
`precision()`	`6`
`fill()`	`widen(' ')`
`getloc()`	a copy of the value returned by `locale()`
`iarray`	a null pointer
`parray`	a null pointer

29.5.5.3 Member functions [basic.ios.members]

```
basic_ostream<charT, traits>* tie() const;
```

1 *Returns:* An output sequence that is *tied* to (synchronized with) the sequence controlled by the stream buffer.

```
basic_ostream<charT, traits>* tie(basic_ostream<charT, traits>* tiestr);
```

2 ~~*Requires:*~~ *Expects:* If `tiestr` is not null, `tiestr` ~~shall not be~~is not reachable by traversing the linked list of tied stream objects starting from `tiestr->tie()`.

3 *Ensures:* `tiestr == tie()`.

4 *Returns:* The previous value of `tie()`.

```
basic_streambuf<charT, traits>* rdbuf() const;
```

5 *Returns:* A pointer to the `streambuf` associated with the stream.

```
basic_streambuf<charT, traits>* rdbuf(basic_streambuf<charT, traits>* sb);
```

6 *Ensures:* `sb == rdbuf()`.

7 *Effects:* Calls `clear()`.

8 *Returns:* The previous value of `rdbuf()`.

```
locale imbue(const locale& loc);
```

9 *Effects:* Calls `ios_base::imbue(loc)` (29.5.3.3) and if `rdbuf() != 0` then `rdbuf()->pubimbue(loc)` (29.6.3.2.1).

10 *Returns:* The prior value of `ios_base::imbue()`.

```
char narrow(char_type c, char dfault) const;
```

11 *Returns:* `use_facet<ctype<char_type>>(getloc()).narrow(c, dfault)`

```
char_type widen(char c) const;
```

12 *Returns:* `use_facet<ctype<char_type>>(getloc()).widen(c)`

```
char_type fill() const;
```

13 *Returns:* The character used to pad (fill) an output conversion to the specified field width.

```
char_type fill(char_type fillch);
```

14 *Ensures:* `traits::eq(fillch, fill())`.

15 *Returns:* The previous value of `fill()`.

```
basic_ios& copyfmt(const basic_ios& rhs);
```

16 *Effects:* If `(this == addressof(rhs))` does nothing. Otherwise assigns to the member objects of `*this` the corresponding member objects of `rhs` as follows:

(16.1) — calls each registered callback pair `(fn, idx)` as `(*fn)(erase_event, *this, idx)`;

(16.2) — then, assigns to the member objects of `*this` the corresponding member objects of `rhs`, except that

(16.2.1) — `rdstate()`, `rdbuf()`, and `exceptions()` are left unchanged;

(16.2.2) — the contents of arrays pointed at by `pword` and `iword` are copied, not the pointers themselves;[296] and

(16.2.3) — if any newly stored pointer values in `*this` point at objects stored outside the object `rhs` and those objects are destroyed when `rhs` is destroyed, the newly stored pointer values are altered to point at newly constructed copies of the objects;

(16.3) — then, calls each callback pair that was copied from `rhs` as `(*fn)(copyfmt_event, *this, idx)`;

(16.4) — then, calls `exceptions(rhs.exceptions())`.

17 [*Note*: The second pass through the callback pairs permits a copied `pword` value to be zeroed, or to have its referent deep copied or reference counted, or to have other special action taken. — *end note*]

18 *Ensures:* The postconditions of this function are indicated in Table 111.

Table 111 — `basic_ios::copyfmt()` effects

Element	Value
`rdbuf()`	*unchanged*
`tie()`	`rhs.tie()`
`rdstate()`	*unchanged*
`exceptions()`	`rhs.exceptions()`
`flags()`	`rhs.flags()`
`width()`	`rhs.width()`
`precision()`	`rhs.precision()`
`fill()`	`rhs.fill()`
`getloc()`	`rhs.getloc()`

19 *Returns:* `*this`.

```
void move(basic_ios& rhs);
```

296) This suggests an infinite amount of copying, but the implementation can keep track of the maximum element of the arrays that is nonzero.

```
void move(basic_ios&& rhs);
```

20 *Ensures:* `*this` shall have the state that `rhs` had before the function call, except that `rdbuf()` shall return 0. `rhs` shall be in a valid but unspecified state, except that `rhs.rdbuf()` shall return the same value as it returned before the function call, and `rhs.tie()` shall return 0.

```
void swap(basic_ios& rhs) noexcept;
```

21 *Effects:* The states of `*this` and `rhs` shall be exchanged, except that `rdbuf()` shall return the same value as it returned before the function call, and `rhs.rdbuf()` shall return the same value as it returned before the function call.

```
void set_rdbuf(basic_streambuf<charT, traits>* sb);
```

22 ~~*Requires:*~~ *Expects:* `sb != nullptr`.

23 *Effects:* Associates the `basic_streambuf` object pointed to by `sb` with this stream without calling `clear()`.

24 *Ensures:* `rdbuf() == sb`.

25 *Throws:* Nothing.

29.5.5.4 Flags functions [iostate.flags]

```
explicit operator bool() const;
```

1 *Returns:* `!fail()`.

```
bool operator!() const;
```

2 *Returns:* `fail()`.

```
iostate rdstate() const;
```

3 *Returns:* The error state of the stream buffer.

```
void clear(iostate state = goodbit);
```

4 *Ensures:* If `rdbuf() != 0` then `state == rdstate()`; otherwise `rdstate() == (state | ios_base::badbit)`.

5 *Effects:* If `((state | (rdbuf() ? goodbit : badbit)) & exceptions()) == 0`, returns. Otherwise, the function throws an object of class `basic_ios::failure` (29.5.3.1.1), constructed with implementation-defined argument values.

```
void setstate(iostate state);
```

6 *Effects:* Calls `clear(rdstate() | state)` (which may throw `basic_ios::failure` (29.5.3.1.1)).

```
bool good() const;
```

7 *Returns:* `rdstate() == 0`

```
bool eof() const;
```

8 *Returns:* `true` if `eofbit` is set in `rdstate()`.

```
bool fail() const;
```

9 *Returns:* `true` if `failbit` or `badbit` is set in `rdstate()`.[297]

```
bool bad() const;
```

10 *Returns:* `true` if `badbit` is set in `rdstate()`.

```
iostate exceptions() const;
```

11 *Returns:* A mask that determines what elements set in `rdstate()` cause exceptions to be thrown.

```
void exceptions(iostate except);
```

12 *Ensures:* `except == exceptions()`.

297) Checking `badbit` also for `fail()` is historical practice.

13 *Effects:* Calls `clear(rdstate())`.

29.5.6 `ios_base` manipulators [std.ios.manip]

29.5.6.1 `fmtflags` manipulators [fmtflags.manip]

1 Each function specified in this subclause is a designated addressable function (**??**).

```
ios_base& boolalpha(ios_base& str);
```

2 *Effects:* Calls `str.setf(ios_base::boolalpha)`.

3 *Returns:* `str`.

```
ios_base& noboolalpha(ios_base& str);
```

4 *Effects:* Calls `str.unsetf(ios_base::boolalpha)`.

5 *Returns:* `str`.

```
ios_base& showbase(ios_base& str);
```

6 *Effects:* Calls `str.setf(ios_base::showbase)`.

7 *Returns:* `str`.

```
ios_base& noshowbase(ios_base& str);
```

8 *Effects:* Calls `str.unsetf(ios_base::showbase)`.

9 *Returns:* `str`.

```
ios_base& showpoint(ios_base& str);
```

10 *Effects:* Calls `str.setf(ios_base::showpoint)`.

11 *Returns:* `str`.

```
ios_base& noshowpoint(ios_base& str);
```

12 *Effects:* Calls `str.unsetf(ios_base::showpoint)`.

13 *Returns:* `str`.

```
ios_base& showpos(ios_base& str);
```

14 *Effects:* Calls `str.setf(ios_base::showpos)`.

15 *Returns:* `str`.

```
ios_base& noshowpos(ios_base& str);
```

16 *Effects:* Calls `str.unsetf(ios_base::showpos)`.

17 *Returns:* `str`.

```
ios_base& skipws(ios_base& str);
```

18 *Effects:* Calls `str.setf(ios_base::skipws)`.

19 *Returns:* `str`.

```
ios_base& noskipws(ios_base& str);
```

20 *Effects:* Calls `str.unsetf(ios_base::skipws)`.

21 *Returns:* `str`.

```
ios_base& uppercase(ios_base& str);
```

22 *Effects:* Calls `str.setf(ios_base::uppercase)`.

23 *Returns:* `str`.

```
ios_base& nouppercase(ios_base& str);
```

24 *Effects:* Calls `str.unsetf(ios_base::uppercase)`.

25 *Returns:* `str`.

`ios_base& unitbuf(ios_base& str);`

26 *Effects:* Calls `str.setf(ios_base::unitbuf)`.

27 *Returns:* `str`.

`ios_base& nounitbuf(ios_base& str);`

28 *Effects:* Calls `str.unsetf(ios_base::unitbuf)`.

29 *Returns:* `str`.

29.5.6.2 `adjustfield` manipulators [adjustfield.manip]

1 Each function specified in this subclause is a designated addressable function (**??**).

`ios_base& internal(ios_base& str);`

2 *Effects:* Calls `str.setf(ios_base::internal, ios_base::adjustfield)`.

3 *Returns:* `str`.

`ios_base& left(ios_base& str);`

4 *Effects:* Calls `str.setf(ios_base::left, ios_base::adjustfield)`.

5 *Returns:* `str`.

`ios_base& right(ios_base& str);`

6 *Effects:* Calls `str.setf(ios_base::right, ios_base::adjustfield)`.

7 *Returns:* `str`.

29.5.6.3 `basefield` manipulators [basefield.manip]

1 Each function specified in this subclause is a designated addressable function (**??**).

`ios_base& dec(ios_base& str);`

2 *Effects:* Calls `str.setf(ios_base::dec, ios_base::basefield)`.

3 *Returns:* `str`[298].

`ios_base& hex(ios_base& str);`

4 *Effects:* Calls `str.setf(ios_base::hex, ios_base::basefield)`.

5 *Returns:* `str`.

`ios_base& oct(ios_base& str);`

6 *Effects:* Calls `str.setf(ios_base::oct, ios_base::basefield)`.

7 *Returns:* `str`.

29.5.6.4 `floatfield` manipulators [floatfield.manip]

1 Each function specified in this subclause is a designated addressable function (**??**).

`ios_base& fixed(ios_base& str);`

2 *Effects:* Calls `str.setf(ios_base::fixed, ios_base::floatfield)`.

3 *Returns:* `str`.

`ios_base& scientific(ios_base& str);`

4 *Effects:* Calls `str.setf(ios_base::scientific, ios_base::floatfield)`.

5 *Returns:* `str`.

`ios_base& hexfloat(ios_base& str);`

6 *Effects:* Calls `str.setf(ios_base::fixed | ios_base::scientific, ios_base::floatfield)`.

7 *Returns:* `str`.

298) The function signature `dec(ios_base&)` can be called by the function signature `basic_ostream& stream::operator<<(ios_base& (*)(ios_base&))` to permit expressions of the form `cout << dec` to change the format flags stored in `cout`.

8 [*Note*: The more obvious use of `ios_base::hex` to specify hexadecimal floating-point format would change the meaning of existing well-defined programs. C++ 2003 gives no meaning to the combination of `fixed` and `scientific`. — *end note*]

```
ios_base& defaultfloat(ios_base& str);
```

9 *Effects:* Calls `str.unsetf(ios_base::floatfield)`.

10 *Returns:* `str`.

29.5.7 Error reporting [error.reporting]

```
error_code make_error_code(io_errc e) noexcept;
```

1 *Returns:* `error_code(static_cast<int>(e), iostream_category())`.

```
error_condition make_error_condition(io_errc e) noexcept;
```

2 *Returns:* `error_condition(static_cast<int>(e), iostream_category())`.

```
const error_category& iostream_category() noexcept;
```

3 *Returns:* A reference to an object of a type derived from class `error_category`.

4 The object's `default_error_condition` and `equivalent` virtual functions shall behave as specified for the class `error_category`. The object's `name` virtual function shall return a pointer to the string `"iostream"`.

29.6 Stream buffers [stream.buffers]

29.6.1 Header <streambuf> synopsis [streambuf.syn]

```
namespace std {
  template<class charT, class traits = char_traits<charT>>
    class basic_streambuf;
  using streambuf  = basic_streambuf<char>;
  using wstreambuf = basic_streambuf<wchar_t>;
}
```

1 The header `<streambuf>` defines types that control input from and output to *character* sequences.

29.6.2 Stream buffer requirements [streambuf.reqts]

1 Stream buffers can impose various constraints on the sequences they control. Some constraints are:

(1.1) — The controlled input sequence can be not readable.

(1.2) — The controlled output sequence can be not writable.

(1.3) — The controlled sequences can be associated with the contents of other representations for character sequences, such as external files.

(1.4) — The controlled sequences can support operations *directly* to or from associated sequences.

(1.5) — The controlled sequences can impose limitations on how the program can read characters from a sequence, write characters to a sequence, put characters back into an input sequence, or alter the stream position.

2 Each sequence is characterized by three pointers which, if non-null, all point into the same `charT` array object. The array object represents, at any moment, a (sub)sequence of characters from the sequence. Operations performed on a sequence alter the values stored in these pointers, perform reads and writes directly to or from associated sequences, and alter "the stream position" and conversion state as needed to maintain this subsequence relationship. The three pointers are:

(2.1) — the *beginning pointer*, or lowest element address in the array (called `xbeg` here);

(2.2) — the *next pointer*, or next element address that is a current candidate for reading or writing (called `xnext` here);

(2.3) — the *end pointer*, or first element address beyond the end of the array (called `xend` here).

3 The following semantic constraints shall always apply for any set of three pointers for a sequence, using the pointer names given immediately above:

(3.1) — If `xnext` is not a null pointer, then `xbeg` and `xend` shall also be non-null pointers into the same `charT` array, as described above; otherwise, `xbeg` and `xend` shall also be null.

(3.2) — If `xnext` is not a null pointer and `xnext < xend` for an output sequence, then a *write position* is available. In this case, `*xnext` shall be assignable as the next element to write (to put, or to store a character value, into the sequence).

(3.3) — If `xnext` is not a null pointer and `xbeg < xnext` for an input sequence, then a *putback position* is available. In this case, `xnext[-1]` shall have a defined value and is the next (preceding) element to store a character that is put back into the input sequence.

(3.4) — If `xnext` is not a null pointer and `xnext < xend` for an input sequence, then a *read position* is available. In this case, `*xnext` shall have a defined value and is the next element to read (to get, or to obtain a character value, from the sequence).

29.6.3 Class template `basic_streambuf` [streambuf]

```
namespace std {
  template<class charT, class traits = char_traits<charT>>
  class basic_streambuf {
  public:
    using char_type   = charT;
    using int_type    = typename traits::int_type;
    using pos_type    = typename traits::pos_type;
    using off_type    = typename traits::off_type;
    using traits_type = traits;

    virtual ~basic_streambuf();

    // 29.6.3.2.1, locales
    locale   pubimbue(const locale& loc);
    locale   getloc() const;

    // 29.6.3.2.2, buffer and positioning
    basic_streambuf* pubsetbuf(char_type* s, streamsize n);
    pos_type pubseekoff(off_type off, ios_base::seekdir way,
                        ios_base::openmode which
                          = ios_base::in | ios_base::out);
    pos_type pubseekpos(pos_type sp,
                        ios_base::openmode which
                          = ios_base::in | ios_base::out);
    int      pubsync();

    // get and put areas
    // 29.6.3.2.3, get area
    streamsize in_avail();
    int_type snextc();
    int_type sbumpc();
    int_type sgetc();
    streamsize sgetn(char_type* s, streamsize n);

    // 29.6.3.2.4, putback
    int_type sputbackc(char_type c);
    int_type sungetc();

    // 29.6.3.2.5, put area
    int_type   sputc(char_type c);
    streamsize sputn(const char_type* s, streamsize n);

  protected:
    basic_streambuf();
    basic_streambuf(const basic_streambuf& rhs);
    basic_streambuf& operator=(const basic_streambuf& rhs);

    void swap(basic_streambuf& rhs);
```

```
    // 29.6.3.3.2, get area access
    char_type* eback() const;
    char_type* gptr()  const;
    char_type* egptr() const;
    void       gbump(int n);
    void       setg(char_type* gbeg, char_type* gnext, char_type* gend);

    // 29.6.3.3.3, put area access
    char_type* pbase() const;
    char_type* pptr() const;
    char_type* epptr() const;
    void       pbump(int n);
    void       setp(char_type* pbeg, char_type* pend);

    // 29.6.3.4, virtual functions
    // 29.6.3.4.1, locales
    virtual void imbue(const locale& loc);

    // 29.6.3.4.2, buffer management and positioning
    virtual basic_streambuf* setbuf(char_type* s, streamsize n);
    virtual pos_type seekoff(off_type off, ios_base::seekdir way,
                             ios_base::openmode which
                               = ios_base::in | ios_base::out);
    virtual pos_type seekpos(pos_type sp,
                             ios_base::openmode which
                               = ios_base::in | ios_base::out);
    virtual int      sync();

    // 29.6.3.4.3, get area
    virtual streamsize showmanyc();
    virtual streamsize xsgetn(char_type* s, streamsize n);
    virtual int_type   underflow();
    virtual int_type   uflow();

    // 29.6.3.4.4, putback
    virtual int_type   pbackfail(int_type c = traits::eof());

    // 29.6.3.4.5, put area
    virtual streamsize xsputn(const char_type* s, streamsize n);
    virtual int_type   overflow(int_type c = traits::eof());
  };
}
```

1 The class template `basic_streambuf` serves as an abstract base class for deriving various *stream buffers* whose objects each control two *character sequences*:

(1.1) — a character *input sequence*;

(1.2) — a character *output sequence*.

29.6.3.1 Constructors [streambuf.cons]

`basic_streambuf();`

1 *Effects:* Constructs an object of class `basic_streambuf<charT, traits>` and initializes:[299]

(1.1) — all its pointer member objects to null pointers,

(1.2) — the `getloc()` member to a copy the global locale, `locale()`, at the time of construction.

2 *Remarks:* Once the `getloc()` member is initialized, results of calling locale member functions, and of members of facets so obtained, can safely be cached until the next time the member `imbue` is called.

299) The default constructor is protected for class `basic_streambuf` to assure that only objects for classes derived from this class may be constructed.

```
basic_streambuf(const basic_streambuf& rhs);
```

3 *Effects:* Constructs a copy of `rhs`.

4 *Ensures:*

(4.1) — `eback() == rhs.eback()`

(4.2) — `gptr() == rhs.gptr()`

(4.3) — `egptr() == rhs.egptr()`

(4.4) — `pbase() == rhs.pbase()`

(4.5) — `pptr() == rhs.pptr()`

(4.6) — `epptr() == rhs.epptr()`

(4.7) — `getloc() == rhs.getloc()`

```
~basic_streambuf();
```

5 *Effects:* None.

29.6.3.2 Public member functions [streambuf.members]

29.6.3.2.1 Locales [streambuf.locales]

```
locale pubimbue(const locale& loc);
```

1 *Ensures:* `loc == getloc()`.

2 *Effects:* Calls `imbue(loc)`.

3 *Returns:* Previous value of `getloc()`.

```
locale getloc() const;
```

4 *Returns:* If `pubimbue()` has ever been called, then the last value of `loc` supplied, otherwise the current global locale, `locale()`, in effect at the time of construction. If called after `pubimbue()` has been called but before `pubimbue` has returned (i.e., from within the call of `imbue()`) then it returns the previous value.

29.6.3.2.2 Buffer management and positioning [streambuf.buffer]

```
basic_streambuf* pubsetbuf(char_type* s, streamsize n);
```

1 *Returns:* `setbuf(s, n)`.

```
pos_type pubseekoff(off_type off, ios_base::seekdir way,
                    ios_base::openmode which
                      = ios_base::in | ios_base::out);
```

2 *Returns:* `seekoff(off, way, which)`.

```
pos_type pubseekpos(pos_type sp,
                    ios_base::openmode which
                      = ios_base::in | ios_base::out);
```

3 *Returns:* `seekpos(sp, which)`.

```
int pubsync();
```

4 *Returns:* `sync()`.

29.6.3.2.3 Get area [streambuf.pub.get]

```
streamsize in_avail();
```

1 *Returns:* If a read position is available, returns `egptr() - gptr()`. Otherwise returns `showmanyc()` (29.6.3.4.3).

```
int_type snextc();
```

2 *Effects:* Calls `sbumpc()`.

3 *Returns:* If that function returns `traits::eof()`, returns `traits::eof()`. Otherwise, returns `sgetc()`.

```
int_type sbumpc();
```

4 *Returns:* If the input sequence read position is not available, returns `uflow()`. Otherwise, returns `traits::to_int_type(*gptr())` and increments the next pointer for the input sequence.

```
int_type sgetc();
```

5 *Returns:* If the input sequence read position is not available, returns `underflow()`. Otherwise, returns `traits::to_int_type(*gptr())`.

```
streamsize sgetn(char_type* s, streamsize n);
```

6 *Returns:* `xsgetn(s, n)`.

29.6.3.2.4 Putback [streambuf.pub.pback]

```
int_type sputbackc(char_type c);
```

1 *Returns:* If the input sequence putback position is not available, or if `traits::eq(c, gptr()[-1])` is `false`, returns `pbackfail(traits::to_int_type(c))`. Otherwise, decrements the next pointer for the input sequence and returns `traits::to_int_type(*gptr())`.

```
int_type sungetc();
```

2 *Returns:* If the input sequence putback position is not available, returns `pbackfail()`. Otherwise, decrements the next pointer for the input sequence and returns `traits::to_int_type(*gptr())`.

29.6.3.2.5 Put area [streambuf.pub.put]

```
int_type sputc(char_type c);
```

1 *Returns:* If the output sequence write position is not available, returns `overflow(traits::to_int_type(c))`. Otherwise, stores `c` at the next pointer for the output sequence, increments the pointer, and returns `traits::to_int_type(c)`.

```
streamsize sputn(const char_type* s, streamsize n);
```

2 *Returns:* `xsputn(s, n)`.

29.6.3.3 Protected member functions [streambuf.protected]

29.6.3.3.1 Assignment [streambuf.assign]

```
basic_streambuf& operator=(const basic_streambuf& rhs);
```

1 *Effects:* Assigns the data members of `rhs` to `*this`.

2 *Ensures:*

(2.1) — `eback() == rhs.eback()`

(2.2) — `gptr() == rhs.gptr()`

(2.3) — `egptr() == rhs.egptr()`

(2.4) — `pbase() == rhs.pbase()`

(2.5) — `pptr() == rhs.pptr()`

(2.6) — `epptr() == rhs.epptr()`

(2.7) — `getloc() == rhs.getloc()`

3 *Returns:* `*this`.

```
void swap(basic_streambuf& rhs);
```

4 *Effects:* Swaps the data members of `rhs` and `*this`.

29.6.3.3.2 Get area access [streambuf.get.area]

```
char_type* eback() const;
```

1 *Returns:* The beginning pointer for the input sequence.

```
char_type* gptr() const;
```

2 *Returns:* The next pointer for the input sequence.

```
char_type* egptr() const;
```

3 *Returns:* The end pointer for the input sequence.

```
void gbump(int n);
```

4 *Effects:* Adds `n` to the next pointer for the input sequence.

```
void setg(char_type* gbeg, char_type* gnext, char_type* gend);
```

5 *Ensures:* `gbeg == eback()`, `gnext == gptr()`, and `gend == egptr()`.

29.6.3.3.3 Put area access [streambuf.put.area]

```
char_type* pbase() const;
```

1 *Returns:* The beginning pointer for the output sequence.

```
char_type* pptr() const;
```

2 *Returns:* The next pointer for the output sequence.

```
char_type* epptr() const;
```

3 *Returns:* The end pointer for the output sequence.

```
void pbump(int n);
```

4 *Effects:* Adds `n` to the next pointer for the output sequence.

```
void setp(char_type* pbeg, char_type* pend);
```

5 *Ensures:* `pbeg == pbase()`, `pbeg == pptr()`, and `pend == epptr()`.

29.6.3.4 Virtual functions [streambuf.virtuals]

29.6.3.4.1 Locales [streambuf.virt.locales]

```
void imbue(const locale&);
```

1 *Effects:* Change any translations based on locale.

2 *Remarks:* Allows the derived class to be informed of changes in locale at the time they occur. Between invocations of this function a class derived from streambuf can safely cache results of calls to locale functions and to members of facets so obtained.

3 *Default behavior:* Does nothing.

29.6.3.4.2 Buffer management and positioning [streambuf.virt.buffer]

```
basic_streambuf* setbuf(char_type* s, streamsize n);
```

1 *Effects:* Influences stream buffering in a way that is defined separately for each class derived from `basic_streambuf` in this Clause (29.8.2.4, 29.9.2.4).

2 *Default behavior:* Does nothing. Returns `this`.

```
pos_type seekoff(off_type off, ios_base::seekdir way,
                 ios_base::openmode which
                  = ios_base::in | ios_base::out);
```

3 *Effects:* Alters the stream positions within one or more of the controlled sequences in a way that is defined separately for each class derived from `basic_streambuf` in this Clause (29.8.2.4, 29.9.2.4).

4 *Default behavior:* Returns `pos_type(off_type(-1))`.

```
pos_type seekpos(pos_type sp,
                 ios_base::openmode which
                  = ios_base::in | ios_base::out);
```

5 *Effects:* Alters the stream positions within one or more of the controlled sequences in a way that is defined separately for each class derived from `basic_streambuf` in this Clause (29.8.2, 29.9.2).

6 *Default behavior:* Returns `pos_type(off_type(-1))`.

```
int sync();
```

7 *Effects:* Synchronizes the controlled sequences with the arrays. That is, if `pbase()` is non-null the characters between `pbase()` and `pptr()` are written to the controlled sequence. The pointers may then be reset as appropriate.

8 *Returns:* `-1` on failure. What constitutes failure is determined by each derived class (29.9.2.4).

9 *Default behavior:* Returns zero.

29.6.3.4.3 Get area [streambuf.virt.get]

```
streamsize showmanyc();
```[300]

1 *Returns:* An estimate of the number of characters available in the sequence, or -1. If it returns a positive value, then successive calls to `underflow()` will not return `traits::eof()` until at least that number of characters have been extracted from the stream. If `showmanyc()` returns -1, then calls to `underflow()` or `uflow()` will fail.[301]

2 *Default behavior:* Returns zero.

3 *Remarks:* Uses `traits::eof()`.

```
streamsize xsgetn(char_type* s, streamsize n);
```

4 *Effects:* Assigns up to `n` characters to successive elements of the array whose first element is designated by `s`. The characters assigned are read from the input sequence as if by repeated calls to `sbumpc()`. Assigning stops when either `n` characters have been assigned or a call to `sbumpc()` would return `traits::eof()`.

5 *Returns:* The number of characters assigned.[302]

6 *Remarks:* Uses `traits::eof()`.

```
int_type underflow();
```

7 *Remarks:* The public members of `basic_streambuf` call this virtual function only if `gptr()` is null or `gptr() >= egptr()`

8 *Returns:* `traits::to_int_type(c)`, where `c` is the first *character* of the *pending sequence*, without moving the input sequence position past it. If the pending sequence is null then the function returns `traits::eof()` to indicate failure.

9 The *pending sequence* of characters is defined as the concatenation of

(9.1) — the empty sequence if `gptr()` is null, otherwise the characters in `[gptr(), egptr())`, followed by

(9.2) — some (possibly empty) sequence of characters read from the input sequence.

10 The *result character* is the first character of the pending sequence if it is non-empty, otherwise the next character that would be read from the input sequence.

11 The *backup sequence* is the empty sequence if `eback()` is null, otherwise the characters in `[eback(), gptr())`.

12 *Effects:* The function sets up the `gptr()` and `egptr()` such that if the pending sequence is non-empty, then `egptr()` is non-null and the characters in `[gptr(), egptr())` are the characters in the pending sequence, otherwise either `gptr()` is null or `gptr() == egptr()`.

13 If `eback()` and `gptr()` are non-null then the function is not constrained as to their contents, but the "usual backup condition" is that either

(13.1) — the backup sequence contains at least `gptr() - eback()` characters, in which case the characters in `[eback(), gptr())` agree with the last `gptr() - eback()` characters of the backup sequence, or

300) The morphemes of `showmanyc` are "es-how-many-see", not "show-manic".
301) `underflow` or `uflow` might fail by throwing an exception prematurely. The intention is not only that the calls will not return `eof()` but that they will return "immediately".
302) Classes derived from `basic_streambuf` can provide more efficient ways to implement `xsgetn()` and `xsputn()` by overriding these definitions from the base class.

(13.2) — the characters in `[gptr() - n, gptr())` agree with the backup sequence (where `n` is the length of the backup sequence).

14 *Default behavior:* Returns `traits::eof()`.

```
int_type uflow();
```

15 *Requires:* The constraints are the same as for `underflow()`, except that the result character shall be transferred from the pending sequence to the backup sequence, and the pending sequence shall not be empty before the transfer.

16 *Default behavior:* Calls `underflow()`. If `underflow()` returns `traits::eof()`, returns `traits::eof()`. Otherwise, returns the value of `traits::to_int_type(*gptr())` and increment the value of the next pointer for the input sequence.

17 *Returns:* `traits::eof()` to indicate failure.

29.6.3.4.4 Putback [streambuf.virt.pback]

```
int_type pbackfail(int_type c = traits::eof());
```

1 *Remarks:* The public functions of `basic_streambuf` call this virtual function only when `gptr()` is null, `gptr() == eback()`, or `traits::eq(traits::to_char_type(c), gptr()[-1])` returns `false`. Other calls shall also satisfy that constraint.

The *pending sequence* is defined as for `underflow()`, with the modifications that

(1.1) — If `traits::eq_int_type(c, traits::eof())` returns `true`, then the input sequence is backed up one character before the pending sequence is determined.

(1.2) — If `traits::eq_int_type(c, traits::eof())` returns `false`, then `c` is prepended. Whether the input sequence is backed up or modified in any other way is unspecified.

2 *Ensures:* On return, the constraints of `gptr()`, `eback()`, and `pptr()` are the same as for `underflow()`.

3 *Returns:* `traits::eof()` to indicate failure. Failure may occur because the input sequence could not be backed up, or if for some other reason the pointers could not be set consistent with the constraints. `pbackfail()` is called only when put back has really failed.

4 Returns some value other than `traits::eof()` to indicate success.

5 *Default behavior:* Returns `traits::eof()`.

29.6.3.4.5 Put area [streambuf.virt.put]

```
streamsize xsputn(const char_type* s, streamsize n);
```

1 *Effects:* Writes up to `n` characters to the output sequence as if by repeated calls to `sputc(c)`. The characters written are obtained from successive elements of the array whose first element is designated by `s`. Writing stops when either `n` characters have been written or a call to `sputc(c)` would return `traits::eof()`. It is unspecified whether the function calls `overflow()` when `pptr() == epptr()` becomes `true` or whether it achieves the same effects by other means.

2 *Returns:* The number of characters written.

```
int_type overflow(int_type c = traits::eof());
```

3 *Effects:* Consumes some initial subsequence of the characters of the *pending sequence*. The pending sequence is defined as the concatenation of

(3.1) — the empty sequence if `pbase()` is null, otherwise the `pptr() - pbase()` characters beginning at `pbase()`, followed by

(3.2) — the empty sequence if `traits::eq_int_type(c, traits::eof())` returns `true`, otherwise the sequence consisting of `c`.

4 *Remarks:* The member functions `sputc()` and `sputn()` call this function in case that no room can be found in the put buffer enough to accommodate the argument character sequence.

5 *Requires:* Every overriding definition of this virtual function shall obey the following constraints:

(5.1) — The effect of consuming a character on the associated output sequence is specified.[303]

(5.2) — Let `r` be the number of characters in the pending sequence not consumed. If `r` is nonzero then `pbase()` and `pptr()` shall be set so that: `pptr() - pbase() == r` and the `r` characters starting at `pbase()` are the associated output stream. In case `r` is zero (all characters of the pending sequence have been consumed) then either `pbase()` is set to `nullptr`, or `pbase()` and `pptr()` are both set to the same non-null value.

(5.3) — The function may fail if either appending some character to the associated output stream fails or if it is unable to establish `pbase()` and `pptr()` according to the above rules.

6 *Returns:* `traits::eof()` or throws an exception if the function fails.

Otherwise, returns some value other than `traits::eof()` to indicate success.[304]

7 *Default behavior:* Returns `traits::eof()`.

29.7 Formatting and manipulators [iostream.format]

29.7.1 Header <istream> synopsis [istream.syn]

```
namespace std {
  template<class charT, class traits = char_traits<charT>>
    class basic_istream;

  using istream  = basic_istream<char>;
  using wistream = basic_istream<wchar_t>;

  template<class charT, class traits = char_traits<charT>>
    class basic_iostream;

  using iostream  = basic_iostream<char>;
  using wiostream = basic_iostream<wchar_t>;

  template<class charT, class traits>
    basic_istream<charT, traits>& ws(basic_istream<charT, traits>& is);

  template<class charT, class traits, class T>
    basic_istream<charT, traits>& operator>>(basic_istream<charT, traits>&& is, T&& x);
}
```

29.7.2 Header <ostream> synopsis [ostream.syn]

```
namespace std {
  template<class charT, class traits = char_traits<charT>>
    class basic_ostream;

  using ostream  = basic_ostream<char>;
  using wostream = basic_ostream<wchar_t>;

  template<class charT, class traits>
    basic_ostream<charT, traits>& endl(basic_ostream<charT, traits>& os);
  template<class charT, class traits>
    basic_ostream<charT, traits>& ends(basic_ostream<charT, traits>& os);
  template<class charT, class traits>
    basic_ostream<charT, traits>& flush(basic_ostream<charT, traits>& os);

  template<class charT, class traits>
    basic_ostream<charT, traits>& emit_on_flush(basic_ostream<charT, traits>& os);
  template<class charT, class traits>
    basic_ostream<charT, traits>& noemit_on_flush(basic_ostream<charT, traits>& os);
  template<class charT, class traits>
    basic_ostream<charT, traits>& flush_emit(basic_ostream<charT, traits>& os);
```

303) That is, for each class derived from an instance of `basic_streambuf` in this Clause (29.8.2, 29.9.2), a specification of how consuming a character effects the associated output sequence is given. There is no requirement on a program-defined class.
304) Typically, `overflow` returns `c` to indicate success, except when `traits::eq_int_type(c, traits::eof())` returns `true`, in which case it returns `traits::not_eof(c)`.

```
  template<class charT, class traits, class T>
    basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>&& os, const T& x);
}
```

29.7.3 Header <iomanip> synopsis [iomanip.syn]

```
namespace std {
  // types T1, T2, ... are unspecified implementation types
  T1 resetiosflags(ios_base::fmtflags mask);
  T2 setiosflags  (ios_base::fmtflags mask);
  T3 setbase(int base);
  template<class charT> T4 setfill(charT c);
  T5 setprecision(int n);
  T6 setw(int n);
  template<class moneyT> T7 get_money(moneyT& mon, bool intl = false);
  template<class moneyT> T8 put_money(const moneyT& mon, bool intl = false);
  template<class charT> T9 get_time(struct tm* tmb, const charT* fmt);
  template<class charT> T10 put_time(const struct tm* tmb, const charT* fmt);

  template<class charT>
    T11 quoted(const charT* s, charT delim = charT('"'), charT escape = charT('\\'));

  template<class charT, class traits, class Allocator>
    T12 quoted(const basic_string<charT, traits, Allocator>& s,
               charT delim = charT('"'), charT escape = charT('\\'));

  template<class charT, class traits, class Allocator>
    T13 quoted(basic_string<charT, traits, Allocator>& s,
               charT delim = charT('"'), charT escape = charT('\\'));

  template<class charT, class traits>
    T14 quoted(basic_string_view<charT, traits> s,
               charT delim = charT('"'), charT escape = charT('\\'));
}
```

29.7.4 Input streams [input.streams]

1 The header <istream> defines two types and a function signature that control input from a stream buffer along with a function template that extracts from stream rvalues.

29.7.4.1 Class template basic_istream [istream]

```
namespace std {
  template<class charT, class traits = char_traits<charT>>
  class basic_istream : virtual public basic_ios<charT, traits> {
  public:
    // types (inherited from basic_ios (29.5.5))
    using char_type   = charT;
    using int_type    = typename traits::int_type;
    using pos_type    = typename traits::pos_type;
    using off_type    = typename traits::off_type;
    using traits_type = traits;

    // 29.7.4.1.1, constructor/destructor
    explicit basic_istream(basic_streambuf<charT, traits>* sb);
    virtual ~basic_istream();

    // 29.7.4.1.3, prefix/suffix
    class sentry;

    // 29.7.4.2, formatted input
    basic_istream<charT, traits>&
      operator>>(basic_istream<charT, traits>& (*pf)(basic_istream<charT, traits>&));
    basic_istream<charT, traits>&
      operator>>(basic_ios<charT, traits>& (*pf)(basic_ios<charT, traits>&));
```

```
  basic_istream<charT, traits>&
    operator>>(ios_base& (*pf)(ios_base&));

  basic_istream<charT, traits>& operator>>(bool& n);
  basic_istream<charT, traits>& operator>>(short& n);
  basic_istream<charT, traits>& operator>>(unsigned short& n);
  basic_istream<charT, traits>& operator>>(int& n);
  basic_istream<charT, traits>& operator>>(unsigned int& n);
  basic_istream<charT, traits>& operator>>(long& n);
  basic_istream<charT, traits>& operator>>(unsigned long& n);
  basic_istream<charT, traits>& operator>>(long long& n);
  basic_istream<charT, traits>& operator>>(unsigned long long& n);
  basic_istream<charT, traits>& operator>>(float& f);
  basic_istream<charT, traits>& operator>>(double& f);
  basic_istream<charT, traits>& operator>>(long double& f);

  basic_istream<charT, traits>& operator>>(void*& p);
  basic_istream<charT, traits>& operator>>(basic_streambuf<char_type, traits>* sb);

  // 29.7.4.3, unformatted input
  streamsize gcount() const;
  int_type get();
  basic_istream<charT, traits>& get(char_type& c);
  basic_istream<charT, traits>& get(char_type* s, streamsize n);
  basic_istream<charT, traits>& get(char_type* s, streamsize n, char_type delim);
  basic_istream<charT, traits>& get(basic_streambuf<char_type, traits>& sb);
  basic_istream<charT, traits>& get(basic_streambuf<char_type, traits>& sb, char_type delim);

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

  basic_istream<charT, traits>& ignore(streamsize n = 1, int_type delim = traits::eof());
  int_type                      peek();
  basic_istream<charT, traits>& read    (char_type* s, streamsize n);
  streamsize                    readsome(char_type* s, streamsize n);

  basic_istream<charT, traits>& putback(char_type c);
  basic_istream<charT, traits>& unget();
  int sync();

  pos_type tellg();
  basic_istream<charT, traits>& seekg(pos_type);
  basic_istream<charT, traits>& seekg(off_type, ios_base::seekdir);

protected:
  // 29.7.4.1.1, copy/move constructor
  basic_istream(const basic_istream& rhs) = delete;
  basic_istream(basic_istream&& rhs);

  // 29.7.4.1.2, assign and swap
  basic_istream& operator=(const basic_istream& rhs) = delete;
  basic_istream& operator=(basic_istream&& rhs);
  void swap(basic_istream& rhs);
};

// 29.7.4.2.3, character extraction templates
template<class charT, class traits>
  basic_istream<charT, traits>& operator>>(basic_istream<charT, traits>&, charT&);
template<class traits>
  basic_istream<char, traits>& operator>>(basic_istream<char, traits>&, unsigned char&);
template<class traits>
  basic_istream<char, traits>& operator>>(basic_istream<char, traits>&, signed char&);
```

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

1 The class template `basic_istream` defines a number of member function signatures that assist in reading and interpreting input from sequences controlled by a stream buffer.

2 Two groups of member function signatures share common properties: the *formatted input functions* (or *extractors*) and the *unformatted input functions.* Both groups of input functions are described as if they obtain (or *extract*) input *characters* by calling `rdbuf()->sbumpc()` or `rdbuf()->sgetc()`. They may use other public members of `istream`.

3 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` (29.5.5.4), before returning.

4 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.

29.7.4.1.1 Constructors [istream.cons]

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

1 *Effects:* Constructs an object of class `basic_istream`, initializing the base class subobject with `basic_ios::init(sb)` (29.5.5.2).

2 *Ensures:* `gcount() == 0`.

```
basic_istream(basic_istream&& rhs);
```

3 *Effects:* Move constructs from the rvalue `rhs`. This is accomplished by default constructing the base class, copying the `gcount()` from `rhs`, calling `basic_ios<charT, traits>::move(rhs)` to initialize the base class, and setting the `gcount()` for `rhs` to 0.

```
virtual ~basic_istream();
```

4 *Effects:* Destroys an object of class `basic_istream`.

5 *Remarks:* Does not perform any operations of `rdbuf()`.

29.7.4.1.2 Assignment and swap [istream.assign]

```
basic_istream& operator=(basic_istream&& rhs);
```

1 *Effects:* As if by `swap(rhs)`.

2 *Returns:* `*this`.

```
void swap(basic_istream& rhs);
```

3 *Effects:* Calls `basic_ios<charT, traits>::swap(rhs)`. Exchanges the values returned by `gcount()` and `rhs.gcount()`.

29.7.4.1.3 Class `basic_istream::sentry` [istream.sentry]

```
namespace std {
  template<class charT, class traits = char_traits<charT>>
  class basic_istream<charT, traits>::sentry {
    using traits_type = traits;
    bool ok_; // exposition only
  public:
    explicit sentry(basic_istream<charT, traits>& is, bool noskipws = false);
    ~sentry();
    explicit operator bool() const { return ok_; }
    sentry(const sentry&) = delete;
```

```
    sentry& operator=(const sentry&) = delete;
  };
}
```

1 The class `sentry` defines a class that is responsible for doing exception safe prefix and suffix operations.

```
explicit sentry(basic_istream<charT, traits>& is, bool noskipws = false);
```

2 *Effects:* If `is.good()` is `false`, calls `is.setstate(failbit)`. Otherwise, prepares for formatted or unformatted input. First, if `is.tie()` is not a null pointer, the function calls `is.tie()->flush()` to synchronize the output sequence with any associated external C stream. Except that this call can be suppressed if the put area of `is.tie()` is empty. Further an implementation is allowed to defer the call to `flush` until a call of `is.rdbuf()->underflow()` occurs. If no such call occurs before the `sentry` object is destroyed, the call to `flush` may be eliminated entirely.[305] If `noskipws` is zero and `is.flags() & ios_base::skipws` is nonzero, the function extracts and discards each character as long as the next available input character `c` is a whitespace character. If `is.rdbuf()->sbumpc()` or `is.rdbuf()->sgetc()` returns `traits::eof()`, the function calls `setstate(failbit | eofbit)` (which may throw `ios_base::failure`).

3 *Remarks:* The constructor

```
explicit sentry(basic_istream<charT, traits>& is, bool noskipws = false)
```

uses the currently imbued locale in `is`, to determine whether the next input character is whitespace or not.

4 To decide if the character `c` is a whitespace character, the constructor performs as if it executes the following code fragment:

```
const ctype<charT>& ctype = use_facet<ctype<charT>>(is.getloc());
if (ctype.is(ctype.space, c) != 0)
  // c is a whitespace character.
```

5 If, after any preparation is completed, `is.good()` is `true`, `ok_ != false` otherwise, `ok_ == false`. During preparation, the constructor may call `setstate(failbit)` (which may throw `ios_base::failure` (29.5.5.4))[306]

```
~sentry();
```

6 *Effects:* None.

```
explicit operator bool() const;
```

7 *Effects:* Returns `ok_`.

29.7.4.2 Formatted input functions [istream.formatted]

29.7.4.2.1 Common requirements [istream.formatted.reqmts]

1 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 turned on[307] 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`.

29.7.4.2.2 Arithmetic extractors [istream.formatted.arithmetic]

```
operator>>(unsigned short& val);
operator>>(unsigned int& val);
operator>>(long& val);
operator>>(unsigned long& val);
operator>>(long long& val);
operator>>(unsigned long long& val);
operator>>(float& val);
operator>>(double& val);
```

305) This will be possible only in functions that are part of the library. The semantics of the constructor used in user code is as specified.
306) The `sentry` constructor and destructor can also perform additional implementation-dependent operations.
307) This is done without causing an `ios::failure` to be thrown.

```
operator>>(long double& val);
operator>>(bool& val);
operator>>(void*& val);
```

1 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 29.7.4.2.1). After a sentry object is constructed, the conversion occurs as if performed by the following code fragment:

```
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);
```

In the above fragment, `loc` stands for the private member of the `basic_ios` class. [*Note*: The first argument provides an object of the `istreambuf_iterator` class which is an iterator pointed to an input stream. It bypasses istreams and uses streambufs directly. — *end note*] Class `locale` relies on this type as its interface to `istream`, so that it does not need to depend directly on `istream`.

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

2 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, lval);
if (lval < numeric_limits<short>::min()) {
  err |= ios_base::failbit;
  val = numeric_limits<short>::min();
} else if (numeric_limits<short>::max() < lval) {
  err |= ios_base::failbit;
  val = numeric_limits<short>::max();
}  else
  val = static_cast<short>(lval);
setstate(err);
```

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

3 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, lval);
if (lval < numeric_limits<int>::min()) {
  err |= ios_base::failbit;
  val = numeric_limits<int>::min();
} else if (numeric_limits<int>::max() < lval) {
  err |= ios_base::failbit;
  val = numeric_limits<int>::max();
}  else
  val = static_cast<int>(lval);
setstate(err);
```

29.7.4.2.3 `basic_istream::operator>>` [istream.extractors]

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

1 *Effects:* None. This extractor does not behave as a formatted input function (as described in 29.7.4.2.1).

2 *Returns:* `pf(*this)`.[308]

308) See, for example, the function signature `ws(basic_istream&)` (29.7.4.4).

```
basic_istream<charT, traits>&
  operator>>(basic_ios<charT, traits>& (*pf)(basic_ios<charT, traits>&));
```

3 *Effects:* Calls `pf(*this)`. This extractor does not behave as a formatted input function (as described in 29.7.4.2.1).

4 *Returns:* `*this`.

```
basic_istream<charT, traits>& operator>>(ios_base& (*pf)(ios_base&));
```

5 *Effects:* Calls `pf(*this)`.[309] This extractor does not behave as a formatted input function (as described in 29.7.4.2.1).

6 *Returns:* `*this`.

```
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]);
```

7 *Effects:* Behaves like a formatted input member (as described in 29.7.4.2.1) of `in`. After a `sentry` object is constructed, `operator>>` extracts characters and stores them into `s`. If `width()` is greater than zero, `n` is `min(size_t(width()), N)`. Otherwise `n` is `N`. `n` is the maximum number of characters stored.

8 Characters are extracted and stored until any of the following occurs:

(8.1) — `n-1` characters are stored;

(8.2) — end of file occurs on the input sequence;

(8.3) — letting `ct` be `use_facet<ctype<charT>>(in.getloc())`, `ct.is(ct.space, c)` is `true`.

9 `operator>>` then stores a null byte (`charT()`) in the next position, which may be the first position if no characters were extracted. `operator>>` then calls `width(0)`.

10 If the function extracted no characters, it calls `setstate(failbit)`, which may throw `ios_base::failure` (29.5.5.4).

11 *Returns:* `in`.

```
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);
```

12 *Effects:* Behaves like a formatted input member (as described in 29.7.4.2.1) of `in`. After a `sentry` object is constructed a character is extracted from `in`, if one is available, and stored in `c`. Otherwise, the function calls `in.setstate(failbit)`.

13 *Returns:* `in`.

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

14 *Effects:* Behaves as an unformatted input function (29.7.4.3). If `sb` is null, calls `setstate(failbit)`, which may throw `ios_base::failure` (29.5.5.4). After a sentry object is constructed, extracts characters from `*this` and inserts them in the output sequence controlled by `sb`. Characters are extracted and inserted until any of the following occurs:

(14.1) — end-of-file occurs on the input sequence;

(14.2) — inserting in the output sequence fails (in which case the character to be inserted is not extracted);

(14.3) — an exception occurs (in which case the exception is caught).

15 If the function inserts no characters, it calls `setstate(failbit)`, which may throw `ios_base::failure` (29.5.5.4). If it inserted no characters because it caught an exception thrown while extracting

309) See, for example, the function signature `dec(ios_base&)` (29.5.6.3).

characters from `*this` and `failbit` is on in `exceptions()` (29.5.5.4), then the caught exception is rethrown.

16 *Returns:* `*this`.

29.7.4.3 Unformatted input functions [istream.unformatted]

1 Each unformatted input function begins execution by constructing 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 `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 exception is thrown during input then `ios::badbit` is turned on[310] 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.

```
streamsize gcount() const;
```

2 *Effects:* None. This member function does not behave as an unformatted input function (as described above).

3 *Returns:* The number of characters extracted by the last unformatted input member function called for the object.

```
int_type get();
```

4 *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` (29.5.5.4),

5 *Returns:* `c` if available, otherwise `traits::eof()`.

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

6 *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`.[311] Otherwise, the function calls `setstate(failbit)` (which may throw `ios_base::failure` (29.5.5.4)).

7 *Returns:* `*this`.

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

8 *Effects:* Behaves as an unformatted input function (as described above). After constructing a sentry object, extracts characters and stores them into successive locations of an array whose first element is designated by `s`.[312] Characters are extracted and stored until any of the following occurs:

(8.1) — `n` is less than one or `n - 1` characters are stored;

(8.2) — end-of-file occurs on the input sequence (in which case the function calls `setstate(eofbit)`);

(8.3) — `traits::eq(c, delim)` for the next available input character `c` (in which case `c` is not extracted).

9 If the function stores no characters, it calls `setstate(failbit)` (which may throw `ios_base::failure` (29.5.5.4)). In any case, if `n` is greater than zero it then stores a null character into the next successive location of the array.

10 *Returns:* `*this`.

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

11 *Effects:* Calls `get(s, n, widen('\n'))`.

12 *Returns:* Value returned by the call.

310) This is done without causing an `ios::failure` to be thrown.
311) Note that this function is not overloaded on types `signed char` and `unsigned char`.
312) Note that this function is not overloaded on types `signed char` and `unsigned char`.

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

13 *Effects:* Behaves as an unformatted input function (as described above). After constructing a sentry object, extracts characters and inserts them in the output sequence controlled by `sb`. Characters are extracted and inserted until any of the following occurs:

(13.1) — end-of-file occurs on the input sequence;

(13.2) — inserting in the output sequence fails (in which case the character to be inserted is not extracted);

(13.3) — `traits::eq(c, delim)` for the next available input character `c` (in which case `c` is not extracted);

(13.4) — an exception occurs (in which case, the exception is caught but not rethrown).

14 If the function inserts no characters, it calls `setstate(failbit)`, which may throw `ios_base::failure` (29.5.5.4).

15 *Returns:* `*this`.

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

16 *Effects:* Calls `get(sb, widen('\n'))`.

17 *Returns:* Value returned by the call.

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

18 *Effects:* Behaves as an unformatted input function (as described above). After constructing a sentry object, extracts characters and stores them into successive locations of an array whose first element is designated by `s`.[313] Characters are extracted and stored until one of the following occurs:

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);[314]
3. `n` is less than one or `n - 1` characters are stored (in which case the function calls `setstate(failbit)`).

19 These conditions are tested in the order shown.[315]

20 If the function extracts no characters, it calls `setstate(failbit)` (which may throw `ios_base::failure` (29.5.5.4)).[316]

21 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.

22 *Returns:* `*this`.

23 [*Example*:

```
#include <iostream>

int main() {
  using namespace std;
  const int line_buffer_size = 100;

  char buffer[line_buffer_size];
  int line_number = 0;
  while (cin.getline(buffer, line_buffer_size, '\n') || cin.gcount()) {
    int count = cin.gcount();
    if (cin.eof())
      cout << "Partial final line";   // cin.fail() is false
    else if (cin.fail()) {
      cout << "Partial long line";
      cin.clear(cin.rdstate() & ~ios_base::failbit);
    } else {
      count--;                        // Don't include newline in count
```

313) Note that this function is not overloaded on types `signed char` and `unsigned char`.

314) Since the final input character is "extracted", it is counted in the `gcount()`, even though it is not stored.

315) This allows an input line which exactly fills the buffer, without setting `failbit`. This is different behavior than the historical AT&T implementation.

316) This implies an empty input line will not cause `failbit` to be set.

```
        cout << "Line " << ++line_number;
      }
      cout << " (" << count << " chars): " << buffer << endl;
    }
  }
```

— *end example*]

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

24 *Returns:* `getline(s, n, widen('\n'))`

`basic_istream<charT, traits>& ignore(streamsize n = 1, int_type delim = traits::eof());`

25 *Effects:* Behaves as an unformatted input function (as described above). After constructing a sentry object, extracts characters and discards them. Characters are extracted until any of the following occurs:

(25.1) — `n != numeric_limits<streamsize>::max()` (**??**) and `n` characters have been extracted so far

(25.2) — end-of-file occurs on the input sequence (in which case the function calls `setstate(eofbit)`, which may throw `ios_base::failure` (29.5.5.4));

(25.3) — `traits::eq_int_type(traits::to_int_type(c), delim)` for the next available input character `c` (in which case `c` is extracted).

26 *Remarks:* The last condition will never occur if `traits::eq_int_type(delim, traits::eof())`.

27 *Returns:* `*this`.

`int_type peek();`

28 *Effects:* Behaves as an unformatted input function (as described above). After constructing a sentry object, reads but does not extract the current input character.

29 *Returns:* `traits::eof()` if `good()` is `false`. Otherwise, returns `rdbuf()->sgetc()`.

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

30 *Effects:* Behaves as an unformatted input function (as described above). After constructing a sentry object, if `!good()` calls `setstate(failbit)` which may throw an exception, and return. Otherwise extracts characters and stores them into successive locations of an array whose first element is designated by `s`.[317] Characters are extracted and stored until either of the following occurs:

(30.1) — `n` characters are stored;

(30.2) — end-of-file occurs on the input sequence (in which case the function calls `setstate(failbit | eofbit)`, which may throw `ios_base::failure` (29.5.5.4)).

31 *Returns:* `*this`.

`streamsize readsome(char_type* s, streamsize n);`

32 *Effects:* Behaves as an unformatted input function (as described above). After constructing a sentry object, if `!good()` calls `setstate(failbit)` which may throw an exception, and return. Otherwise extracts characters and stores them into successive locations of an array whose first element is designated by `s`. If `rdbuf()->in_avail() == -1`, calls `setstate(eofbit)` (which may throw `ios_base::failure` (29.5.5.4)), and extracts no characters;

(32.1) — If `rdbuf()->in_avail() == 0`, extracts no characters

(32.2) — If `rdbuf()->in_avail() > 0`, extracts `min(rdbuf()->in_avail(), n))`.

33 *Returns:* The number of characters extracted.

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

34 *Effects:* Behaves as an unformatted input function (as described above), except that the function first clears `eofbit`. After constructing a sentry object, if `!good()` calls `setstate(failbit)` which may throw an exception, and return. If `rdbuf()` is not null, calls `rdbuf()->sputbackc(c)`. If `rdbuf()`

317) Note that this function is not overloaded on types `signed char` and `unsigned char`.

is null, or if `sputbackc` returns `traits::eof()`, calls `setstate(badbit)` (which may throw `ios_base::failure` (29.5.5.4)). [*Note*: This function extracts no characters, so the value returned by the next call to `gcount()` is 0. — *end note*]

35 *Returns:* `*this`.

```
basic_istream<charT, traits>& unget();
```

36 *Effects:* Behaves as an unformatted input function (as described above), except that the function first clears `eofbit`. After constructing a sentry object, if `!good()` calls `setstate(failbit)` which may throw an exception, and return. If `rdbuf()` is not null, calls `rdbuf()->sungetc()`. If `rdbuf()` is null, or if `sungetc` returns `traits::eof()`, calls `setstate(badbit)` (which may throw `ios_base::failure` (29.5.5.4)). [*Note*: This function extracts no characters, so the value returned by the next call to `gcount()` is 0. — *end note*]

37 *Returns:* `*this`.

```
int sync();
```

38 *Effects:* Behaves as an unformatted input function (as described above), except that it does not count the number of characters extracted and does not affect the value returned by subsequent calls to `gcount()`. After constructing a sentry object, if `rdbuf()` is a null pointer, returns `-1`. Otherwise, calls `rdbuf()->pubsync()` and, if that function returns `-1` calls `setstate(badbit)` (which may throw `ios_base::failure` (29.5.5.4), and returns `-1`. Otherwise, returns zero.

```
pos_type tellg();
```

39 *Effects:* Behaves as an unformatted input function (as described above), except that it does not count the number of characters extracted and does not affect the value returned by subsequent calls to `gcount()`.

40 *Returns:* After constructing a sentry object, if `fail() != false`, returns `pos_type(-1)` to indicate failure. Otherwise, returns `rdbuf()->pubseekoff(0, cur, in)`.

```
basic_istream<charT, traits>& seekg(pos_type pos);
```

41 *Effects:* Behaves as an unformatted input function (as described above), except that the function first clears `eofbit`, it does not count the number of characters extracted, and it does not affect the value returned by subsequent calls to `gcount()`. After constructing a sentry object, if `fail() != true`, executes `rdbuf()->pubseekpos(pos, ios_base::in)`. In case of failure, the function calls `setstate(failbit)` (which may throw `ios_base::failure`).

42 *Returns:* `*this`.

```
basic_istream<charT, traits>& seekg(off_type off, ios_base::seekdir dir);
```

43 *Effects:* Behaves as an unformatted input function (as described above), except that the function first clears `eofbit`, does not count the number of characters extracted, and does not affect the value returned by subsequent calls to `gcount()`. After constructing a sentry object, if `fail() != true`, executes `rdbuf()->pubseekoff(off, dir, ios_base::in)`. In case of failure, the function calls `setstate(failbit)` (which may throw `ios_base::failure`).

44 *Returns:* `*this`.

29.7.4.4 Standard `basic_istream` manipulators [istream.manip]

1 Each instantiation of the function template specified in this subclause is a designated addressable function (**??**).

```
template<class charT, class traits>
  basic_istream<charT, traits>& ws(basic_istream<charT, traits>& is);
```

2 *Effects:* Behaves as an unformatted input function (29.7.4.3), except that it does not count the number of characters extracted and does not affect the value returned by subsequent calls to `is.gcount()`. After constructing a sentry object extracts characters as long as the next available character `c` is whitespace or until there are no more characters in the sequence. Whitespace characters are distinguished with the same criterion as used by `sentry::sentry` (29.7.4.1.3). If `ws` stops extracting characters because there are no more available it sets `eofbit`, but not `failbit`.

3 *Returns:* `is`.

29.7.4.5 Rvalue stream extraction [istream.rvalue]

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

1 *Constraints:* The expression `is >> std::forward<T>(x)` is well-formed.

2 *Effects:* Equivalent to:

```
is >> std::forward<T>(x);
return is;
```

3 *Remarks:* This function shall not participate in overload resolution unless the expression `is >> std::forward<T>(x)` is well-formed.

29.7.4.6 Class template basic_iostream [iostreamclass]

```
namespace std {
  template<class charT, class traits = char_traits<charT>>
  class basic_iostream
    : public basic_istream<charT, traits>,
      public basic_ostream<charT, traits> {
  public:
    using char_type   = charT;
    using int_type    = typename traits::int_type;
    using pos_type    = typename traits::pos_type;
    using off_type    = typename traits::off_type;
    using traits_type = traits;

    // 29.7.4.6.1, constructor
    explicit basic_iostream(basic_streambuf<charT, traits>* sb);

    // 29.7.4.6.2, destructor
    virtual ~basic_iostream();

  protected:
    // 29.7.4.6.1, constructor
    basic_iostream(const basic_iostream& rhs) = delete;
    basic_iostream(basic_iostream&& rhs);

    // 29.7.4.6.3, assign and swap
    basic_iostream& operator=(const basic_iostream& rhs) = delete;
    basic_iostream& operator=(basic_iostream&& rhs);
    void swap(basic_iostream& rhs);
  };
}
```

1 The class template `basic_iostream` inherits a number of functions that allow reading input and writing output to sequences controlled by a stream buffer.

29.7.4.6.1 Constructors [iostream.cons]

```
explicit basic_iostream(basic_streambuf<charT, traits>* sb);
```

1 *Effects:* Constructs an object of class `basic_iostream`, initializing the base class subobjects with `basic_istream<charT, traits>(sb)` (29.7.4.1) and `basic_ostream<charT, traits>(sb)` (29.7.5.1).

2 *Ensures:* `rdbuf() == sb` and `gcount() == 0`.

```
basic_iostream(basic_iostream&& rhs);
```

3 *Effects:* Move constructs from the rvalue `rhs` by constructing the `basic_istream` base class with `move(rhs)`.

29.7.4.6.2 Destructor [iostream.dest]

```
virtual ~basic_iostream();
```

1 *Effects:* Destroys an object of class `basic_iostream`.

2 *Remarks:* Does not perform any operations on `rdbuf()`.

29.7.4.6.3 Assignment and swap [iostream.assign]

```
basic_iostream& operator=(basic_iostream&& rhs);
```

1 *Effects:* As if by `swap(rhs)`.

```
void swap(basic_iostream& rhs);
```

2 *Effects:* Calls `basic_istream<charT, traits>::swap(rhs)`.

29.7.5 Output streams [output.streams]

1 The header `<ostream>` defines a type and several function signatures that control output to a stream buffer along with a function template that inserts into stream rvalues.

29.7.5.1 Class template `basic_ostream` [ostream]

```
namespace std {
  template<class charT, class traits = char_traits<charT>>
  class basic_ostream : virtual public basic_ios<charT, traits> {
  public:
    // types (inherited from basic_ios (29.5.5))
    using char_type   = charT;
    using int_type    = typename traits::int_type;
    using pos_type    = typename traits::pos_type;
    using off_type    = typename traits::off_type;
    using traits_type = traits;

    // 29.7.5.1.1, constructor/destructor
    explicit basic_ostream(basic_streambuf<char_type, traits>* sb);
    virtual ~basic_ostream();

    // 29.7.5.1.3, prefix/suffix
    class sentry;

    // 29.7.5.2, formatted output
    basic_ostream<charT, traits>&
      operator<<(basic_ostream<charT, traits>& (*pf)(basic_ostream<charT, traits>&));
    basic_ostream<charT, traits>&
      operator<<(basic_ios<charT, traits>& (*pf)(basic_ios<charT, traits>&));
    basic_ostream<charT, traits>&
      operator<<(ios_base& (*pf)(ios_base&));

    basic_ostream<charT, traits>& operator<<(bool n);
    basic_ostream<charT, traits>& operator<<(short n);
    basic_ostream<charT, traits>& operator<<(unsigned short n);
    basic_ostream<charT, traits>& operator<<(int n);
    basic_ostream<charT, traits>& operator<<(unsigned int n);
    basic_ostream<charT, traits>& operator<<(long n);
    basic_ostream<charT, traits>& operator<<(unsigned long n);
    basic_ostream<charT, traits>& operator<<(long long n);
    basic_ostream<charT, traits>& operator<<(unsigned long long n);
    basic_ostream<charT, traits>& operator<<(float f);
    basic_ostream<charT, traits>& operator<<(double f);
    basic_ostream<charT, traits>& operator<<(long double f);

    basic_ostream<charT, traits>& operator<<(const void* p);
    basic_ostream<charT, traits>& operator<<(nullptr_t);
    basic_ostream<charT, traits>& operator<<(basic_streambuf<char_type, traits>* sb);

    // 29.7.5.3, unformatted output
    basic_ostream<charT, traits>& put(char_type c);
    basic_ostream<charT, traits>& write(const char_type* s, streamsize n);

    basic_ostream<charT, traits>& flush();
```

```
    // 29.7.5.1.4, seeks
    pos_type tellp();
    basic_ostream<charT, traits>& seekp(pos_type);
    basic_ostream<charT, traits>& seekp(off_type, ios_base::seekdir);

  protected:
    // 29.7.5.1.1, copy/move constructor
    basic_ostream(const basic_ostream& rhs) = delete;
    basic_ostream(basic_ostream&& rhs);

    // 29.7.5.1.2, assign and swap
    basic_ostream& operator=(const basic_ostream& rhs) = delete;
    basic_ostream& operator=(basic_ostream&& rhs);
    void swap(basic_ostream& rhs);
  };

  // 29.7.5.2.4, character inserters
  template<class charT, class traits>
    basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>&, charT);
  template<class charT, class traits>
    basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>&, char);
  template<class traits>
    basic_ostream<char, traits>& operator<<(basic_ostream<char, traits>&, char);

  template<class traits>
    basic_ostream<char, traits>& operator<<(basic_ostream<char, traits>&, signed char);
  template<class traits>
    basic_ostream<char, traits>& operator<<(basic_ostream<char, traits>&, unsigned char);

  template<class charT, class traits>
    basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>&, const charT*);
  template<class charT, class traits>
    basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>&, const char*);
  template<class traits>
    basic_ostream<char, traits>& operator<<(basic_ostream<char, traits>&, const char*);

  template<class traits>
    basic_ostream<char, traits>& operator<<(basic_ostream<char, traits>&, const signed char*);
  template<class traits>
    basic_ostream<char, traits>& operator<<(basic_ostream<char, traits>&, const unsigned char*);
}
```

1 The class template `basic_ostream` defines a number of member function signatures that assist in formatting and writing output to output sequences controlled by a stream buffer.

2 Two groups of member function signatures share common properties: the *formatted output functions* (or *inserters*) and the *unformatted output functions.* Both groups of output functions generate (or *insert*) output *characters* by actions equivalent to calling `rdbuf()->sputc(int_type)`. They may use other public members of `basic_ostream` except that they shall not invoke any virtual members of `rdbuf()` except `overflow()`, `xsputn()`, and `sync()`.

3 If one of these called functions throws an exception, then unless explicitly noted otherwise the output function sets `badbit` in error state. If `badbit` is on in `exceptions()`, the output function rethrows the exception without completing its actions, otherwise it does not throw anything and treat as an error.

29.7.5.1.1 Constructors [ostream.cons]

```
explicit basic_ostream(basic_streambuf<charT, traits>* sb);
```

1 *Effects:* Constructs an object of class `basic_ostream`, initializing the base class subobject with `basic_ios<charT, traits>::init(sb)` (29.5.5.2).

2 *Ensures:* `rdbuf() == sb`.

```
basic_ostream(basic_ostream&& rhs);
```

3 *Effects:* Move constructs from the rvalue `rhs`. This is accomplished by default constructing the base class and calling `basic_ios<charT, traits>::move(rhs)` to initialize the base class.

```
virtual ~basic_ostream();
```

4 *Effects:* Destroys an object of class `basic_ostream`.

5 *Remarks:* Does not perform any operations on `rdbuf()`.

29.7.5.1.2 Assignment and swap [ostream.assign]

```
basic_ostream& operator=(basic_ostream&& rhs);
```

1 *Effects:* As if by `swap(rhs)`.

2 *Returns:* `*this`.

```
void swap(basic_ostream& rhs);
```

3 *Effects:* Calls `basic_ios<charT, traits>::swap(rhs)`.

29.7.5.1.3 Class `basic_ostream::sentry` [ostream.sentry]

```
namespace std {
  template<class charT, class traits = char_traits<charT>>
  class basic_ostream<charT, traits>::sentry {
    bool ok_; // exposition only
  public:
    explicit sentry(basic_ostream<charT, traits>& os);
    ~sentry();
    explicit operator bool() const { return ok_; }

    sentry(const sentry&) = delete;
    sentry& operator=(const sentry&) = delete;
  };
}
```

1 The class `sentry` defines a class that is responsible for doing exception safe prefix and suffix operations.

```
explicit sentry(basic_ostream<charT, traits>& os);
```

2 If `os.good()` is nonzero, prepares for formatted or unformatted output. If `os.tie()` is not a null pointer, calls `os.tie()->flush()`.[318]

3 If, after any preparation is completed, `os.good()` is `true`, `ok_ == true` otherwise, `ok_ == false`. During preparation, the constructor may call `setstate(failbit)` (which may throw `ios_base::failure` (29.5.5.4))[319]

```
~sentry();
```

4 If `(os.flags() & ios_base::unitbuf) && !uncaught_exceptions() && os.good()` is `true`, calls `os.rdbuf()->pubsync()`. If that function returns -1, sets `badbit` in `os.rdstate()` without propagating an exception.

```
explicit operator bool() const;
```

5 *Effects:* Returns `ok_`.

29.7.5.1.4 Seek members [ostream.seeks]

1 Each seek member function begins execution by constructing an object of class `sentry`. It returns by destroying the `sentry` object.

```
pos_type tellp();
```

2 *Returns:* If `fail() != false`, returns `pos_type(-1)` to indicate failure. Otherwise, returns `rdbuf()->pubseekoff(0, cur, out)`.

318) The call `os.tie()->flush()` does not necessarily occur if the function can determine that no synchronization is necessary.
319) The `sentry` constructor and destructor can also perform additional implementation-dependent operations.

```
basic_ostream<charT, traits>& seekp(pos_type pos);
```

3 *Effects:* If `fail() != true`, executes `rdbuf()->pubseekpos(pos, ios_base::out)`. In case of failure, the function calls `setstate(failbit)` (which may throw `ios_base::failure`).

4 *Returns:* `*this`.

```
basic_ostream<charT, traits>& seekp(off_type off, ios_base::seekdir dir);
```

5 *Effects:* If `fail() != true`, executes `rdbuf()->pubseekoff(off, dir, ios_base::out)`. In case of failure, the function calls `setstate(failbit)` (which may throw `ios_base::failure`).

6 *Returns:* `*this`.

29.7.5.2 Formatted output functions [ostream.formatted]

29.7.5.2.1 Common requirements [ostream.formatted.reqmts]

1 Each formatted output function begins execution by constructing an object of class `sentry`. If this object returns `true` when converted to a value of type `bool`, the function endeavors to generate the requested output. If the generation fails, then the formatted output function does `setstate(ios_base::failbit)`, which might throw an exception. If an exception is thrown during output, then `ios::badbit` is turned on[320] in `*this`'s error state. If `(exceptions()&badbit) != 0` then the exception is rethrown. Whether or not an exception is thrown, the `sentry` object is destroyed before leaving the formatted output function. If no exception is thrown, the result of the formatted output function is `*this`.

2 The descriptions of the individual formatted output functions describe how they perform output and do not mention the `sentry` object.

3 If a formatted output function of a stream `os` determines padding, it does so as follows. Given a `charT` character sequence `seq` where `charT` is the character type of the stream, if the length of `seq` is less than `os.width()`, then enough copies of `os.fill()` are added to this sequence as necessary to pad to a width of `os.width()` characters. If `(os.flags() & ios_base::adjustfield) == ios_base::left` is `true`, the fill characters are placed after the character sequence; otherwise, they are placed before the character sequence.

29.7.5.2.2 Arithmetic inserters [ostream.inserters.arithmetic]

```
operator<<(bool val);
operator<<(short val);
operator<<(unsigned short val);
operator<<(int val);
operator<<(unsigned int val);
operator<<(long val);
operator<<(unsigned long val);
operator<<(long long val);
operator<<(unsigned long long val);
operator<<(float val);
operator<<(double val);
operator<<(long double val);
operator<<(const void* val);
```

1 *Effects:* The classes `num_get<>` and `num_put<>` handle locale-dependent numeric formatting and parsing. These inserter functions use the imbued `locale` value to perform numeric formatting. When `val` is of type `bool`, `long`, `unsigned long`, `long long`, `unsigned long long`, `double`, `long double`, or `const void*`, the formatting conversion occurs as if it performed the following code fragment:

```
bool failed = use_facet<
  num_put<charT, ostreambuf_iterator<charT, traits>>
    >(getloc()).put(*this, *this, fill(), val).failed();
```

When `val` is of type `short` the formatting conversion occurs as if it performed the following code fragment:

```
ios_base::fmtflags baseflags = ios_base::flags() & ios_base::basefield;
```

320) without causing an `ios::failure` to be thrown.

```
bool failed = use_facet<
  num_put<charT, ostreambuf_iterator<charT, traits>>
    >(getloc()).put(*this, *this, fill(),
    baseflags == ios_base::oct || baseflags == ios_base::hex
      ? static_cast<long>(static_cast<unsigned short>(val))
      : static_cast<long>(val)).failed();
```

When `val` is of type `int` the formatting conversion occurs as if it performed the following code fragment:

```
ios_base::fmtflags baseflags = ios_base::flags() & ios_base::basefield;
bool failed = use_facet<
  num_put<charT, ostreambuf_iterator<charT, traits>>
    >(getloc()).put(*this, *this, fill(),
    baseflags == ios_base::oct || baseflags == ios_base::hex
      ? static_cast<long>(static_cast<unsigned int>(val))
      : static_cast<long>(val)).failed();
```

When `val` is of type `unsigned short` or `unsigned int` the formatting conversion occurs as if it performed the following code fragment:

```
bool failed = use_facet<
  num_put<charT, ostreambuf_iterator<charT, traits>>
    >(getloc()).put(*this, *this, fill(),
      static_cast<unsigned long>(val)).failed();
```

When `val` is of type `float` the formatting conversion occurs as if it performed the following code fragment:

```
bool failed = use_facet<
  num_put<charT, ostreambuf_iterator<charT, traits>>
    >(getloc()).put(*this, *this, fill(),
      static_cast<double>(val)).failed();
```

2 The first argument provides an object of the `ostreambuf_iterator<>` class which is an iterator for class `basic_ostream<>`. It bypasses `ostream`s and uses `streambuf`s directly. Class `locale` relies on these types as its interface to iostreams, since for flexibility it has been abstracted away from direct dependence on `ostream`. The second parameter is a reference to the base class subobject of type `ios_base`. It provides formatting specifications such as field width, and a locale from which to obtain other facets. If `failed` is `true` then does `setstate(badbit)`, which may throw an exception, and returns.

3 *Returns:* `*this`.

29.7.5.2.3 `basic_ostream::operator<<` [ostream.inserters]

```
basic_ostream<charT, traits>&
  operator<<(basic_ostream<charT, traits>& (*pf)(basic_ostream<charT, traits>&));
```

1 *Effects:* None. Does not behave as a formatted output function (as described in 29.7.5.2.1).

2 *Returns:* `pf(*this)`.[321]

```
basic_ostream<charT, traits>&
  operator<<(basic_ios<charT, traits>& (*pf)(basic_ios<charT, traits>&));
```

3 *Effects:* Calls `pf(*this)`. This inserter does not behave as a formatted output function (as described in 29.7.5.2.1).

4 *Returns:* `*this`.[322]

```
basic_ostream<charT, traits>& operator<<(ios_base& (*pf)(ios_base&));
```

5 *Effects:* Calls `pf(*this)`. This inserter does not behave as a formatted output function (as described in 29.7.5.2.1).

6 *Returns:* `*this`.

321) See, for example, the function signature `endl(basic_ostream&)` (29.7.5.4).
322) See, for example, the function signature `dec(ios_base&)` (29.5.6.3).

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

7 *Effects:* Behaves as an unformatted output function (29.7.5.3). After the sentry object is constructed, if `sb` is null calls `setstate(badbit)` (which may throw `ios_base::failure`).

8 Gets characters from `sb` and inserts them in `*this`. Characters are read from `sb` and inserted until any of the following occurs:

(8.1) — end-of-file occurs on the input sequence;

(8.2) — inserting in the output sequence fails (in which case the character to be inserted is not extracted);

(8.3) — an exception occurs while getting a character from `sb`.

9 If the function inserts no characters, it calls `setstate(failbit)` (which may throw `ios_base::failure` (29.5.5.4)). If an exception was thrown while extracting a character, the function sets `failbit` in error state, and if `failbit` is on in `exceptions()` the caught exception is rethrown.

10 *Returns:* `*this`.

```
basic_ostream<charT, traits>& operator<<(nullptr_t);
```

11 *Effects:* Equivalent to:

```
return *this << s;
```

where `s` is an implementation-defined NTCTS (**??**).

29.7.5.2.4 Character inserter function templates [ostream.inserters.character]

```
template<class charT, class traits>
  basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>& out, charT c);
template<class charT, class traits>
  basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>& out, char c);
// specialization
template<class traits>
  basic_ostream<char, traits>& operator<<(basic_ostream<char, traits>& out, char c);
// signed and unsigned
template<class traits>
  basic_ostream<char, traits>& operator<<(basic_ostream<char, traits>& out, signed char c);
template<class traits>
  basic_ostream<char, traits>& operator<<(basic_ostream<char, traits>& out, unsigned char c);
```

1 *Effects:* Behaves as a formatted output function (29.7.5.2.1) of `out`. Constructs a character sequence `seq`. If `c` has type `char` and the character type of the stream is not `char`, then `seq` consists of `out.widen(c)`; otherwise `seq` consists of `c`. Determines padding for `seq` as described in 29.7.5.2.1. Inserts `seq` into `out`. Calls `os.width(0)`.

2 *Returns:* `out`.

```
template<class charT, class traits>
  basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>& out, const charT* s);
template<class charT, class traits>
  basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>& out, const char* s);
template<class traits>
  basic_ostream<char, traits>& operator<<(basic_ostream<char, traits>& out, const char* s);
template<class traits>
  basic_ostream<char, traits>& operator<<(basic_ostream<char, traits>& out, const signed char* s);
template<class traits>
  basic_ostream<char, traits>& operator<<(basic_ostream<char, traits>& out,
                                          const unsigned char* s);
```

3 ~~*Requires:*~~ *Expects:* `s` ~~shall not be~~is not a null pointer.

4 *Effects:* Behaves like a formatted inserter (as described in 29.7.5.2.1) of `out`. Creates a character sequence `seq` of `n` characters starting at `s`, each widened using `out.widen()` (29.5.5.3), where `n` is the number that would be computed as if by:

(4.1) — `traits::length(s)` for the overload where the first argument is of type `basic_ostream<charT, traits>&` and the second is of type `const charT*`, and also for the overload where the first argument is of type `basic_ostream<char, traits>&` and the second is of type `const char*`,

(4.2) — `char_traits<char>::length(s)` for the overload where the first argument is of type `basic_ostream<charT, traits>&` and the second is of type `const char*`,

(4.3) — `traits::length(reinterpret_cast<const char*>(s))` for the other two overloads.

Determines padding for `seq` as described in 29.7.5.2.1. Inserts `seq` into `out`. Calls `width(0)`.

5 *Returns:* `out`.

29.7.5.3 Unformatted output functions [ostream.unformatted]

1 Each unformatted output function begins execution by constructing an object of class `sentry`. If this object returns `true`, while converting to a value of type `bool`, the function endeavors to generate the requested output. If an exception is thrown during output, then `ios::badbit` is turned on[323] in `*this`'s error state. If `(exceptions() & badbit) != 0` then the exception is rethrown. In any case, the unformatted output function ends by destroying the sentry object, then, if no exception was thrown, returning the value specified for the unformatted output function.

```
basic_ostream<charT, traits>& put(char_type c);
```

2 *Effects:* Behaves as an unformatted output function (as described above). After constructing a sentry object, inserts the character `c`, if possible.[324]

3 Otherwise, calls `setstate(badbit)` (which may throw `ios_base::failure` (29.5.5.4)).

4 *Returns:* `*this`.

```
basic_ostream& write(const char_type* s, streamsize n);
```

5 *Effects:* Behaves as an unformatted output function (as described above). After constructing a sentry object, obtains characters to insert from successive locations of an array whose first element is designated by `s`.[325] Characters are inserted until either of the following occurs:

(5.1) — `n` characters are inserted;

(5.2) — inserting in the output sequence fails (in which case the function calls `setstate(badbit)`, which may throw `ios_base::failure` (29.5.5.4)).

6 *Returns:* `*this`.

```
basic_ostream& flush();
```

7 *Effects:* Behaves as an unformatted output function (as described above). If `rdbuf()` is not a null pointer, constructs a sentry object. If this object returns `true` when converted to a value of type `bool` the function calls `rdbuf()->pubsync()`. If that function returns -1 calls `setstate(badbit)` (which may throw `ios_base::failure` (29.5.5.4)). Otherwise, if the sentry object returns `false`, does nothing.

8 *Returns:* `*this`.

29.7.5.4 Standard manipulators [ostream.manip]

1 Each instantiation of any of the function templates specified in this subclause is a designated addressable function (**??**).

```
template<class charT, class traits>
  basic_ostream<charT, traits>& endl(basic_ostream<charT, traits>& os);
```

2 *Effects:* Calls `os.put(os.widen('\n'))`, then `os.flush()`.

3 *Returns:* `os`.

```
template<class charT, class traits>
  basic_ostream<charT, traits>& ends(basic_ostream<charT, traits>& os);
```

4 *Effects:* Inserts a null character into the output sequence: calls `os.put(charT())`.

5 *Returns:* `os`.

323) without causing an `ios::failure` to be thrown.
324) Note that this function is not overloaded on types `signed char` and `unsigned char`.
325) Note that this function is not overloaded on types `signed char` and `unsigned char`.

```
template<class charT, class traits>
  basic_ostream<charT, traits>& flush(basic_ostream<charT, traits>& os);
```

6 *Effects:* Calls `os.flush()`.

7 *Returns:* `os`.

```
template<class charT, class traits>
  basic_ostream<charT, traits>& emit_on_flush(basic_ostream<charT, traits>& os);
```

8 *Effects:* If `os.rdbuf()` is a `basic_syncbuf<charT, traits, Allocator>*`, called `buf` for the purpose of exposition, calls `buf->set_emit_on_sync(true)`. Otherwise this manipulator has no effect. [*Note*: To work around the issue that the `Allocator` template argument cannot be deduced, implementations can introduce an intermediate base class to `basic_syncbuf` that manages its `emit_on_sync` flag. — *end note*]

9 *Returns:* `os`.

```
template<class charT, class traits>
  basic_ostream<charT, traits>& noemit_on_flush(basic_ostream<charT, traits>& os);
```

10 *Effects:* If `os.rdbuf()` is a `basic_syncbuf<charT, traits, Allocator>*`, called `buf` for the purpose of exposition, calls `buf->set_emit_on_sync(false)`. Otherwise this manipulator has no effect.

11 *Returns:* `os`.

```
template<class charT, class traits>
  basic_ostream<charT, traits>& flush_emit(basic_ostream<charT, traits>& os);
```

12 *Effects:* Calls `os.flush()`. Then, if `os.rdbuf()` is a `basic_syncbuf<charT, traits, Allocator>*`, called `buf` for the purpose of exposition, calls `buf->emit()`.

13 *Returns:* `os`.

29.7.5.5 Rvalue stream insertion [ostream.rvalue]

```
template<class charT, class traits, class T>
  basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>&& os, const T& x);
```

1 *Constraints:* The expression `os << x` is well-formed.

2 *Effects:* As if by: `os << x;`

3 *Returns:* `os`.

4 *Remarks:* This function shall not participate in overload resolution unless the expression `os << x` is well-formed.

29.7.6 Standard manipulators [std.manip]

1 The header `<iomanip>` defines several functions that support extractors and inserters that alter information maintained by class `ios_base` and its derived classes.

```
unspecified resetiosflags(ios_base::fmtflags mask);
```

2 *Returns:* An object of unspecified type such that if `out` is an object of type `basic_ostream<charT, traits>` then the expression `out << resetiosflags(mask)` behaves as if it called `f(out, mask)`, or if `in` is an object of type `basic_istream<charT, traits>` then the expression `in >> resetiosflags(mask)` behaves as if it called `f(in, mask)`, where the function `f` is defined as:[326]

```
void f(ios_base& str, ios_base::fmtflags mask) {
  // reset specified flags
  str.setf(ios_base::fmtflags(0), mask);
}
```

The expression `out << resetiosflags(mask)` shall have type `basic_ostream<charT, traits>&` and value `out`. The expression `in >> resetiosflags(mask)` shall have type `basic_istream<charT, traits>&` and value `in`.

326) The expression `cin >> resetiosflags(ios_base::skipws)` clears `ios_base::skipws` in the format flags stored in the `basic_istream<charT, traits>` object `cin` (the same as `cin >> noskipws`), and the expression `cout << resetiosflags(ios_base::showbase)` clears `ios_base::showbase` in the format flags stored in the `basic_ostream<charT, traits>` object `cout` (the same as `cout << noshowbase`).

unspecified `setiosflags(ios_base::fmtflags mask);`

3 *Returns:* An object of unspecified type such that if `out` is an object of type `basic_ostream<charT, traits>` then the expression `out << setiosflags(mask)` behaves as if it called `f(out, mask)`, or if `in` is an object of type `basic_istream<charT, traits>` then the expression `in >> setiosflags(mask)` behaves as if it called `f(in, mask)`, where the function `f` is defined as:

```
void f(ios_base& str, ios_base::fmtflags mask) {
  // set specified flags
  str.setf(mask);
}
```

The expression `out << setiosflags(mask)` shall have type `basic_ostream<charT, traits>&` and value `out`. The expression `in >> setiosflags(mask)` shall have type `basic_istream<charT, traits>&` and value `in`.

unspecified `setbase(int base);`

4 *Returns:* An object of unspecified type such that if `out` is an object of type `basic_ostream<charT, traits>` then the expression `out << setbase(base)` behaves as if it called `f(out, base)`, or if `in` is an object of type `basic_istream<charT, traits>` then the expression `in >> setbase(base)` behaves as if it called `f(in, base)`, where the function `f` is defined as:

```
void f(ios_base& str, int base) {
  // set basefield
  str.setf(base ==  8 ? ios_base::oct :
      base == 10 ? ios_base::dec :
      base == 16 ? ios_base::hex :
      ios_base::fmtflags(0), ios_base::basefield);
}
```

The expression `out << setbase(base)` shall have type `basic_ostream<charT, traits>&` and value `out`. The expression `in >> setbase(base)` shall have type `basic_istream<charT, traits>&` and value `in`.

unspecified `setfill(char_type c);`

5 *Returns:* An object of unspecified type such that if `out` is an object of type `basic_ostream<charT, traits>` and `c` has type `charT` then the expression `out << setfill(c)` behaves as if it called `f(out, c)`, where the function `f` is defined as:

```
template<class charT, class traits>
void f(basic_ios<charT, traits>& str, charT c) {
  // set fill character
  str.fill(c);
}
```

The expression `out << setfill(c)` shall have type `basic_ostream<charT, traits>&` and value `out`.

unspecified `setprecision(int n);`

6 *Returns:* An object of unspecified type such that if `out` is an object of type `basic_ostream<charT, traits>` then the expression `out << setprecision(n)` behaves as if it called `f(out, n)`, or if `in` is an object of type `basic_istream<charT, traits>` then the expression `in >> setprecision(n)` behaves as if it called `f(in, n)`, where the function `f` is defined as:

```
void f(ios_base& str, int n) {
  // set precision
  str.precision(n);
}
```

The expression `out << setprecision(n)` shall have type `basic_ostream<charT, traits>&` and value `out`. The expression `in >> setprecision(n)` shall have type `basic_istream<charT, traits>&` and value `in`.

unspecified `setw(int n);`

7 *Returns:* An object of unspecified type such that if `out` is an instance of `basic_ostream<charT, traits>` then the expression `out << setw(n)` behaves as if it called `f(out, n)`, or if `in` is an object

of type `basic_istream<charT, traits>` then the expression `in >> setw(n)` behaves as if it called `f(in, n)`, where the function `f` is defined as:

```
void f(ios_base& str, int n) {
  // set width
  str.width(n);
}
```

The expression `out << setw(n)` shall have type `basic_ostream<charT, traits>&` and value `out`. The expression `in >> setw(n)` shall have type `basic_istream<charT, traits>&` and value `in`.

29.7.7 Extended manipulators [ext.manip]

1 The header `<iomanip>` defines several functions that support extractors and inserters that allow for the parsing and formatting of sequences and values for money and time.

```
template<class moneyT> unspecified get_money(moneyT& mon, bool intl = false);
```

2 ~~*Requires:*~~ *Mandates:* The type `moneyT` ~~shall be~~is either `long double` or a specialization of the `basic_string` template (??).

3 *Effects:* The expression `in >> get_money(mon, intl)` described below behaves as a formatted input function (29.7.4.2.1).

4 *Returns:* An object of unspecified type such that if `in` is an object of type `basic_istream<charT, traits>` then the expression `in >> get_money(mon, intl)` behaves as if it called `f(in, mon, intl)`, where the function `f` is defined as:

```
template<class charT, class traits, class moneyT>
void f(basic_ios<charT, traits>& str, moneyT& mon, bool intl) {
  using Iter     = istreambuf_iterator<charT, traits>;
  using MoneyGet = money_get<charT, Iter>;

  ios_base::iostate err = ios_base::goodbit;
  const MoneyGet& mg = use_facet<MoneyGet>(str.getloc());

  mg.get(Iter(str.rdbuf()), Iter(), intl, str, err, mon);

  if (ios_base::goodbit != err)
    str.setstate(err);
}
```

The expression `in >> get_money(mon, intl)` shall have type `basic_istream<charT, traits>&` and value `in`.

```
template<class moneyT> unspecified put_money(const moneyT& mon, bool intl = false);
```

5 ~~*Requires:*~~ *Mandates:* The type `moneyT` ~~shall be~~is either `long double` or a specialization of the `basic_string` template (??).

6 *Returns:* An object of unspecified type such that if `out` is an object of type `basic_ostream<charT, traits>` then the expression `out << put_money(mon, intl)` behaves as a formatted output function (29.7.5.2.1) that calls `f(out, mon, intl)`, where the function `f` is defined as:

```
template<class charT, class traits, class moneyT>
void f(basic_ios<charT, traits>& str, const moneyT& mon, bool intl) {
  using Iter     = ostreambuf_iterator<charT, traits>;
  using MoneyPut = money_put<charT, Iter>;

  const MoneyPut& mp = use_facet<MoneyPut>(str.getloc());
  const Iter end = mp.put(Iter(str.rdbuf()), intl, str, str.fill(), mon);

  if (end.failed())
    str.setstate(ios::badbit);
}
```

The expression `out << put_money(mon, intl)` shall have type `basic_ostream<charT, traits>&` and value `out`.

```
template<class charT> unspecified get_time(struct tm* tmb, const charT* fmt);
```

7 ~~*Requires:*~~ *Expects:* The argument `tmb` ~~shall be~~is a valid pointer to an object of type `struct tm`. The argument `fmt` ~~shall be~~is a valid pointer to an array of objects of type `charT` with `char_traits<charT>::length(fmt)` elements.

8 *Returns:* An object of unspecified type such that if `in` is an object of type `basic_istream<charT, traits>` then the expression `in >> get_time(tmb, fmt)` behaves as if it called `f(in, tmb, fmt)`, where the function `f` is defined as:

```
template<class charT, class traits>
void f(basic_ios<charT, traits>& str, struct tm* tmb, const charT* fmt) {
  using Iter    = istreambuf_iterator<charT, traits>;
  using TimeGet = time_get<charT, Iter>;

  ios_base::iostate err = ios_base::goodbit;
  const TimeGet& tg = use_facet<TimeGet>(str.getloc());

  tg.get(Iter(str.rdbuf()), Iter(), str, err, tmb,
    fmt, fmt + traits::length(fmt));

  if (err != ios_base::goodbit)
    str.setstate(err);
}
```

The expression `in >> get_time(tmb, fmt)` shall have type `basic_istream<charT, traits>&` and value `in`.

```
template<class charT> unspecified put_time(const struct tm* tmb, const charT* fmt);
```

9 ~~*Requires:*~~ *Expects:* The argument `tmb` ~~shall be~~is a valid pointer to an object of type `struct tm`, and the argument `fmt` ~~shall be~~is a valid pointer to an array of objects of type `charT` with `char_traits<charT>::length(fmt)` elements.

10 *Returns:* An object of unspecified type such that if `out` is an object of type `basic_ostream<charT, traits>` then the expression `out << put_time(tmb, fmt)` behaves as if it called `f(out, tmb, fmt)`, where the function `f` is defined as:

```
template<class charT, class traits>
void f(basic_ios<charT, traits>& str, const struct tm* tmb, const charT* fmt) {
  using Iter    = ostreambuf_iterator<charT, traits>;
  using TimePut = time_put<charT, Iter>;

  const TimePut& tp = use_facet<TimePut>(str.getloc());
  const Iter end = tp.put(Iter(str.rdbuf()), str, str.fill(), tmb,
    fmt, fmt + traits::length(fmt));

  if (end.failed())
    str.setstate(ios_base::badbit);
}
```

The expression `out << put_time(tmb, fmt)` shall have type `basic_ostream<charT, traits>&` and value `out`.

29.7.8 Quoted manipulators [quoted.manip]

1 [*Note*: Quoted manipulators provide string insertion and extraction of quoted strings (for example, XML and CSV formats). Quoted manipulators are useful in ensuring that the content of a string with embedded spaces remains unchanged if inserted and then extracted via stream I/O. — *end note*]

```
template<class charT>
  unspecified quoted(const charT* s, charT delim = charT('"'), charT escape = charT('\\'));
template<class charT, class traits, class Allocator>
  unspecified quoted(const basic_string<charT, traits, Allocator>& s,
                     charT delim = charT('"'), charT escape = charT('\\'));
```

```
template<class charT, class traits>
  unspecified quoted(basic_string_view<charT, traits> s,
                     charT delim = charT('"'), charT escape = charT('\\'));
```

2 *Returns:* An object of unspecified type such that if `out` is an instance of `basic_ostream` with member type `char_type` the same as `charT` and with member type `traits_type`, which in the second and third forms is the same as `traits`, then the expression `out << quoted(s, delim, escape)` behaves as a formatted output function (29.7.5.2.1) of `out`. This forms a character sequence `seq`, initially consisting of the following elements:

(2.1) — `delim`.

(2.2) — Each character in `s`. If the character to be output is equal to `escape` or `delim`, as determined by `traits_type::eq`, first output `escape`.

(2.3) — `delim`.

Let `x` be the number of elements initially in `seq`. Then padding is determined for `seq` as described in 29.7.5.2.1, `seq` is inserted as if by calling `out.rdbuf()->sputn(seq, n)`, where `n` is the larger of `out.width()` and `x`, and `out.width(0)` is called. The expression `out << quoted(s, delim, escape)` shall have type `basic_ostream<charT, traits>&` and value `out`.

```
template<class charT, class traits, class Allocator>
  unspecified quoted(basic_string<charT, traits, Allocator>& s,
                     charT delim = charT('"'), charT escape = charT('\\'));
```

3 *Returns:* An object of unspecified type such that:

(3.1) — If `in` is an instance of `basic_istream` with member types `char_type` and `traits_type` the same as `charT` and `traits`, respectively, then the expression `in >> quoted(s, delim, escape)` behaves as if it extracts the following characters from `in` using `operator>>(basic_istream<charT, traits>&, charT&)` (29.7.4.2.3) which may throw `ios_base::failure` (29.5.3.1.1):

(3.1.1) — If the first character extracted is equal to `delim`, as determined by `traits_type::eq`, then:

(3.1.1.1) — Turn off the `skipws` flag.

(3.1.1.2) — `s.clear()`

(3.1.1.3) — Until an unescaped `delim` character is reached or `!in`, extract characters from `in` and append them to `s`, except that if an `escape` is reached, ignore it and append the next character to `s`.

(3.1.1.4) — Discard the final `delim` character.

(3.1.1.5) — Restore the `skipws` flag to its original value.

(3.1.2) — Otherwise, `in >> s`.

(3.2) — If `out` is an instance of `basic_ostream` with member types `char_type` and `traits_type` the same as `charT` and `traits`, respectively, then the expression `out << quoted(s, delim, escape)` behaves as specified for the `const basic_string<charT, traits, Allocator>&` overload of the `quoted` function.

The expression `in >> quoted(s, delim, escape)` shall have type `basic_istream<charT, traits>&` and value `in`. The expression `out << quoted(s, delim, escape)` shall have type `basic_ostream <charT, traits>&` and value `out`.

29.8 String-based streams [string.streams]

29.8.1 Header <sstream> synopsis [sstream.syn]

```
namespace std {
  template<class charT, class traits = char_traits<charT>,
           class Allocator = allocator<charT>>
    class basic_stringbuf;

  using stringbuf  = basic_stringbuf<char>;
  using wstringbuf = basic_stringbuf<wchar_t>;
```

```
  template<class charT, class traits = char_traits<charT>,
           class Allocator = allocator<charT>>
    class basic_istringstream;

  using istringstream  = basic_istringstream<char>;
  using wistringstream = basic_istringstream<wchar_t>;

  template<class charT, class traits = char_traits<charT>,
           class Allocator = allocator<charT>>
    class basic_ostringstream;
  using ostringstream  = basic_ostringstream<char>;
  using wostringstream = basic_ostringstream<wchar_t>;

  template<class charT, class traits = char_traits<charT>,
           class Allocator = allocator<charT>>
    class basic_stringstream;
  using stringstream  = basic_stringstream<char>;
  using wstringstream = basic_stringstream<wchar_t>;
}
```

1 The header `<sstream>` defines four class templates and eight types that associate stream buffers with objects of class `basic_string`, as described in **??**.

29.8.2 Class template `basic_stringbuf` [stringbuf]

```
namespace std {
  template<class charT, class traits = char_traits<charT>,
           class Allocator = allocator<charT>>
  class basic_stringbuf : public basic_streambuf<charT, traits> {
  public:
    using char_type      = charT;
    using int_type       = typename traits::int_type;
    using pos_type       = typename traits::pos_type;
    using off_type       = typename traits::off_type;
    using traits_type    = traits;
    using allocator_type = Allocator;

    // 29.8.2.1, constructors
    basic_stringbuf() : basic_stringbuf(ios_base::in | ios_base::out) {}
    explicit basic_stringbuf(ios_base::openmode which);
    explicit basic_stringbuf(
      const basic_string<charT, traits, Allocator>& str,
      ios_base::openmode which = ios_base::in | ios_base::out);
    basic_stringbuf(const basic_stringbuf& rhs) = delete;
    basic_stringbuf(basic_stringbuf&& rhs);

    // 29.8.2.2, assign and swap
    basic_stringbuf& operator=(const basic_stringbuf& rhs) = delete;
    basic_stringbuf& operator=(basic_stringbuf&& rhs);
    void swap(basic_stringbuf& rhs);

    // 29.8.2.3, get and set
    basic_string<charT, traits, Allocator> str() const;
    void str(const basic_string<charT, traits, Allocator>& s);

  protected:
    // 29.8.2.4, overridden virtual functions
    int_type underflow() override;
    int_type pbackfail(int_type c = traits::eof()) override;
    int_type overflow (int_type c = traits::eof()) override;
    basic_streambuf<charT, traits>* setbuf(charT*, streamsize) override;

    pos_type seekoff(off_type off, ios_base::seekdir way,
                     ios_base::openmode which
                      = ios_base::in | ios_base::out) override;
```

```
    pos_type seekpos(pos_type sp,
                     ios_base::openmode which
                      = ios_base::in | ios_base::out) override;

  private:
    ios_base::openmode mode;  // exposition only
  };

  template<class charT, class traits, class Allocator>
    void swap(basic_stringbuf<charT, traits, Allocator>& x,
              basic_stringbuf<charT, traits, Allocator>& y);
}
```

1 The class `basic_stringbuf` is derived from `basic_streambuf` to associate possibly the input sequence and possibly the output sequence with a sequence of arbitrary *characters*. The sequence can be initialized from, or made available as, an object of class `basic_string`.

2 For the sake of exposition, the maintained data is presented here as:

(2.1) — `ios_base::openmode mode`, has `in` set if the input sequence can be read, and `out` set if the output sequence can be written.

29.8.2.1 Constructors [stringbuf.cons]

```
explicit basic_stringbuf(ios_base::openmode which);
```

1 *Effects:* Constructs an object of class `basic_stringbuf`, initializing the base class with `basic_streambuf()` (29.6.3.1), and initializing `mode` with `which`. It is implementation-defined whether the sequence pointers (`eback()`, `gptr()`, `egptr()`, `pbase()`, `pptr()`, `epptr()`) are initialized to null pointers.

2 *Ensures:* `str() == ""`.

```
explicit basic_stringbuf(
  const basic_string<charT, traits, Allocator>& s,
  ios_base::openmode which = ios_base::in | ios_base::out);
```

3 *Effects:* Constructs an object of class `basic_stringbuf`, initializing the base class with `basic_streambuf()` (29.6.3.1), and initializing `mode` with `which`. Then calls `str(s)`.

```
basic_stringbuf(basic_stringbuf&& rhs);
```

4 *Effects:* Move constructs from the rvalue `rhs`. It is implementation-defined whether the sequence pointers in `*this` (`eback()`, `gptr()`, `egptr()`, `pbase()`, `pptr()`, `epptr()`) obtain the values which `rhs` had. Whether they do or not, `*this` and `rhs` reference separate buffers (if any at all) after the construction. The openmode, locale and any other state of `rhs` is also copied.

5 *Ensures:* Let `rhs_p` refer to the state of `rhs` just prior to this construction and let `rhs_a` refer to the state of `rhs` just after this construction.

(5.1) — `str() == rhs_p.str()`

(5.2) — `gptr() - eback() == rhs_p.gptr() - rhs_p.eback()`

(5.3) — `egptr() - eback() == rhs_p.egptr() - rhs_p.eback()`

(5.4) — `pptr() - pbase() == rhs_p.pptr() - rhs_p.pbase()`

(5.5) — `epptr() - pbase() == rhs_p.epptr() - rhs_p.pbase()`

(5.6) — `if (eback()) eback() != rhs_a.eback()`

(5.7) — `if (gptr()) gptr() != rhs_a.gptr()`

(5.8) — `if (egptr()) egptr() != rhs_a.egptr()`

(5.9) — `if (pbase()) pbase() != rhs_a.pbase()`

(5.10) — `if (pptr()) pptr() != rhs_a.pptr()`

(5.11) — `if (epptr()) epptr() != rhs_a.epptr()`

29.8.2.2 Assignment and swap [stringbuf.assign]

```
basic_stringbuf& operator=(basic_stringbuf&& rhs);
```

1 *Effects:* After the move assignment `*this` has the observable state it would have had if it had been move constructed from `rhs` (see 29.8.2.1).

2 *Returns:* `*this`.

```
void swap(basic_stringbuf& rhs);
```

3 *Effects:* Exchanges the state of `*this` and `rhs`.

```
template<class charT, class traits, class Allocator>
  void swap(basic_stringbuf<charT, traits, Allocator>& x,
            basic_stringbuf<charT, traits, Allocator>& y);
```

4 *Effects:* As if by `x.swap(y)`.

29.8.2.3 Member functions [stringbuf.members]

```
basic_string<charT, traits, Allocator> str() const;
```

1 *Returns:* A `basic_string` object whose content is equal to the `basic_stringbuf` underlying character sequence. If the `basic_stringbuf` was created only in input mode, the resultant `basic_string` contains the character sequence in the range `[eback(), egptr())`. If the `basic_stringbuf` was created with `which & ios_base::out` being nonzero then the resultant `basic_string` contains the character sequence in the range `[pbase(), high_mark)`, where `high_mark` represents the position one past the highest initialized character in the buffer. Characters can be initialized by writing to the stream, by constructing the `basic_stringbuf` with a `basic_string`, or by calling the `str(basic_string)` member function. In the case of calling the `str(basic_string)` member function, all characters initialized prior to the call are now considered uninitialized (except for those characters re-initialized by the new `basic_string`). Otherwise the `basic_stringbuf` has been created in neither input nor output mode and a zero length `basic_string` is returned.

```
void str(const basic_string<charT, traits, Allocator>& s);
```

2 *Effects:* Copies the content of `s` into the `basic_stringbuf` underlying character sequence and initializes the input and output sequences according to `mode`.

3 *Ensures:* If `mode & ios_base::out` is nonzero, `pbase()` points to the first underlying character and `epptr() >= pbase() + s.size()` holds; in addition, if `mode & ios_base::ate` is nonzero, `pptr() == pbase() + s.size()` holds, otherwise `pptr() == pbase()` is `true`. If `mode & ios_base::in` is nonzero, `eback()` points to the first underlying character, and both `gptr() == eback()` and `egptr() == eback() + s.size()` hold.

29.8.2.4 Overridden virtual functions [stringbuf.virtuals]

```
int_type underflow() override;
```

1 *Returns:* If the input sequence has a read position available, returns `traits::to_int_type(*gptr())`. Otherwise, returns `traits::eof()`. Any character in the underlying buffer which has been initialized is considered to be part of the input sequence.

```
int_type pbackfail(int_type c = traits::eof()) override;
```

2 *Effects:* Puts back the character designated by `c` to the input sequence, if possible, in one of three ways:

(2.1) — If `traits::eq_int_type(c, traits::eof())` returns `false` and if the input sequence has a putback position available, and if `traits::eq(to_char_type(c), gptr()[-1])` returns `true`, assigns `gptr() - 1` to `gptr()`.

Returns: `c`.

(2.2) — If `traits::eq_int_type(c, traits::eof())` returns `false` and if the input sequence has a putback position available, and if `mode & ios_base::out` is nonzero, assigns `c` to `*--gptr()`.

Returns: `c`.

(2.3) — If `traits::eq_int_type(c, traits::eof())` returns `true` and if the input sequence has a putback position available, assigns `gptr() - 1` to `gptr()`.

Returns: `traits::not_eof(c)`.

3 *Returns:* As specified above, or `traits::eof()` to indicate failure.

4 *Remarks:* If the function can succeed in more than one of these ways, it is unspecified which way is chosen.

```
int_type overflow(int_type c = traits::eof()) override;
```

5 *Effects:* Appends the character designated by `c` to the output sequence, if possible, in one of two ways:

(5.1) — If `traits::eq_int_type(c, traits::eof())` returns `false` and if either the output sequence has a write position available or the function makes a write position available (as described below), the function calls `sputc(c)`.

Signals success by returning `c`.

(5.2) — If `traits::eq_int_type(c, traits::eof())` returns `true`, there is no character to append.

Signals success by returning a value other than `traits::eof()`.

6 *Remarks:* The function can alter the number of write positions available as a result of any call.

7 *Returns:* As specified above, or `traits::eof()` to indicate failure.

8 The function can make a write position available only if `(mode & ios_base::out) != 0`. To make a write position available, the function reallocates (or initially allocates) an array object with a sufficient number of elements to hold the current array object (if any), plus at least one additional write position. If `(mode & ios_base::in) != 0`, the function alters the read end pointer `egptr()` to point just past the new write position.

```
pos_type seekoff(off_type off, ios_base::seekdir way,
                 ios_base::openmode which
                   = ios_base::in | ios_base::out) override;
```

9 *Effects:* Alters the stream position within one of the controlled sequences, if possible, as indicated in Table 112.

Table 112 — `seekoff` positioning

Conditions	Result
`(which & ios_base::in) == ios_base::in`	positions the input sequence
`(which & ios_base::out) == ios_base::out`	positions the output sequence
`(which & (ios_base::in | ios_base::out)) == (ios_base::in | ios_base::out)` and either `way == ios_base::beg` or `way == ios_base::end`	positions both the input and the output sequences
Otherwise	the positioning operation fails.

10 For a sequence to be positioned, the function determines `newoff` as indicated in Table 113. If the sequence's next pointer (either `gptr()` or `pptr()`) is a null pointer and `newoff` is nonzero, the positioning operation fails.

11 If `(newoff + off) < 0`, or if `newoff + off` refers to an uninitialized character (29.8.2.3), the positioning operation fails. Otherwise, the function assigns `xbeg + newoff + off` to the next pointer `xnext`.

12 *Returns:* `pos_type(newoff)`, constructed from the resultant offset `newoff` (of type `off_type`), that stores the resultant stream position, if possible. If the positioning operation fails, or if the constructed object cannot represent the resultant stream position, the return value is `pos_type(off_type(-1))`.

Table 113 — `newoff` values

Condition	`newoff` Value
`way == ios_base::beg`	0
`way == ios_base::cur`	the next pointer minus the beginning pointer (`xnext - xbeg`).
`way == ios_base::end`	the high mark pointer minus the beginning pointer (`high_mark - xbeg`).

```
pos_type seekpos(pos_type sp,
                 ios_base::openmode which
                   = ios_base::in | ios_base::out) override;
```

13 *Effects:* Equivalent to `seekoff(off_type(sp), ios_base::beg, which)`.

14 *Returns:* `sp` to indicate success, or `pos_type(off_type(-1))` to indicate failure.

```
basic_streambuf<charT, traits>* setbuf(charT* s, streamsize n);
```

15 *Effects:* implementation-defined, except that `setbuf(0, 0)` has no effect.

16 *Returns:* `this`.

29.8.3 Class template `basic_istringstream` [istringstream]

```
namespace std {
  template<class charT, class traits = char_traits<charT>,
           class Allocator = allocator<charT>>
  class basic_istringstream : public basic_istream<charT, traits> {
  public:
    using char_type      = charT;
    using int_type       = typename traits::int_type;
    using pos_type       = typename traits::pos_type;
    using off_type       = typename traits::off_type;
    using traits_type    = traits;
    using allocator_type = Allocator;

    // 29.8.3.1, constructors
    basic_istringstream() : basic_istringstream(ios_base::in) {}
    explicit basic_istringstream(ios_base::openmode which);
    explicit basic_istringstream(
      const basic_string<charT, traits, Allocator>& str,
      ios_base::openmode which = ios_base::in);
    basic_istringstream(const basic_istringstream& rhs) = delete;
    basic_istringstream(basic_istringstream&& rhs);

    // 29.8.3.2, assign and swap
    basic_istringstream& operator=(const basic_istringstream& rhs) = delete;
    basic_istringstream& operator=(basic_istringstream&& rhs);
    void swap(basic_istringstream& rhs);

    // 29.8.3.3, members
    basic_stringbuf<charT, traits, Allocator>* rdbuf() const;

    basic_string<charT, traits, Allocator> str() const;
    void str(const basic_string<charT, traits, Allocator>& s);
  private:
    basic_stringbuf<charT, traits, Allocator> sb; // exposition only
  };

  template<class charT, class traits, class Allocator>
    void swap(basic_istringstream<charT, traits, Allocator>& x,
              basic_istringstream<charT, traits, Allocator>& y);
}
```

1 The class `basic_istringstream<charT, traits, Allocator>` supports reading objects of class `basic_string<charT, traits, Allocator>`. It uses a `basic_stringbuf<charT, traits, Allocator>` object to control the associated storage. For the sake of exposition, the maintained data is presented here as:

(1.1) — `sb`, the `stringbuf` object.

29.8.3.1 Constructors [istringstream.cons]

```
explicit basic_istringstream(ios_base::openmode which);
```

1 *Effects:* Constructs an object of class `basic_istringstream<charT, traits>`, initializing the base class with `basic_istream<charT, traits>(addressof(sb))` (29.7.4.1) and initializing `sb` with `basic_stringbuf<charT, traits, Allocator>(which | ios_base::in)` (29.8.2.1).

```
explicit basic_istringstream(
  const basic_string<charT, traits, Allocator>& str,
  ios_base::openmode which = ios_base::in);
```

2 *Effects:* Constructs an object of class `basic_istringstream<charT, traits>`, initializing the base class with `basic_istream<charT, traits>(addressof(sb))` (29.7.4.1) and initializing `sb` with `basic_stringbuf<charT, traits, Allocator>(str, which | ios_base::in)` (29.8.2.1).

```
basic_istringstream(basic_istringstream&& rhs);
```

3 *Effects:* Move constructs from the rvalue `rhs`. This is accomplished by move constructing the base class, and the contained `basic_stringbuf`. Next `basic_istream<charT, traits>::set_rdbuf(addressof(sb))` is called to install the contained `basic_stringbuf`.

29.8.3.2 Assignment and swap [istringstream.assign]

```
basic_istringstream& operator=(basic_istringstream&& rhs);
```

1 *Effects:* Move assigns the base and members of `*this` from the base and corresponding members of `rhs`.

2 *Returns:* `*this`.

```
void swap(basic_istringstream& rhs);
```

3 *Effects:* Exchanges the state of `*this` and `rhs` by calling `basic_istream<charT, traits>::swap(rhs)` and `sb.swap(rhs.sb)`.

```
template<class charT, class traits, class Allocator>
  void swap(basic_istringstream<charT, traits, Allocator>& x,
            basic_istringstream<charT, traits, Allocator>& y);
```

4 *Effects:* As if by `x.swap(y)`.

29.8.3.3 Member functions [istringstream.members]

```
basic_stringbuf<charT, traits, Allocator>* rdbuf() const;
```

1 *Returns:* `const_cast<basic_stringbuf<charT, traits, Allocator>*>(addressof(sb))`.

```
basic_string<charT, traits, Allocator> str() const;
```

2 *Returns:* `rdbuf()->str()`.

```
void str(const basic_string<charT, traits, Allocator>& s);
```

3 *Effects:* Calls `rdbuf()->str(s)`.

29.8.4 Class template `basic_ostringstream` [ostringstream]

```
namespace std {
  template<class charT, class traits = char_traits<charT>,
           class Allocator = allocator<charT>>
  class basic_ostringstream : public basic_ostream<charT, traits> {
  public:
    using char_type      = charT;
    using int_type       = typename traits::int_type;
    using pos_type       = typename traits::pos_type;
```

```
    using off_type       = typename traits::off_type;
    using traits_type    = traits;
    using allocator_type = Allocator;

    // 29.8.4.1, constructors
    basic_ostringstream() : basic_ostringstream(ios_base::out) {}
    explicit basic_ostringstream(ios_base::openmode which);
    explicit basic_ostringstream(
      const basic_string<charT, traits, Allocator>& str,
      ios_base::openmode which = ios_base::out);
    basic_ostringstream(const basic_ostringstream& rhs) = delete;
    basic_ostringstream(basic_ostringstream&& rhs);

    // 29.8.4.2, assign and swap
    basic_ostringstream& operator=(const basic_ostringstream& rhs) = delete;
    basic_ostringstream& operator=(basic_ostringstream&& rhs);
    void swap(basic_ostringstream& rhs);

    // 29.8.4.3, members
    basic_stringbuf<charT, traits, Allocator>* rdbuf() const;

    basic_string<charT, traits, Allocator> str() const;
    void str(const basic_string<charT, traits, Allocator>& s);
  private:
    basic_stringbuf<charT, traits, Allocator> sb; // exposition only
  };

  template<class charT, class traits, class Allocator>
    void swap(basic_ostringstream<charT, traits, Allocator>& x,
              basic_ostringstream<charT, traits, Allocator>& y);
}
```

1 The class `basic_ostringstream<charT, traits, Allocator>` supports writing objects of class `basic_-string<charT, traits, Allocator>`. It uses a `basic_stringbuf` object to control the associated storage. For the sake of exposition, the maintained data is presented here as:

(1.1) — `sb`, the `stringbuf` object.

29.8.4.1 Constructors [ostringstream.cons]

```
explicit basic_ostringstream(ios_base::openmode which);
```

1 *Effects:* Constructs an object of class `basic_ostringstream<charT, traits>`, initializing the base class with `basic_ostream<charT, traits>(addressof(sb))` (29.7.5.1) and initializing `sb` with `basic_stringbuf<charT, traits, Allocator>(which | ios_base::out)` (29.8.2.1).

```
explicit basic_ostringstream(
  const basic_string<charT, traits, Allocator>& str,
  ios_base::openmode which = ios_base::out);
```

2 *Effects:* Constructs an object of class `basic_ostringstream<charT, traits>`, initializing the base class with `basic_ostream<charT, traits>(addressof(sb))` (29.7.5.1) and initializing `sb` with `basic_stringbuf<charT, traits, Allocator>(str, which | ios_base::out)` (29.8.2.1).

```
basic_ostringstream(basic_ostringstream&& rhs);
```

3 *Effects:* Move constructs from the rvalue `rhs`. This is accomplished by move constructing the base class, and the contained `basic_stringbuf`. Next `basic_ostream<charT, traits>::set_-rdbuf(addressof(sb))` is called to install the contained `basic_stringbuf`.

29.8.4.2 Assignment and swap [ostringstream.assign]

```
basic_ostringstream& operator=(basic_ostringstream&& rhs);
```

1 *Effects:* Move assigns the base and members of `*this` from the base and corresponding members of `rhs`.

2 *Returns:* `*this`.

```
void swap(basic_ostringstream& rhs);
```

3 *Effects:* Exchanges the state of `*this` and `rhs` by calling `basic_ostream<charT, traits>::swap(rhs)` and `sb.swap(rhs.sb)`.

```
template<class charT, class traits, class Allocator>
  void swap(basic_ostringstream<charT, traits, Allocator>& x,
            basic_ostringstream<charT, traits, Allocator>& y);
```

4 *Effects:* As if by `x.swap(y)`.

29.8.4.3 Member functions [ostringstream.members]

```
basic_stringbuf<charT, traits, Allocator>* rdbuf() const;
```

1 *Returns:* `const_cast<basic_stringbuf<charT, traits, Allocator>*>(addressof(sb))`.

```
basic_string<charT, traits, Allocator> str() const;
```

2 *Returns:* `rdbuf()->str()`.

```
void str(const basic_string<charT, traits, Allocator>& s);
```

3 *Effects:* Calls `rdbuf()->str(s)`.

29.8.5 Class template `basic_stringstream` [stringstream]

```
namespace std {
  template<class charT, class traits = char_traits<charT>,
           class Allocator = allocator<charT>>
  class basic_stringstream : public basic_iostream<charT, traits> {
  public:
    using char_type      = charT;
    using int_type       = typename traits::int_type;
    using pos_type       = typename traits::pos_type;
    using off_type       = typename traits::off_type;
    using traits_type    = traits;
    using allocator_type = Allocator;

    // 29.8.5.1, constructors
    basic_stringstream() : basic_stringstream(ios_base::out | ios_base::in) {}
    explicit basic_stringstream(ios_base::openmode which);
    explicit basic_stringstream(
      const basic_string<charT, traits, Allocator>& str,
      ios_base::openmode which = ios_base::out | ios_base::in);
    basic_stringstream(const basic_stringstream& rhs) = delete;
    basic_stringstream(basic_stringstream&& rhs);

    // 29.8.5.2, assign and swap
    basic_stringstream& operator=(const basic_stringstream& rhs) = delete;
    basic_stringstream& operator=(basic_stringstream&& rhs);
    void swap(basic_stringstream& rhs);

    // 29.8.5.3, members
    basic_stringbuf<charT, traits, Allocator>* rdbuf() const;
    basic_string<charT, traits, Allocator> str() const;
    void str(const basic_string<charT, traits, Allocator>& str);

  private:
    basic_stringbuf<charT, traits> sb;  // exposition only
  };

  template<class charT, class traits, class Allocator>
    void swap(basic_stringstream<charT, traits, Allocator>& x,
              basic_stringstream<charT, traits, Allocator>& y);
}
```

1 The class template `basic_stringstream<charT, traits>` supports reading and writing from objects of class `basic_string<charT, traits, Allocator>`. It uses a `basic_stringbuf<charT, traits, Allocator>` object to control the associated sequence. For the sake of exposition, the maintained data is presented here as

(1.1) — `sb`, the `stringbuf` object.

29.8.5.1 Constructors [stringstream.cons]

```
explicit basic_stringstream(ios_base::openmode which);
```

1 *Effects:* Constructs an object of class `basic_stringstream<charT, traits>`, initializing the base class with `basic_iostream<charT, traits>(addressof(sb))` (29.7.4.6.1) and initializing `sb` with `basic_stringbuf<charT, traits, Allocator>(which)`.

```
explicit basic_stringstream(
  const basic_string<charT, traits, Allocator>& str,
  ios_base::openmode which = ios_base::out | ios_base::in);
```

2 *Effects:* Constructs an object of class `basic_stringstream<charT, traits>`, initializing the base class with `basic_iostream<charT, traits>(addressof(sb))` (29.7.4.6.1) and initializing `sb` with `basic_stringbuf<charT, traits, Allocator>(str, which)`.

```
basic_stringstream(basic_stringstream&& rhs);
```

3 *Effects:* Move constructs from the rvalue `rhs`. This is accomplished by move constructing the base class, and the contained `basic_stringbuf`. Next `basic_istream<charT, traits>::set_rdbuf(addressof(sb))` is called to install the contained `basic_stringbuf`.

29.8.5.2 Assignment and swap [stringstream.assign]

```
basic_stringstream& operator=(basic_stringstream&& rhs);
```

1 *Effects:* Move assigns the base and members of `*this` from the base and corresponding members of `rhs`.

2 *Returns:* `*this`.

```
void swap(basic_stringstream& rhs);
```

3 *Effects:* Exchanges the state of `*this` and `rhs` by calling `basic_iostream<charT,traits>::swap(rhs)` and `sb.swap(rhs.sb)`.

```
template<class charT, class traits, class Allocator>
  void swap(basic_stringstream<charT, traits, Allocator>& x,
            basic_stringstream<charT, traits, Allocator>& y);
```

4 *Effects:* As if by `x.swap(y)`.

29.8.5.3 Member functions [stringstream.members]

```
basic_stringbuf<charT, traits, Allocator>* rdbuf() const;
```

1 *Returns:* `const_cast<basic_stringbuf<charT, traits, Allocator>*>(addressof(sb))`.

```
basic_string<charT, traits, Allocator> str() const;
```

2 *Returns:* `rdbuf()->str()`.

```
void str(const basic_string<charT, traits, Allocator>& str);
```

3 *Effects:* Calls `rdbuf()->str(str)`.

29.9 File-based streams [file.streams]

29.9.1 Header <fstream> synopsis [fstream.syn]

```
namespace std {
  template<class charT, class traits = char_traits<charT>>
    class basic_filebuf;
  using filebuf  = basic_filebuf<char>;
  using wfilebuf = basic_filebuf<wchar_t>;
```

```
  template<class charT, class traits = char_traits<charT>>
    class basic_ifstream;
  using ifstream  = basic_ifstream<char>;
  using wifstream = basic_ifstream<wchar_t>;

  template<class charT, class traits = char_traits<charT>>
    class basic_ofstream;
  using ofstream  = basic_ofstream<char>;
  using wofstream = basic_ofstream<wchar_t>;

  template<class charT, class traits = char_traits<charT>>
    class basic_fstream;
  using fstream  = basic_fstream<char>;
  using wfstream = basic_fstream<wchar_t>;
}
```

1 The header `<fstream>` defines four class templates and eight types that associate stream buffers with files and assist reading and writing files.

2 [*Note*: The class template `basic_filebuf` treats a file as a source or sink of bytes. In an environment that uses a large character set, the file typically holds multibyte character sequences and the `basic_filebuf` object converts those multibyte sequences into wide character sequences. — *end note*]

3 In this subclause, member functions taking arguments of `const filesystem::path::value_type*` are only be provided on systems where `filesystem::path::value_type` (29.11.7) is not `char`. [*Note*: These functions enable class `path` support for systems with a wide native path character type, such as `wchar_t`. — *end note*]

29.9.2 Class template `basic_filebuf` [filebuf]

```
namespace std {
  template<class charT, class traits = char_traits<charT>>
  class basic_filebuf : public basic_streambuf<charT, traits> {
  public:
    using char_type   = charT;
    using int_type    = typename traits::int_type;
    using pos_type    = typename traits::pos_type;
    using off_type    = typename traits::off_type;
    using traits_type = traits;

    // 29.9.2.1, constructors/destructor
    basic_filebuf();
    basic_filebuf(const basic_filebuf& rhs) = delete;
    basic_filebuf(basic_filebuf&& rhs);
    virtual ~basic_filebuf();

    // 29.9.2.2, assign and swap
    basic_filebuf& operator=(const basic_filebuf& rhs) = delete;
    basic_filebuf& operator=(basic_filebuf&& rhs);
    void swap(basic_filebuf& rhs);

    // 29.9.2.3, members
    bool is_open() const;
    basic_filebuf* open(const char* s, ios_base::openmode mode);
    basic_filebuf* open(const filesystem::path::value_type* s,
                        ios_base::openmode mode);  // wide systems only; see 29.9.1
    basic_filebuf* open(const string& s,
                        ios_base::openmode mode);
    basic_filebuf* open(const filesystem::path& s,
                        ios_base::openmode mode);
    basic_filebuf* close();

  protected:
    // 29.9.2.4, overridden virtual functions
    streamsize showmanyc() override;
    int_type underflow() override;
    int_type uflow() override;
```

```
    int_type pbackfail(int_type c = traits::eof()) override;
    int_type overflow (int_type c = traits::eof()) override;

    basic_streambuf<charT, traits>* setbuf(char_type* s,
                                           streamsize n) override;
    pos_type seekoff(off_type off, ios_base::seekdir way,
                     ios_base::openmode which
                      = ios_base::in | ios_base::out) override;
    pos_type seekpos(pos_type sp,
                     ios_base::openmode which
                      = ios_base::in | ios_base::out) override;
    int      sync() override;
    void     imbue(const locale& loc) override;
  };

  template<class charT, class traits>
    void swap(basic_filebuf<charT, traits>& x,
              basic_filebuf<charT, traits>& y);
}
```

1 The class `basic_filebuf<charT, traits>` associates both the input sequence and the output sequence with a file.

2 The restrictions on reading and writing a sequence controlled by an object of class `basic_filebuf<charT, traits>` are the same as for reading and writing with the C standard library `FILE`s.

3 In particular:

(3.1) — If the file is not open for reading the input sequence cannot be read.

(3.2) — If the file is not open for writing the output sequence cannot be written.

(3.3) — A joint file position is maintained for both the input sequence and the output sequence.

4 An instance of `basic_filebuf` behaves as described in 29.9.2 provided `traits::pos_type` is `fpos<traits::state_type>`. Otherwise the behavior is undefined.

5 In order to support file I/O and multibyte/wide character conversion, conversions are performed using members of a facet, referred to as `a_codecvt` in following subclauses, obtained as if by

```
const codecvt<charT, char, typename traits::state_type>& a_codecvt =
  use_facet<codecvt<charT, char, typename traits::state_type>>(getloc());
```

29.9.2.1 Constructors [filebuf.cons]

`basic_filebuf();`

1 *Effects:* Constructs an object of class `basic_filebuf<charT, traits>`, initializing the base class with `basic_streambuf<charT, traits>()` (29.6.3.1).

2 *Ensures:* `is_open() == false`.

`basic_filebuf(basic_filebuf&& rhs);`

3 *Effects:* Move constructs from the rvalue `rhs`. It is implementation-defined whether the sequence pointers in `*this` (`eback()`, `gptr()`, `egptr()`, `pbase()`, `pptr()`, `epptr()`) obtain the values which `rhs` had. Whether they do or not, `*this` and `rhs` reference separate buffers (if any at all) after the construction. Additionally `*this` references the file which `rhs` did before the construction, and `rhs` references no file after the construction. The openmode, locale and any other state of `rhs` is also copied.

4 *Ensures:* Let `rhs_p` refer to the state of `rhs` just prior to this construction and let `rhs_a` refer to the state of `rhs` just after this construction.

(4.1) — `is_open() == rhs_p.is_open()`

(4.2) — `rhs_a.is_open() == false`

(4.3) — `gptr() - eback() == rhs_p.gptr() - rhs_p.eback()`

(4.4) — `egptr() - eback() == rhs_p.egptr() - rhs_p.eback()`

(4.5) — `pptr() - pbase() == rhs_p.pptr() - rhs_p.pbase()`

(4.6) — `epptr() - pbase() == rhs_p.epptr() - rhs_p.pbase()`

(4.7) — `if (eback()) eback() != rhs_a.eback()`

(4.8) — `if (gptr()) gptr() != rhs_a.gptr()`

(4.9) — `if (egptr()) egptr() != rhs_a.egptr()`

(4.10) — `if (pbase()) pbase() != rhs_a.pbase()`

(4.11) — `if (pptr()) pptr() != rhs_a.pptr()`

(4.12) — `if (epptr()) epptr() != rhs_a.epptr()`

```
virtual ~basic_filebuf();
```

5 *Effects:* Destroys an object of class `basic_filebuf<charT, traits>`. Calls `close()`. If an exception occurs during the destruction of the object, including the call to `close()`, the exception is caught but not rethrown (see **??**).

29.9.2.2 Assignment and swap [filebuf.assign]

```
basic_filebuf& operator=(basic_filebuf&& rhs);
```

1 *Effects:* Calls `close()` then move assigns from `rhs`. After the move assignment `*this` has the observable state it would have had if it had been move constructed from `rhs` (see 29.9.2.1).

2 *Returns:* `*this`.

```
void swap(basic_filebuf& rhs);
```

3 *Effects:* Exchanges the state of `*this` and `rhs`.

```
template<class charT, class traits>
  void swap(basic_filebuf<charT, traits>& x,
            basic_filebuf<charT, traits>& y);
```

4 *Effects:* As if by `x.swap(y)`.

29.9.2.3 Member functions [filebuf.members]

```
bool is_open() const;
```

1 *Returns:* `true` if a previous call to `open` succeeded (returned a non-null value) and there has been no intervening call to close.

```
basic_filebuf* open(const char* s, ios_base::openmode mode);
basic_filebuf* open(const filesystem::path::value_type* s,
                    ios_base::openmode mode);  // wide systems only; see 29.9.1
```

2 *Expects:* `s` ~~shall point to~~is a NTCTS (**??**).

3 *Effects:* If `is_open() != false`, returns a null pointer. Otherwise, initializes the `filebuf` as required. It then opens the file to which `s` resolves, if possible, as if by a call to `fopen` with the second argument determined from `mode & ~ios_base::ate` as indicated in Table 114. If `mode` is not some combination of flags shown in the table then the open fails.

4 If the open operation succeeds and `(mode & ios_base::ate) != 0`, positions the file to the end (as if by calling `fseek(file, 0, SEEK_END)`, where `file` is the pointer returned by calling `fopen`).[327]

5 If the repositioning operation fails, calls `close()` and returns a null pointer to indicate failure.

6 *Returns:* `this` if successful, a null pointer otherwise.

```
basic_filebuf* open(const string& s, ios_base::openmode mode);
basic_filebuf* open(const filesystem::path& s, ios_base::openmode mode);
```

7 *Returns:* `open(s.c_str(), mode);`

327) The macro `SEEK_END` is defined, and the function signatures `fopen(const char*, const char*)` and `fseek(FILE*, long, int)` are declared, in `<cstdio>` (29.12.1).

Table 114 — File open modes

`ios_base` flag combination					`stdio` equivalent
`binary`	`in`	`out`	`trunc`	`app`	
		+			`"w"`
		+		+	`"a"`
				+	`"a"`
		+	+		`"w"`
	+				`"r"`
	+	+			`"r+"`
	+	+	+		`"w+"`
	+	+		+	`"a+"`
	+			+	`"a+"`
+		+			`"wb"`
+		+		+	`"ab"`
+				+	`"ab"`
+		+	+		`"wb"`
+	+				`"rb"`
+	+	+			`"r+b"`
+	+	+	+		`"w+b"`
+	+	+		+	`"a+b"`
+	+			+	`"a+b"`

```
basic_filebuf* close();
```

8 *Effects:* If `is_open() == false`, returns a null pointer. If a put area exists, calls `overflow(traits::eof())` to flush characters. If the last virtual member function called on `*this` (between `underflow`, `overflow`, `seekoff`, and `seekpos`) was `overflow` then calls `a_codecvt.unshift` (possibly several times) to determine a termination sequence, inserts those characters and calls `overflow(traits::eof())` again. Finally, regardless of whether any of the preceding calls fails or throws an exception, the function closes the file (as if by calling `fclose(file)`). If any of the calls made by the function, including `fclose`, fails, `close` fails by returning a null pointer. If one of these calls throws an exception, the exception is caught and rethrown after closing the file.

9 *Returns:* `this` on success, a null pointer otherwise.

10 *Ensures:* `is_open() == false`.

29.9.2.4 Overridden virtual functions [filebuf.virtuals]

```
streamsize showmanyc() override;
```

1 *Effects:* Behaves the same as `basic_streambuf::showmanyc()` (29.6.3.4).

2 *Remarks:* An implementation might well provide an overriding definition for this function signature if it can determine that more characters can be read from the input sequence.

```
int_type underflow() override;
```

3 *Effects:* Behaves according to the description of `basic_streambuf<charT, traits>::underflow()`, with the specialization that a sequence of characters is read from the input sequence as if by reading from the associated file into an internal buffer (`extern_buf`) and then as if by doing:

```
char   extern_buf[XSIZE];
char*  extern_end;
charT  intern_buf[ISIZE];
charT* intern_end;
codecvt_base::result r =
  a_codecvt.in(state, extern_buf, extern_buf+XSIZE, extern_end,
               intern_buf, intern_buf+ISIZE, intern_end);
```

This shall be done in such a way that the class can recover the position (`fpos_t`) corresponding to each character between `intern_buf` and `intern_end`. If the value of `r` indicates that `a_codecvt.in()` ran out of space in `intern_buf`, retry with a larger `intern_buf`.

```
int_type uflow() override;
```

4 *Effects:* Behaves according to the description of `basic_streambuf<charT, traits>::uflow()`, with the specialization that a sequence of characters is read from the input with the same method as used by `underflow`.

```
int_type pbackfail(int_type c = traits::eof()) override;
```

5 *Effects:* Puts back the character designated by `c` to the input sequence, if possible, in one of three ways:

(5.1) — If `traits::eq_int_type(c, traits::eof())` returns `false` and if the function makes a putback position available and if `traits::eq(to_char_type(c), gptr()[-1])` returns `true`, decrements the next pointer for the input sequence, `gptr()`.

Returns: `c`.

(5.2) — If `traits::eq_int_type(c, traits::eof())` returns `false` and if the function makes a putback position available and if the function is permitted to assign to the putback position, decrements the next pointer for the input sequence, and stores `c` there.

Returns: `c`.

(5.3) — If `traits::eq_int_type(c, traits::eof())` returns `true`, and if either the input sequence has a putback position available or the function makes a putback position available, decrements the next pointer for the input sequence, `gptr()`.

Returns: `traits::not_eof(c)`.

6 *Returns:* As specified above, or `traits::eof()` to indicate failure.

7 *Remarks:* If `is_open() == false`, the function always fails.

8 The function does not put back a character directly to the input sequence.

9 If the function can succeed in more than one of these ways, it is unspecified which way is chosen. The function can alter the number of putback positions available as a result of any call.

```
int_type overflow(int_type c = traits::eof()) override;
```

10 *Effects:* Behaves according to the description of `basic_streambuf<charT, traits>::overflow(c)`, except that the behavior of "consuming characters" is performed by first converting as if by:

```
charT* b = pbase();
charT* p = pptr();
charT* end;
char   xbuf[XSIZE];
char*  xbuf_end;
codecvt_base::result r =
  a_codecvt.out(state, b, p, end, xbuf, xbuf+XSIZE, xbuf_end);
```

and then

(10.1) — If `r == codecvt_base::error` then fail.

(10.2) — If `r == codecvt_base::noconv` then output characters from `b` up to (and not including) `p`.

(10.3) — If `r == codecvt_base::partial` then output to the file characters from `xbuf` up to `xbuf_end`, and repeat using characters from `end` to `p`. If output fails, fail (without repeating).

(10.4) — Otherwise output from `xbuf` to `xbuf_end`, and fail if output fails. At this point if `b != p` and `b == end` (`xbuf` isn't large enough) then increase `XSIZE` and repeat from the beginning.

11 *Returns:* `traits::not_eof(c)` to indicate success, and `traits::eof()` to indicate failure. If `is_open() == false`, the function always fails.

```
basic_streambuf* setbuf(char_type* s, streamsize n) override;
```

12 *Effects:* If `setbuf(0, 0)` is called on a stream before any I/O has occurred on that stream, the stream becomes unbuffered. Otherwise the results are implementation-defined. "Unbuffered" means that `pbase()` and `pptr()` always return null and output to the file should appear as soon as possible.

```
pos_type seekoff(off_type off, ios_base::seekdir way,
                 ios_base::openmode which
                   = ios_base::in | ios_base::out) override;
```

13 *Effects:* Let `width` denote `a_codecvt.encoding()`. If `is_open() == false`, or `off != 0 && width <= 0`, then the positioning operation fails. Otherwise, if `way != basic_ios::cur` or `off != 0`, and if the last operation was output, then update the output sequence and write any unshift sequence. Next, seek to the new position: if `width > 0`, call `fseek(file, width * off, whence)`, otherwise call `fseek(file, 0, whence)`.

14 *Remarks:* "The last operation was output" means either the last virtual operation was overflow or the put buffer is non-empty. "Write any unshift sequence" means, if `width` if less than zero then call `a_codecvt.unshift(state, xbuf, xbuf+XSIZE, xbuf_end)` and output the resulting unshift sequence. The function determines one of three values for the argument `whence`, of type `int`, as indicated in Table 115.

Table 115 — `seekoff` effects

`way` Value	`stdio` Equivalent
`basic_ios::beg`	`SEEK_SET`
`basic_ios::cur`	`SEEK_CUR`
`basic_ios::end`	`SEEK_END`

15 *Returns:* A newly constructed `pos_type` object that stores the resultant stream position, if possible. If the positioning operation fails, or if the object cannot represent the resultant stream position, returns `pos_type(off_type(-1))`.

```
pos_type seekpos(pos_type sp,
                 ios_base::openmode which
                   = ios_base::in | ios_base::out) override;
```

16 Alters the file position, if possible, to correspond to the position stored in `sp` (as described below). Altering the file position performs as follows:

1. if `(om & ios_base::out) != 0`, then update the output sequence and write any unshift sequence;
2. set the file position to `sp` as if by a call to `fsetpos`;
3. if `(om & ios_base::in) != 0`, then update the input sequence;

where `om` is the open mode passed to the last call to `open()`. The operation fails if `is_open()` returns `false`.

17 If `sp` is an invalid stream position, or if the function positions neither sequence, the positioning operation fails. If `sp` has not been obtained by a previous successful call to one of the positioning functions (`seekoff` or `seekpos`) on the same file the effects are undefined.

18 *Returns:* `sp` on success. Otherwise returns `pos_type(off_type(-1))`.

```
int sync() override;
```

19 *Effects:* If a put area exists, calls `filebuf::overflow` to write the characters to the file, then flushes the file as if by calling `fflush(file)`. If a get area exists, the effect is implementation-defined.

```
void imbue(const locale& loc) override;
```

20 ~~*Requires:*~~ *Expects:* If the file is not positioned at its beginning and the encoding of the current locale as determined by `a_codecvt.encoding()` is state-dependent (**??**) then that facet is the same as the corresponding facet of `loc`.

21 *Effects:* Causes characters inserted or extracted after this call to be converted according to `loc` until another call of `imbue`.

22 *Remarks:* This may require reconversion of previously converted characters. This in turn may require the implementation to be able to reconstruct the original contents of the file.

29.9.3 Class template `basic_ifstream` [ifstream]

```
namespace std {
  template<class charT, class traits = char_traits<charT>>
  class basic_ifstream : public basic_istream<charT, traits> {
  public:
    using char_type   = charT;
    using int_type    = typename traits::int_type;
    using pos_type    = typename traits::pos_type;
    using off_type    = typename traits::off_type;
    using traits_type = traits;

    // 29.9.3.1, constructors
    basic_ifstream();
    explicit basic_ifstream(const char* s,
                            ios_base::openmode mode = ios_base::in);
    explicit basic_ifstream(const filesystem::path::value_type* s,
                            ios_base::openmode mode = ios_base::in);  // wide systems only; see 29.9.1
    explicit basic_ifstream(const string& s,
                            ios_base::openmode mode = ios_base::in);
    explicit basic_ifstream(const filesystem::path& s,
                            ios_base::openmode mode = ios_base::in);
    basic_ifstream(const basic_ifstream& rhs) = delete;
    basic_ifstream(basic_ifstream&& rhs);

    // 29.9.3.2, assign and swap
    basic_ifstream& operator=(const basic_ifstream& rhs) = delete;
    basic_ifstream& operator=(basic_ifstream&& rhs);
    void swap(basic_ifstream& rhs);

    // 29.9.3.3, members
    basic_filebuf<charT, traits>* rdbuf() const;

    bool is_open() const;
    void open(const char* s, ios_base::openmode mode = ios_base::in);
    void open(const filesystem::path::value_type* s,
              ios_base::openmode mode = ios_base::in);  // wide systems only; see 29.9.1
    void open(const string& s, ios_base::openmode mode = ios_base::in);
    void open(const filesystem::path& s, ios_base::openmode mode = ios_base::in);
    void close();
  private:
    basic_filebuf<charT, traits> sb; // exposition only
  };

  template<class charT, class traits>
    void swap(basic_ifstream<charT, traits>& x,
              basic_ifstream<charT, traits>& y);
}
```

1 The class `basic_ifstream<charT, traits>` supports reading from named files. It uses a `basic_filebuf<charT, traits>` object to control the associated sequence. For the sake of exposition, the maintained data is presented here as:

(1.1) — `sb`, the `filebuf` object.

29.9.3.1 Constructors [ifstream.cons]

```
basic_ifstream();
```

1 *Effects:* Constructs an object of class `basic_ifstream<charT, traits>`, initializing the base class with `basic_istream<charT, traits>(addressof(sb))` (29.7.4.1.1) and initializing `sb` with `basic_filebuf<charT, traits>()` (29.9.2.1).

```
explicit basic_ifstream(const char* s,
                        ios_base::openmode mode = ios_base::in);
```

```
explicit basic_ifstream(const filesystem::path::value_type* s,
                        ios_base::openmode mode = ios_base::in);  // wide systems only; see 29.9.1
```

2 *Effects:* Constructs an object of class `basic_ifstream<charT, traits>`, initializing the base class with `basic_istream<charT, traits>(addressof(sb))` (29.7.4.1.1) and initializing `sb` with `basic_filebuf<charT, traits>()` (29.9.2.1), then calls `rdbuf()->open(s, mode | ios_base::in)`. If that function returns a null pointer, calls `setstate(failbit)`.

```
explicit basic_ifstream(const string& s,
                        ios_base::openmode mode = ios_base::in);
explicit basic_ifstream(const filesystem::path& s,
                        ios_base::openmode mode = ios_base::in);
```

3 *Effects:* The same as `basic_ifstream(s.c_str(), mode)`.

```
basic_ifstream(basic_ifstream&& rhs);
```

4 *Effects:* Move constructs from the rvalue `rhs`. This is accomplished by move constructing the base class, and the contained `basic_filebuf`. Next `basic_istream<charT, traits>::set_rdbuf(addressof(sb))` is called to install the contained `basic_filebuf`.

29.9.3.2 Assignment and swap [ifstream.assign]

```
basic_ifstream& operator=(basic_ifstream&& rhs);
```

1 *Effects:* Move assigns the base and members of `*this` from the base and corresponding members of `rhs`.

2 *Returns:* `*this`.

```
void swap(basic_ifstream& rhs);
```

3 *Effects:* Exchanges the state of `*this` and `rhs` by calling `basic_istream<charT, traits>::swap(rhs)` and `sb.swap(rhs.sb)`.

```
template<class charT, class traits>
  void swap(basic_ifstream<charT, traits>& x,
            basic_ifstream<charT, traits>& y);
```

4 *Effects:* As if by `x.swap(y)`.

29.9.3.3 Member functions [ifstream.members]

```
basic_filebuf<charT, traits>* rdbuf() const;
```

1 *Returns:* `const_cast<basic_filebuf<charT, traits>*>(addressof(sb))`.

```
bool is_open() const;
```

2 *Returns:* `rdbuf()->is_open()`.

```
void open(const char* s, ios_base::openmode mode = ios_base::in);
void open(const filesystem::path::value_type* s,
          ios_base::openmode mode = ios_base::in);  // wide systems only; see 29.9.1
```

3 *Effects:* Calls `rdbuf()->open(s, mode | ios_base::in)`. If that function does not return a null pointer calls `clear()`, otherwise calls `setstate(failbit)` (which may throw `ios_base::failure`) (29.5.5.4).

```
void open(const string& s, ios_base::openmode mode = ios_base::in);
void open(const filesystem::path& s, ios_base::openmode mode = ios_base::in);
```

4 *Effects:* Calls `open(s.c_str(), mode)`.

```
void close();
```

5 *Effects:* Calls `rdbuf()->close()` and, if that function returns a null pointer, calls `setstate(failbit)` (which may throw `ios_base::failure`) (29.5.5.4).

29.9.4 Class template `basic_ofstream` [ofstream]

```
namespace std {
  template<class charT, class traits = char_traits<charT>>
  class basic_ofstream : public basic_ostream<charT, traits> {
  public:
    using char_type   = charT;
    using int_type    = typename traits::int_type;
    using pos_type    = typename traits::pos_type;
    using off_type    = typename traits::off_type;
    using traits_type = traits;

    // 29.9.4.1, constructors
    basic_ofstream();
    explicit basic_ofstream(const char* s,
                            ios_base::openmode mode = ios_base::out);
    explicit basic_ofstream(const filesystem::path::value_type* s,
                            ios_base::openmode mode = ios_base::out);  // wide systems only; see 29.9.1
    explicit basic_ofstream(const string& s,
                            ios_base::openmode mode = ios_base::out);
    explicit basic_ofstream(const filesystem::path& s,
                            ios_base::openmode mode = ios_base::out);
    basic_ofstream(const basic_ofstream& rhs) = delete;
    basic_ofstream(basic_ofstream&& rhs);

    // 29.9.4.2, assign and swap
    basic_ofstream& operator=(const basic_ofstream& rhs) = delete;
    basic_ofstream& operator=(basic_ofstream&& rhs);
    void swap(basic_ofstream& rhs);

    // 29.9.4.3, members
    basic_filebuf<charT, traits>* rdbuf() const;

    bool is_open() const;
    void open(const char* s, ios_base::openmode mode = ios_base::out);
    void open(const filesystem::path::value_type* s,
              ios_base::openmode mode = ios_base::out);  // wide systems only; see 29.9.1
    void open(const string& s, ios_base::openmode mode = ios_base::out);
    void open(const filesystem::path& s, ios_base::openmode mode = ios_base::out);
    void close();
  private:
    basic_filebuf<charT, traits> sb; // exposition only
  };

  template<class charT, class traits>
    void swap(basic_ofstream<charT, traits>& x,
              basic_ofstream<charT, traits>& y);
}
```

1 The class `basic_ofstream<charT, traits>` supports writing to named files. It uses a `basic_filebuf<charT, traits>` object to control the associated sequence. For the sake of exposition, the maintained data is presented here as:

(1.1) — `sb`, the `filebuf` object.

29.9.4.1 Constructors [ofstream.cons]

```
basic_ofstream();
```

1 *Effects:* Constructs an object of class `basic_ofstream<charT, traits>`, initializing the base class with `basic_ostream<charT, traits>(addressof(sb))` (29.7.5.1.1) and initializing `sb` with `basic_filebuf<charT, traits>()` (29.9.2.1).

```
explicit basic_ofstream(const char* s,
                        ios_base::openmode mode = ios_base::out);
```

```
explicit basic_ofstream(const filesystem::path::value_type* s,
                        ios_base::openmode mode = ios_base::out); // wide systems only; see 29.9.1
```

2 *Effects:* Constructs an object of class `basic_ofstream<charT, traits>`, initializing the base class with `basic_ostream<charT, traits>(addressof(sb))` (29.7.5.1.1) and initializing `sb` with `basic_filebuf<charT, traits>()` (29.9.2.1), then calls `rdbuf()->open(s, mode | ios_base::out)`. If that function returns a null pointer, calls `setstate(failbit)`.

```
explicit basic_ofstream(const string& s,
                        ios_base::openmode mode = ios_base::out);
explicit basic_ofstream(const filesystem::path& s,
                        ios_base::openmode mode = ios_base::out);
```

3 *Effects:* The same as `basic_ofstream(s.c_str(), mode)`.

```
basic_ofstream(basic_ofstream&& rhs);
```

4 *Effects:* Move constructs from the rvalue `rhs`. This is accomplished by move constructing the base class, and the contained `basic_filebuf`. Next `basic_ostream<charT, traits>::set_rdbuf(addressof(sb))` is called to install the contained `basic_filebuf`.

29.9.4.2 Assignment and swap [ofstream.assign]

```
basic_ofstream& operator=(basic_ofstream&& rhs);
```

1 *Effects:* Move assigns the base and members of `*this` from the base and corresponding members of `rhs`.

2 *Returns:* `*this`.

```
void swap(basic_ofstream& rhs);
```

3 *Effects:* Exchanges the state of `*this` and `rhs` by calling `basic_ostream<charT, traits>::swap(rhs)` and `sb.swap(rhs.sb)`.

```
template<class charT, class traits>
  void swap(basic_ofstream<charT, traits>& x,
            basic_ofstream<charT, traits>& y);
```

4 *Effects:* As if by `x.swap(y)`.

29.9.4.3 Member functions [ofstream.members]

```
basic_filebuf<charT, traits>* rdbuf() const;
```

1 *Returns:* `const_cast<basic_filebuf<charT, traits>*>(addressof(sb))`.

```
bool is_open() const;
```

2 *Returns:* `rdbuf()->is_open()`.

```
void open(const char* s, ios_base::openmode mode = ios_base::out);
void open(const filesystem::path::value_type* s,
          ios_base::openmode mode = ios_base::out);  // wide systems only; see 29.9.1
```

3 *Effects:* Calls `rdbuf()->open(s, mode | ios_base::out)`. If that function does not return a null pointer calls `clear()`, otherwise calls `setstate(failbit)` (which may throw `ios_base::failure`) (29.5.5.4).

```
void close();
```

4 *Effects:* Calls `rdbuf()->close()` and, if that function fails (returns a null pointer), calls `setstate(failbit)` (which may throw `ios_base::failure`) (29.5.5.4).

```
void open(const string& s, ios_base::openmode mode = ios_base::out);
void open(const filesystem::path& s, ios_base::openmode mode = ios_base::out);
```

5 *Effects:* Calls `open(s.c_str(), mode)`.

29.9.5 Class template `basic_fstream` [fstream]

```
namespace std {
  template<class charT, class traits = char_traits<charT>>
  class basic_fstream : public basic_iostream<charT, traits> {
  public:
    using char_type   = charT;
    using int_type    = typename traits::int_type;
    using pos_type    = typename traits::pos_type;
    using off_type    = typename traits::off_type;
    using traits_type = traits;

    // 29.9.5.1, constructors
    basic_fstream();
    explicit basic_fstream(
      const char* s,
      ios_base::openmode mode = ios_base::in | ios_base::out);
    explicit basic_fstream(
      const filesystem::path::value_type* s,
      ios_base::openmode mode = ios_base::in|ios_base::out);  // wide systems only; see 29.9.1
    explicit basic_fstream(
      const string& s,
      ios_base::openmode mode = ios_base::in | ios_base::out);
    explicit basic_fstream(
      const filesystem::path& s,
      ios_base::openmode mode = ios_base::in | ios_base::out);
    basic_fstream(const basic_fstream& rhs) = delete;
    basic_fstream(basic_fstream&& rhs);

    // 29.9.5.2, assign and swap
    basic_fstream& operator=(const basic_fstream& rhs) = delete;
    basic_fstream& operator=(basic_fstream&& rhs);
    void swap(basic_fstream& rhs);

    // 29.9.5.3, members
    basic_filebuf<charT, traits>* rdbuf() const;
    bool is_open() const;
    void open(
      const char* s,
      ios_base::openmode mode = ios_base::in | ios_base::out);
    void open(
      const filesystem::path::value_type* s,
      ios_base::openmode mode = ios_base::in|ios_base::out);  // wide systems only; see 29.9.1
    void open(
      const string& s,
      ios_base::openmode mode = ios_base::in | ios_base::out);
    void open(
      const filesystem::path& s,
      ios_base::openmode mode = ios_base::in | ios_base::out);
    void close();

  private:
    basic_filebuf<charT, traits> sb; // exposition only
  };

  template<class charT, class traits>
    void swap(basic_fstream<charT, traits>& x,
              basic_fstream<charT, traits>& y);
}
```

1 The class template `basic_fstream<charT, traits>` supports reading and writing from named files. It uses a `basic_filebuf<charT, traits>` object to control the associated sequences. For the sake of exposition, the maintained data is presented here as:

(1.1) — `sb`, the `basic_filebuf` object.

29.9.5.1 Constructors [fstream.cons]

```
basic_fstream();
```

1 *Effects:* Constructs an object of class `basic_fstream<charT, traits>`, initializing the base class with `basic_iostream<charT, traits>(addressof(sb))` (29.7.4.6.1) and initializing `sb` with `basic_filebuf<charT, traits>()`.

```
explicit basic_fstream(
  const char* s,
  ios_base::openmode mode = ios_base::in | ios_base::out);
explicit basic_fstream(
  const filesystem::path::value_type* s,
  ios_base::openmode mode = ios_base::in | ios_base::out);  // wide systems only; see 29.9.1
```

2 *Effects:* Constructs an object of class `basic_fstream<charT, traits>`, initializing the base class with `basic_iostream<charT, traits>(addressof(sb))` (29.7.4.6.1) and initializing `sb` with `basic_filebuf<charT, traits>()`. Then calls `rdbuf()->open(s, mode)`. If that function returns a null pointer, calls `setstate(failbit)`.

```
explicit basic_fstream(
  const string& s,
  ios_base::openmode mode = ios_base::in | ios_base::out);
explicit basic_fstream(
  const filesystem::path& s,
  ios_base::openmode mode = ios_base::in | ios_base::out);
```

3 *Effects:* The same as `basic_fstream(s.c_str(), mode)`.

```
basic_fstream(basic_fstream&& rhs);
```

4 *Effects:* Move constructs from the rvalue `rhs`. This is accomplished by move constructing the base class, and the contained `basic_filebuf`. Next `basic_istream<charT, traits>::set_rdbuf(addressof(sb))` is called to install the contained `basic_filebuf`.

29.9.5.2 Assignment and swap [fstream.assign]

```
basic_fstream& operator=(basic_fstream&& rhs);
```

1 *Effects:* Move assigns the base and members of `*this` from the base and corresponding members of `rhs`.

2 *Returns:* `*this`.

```
void swap(basic_fstream& rhs);
```

3 *Effects:* Exchanges the state of `*this` and `rhs` by calling `basic_iostream<charT,traits>::swap(rhs)` and `sb.swap(rhs.sb)`.

```
template<class charT, class traits>
  void swap(basic_fstream<charT, traits>& x,
            basic_fstream<charT, traits>& y);
```

4 *Effects:* As if by `x.swap(y)`.

29.9.5.3 Member functions [fstream.members]

```
basic_filebuf<charT, traits>* rdbuf() const;
```

1 *Returns:* `const_cast<basic_filebuf<charT, traits>*>(addressof(sb))`.

```
bool is_open() const;
```

2 *Returns:* `rdbuf()->is_open()`.

```
void open(
  const char* s,
  ios_base::openmode mode = ios_base::in | ios_base::out);
```

```
void open(
  const filesystem::path::value_type* s,
  ios_base::openmode mode = ios_base::in | ios_base::out);  // wide systems only; see 29.9.1
```

3 *Effects:* Calls `rdbuf()->open(s, mode)`. If that function does not return a null pointer calls `clear()`, otherwise calls `setstate(failbit)` (which may throw `ios_base::failure`) (29.5.5.4).

```
void open(
  const string& s,
  ios_base::openmode mode = ios_base::in | ios_base::out);
void open(
  const filesystem::path& s,
  ios_base::openmode mode = ios_base::in | ios_base::out);
```

4 *Effects:* Calls `open(s.c_str(), mode)`.

```
void close();
```

5 *Effects:* Calls `rdbuf()->close()` and, if that function returns a null pointer, calls `setstate(failbit)` (which may throw `ios_base::failure`) (29.5.5.4).

29.10 Synchronized output streams [syncstream]

29.10.1 Header <syncstream> synopsis [syncstream.syn]

```
#include <ostream>  // see 29.7.2

namespace std {
  template<class charT, class traits, class Allocator>
    class basic_syncbuf;

  using syncbuf = basic_syncbuf<char>;
  using wsyncbuf = basic_syncbuf<wchar_t>;

  template<class charT, class traits, class Allocator>
    class basic_osyncstream;

  using osyncstream = basic_osyncstream<char>;
  using wosyncstream = basic_osyncstream<wchar_t>;
}
```

1 The header `<syncstream>` provides a mechanism to synchronize execution agents writing to the same stream.

29.10.2 Class template basic_syncbuf [syncstream.syncbuf]

29.10.2.1 Overview [syncstream.syncbuf.overview]

```
namespace std {
  template<class charT, class traits, class Allocator>
  class basic_syncbuf : public basic_streambuf<charT, traits> {
  public:
    using char_type      = charT;
    using int_type       = typename traits::int_type;
    using pos_type       = typename traits::pos_type;
    using off_type       = typename traits::off_type;
    using traits_type    = traits;
    using allocator_type = Allocator;

    using streambuf_type = basic_streambuf<charT, traits>;

    // 29.10.2.2, construction and destruction
    explicit basic_syncbuf(streambuf_type* obuf = nullptr)
      : basic_syncbuf(obuf, Allocator()) {}
    basic_syncbuf(streambuf_type*, const Allocator&);
    basic_syncbuf(basic_syncbuf&&);
    ~basic_syncbuf();
```

```
    // 29.10.2.3, assignment and swap
    basic_syncbuf& operator=(basic_syncbuf&&);
    void swap(basic_syncbuf&);

    // 29.10.2.4, member functions
    bool emit();
    streambuf_type* get_wrapped() const noexcept;
    allocator_type get_allocator() const noexcept;
    void set_emit_on_sync(bool) noexcept;

  protected:
    // 29.10.2.5, overridden virtual functions
    int sync() override;

  private:
    streambuf_type* wrapped;     // exposition only
    bool emit_on_sync{};         // exposition only
  };

  // 29.10.2.6, specialized algorithms
  template<class charT, class traits, class Allocator>
    void swap(basic_syncbuf<charT, traits, Allocator>&,
              basic_syncbuf<charT, traits, Allocator>&);
}
```

1 Class template `basic_syncbuf` stores character data written to it, known as the associated output, into internal buffers allocated using the object's allocator. The associated output is transferred to the wrapped stream buffer object `*wrapped` when `emit()` is called or when the `basic_syncbuf` object is destroyed. Such transfers are atomic with respect to transfers by other `basic_syncbuf` objects with the same wrapped stream buffer object.

29.10.2.2 Construction and destruction [syncstream.syncbuf.cons]

```
basic_syncbuf(streambuf_type* obuf, const Allocator& allocator);
```

1 *Effects:* ~~Constructs the `basic_syncbuf` object and~~ sets `wrapped` to `obuf`.

2 *Remarks:* A copy of `allocator` is used to allocate memory for internal buffers holding the associated output.

3 *Throws:* Nothing unless an exception is thrown by the construction of a mutex or by memory allocation.

4 *Ensures:* `get_wrapped() == obuf` and `get_allocator() == allocator` are `true`.

```
basic_syncbuf(basic_syncbuf&& other);
```

5 *Effects:* Move constructs from `other` (Table **??**).

6 *Ensures:* The value returned by `this->get_wrapped()` is the value returned by `other.get_wrapped()` prior to calling this constructor. Output stored in `other` prior to calling this constructor will be stored in `*this` afterwards. `other.rdbuf()->pbase() == other.rdbuf()->pptr()` and `other.get_wrapped() == nullptr` are `true`.

7 *Remarks:* This constructor disassociates `other` from its wrapped stream buffer, ensuring destruction of `other` produces no output.

```
~basic_syncbuf();
```

8 *Effects:* Calls `emit()`.

9 *Throws:* Nothing. If an exception is thrown from `emit()`, the destructor catches and ignores that exception.

29.10.2.3 Assignment and swap [syncstream.syncbuf.assign]

```
basic_syncbuf& operator=(basic_syncbuf&& rhs) noexcept;
```

1 *Effects:* Calls `emit()` then move assigns from `rhs`. After the move assignment `*this` has the observable state it would have had if it had been move constructed from `rhs` (29.10.2.2).

2 *Returns:* `*this`.

3 *Ensures:*

(3.1) — `rhs.get_wrapped() == nullptr` is `true`.

(3.2) — `this->get_allocator() == rhs.get_allocator()` is `true` when

```
allocator_traits<Allocator>::propagate_on_container_move_assignment::value
```

is `true`; otherwise, the allocator is unchanged.

4 *Remarks:* This assignment operator disassociates `rhs` from its wrapped stream buffer, ensuring destruction of `rhs` produces no output.

```
void swap(basic_syncbuf& other) noexcept;
```

5 ~~*Requires:*~~ *Expects:* Either `allocator_traits<Allocator>::propagate_on_container_swap::value` is `true` or `this->get_allocator() == other.get_allocator()` is `true`.

6 *Effects:* Exchanges the state of `*this` and `other`.

29.10.2.4 Member functions [syncstream.syncbuf.members]

```
bool emit();
```

1 *Effects:* Atomically transfers the associated output of `*this` to the stream buffer `*wrapped`, so that it appears in the output stream as a contiguous sequence of characters. `wrapped->pubsync()` is called if and only if a call was made to `sync()` since the most recent call to `emit()`, if any.

2 *Returns:* `true` if all of the following conditions hold; otherwise `false`:

(2.1) — `wrapped == nullptr` is `false`.

(2.2) — All of the characters in the associated output were successfully transferred.

(2.3) — The call to `wrapped->pubsync()` (if any) succeeded.

3 *Ensures:* On success, the associated output is empty.

4 *Synchronization:* All `emit()` calls transferring characters to the same stream buffer object appear to execute in a total order consistent with the "happens before" relation (**??**), where each `emit()` call synchronizes with subsequent `emit()` calls in that total order.

5 *Remarks:* May call member functions of `wrapped` while holding a lock uniquely associated with `wrapped`.

```
streambuf_type* get_wrapped() const noexcept;
```

6 *Returns:* `wrapped`.

```
allocator_type get_allocator() const noexcept;
```

7 *Returns:* A copy of the allocator that was set in the constructor or assignment operator.

```
void set_emit_on_sync(bool b) noexcept;
```

8 *Effects:* `emit_on_sync = b`.

29.10.2.5 Overridden virtual functions [syncstream.syncbuf.virtuals]

```
int sync() override;
```

1 *Effects:* Records that the wrapped stream buffer is to be flushed. Then, if `emit_on_sync` is `true`, calls `emit()`. [*Note*: If `emit_on_sync` is `false`, the actual flush is delayed until a call to `emit()`. — *end note*]

2 *Returns:* If `emit()` was called and returned `false`, returns `-1`; otherwise `0`.

29.10.2.6 Specialized algorithms [syncstream.syncbuf.special]

```
template<class charT, class traits, class Allocator>
  void swap(basic_syncbuf<charT, traits, Allocator>& a,
            basic_syncbuf<charT, traits, Allocator>& b) noexcept;
```

1 *Effects:* Equivalent to `a.swap(b)`.

29.10.3 Class template `basic_osyncstream` [syncstream.osyncstream]

29.10.3.1 Overview [syncstream.osyncstream.overview]

```
namespace std {
  template<class charT, class traits, class Allocator>
  class basic_osyncstream : public basic_ostream<charT, traits> {
  public:
    using char_type   = charT;
    using int_type    = typename traits::int_type;
    using pos_type    = typename traits::pos_type;
    using off_type    = typename traits::off_type;
    using traits_type = traits;

    using allocator_type = Allocator;
    using streambuf_type = basic_streambuf<charT, traits>;
    using syncbuf_type   = basic_syncbuf<charT, traits, Allocator>;

    // 29.10.3.2, construction and destruction
    basic_osyncstream(streambuf_type*, const Allocator&);
    explicit basic_osyncstream(streambuf_type* obuf)
      : basic_osyncstream(obuf, Allocator()) {}
    basic_osyncstream(basic_ostream<charT, traits>& os, const Allocator& allocator)
      : basic_osyncstream(os.rdbuf(), allocator) {}
    explicit basic_osyncstream(basic_ostream<charT, traits>& os)
      : basic_osyncstream(os, Allocator()) {}
    basic_osyncstream(basic_osyncstream&&) noexcept;
    ~basic_osyncstream();

    // 29.10.3.3, assignment
    basic_osyncstream& operator=(basic_osyncstream&&) noexcept;

    // 29.10.3.4, member functions
    void emit();
    streambuf_type* get_wrapped() const noexcept;
    syncbuf_type* rdbuf() const noexcept { return const_cast<syncbuf_type*>(addressof(sb)); }

  private:
    syncbuf_type sb;     // exposition only
  };
}
```

1 `Allocator` shall ~~satisfy~~meet the *Cpp17Allocator* requirements (Table **??**).

2 [*Example*: A named variable can be used within a block statement for streaming.

```
{
  osyncstream bout(cout);
  bout << "Hello, ";
  bout << "World!";
  bout << endl; // flush is noted
  bout << "and more!\n";
}   // characters are transferred and cout is flushed
```

— *end example*]

3 [*Example*: A temporary object can be used for streaming within a single statement.

```
osyncstream(cout) << "Hello, " << "World!" << '\n';
```

In this example, `cout` is not flushed. — *end example*]

29.10.3.2 Construction and destruction [syncstream.osyncstream.cons]

```
basic_osyncstream(streambuf_type* buf, const Allocator& allocator);
```

1 *Effects:* Initializes `sb` from `buf` and `allocator`. Initializes the base class with `basic_ostream<charT, traits>(addressof(sb))`.

2 [*Note*: The member functions of the provided stream buffer might be called from `emit()` while a lock is held. Care should be taken to ensure that this does not result in deadlock. — *end note*]

3 *Ensures:* `get_wrapped() == buf` is `true`.

```
basic_osyncstream(basic_osyncstream&& other) noexcept;
```

4 *Effects:* Move constructs the base class and `sb` from the corresponding subobjects of `other`, and calls `basic_ostream<charT, traits>::set_rdbuf(addressof(sb))`.

5 *Ensures:* The value returned by `get_wrapped()` is the value returned by `os.get_wrapped()` prior to calling this constructor. `nullptr == other.get_wrapped()` is `true`.

```
~basic_osyncstream();
```

6 *Effects:* Calls `emit()`. If an exception is thrown from `emit()`, that exception is caught and ignored.

29.10.3.3 Assignment [syncstream.osyncstream.assign]

```
basic_osyncstream& operator=(basic_osyncstream&& rhs) noexcept;
```

1 *Effects:* First, calls `emit()`. If an exception is thrown from `emit()`, that exception is caught and ignored. Move assigns `sb` from `rhs.sb`. [*Note*: This disassociates `rhs` from its wrapped stream buffer ensuring destruction of `rhs` produces no output. — *end note*]

2 *Ensures:* `nullptr == rhs.get_wrapped()` is `true`. `get_wrapped()` returns the value previously returned by `rhs.get_wrapped()`.

29.10.3.4 Member functions [syncstream.osyncstream.members]

```
void emit();
```

1 *Effects:* Calls `sb.emit()`. If that call returns `false`, calls `setstate(ios::badbit)`.

2 [*Example*: A flush on a `basic_osyncstream` does not flush immediately:

```
{
  osyncstream bout(cout);
  bout << "Hello," << '\n';      // no flush
  bout.emit();                   // characters transferred; cout not flushed
  bout << "World!" << endl;      // flush noted; cout not flushed
  bout.emit();                   // characters transferred; cout flushed
  bout << "Greetings." << '\n'; // no flush
}   // characters transferred; cout not flushed
```

— *end example*]

3 [*Example*: The function `emit()` can be used to handle exceptions from operations on the underlying stream.

```
{
  osyncstream bout(cout);
  bout << "Hello, " << "World!" << '\n';
  try {
    bout.emit();
  } catch (...) {
    // handle exception
  }
}
```

— *end example*]

```
streambuf_type* get_wrapped() const noexcept;
```

4 *Returns:* `sb.get_wrapped()`.

5 [*Example*: Obtaining the wrapped stream buffer with `get_wrapped()` allows wrapping it again with an `osyncstream`. For example,

```
{
  osyncstream bout1(cout);
  bout1 << "Hello, ";
```

```
  {
    osyncstream(bout1.get_wrapped()) << "Goodbye, " << "Planet!" << '\n';
  }
  bout1 << "World!" << '\n';
}
```

produces the *uninterleaved* output

```
Goodbye, Planet!
Hello, World!
```

— *end example*]

29.11 File systems [filesystems]

29.11.1 General [fs.general]

1 This subclause describes operations on file systems and their components, such as paths, regular files, and directories.

2 A *file system* is a collection of files and their attributes.

3 A *file* is an object within a file system that holds user or system data. Files can be written to, or read from, or both. A file has certain attributes, including type. File types include regular files and directories. Other types of files, such as symbolic links, may be supported by the implementation.

4 A *directory* is a file within a file system that acts as a container of directory entries that contain information about other files, possibly including other directory files. The *parent directory* of a directory is the directory that both contains a directory entry for the given directory and is represented by the dot-dot filename (29.11.7.1) in the given directory. The *parent directory* of other types of files is a directory containing a directory entry for the file under discussion.

5 A *link* is an object that associates a filename with a file. Several links can associate names with the same file. A *hard link* is a link to an existing file. Some file systems support multiple hard links to a file. If the last hard link to a file is removed, the file itself is removed. [*Note*: A hard link can be thought of as a shared-ownership smart pointer to a file. — *end note*] A *symbolic link* is a type of file with the property that when the file is encountered during pathname resolution (29.11.7), a string stored by the file is used to modify the pathname resolution. [*Note*: Symbolic links are often called symlinks. A symbolic link can be thought of as a raw pointer to a file. If the file pointed to does not exist, the symbolic link is said to be a "dangling" symbolic link. — *end note*]

29.11.2 Conformance [fs.conformance]

1 Conformance is specified in terms of behavior. Ideal behavior is not always implementable, so the conformance subclauses take that into account.

29.11.2.1 POSIX conformance [fs.conform.9945]

1 Some behavior is specified by reference to POSIX (29.11.3). How such behavior is actually implemented is unspecified. [*Note*: This constitutes an "as if" rule allowing implementations to call native operating system or other APIs. — *end note*]

2 Implementations should provide such behavior as it is defined by POSIX. Implementations shall document any behavior that differs from the behavior defined by POSIX. Implementations that do not support exact POSIX behavior should provide behavior as close to POSIX behavior as is reasonable given the limitations of actual operating systems and file systems. If an implementation cannot provide any reasonable behavior, the implementation shall report an error as specified in 29.11.6. [*Note*: This allows users to rely on an exception being thrown or an error code being set when an implementation cannot provide any reasonable behavior. — *end note*]

3 Implementations are not required to provide behavior that is not supported by a particular file system. [*Example*: The FAT file system used by some memory cards, camera memory, and floppy disks does not support hard links, symlinks, and many other features of more capable file systems, so implementations are not required to support those features on the FAT file system but instead are required to report an error as described above. — *end example*]

29.11.2.2 Operating system dependent behavior conformance [fs.conform.os]

1 Behavior that is specified as being *operating system dependent* is dependent upon the behavior and characteristics of an operating system. The operating system an implementation is dependent upon is implementation-defined.

2 It is permissible for an implementation to be dependent upon an operating system emulator rather than the actual underlying operating system.

29.11.2.3 File system race behavior [fs.race.behavior]

1 A *file system race* is the condition that occurs when multiple threads, processes, or computers interleave access and modification of the same object within a file system. Behavior is undefined if calls to functions provided by this subclause introduce a file system race.

2 If the possibility of a file system race would make it unreliable for a program to test for a precondition before calling a function described herein, *Requires:* is not specified for the function. [*Note*: As a design practice, preconditions are not specified when it is unreasonable for a program to detect them prior to calling the function. — *end note*]

29.11.3 Normative references [fs.norm.ref]

1 This subclause mentions commercially available operating systems for purposes of exposition.[328]

29.11.4 Requirements [fs.req]

1 Throughout this subclause, `char`, `wchar_t`, `char8_t`, `char16_t`, and `char32_t` are collectively called *encoded character types*.

2 Functions with template parameters named `EcharT` shall not participate in overload resolution unless `EcharT` is one of the encoded character types.

3 Template parameters named `InputIterator` shall satisfy the *Cpp17InputIterator* requirements (**??**) and shall have a value type that is one of the encoded character types.

4 [*Note*: Use of an encoded character type implies an associated character set and encoding. Since `signed char` and `unsigned char` have no implied character set and encoding, they are not included as permitted types. — *end note*]

5 Template parameters named `Allocator` shall satisfy the *Cpp17Allocator* requirements (Table **??**).

29.11.4.1 Namespaces and headers [fs.req.namespace]

1 Unless otherwise specified, references to entities described in this subclause are assumed to be qualified with `::std::filesystem::`.

29.11.5 Header `<filesystem>` synopsis [fs.filesystem.syn]

```
namespace std::filesystem {
  // 29.11.7, paths
  class path;

  // 29.11.7.7, path non-member functions
  void swap(path& lhs, path& rhs) noexcept;
  size_t hash_value(const path& p) noexcept;

  // 29.11.8, filesystem errors
  class filesystem_error;

  // 29.11.11, directory entries
  class directory_entry;

  // 29.11.12, directory iterators
  class directory_iterator;
```

328) POSIX® is a registered trademark of The IEEE. Windows® is a registered trademark of Microsoft Corporation. This information is given for the convenience of users of this document and does not constitute an endorsement by ISO or IEC of these products.

```
  // 29.11.12.2, range access for directory iterators
  directory_iterator begin(directory_iterator iter) noexcept;
  directory_iterator end(const directory_iterator&) noexcept;

  // 29.11.13, recursive directory iterators
  class recursive_directory_iterator;

  // 29.11.13.2, range access for recursive directory iterators
  recursive_directory_iterator begin(recursive_directory_iterator iter) noexcept;
  recursive_directory_iterator end(const recursive_directory_iterator&) noexcept;

  // 29.11.10, file status
  class file_status;

  struct space_info {
    uintmax_t capacity;
    uintmax_t free;
    uintmax_t available;
  };

  // 29.11.9, enumerations
  enum class file_type;
  enum class perms;
  enum class perm_options;
  enum class copy_options;
  enum class directory_options;

  using file_time_type = chrono::time_point<chrono::file_clock>;

  // 29.11.14, filesystem operations
  path absolute(const path& p);
  path absolute(const path& p, error_code& ec);

  path canonical(const path& p);
  path canonical(const path& p, error_code& ec);

  void copy(const path& from, const path& to);
  void copy(const path& from, const path& to, error_code& ec);
  void copy(const path& from, const path& to, copy_options options);
  void copy(const path& from, const path& to, copy_options options,
            error_code& ec);

  bool copy_file(const path& from, const path& to);
  bool copy_file(const path& from, const path& to, error_code& ec);
  bool copy_file(const path& from, const path& to, copy_options option);
  bool copy_file(const path& from, const path& to, copy_options option,
                 error_code& ec);

  void copy_symlink(const path& existing_symlink, const path& new_symlink);
  void copy_symlink(const path& existing_symlink, const path& new_symlink,
                    error_code& ec) noexcept;

  bool create_directories(const path& p);
  bool create_directories(const path& p, error_code& ec);

  bool create_directory(const path& p);
  bool create_directory(const path& p, error_code& ec) noexcept;

  bool create_directory(const path& p, const path& attributes);
  bool create_directory(const path& p, const path& attributes,
                        error_code& ec) noexcept;

  void create_directory_symlink(const path& to, const path& new_symlink);
```

```
void create_directory_symlink(const path& to, const path& new_symlink,
                              error_code& ec) noexcept;

void create_hard_link(const path& to, const path& new_hard_link);
void create_hard_link(const path& to, const path& new_hard_link,
                      error_code& ec) noexcept;

void create_symlink(const path& to, const path& new_symlink);
void create_symlink(const path& to, const path& new_symlink,
                    error_code& ec) noexcept;

path current_path();
path current_path(error_code& ec);
void current_path(const path& p);
void current_path(const path& p, error_code& ec) noexcept;

bool equivalent(const path& p1, const path& p2);
bool equivalent(const path& p1, const path& p2, error_code& ec) noexcept;

bool exists(file_status s) noexcept;
bool exists(const path& p);
bool exists(const path& p, error_code& ec) noexcept;

uintmax_t file_size(const path& p);
uintmax_t file_size(const path& p, error_code& ec) noexcept;

uintmax_t hard_link_count(const path& p);
uintmax_t hard_link_count(const path& p, error_code& ec) noexcept;

bool is_block_file(file_status s) noexcept;
bool is_block_file(const path& p);
bool is_block_file(const path& p, error_code& ec) noexcept;

bool is_character_file(file_status s) noexcept;
bool is_character_file(const path& p);
bool is_character_file(const path& p, error_code& ec) noexcept;

bool is_directory(file_status s) noexcept;
bool is_directory(const path& p);
bool is_directory(const path& p, error_code& ec) noexcept;

bool is_empty(const path& p);
bool is_empty(const path& p, error_code& ec);

bool is_fifo(file_status s) noexcept;
bool is_fifo(const path& p);
bool is_fifo(const path& p, error_code& ec) noexcept;

bool is_other(file_status s) noexcept;
bool is_other(const path& p);
bool is_other(const path& p, error_code& ec) noexcept;

bool is_regular_file(file_status s) noexcept;
bool is_regular_file(const path& p);
bool is_regular_file(const path& p, error_code& ec) noexcept;

bool is_socket(file_status s) noexcept;
bool is_socket(const path& p);
bool is_socket(const path& p, error_code& ec) noexcept;

bool is_symlink(file_status s) noexcept;
bool is_symlink(const path& p);
bool is_symlink(const path& p, error_code& ec) noexcept;
```

```
  file_time_type last_write_time(const path& p);
  file_time_type last_write_time(const path& p, error_code& ec) noexcept;
  void last_write_time(const path& p, file_time_type new_time);
  void last_write_time(const path& p, file_time_type new_time,
                       error_code& ec) noexcept;

  void permissions(const path& p, perms prms, perm_options opts=perm_options::replace);
  void permissions(const path& p, perms prms, error_code& ec) noexcept;
  void permissions(const path& p, perms prms, perm_options opts, error_code& ec);

  path proximate(const path& p, error_code& ec);
  path proximate(const path& p, const path& base = current_path());
  path proximate(const path& p, const path& base, error_code& ec);

  path read_symlink(const path& p);
  path read_symlink(const path& p, error_code& ec);

  path relative(const path& p, error_code& ec);
  path relative(const path& p, const path& base = current_path());
  path relative(const path& p, const path& base, error_code& ec);

  bool remove(const path& p);
  bool remove(const path& p, error_code& ec) noexcept;

  uintmax_t remove_all(const path& p);
  uintmax_t remove_all(const path& p, error_code& ec);

  void rename(const path& from, const path& to);
  void rename(const path& from, const path& to, error_code& ec) noexcept;

  void resize_file(const path& p, uintmax_t size);
  void resize_file(const path& p, uintmax_t size, error_code& ec) noexcept;

  space_info space(const path& p);
  space_info space(const path& p, error_code& ec) noexcept;

  file_status status(const path& p);
  file_status status(const path& p, error_code& ec) noexcept;

  bool status_known(file_status s) noexcept;

  file_status symlink_status(const path& p);
  file_status symlink_status(const path& p, error_code& ec) noexcept;

  path temp_directory_path();
  path temp_directory_path(error_code& ec);

  path weakly_canonical(const path& p);
  path weakly_canonical(const path& p, error_code& ec);
}
```

1 Implementations should ensure that the resolution and range of `file_time_type` reflect the operating system dependent resolution and range of file time values.

29.11.6 Error reporting [fs.err.report]

1 Filesystem library functions often provide two overloads, one that throws an exception to report file system errors, and another that sets an `error_code`. [*Note*: This supports two common use cases:

(1.1) — Uses where file system errors are truly exceptional and indicate a serious failure. Throwing an exception is an appropriate response.

(1.2) — Uses where file system errors are routine and do not necessarily represent failure. Returning an error code is the most appropriate response. This allows application specific error handling, including simply ignoring the error.

— *end note*]

2 Functions not having an argument of type `error_code&` handle errors as follows, unless otherwise specified:

(2.1) — When a call by the implementation to an operating system or other underlying API results in an error that prevents the function from meeting its specifications, an exception of type `filesystem_error` shall be thrown. For functions with a single path argument, that argument shall be passed to the `filesystem_error` constructor with a single path argument. For functions with two path arguments, the first of these arguments shall be passed to the `filesystem_error` constructor as the `path1` argument, and the second shall be passed as the `path2` argument. The `filesystem_error` constructor's `error_code` argument is set as appropriate for the specific operating system dependent error.

(2.2) — Failure to allocate storage is reported by throwing an exception as described in **??**.

(2.3) — Destructors throw nothing.

3 Functions having an argument of type `error_code&` handle errors as follows, unless otherwise specified:

(3.1) — If a call by the implementation to an operating system or other underlying API results in an error that prevents the function from meeting its specifications, the `error_code&` argument is set as appropriate for the specific operating system dependent error. Otherwise, `clear()` is called on the `error_code&` argument.

29.11.7 Class `path` [fs.class.path]

1 An object of class `path` represents a path and contains a pathname. Such an object is concerned only with the lexical and syntactic aspects of a path. The path does not necessarily exist in external storage, and the pathname is not necessarily valid for the current operating system or for a particular file system.

2 [*Note*: Class `path` is used to support the differences between the string types used by different operating systems to represent pathnames, and to perform conversions between encodings when necessary. — *end note*]

3 A *path* is a sequence of elements that identify the location of a file within a filesystem. The elements are the *root-name*$_{opt}$, *root-directory*$_{opt}$, and an optional sequence of *filename*s (29.11.7.1). The maximum number of elements in the sequence is operating system dependent (29.11.2.2).

4 An *absolute path* is a path that unambiguously identifies the location of a file without reference to an additional starting location. The elements of a path that determine if it is absolute are operating system dependent. A *relative path* is a path that is not absolute, and as such, only unambiguously identifies the location of a file when resolved relative to an implied starting location. The elements of a path that determine if it is relative are operating system dependent. [*Note*: Pathnames "." and ".." are relative paths. — *end note*]

5 A *pathname* is a character string that represents the name of a path. Pathnames are formatted according to the generic pathname format grammar (29.11.7.1) or according to an operating system dependent *native pathname format* accepted by the host operating system.

6 *Pathname resolution* is the operating system dependent mechanism for resolving a pathname to a particular file in a file hierarchy. There may be multiple pathnames that resolve to the same file. [*Example*: POSIX specifies the mechanism in section 4.11, Pathname resolution. — *end example*]

```
namespace std::filesystem {
  class path {
  public:
    using value_type  = see below;
    using string_type = basic_string<value_type>;
    static constexpr value_type preferred_separator = see below;

    // 29.11.9.1, enumeration format
    enum format;

    // 29.11.7.4.1, constructors and destructor
    path() noexcept;
    path(const path& p);
    path(path&& p) noexcept;
    path(string_type&& source, format fmt = auto_format);
```

```
template<class Source>
  path(const Source& source, format fmt = auto_format);
template<class InputIterator>
  path(InputIterator first, InputIterator last, format fmt = auto_format);
template<class Source>
  path(const Source& source, const locale& loc, format fmt = auto_format);
template<class InputIterator>
  path(InputIterator first, InputIterator last, const locale& loc, format fmt = auto_format);
~path();

// 29.11.7.4.2, assignments
path& operator=(const path& p);
path& operator=(path&& p) noexcept;
path& operator=(string_type&& source);
path& assign(string_type&& source);
template<class Source>
  path& operator=(const Source& source);
template<class Source>
  path& assign(const Source& source);
template<class InputIterator>
  path& assign(InputIterator first, InputIterator last);

// 29.11.7.4.3, appends
path& operator/=(const path& p);
template<class Source>
  path& operator/=(const Source& source);
template<class Source>
  path& append(const Source& source);
template<class InputIterator>
  path& append(InputIterator first, InputIterator last);

// 29.11.7.4.4, concatenation
path& operator+=(const path& x);
path& operator+=(const string_type& x);
path& operator+=(basic_string_view<value_type> x);
path& operator+=(const value_type* x);
path& operator+=(value_type x);
template<class Source>
  path& operator+=(const Source& x);
template<class EcharT>
  path& operator+=(EcharT x);
template<class Source>
  path& concat(const Source& x);
template<class InputIterator>
  path& concat(InputIterator first, InputIterator last);

// 29.11.7.4.5, modifiers
void  clear() noexcept;
path& make_preferred();
path& remove_filename();
path& replace_filename(const path& replacement);
path& replace_extension(const path& replacement = path());
void  swap(path& rhs) noexcept;

// 29.11.7.7, non-member operators
friend bool operator==(const path& lhs, const path& rhs) noexcept;
friend bool operator!=(const path& lhs, const path& rhs) noexcept;
friend bool operator< (const path& lhs, const path& rhs) noexcept;
friend bool operator<=(const path& lhs, const path& rhs) noexcept;
friend bool operator> (const path& lhs, const path& rhs) noexcept;
friend bool operator>=(const path& lhs, const path& rhs) noexcept;

friend path operator/ (const path& lhs, const path& rhs);
```

```
// 29.11.7.4.6, native format observers
const string_type& native() const noexcept;
const value_type*  c_str() const noexcept;
operator string_type() const;

template<class EcharT, class traits = char_traits<EcharT>,
         class Allocator = allocator<EcharT>>
  basic_string<EcharT, traits, Allocator>
    string(const Allocator& a = Allocator()) const;
std::string    string() const;
std::wstring   wstring() const;
std::u8string  u8string() const;
std::u16string u16string() const;
std::u32string u32string() const;

// 29.11.7.4.7, generic format observers
template<class EcharT, class traits = char_traits<EcharT>,
         class Allocator = allocator<EcharT>>
  basic_string<EcharT, traits, Allocator>
    generic_string(const Allocator& a = Allocator()) const;
std::string    generic_string() const;
std::wstring   generic_wstring() const;
std::u8string  generic_u8string() const;
std::u16string generic_u16string() const;
std::u32string generic_u32string() const;

// 29.11.7.4.8, compare
int compare(const path& p) const noexcept;
int compare(const string_type& s) const;
int compare(basic_string_view<value_type> s) const;
int compare(const value_type* s) const;

// 29.11.7.4.9, decomposition
path root_name() const;
path root_directory() const;
path root_path() const;
path relative_path() const;
path parent_path() const;
path filename() const;
path stem() const;
path extension() const;

// 29.11.7.4.10, query
[[nodiscard]] bool empty() const noexcept;
bool has_root_name() const;
bool has_root_directory() const;
bool has_root_path() const;
bool has_relative_path() const;
bool has_parent_path() const;
bool has_filename() const;
bool has_stem() const;
bool has_extension() const;
bool is_absolute() const;
bool is_relative() const;

// 29.11.7.4.11, generation
path lexically_normal() const;
path lexically_relative(const path& base) const;
path lexically_proximate(const path& base) const;

// 29.11.7.5, iterators
class iterator;
using const_iterator = iterator;
```

```
    iterator begin() const;
    iterator end() const;

    // 29.11.7.6, path inserter and extractor
    template<class charT, class traits>
      friend basic_ostream<charT, traits>&
        operator<<(basic_ostream<charT, traits>& os, const path& p);
    template<class charT, class traits>
      friend basic_istream<charT, traits>&
        operator>>(basic_istream<charT, traits>& is, path& p);
  };
}
```

7 `value_type` is a `typedef` for the operating system dependent encoded character type used to represent pathnames.

8 The value of the `preferred_separator` member is the operating system dependent *preferred-separator* character (29.11.7.1).

9 [*Example*: For POSIX-based operating systems, `value_type` is `char` and `preferred_separator` is the slash character (`'/'`). For Windows-based operating systems, `value_type` is `wchar_t` and `preferred_separator` is the backslash character (`L'\\'`). — *end example*]

29.11.7.1 Generic pathname format [fs.path.generic]

pathname:
root-name$_{opt}$ *root-directory*$_{opt}$ *relative-path*

root-name:
operating system dependent sequences of characters
implementation-defined sequences of characters

root-directory:
directory-separator

relative-path:
filename
filename directory-separator relative-path
an empty path

filename:
non-empty sequence of characters other than *directory-separator* characters

directory-separator:
preferred-separator directory-separator$_{opt}$
fallback-separator directory-separator$_{opt}$

preferred-separator:
operating system dependent directory separator character

fallback-separator:
`/`, if *preferred-separator* is not `/`

1 A *filename* is the name of a file. The *dot* and *dot-dot* filenames, consisting solely of one and two period characters respectively, have special meaning. The following characteristics of filenames are operating system dependent:

(1.1) — The permitted characters. [*Example*: Some operating systems prohibit the ASCII control characters (0x00 – 0x1F) in filenames. — *end example*] [*Note*: For wide portability, users may wish to limit *filename* characters to the POSIX Portable Filename Character Set:
`A B C D E F G H I J K L M N O P Q R S T U V W X Y Z`
`a b c d e f g h i j k l m n o p q r s t u v w x y z`
`0 1 2 3 4 5 6 7 8 9 . _ -` — *end note*]

(1.2) — The maximum permitted length.

(1.3) — Filenames that are not permitted.

(1.4) — Filenames that have special meaning.

(1.5) — Case awareness and sensitivity during path resolution.

(1.6) — Special rules that may apply to file types other than regular files, such as directories.

2 Except in a *root-name*, multiple successive *directory-separator* characters are considered to be the same as one *directory-separator* character.

3 The dot filename is treated as a reference to the current directory. The dot-dot filename is treated as a reference to the parent directory. What the dot-dot filename refers to relative to *root-directory* is implementation-defined. Specific filenames may have special meanings for a particular operating system.

4 A *root-name* identifies the starting location for pathname resolution (29.11.7). If there are no operating system dependent *root-name*s, at least one implementation-defined *root-name* is required. [*Note*: Many operating systems define a name beginning with two *directory-separator* characters as a *root-name* that identifies network or other resource locations. Some operating systems define a single letter followed by a colon as a drive specifier – a *root-name* identifying a specific device such as a disk drive. — *end note*]

5 If a *root-name* is otherwise ambiguous, the possibility with the longest sequence of characters is chosen. [*Note*: On a POSIX-like operating system, it is impossible to have a *root-name* and a *relative-path* without an intervening *root-directory* element. — *end note*]

6 *Normalization* of a generic format pathname means:

1. If the path is empty, stop.
2. Replace each slash character in the *root-name* with a *preferred-separator*.
3. Replace each *directory-separator* with a *preferred-separator*. [*Note*: The generic pathname grammar (29.11.7.1) defines *directory-separator* as one or more slashes and *preferred-separator*s. — *end note*]
4. Remove each dot filename and any immediately following *directory-separator*.
5. As long as any appear, remove a non-dot-dot filename immediately followed by a *directory-separator* and a dot-dot filename, along with any immediately following *directory-separator*.
6. If there is a *root-directory*, remove all dot-dot filenames and any *directory-separator*s immediately following them. [*Note*: These dot-dot filenames attempt to refer to nonexistent parent directories. — *end note*]
7. If the last filename is dot-dot, remove any trailing *directory-separator*.
8. If the path is empty, add a dot.

The result of normalization is a path in *normal form*, which is said to be *normalized*.

29.11.7.2 Conversions [fs.path.cvt]

29.11.7.2.1 Argument format conversions [fs.path.fmt.cvt]

1 [*Note*: The format conversions described in this subclause are not applied on POSIX-based operating systems because on these systems:

(1.1) — The generic format is acceptable as a native path.

(1.2) — There is no need to distinguish between native format and generic format in function arguments.

(1.3) — Paths for regular files and paths for directories share the same syntax.

— *end note*]

2 Several functions are defined to accept *detected-format* arguments, which are character sequences. A detected-format argument represents a path using either a pathname in the generic format (29.11.7.1) or a pathname in the native format (29.11.7). Such an argument is taken to be in the generic format if and only if it matches the generic format and is not acceptable to the operating system as a native path.

3 [*Note*: Some operating systems may have no unambiguous way to distinguish between native format and generic format arguments. This is by design as it simplifies use for operating systems that do not require disambiguation. An implementation for an operating system where disambiguation is required is permitted to distinguish between the formats. — *end note*]

4 Pathnames are converted as needed between the generic and native formats in an operating-system-dependent manner. Let *G(n)* and *N(g)* in a mathematical sense be the implementation's functions that convert native-to-generic and generic-to-native formats respectively. If $g=G(n)$ for some n, then $G(N(g))=g$; if $n=N(g)$ for some g, then $N(G(n))=n$. [*Note*: Neither G nor N need be invertible. — *end note*]

5 If the native format requires paths for regular files to be formatted differently from paths for directories, the path shall be treated as a directory path if its last element is a *directory-separator*, otherwise it shall be treated as a path to a regular file.

6 [*Note*: A path stores a native format pathname (29.11.7.4.6) and acts as if it also stores a generic format pathname, related as given below. The implementation may generate the generic format pathname based on the native format pathname (and possibly other information) when requested. — *end note*]

7 When a path is constructed from or is assigned a single representation separate from any path, the other representation is selected by the appropriate conversion function (G or N).

8 When the (new) value p of one representation of a path is derived from the representation of that or another path, a value q is chosen for the other representation. The value q converts to p (by G or N as appropriate) if any such value does so; q is otherwise unspecified. [*Note*: If q is the result of converting any path at all, it is the result of converting p. — *end note*]

29.11.7.2.2 Type and encoding conversions [fs.path.type.cvt]

1 The *native encoding* of an ordinary character string is the operating system dependent current encoding for pathnames (29.11.7). The *native encoding* for wide character strings is the implementation-defined execution wide-character set encoding (**??**).

2 For member function arguments that take character sequences representing paths and for member functions returning strings, value type and encoding conversion is performed if the value type of the argument or return value differs from `path::value_type`. For the argument or return value, the method of conversion and the encoding to be converted to is determined by its value type:

(2.1) — `char`: The encoding is the native ordinary encoding. The method of conversion, if any, is operating system dependent. [*Note*: For POSIX-based operating systems `path::value_type` is `char` so no conversion from `char` value type arguments or to `char` value type return values is performed. For Windows-based operating systems, the native ordinary encoding is determined by calling a Windows API function. — *end note*] [*Note*: This results in behavior identical to other C and C++ standard library functions that perform file operations using ordinary character strings to identify paths. Changing this behavior would be surprising and error prone. — *end note*]

(2.2) — `wchar_t`: The encoding is the native wide encoding. The method of conversion is unspecified. [*Note*: For Windows-based operating systems `path::value_type` is `wchar_t` so no conversion from `wchar_t` value type arguments or to `wchar_t` value type return values is performed. — *end note*]

(2.3) — `char8_t`: The encoding is UTF-8. The method of conversion is unspecified.

(2.4) — `char16_t`: The encoding is UTF-16. The method of conversion is unspecified.

(2.5) — `char32_t`: The encoding is UTF-32. The method of conversion is unspecified.

3 If the encoding being converted to has no representation for source characters, the resulting converted characters, if any, are unspecified. Implementations should not modify member function arguments if already of type `path::value_type`.

29.11.7.3 Requirements [fs.path.req]

1 In addition to the requirements (29.11.4), function template parameters named `Source` shall be one of:

(1.1) — `basic_string<EcharT, traits, Allocator>`. A function argument `const Source& source` shall have an effective range `[source.begin(), source.end())`.

(1.2) — `basic_string_view<EcharT, traits>`. A function argument `const Source& source` shall have an effective range `[source.begin(), source.end())`.

(1.3) — A type meeting the *Cpp17InputIterator* requirements that iterates over a NTCTS. The value type shall be an encoded character type. A function argument `const Source& source` shall have an effective range `[source, end)` where `end` is the first iterator value with an element value equal to `iterator_traits<Source>::value_type()`.

(1.4) — A character array that after array-to-pointer decay results in a pointer to the start of a NTCTS. The value type shall be an encoded character type. A function argument `const Source& source` shall have an effective range `[source, end)` where `end` is the first iterator value with an element value equal to `iterator_traits<decay_t<Source>>::value_type()`.

2 Functions taking template parameters named `Source` shall not participate in overload resolution unless either

(2.1) — `Source` is a specialization of `basic_string` or `basic_string_view`, or

(2.2) — the *qualified-id* `iterator_traits<decay_t<Source>>::value_type` is valid and denotes a possibly `const` encoded character type (**??**).

3 [*Note*: See path conversions (29.11.7.2) for how the value types above and their encodings convert to `path::value_type` and its encoding. — *end note*]

4 Arguments of type `Source` shall not be null pointers.

29.11.7.4 Members [fs.path.member]

29.11.7.4.1 Constructors [fs.path.construct]

```
path() noexcept;
```

1 *Effects:* Constructs an object of class `path`.

2 *Ensures:* `empty() == true`.

```
path(const path& p);
path(path&& p) noexcept;
```

3 *Effects:* Constructs an object of class `path` having the same pathname in the native and generic formats, respectively, as the original value of `p`. In the second form, `p` is left in a valid but unspecified state.

```
path(string_type&& source, format fmt = auto_format);
```

4 *Effects:* Constructs an object of class `path` for which the pathname in the detected-format of `source` has the original value of `source` (29.11.7.2.1), converting format if required (29.11.7.2.1). `source` is left in a valid but unspecified state.

```
template<class Source>
  path(const Source& source, format fmt = auto_format);
template<class InputIterator>
  path(InputIterator first, InputIterator last, format fmt = auto_format);
```

5 *Effects:* Let `s` be the effective range of `source` (29.11.7.3) or the range `[first, last)`, with the encoding converted if required (29.11.7.2). Finds the detected-format of `s` (29.11.7.2.1) and constructs an object of class `path` for which the pathname in that format is `s`.

```
template<class Source>
  path(const Source& source, const locale& loc, format fmt = auto_format);
template<class InputIterator>
  path(InputIterator first, InputIterator last, const locale& loc, format fmt = auto_format);
```

6 ~~*Requires:*~~ *Mandates:* The value type of `Source` and `InputIterator` is `char`.

7 *Effects:* Let `s` be the effective range of `source` or the range `[first, last)`, after converting the encoding as follows:

(7.1) — If `value_type` is `wchar_t`, converts to the native wide encoding (29.11.7.2.2) using the `codecvt<wchar_t, char, mbstate_t>` facet of `loc`.

(7.2) — Otherwise a conversion is performed using the `codecvt<wchar_t, char, mbstate_t>` facet of `loc`, and then a second conversion to the current ordinary encoding.

8 Finds the detected-format of `s` (29.11.7.2.1) and constructs an object of class `path` for which the pathname in that format is `s`.

[*Example*: A string is to be read from a database that is encoded in ISO/IEC 8859-1, and used to create a directory:

```
namespace fs = std::filesystem;
std::string latin1_string = read_latin1_data();
codecvt_8859_1<wchar_t> latin1_facet;
std::locale latin1_locale(std::locale(), latin1_facet);
fs::create_directory(fs::path(latin1_string, latin1_locale));
```

For POSIX-based operating systems, the path is constructed by first using `latin1_facet` to convert ISO/IEC 8859-1 encoded `latin1_string` to a wide character string in the native wide encoding (29.11.7.2.2). The resulting wide string is then converted to an ordinary character pathname string in the current native ordinary encoding. If the native wide encoding is UTF-16 or UTF-32, and the current native ordinary encoding is UTF-8, all of the characters in the ISO/IEC 8859-1 character set will be converted to their Unicode representation, but for other native ordinary encodings some characters may have no representation.

For Windows-based operating systems, the path is constructed by using `latin1_facet` to convert ISO/IEC 8859-1 encoded `latin1_string` to a UTF-16 encoded wide character pathname string. All of the characters in the ISO/IEC 8859-1 character set will be converted to their Unicode representation. — *end example*]

29.11.7.4.2 Assignments [fs.path.assign]

```
path& operator=(const path& p);
```

1 *Effects:* If `*this` and `p` are the same object, has no effect. Otherwise, sets both respective pathnames of `*this` to the respective pathnames of `p`.

2 *Returns:* `*this`.

```
path& operator=(path&& p) noexcept;
```

3 *Effects:* If `*this` and `p` are the same object, has no effect. Otherwise, sets both respective pathnames of `*this` to the respective pathnames of `p`. `p` is left in a valid but unspecified state. [*Note*: A valid implementation is `swap(p)`. — *end note*]

4 *Returns:* `*this`.

```
path& operator=(string_type&& source);
path& assign(string_type&& source);
```

5 *Effects:* Sets the pathname in the detected-format of `source` to the original value of `source`. `source` is left in a valid but unspecified state.

6 *Returns:* `*this`.

```
template<class Source>
  path& operator=(const Source& source);
template<class Source>
  path& assign(const Source& source);
template<class InputIterator>
  path& assign(InputIterator first, InputIterator last);
```

7 *Effects:* Let `s` be the effective range of `source` (29.11.7.3) or the range `[first, last)`, with the encoding converted if required (29.11.7.2). Finds the detected-format of `s` (29.11.7.2.1) and sets the pathname in that format to `s`.

8 *Returns:* `*this`.

29.11.7.4.3 Appends [fs.path.append]

1 The append operations use `operator/=` to denote their semantic effect of appending *preferred-separator* when needed.

```
path& operator/=(const path& p);
```

2 *Effects:* If `p.is_absolute() || (p.has_root_name() && p.root_name() != root_name())`, then `operator=(p)`.

3 Otherwise, modifies `*this` as if by these steps:

(3.1) — If `p.has_root_directory()`, then removes any root directory and relative path from the generic format pathname. Otherwise, if `!has_root_directory() && is_absolute()` is `true` or if `has_filename()` is `true`, then appends `path::preferred_separator` to the generic format pathname.

(3.2) — Then appends the native format pathname of `p`, omitting any *root-name* from its generic format pathname, to the native format pathname.

4 [*Example*: Even if `//host` is interpreted as a *root-name*, both of the paths `path("//host")/"foo"` and `path("//host/")/"foo"` equal `"//host/foo"`.

Expression examples:

```
// On POSIX,
path("foo") / "";      // yields "foo/"
path("foo") / "/bar"; // yields "/bar"
// On Windows, backslashes replace slashes in the above yields
```

```
// On Windows,
path("foo") / "c:/bar";  // yields "c:/bar"
path("foo") / "c:";      // yields "c:"
path("c:") / "";         // yields "c:"
path("c:foo") / "/bar";  // yields "c:/bar"
path("c:foo") / "c:bar"; // yields "c:foo/bar"
```

— *end example*]

5 *Returns:* `*this`.

```
template<class Source>
  path& operator/=(const Source& source);
template<class Source>
  path& append(const Source& source);
```

6 *Effects:* Equivalent to: `return operator/=(path(source));`

```
template<class InputIterator>
  path& append(InputIterator first, InputIterator last);
```

7 *Effects:* Equivalent to: `return operator/=(path(first, last));`

29.11.7.4.4 Concatenation [fs.path.concat]

```
path& operator+=(const path& x);
path& operator+=(const string_type& x);
path& operator+=(basic_string_view<value_type> x);
path& operator+=(const value_type* x);
path& operator+=(value_type x);
template<class Source>
  path& operator+=(const Source& x);
template<class EcharT>
  path& operator+=(EcharT x);
template<class Source>
  path& concat(const Source& x);
```

1 *Effects:* Appends `path(x).native()` to the pathname in the native format. [*Note*: This directly manipulates the value of `native()` and may not be portable between operating systems. — *end note*]

2 *Returns:* `*this`.

```
template<class InputIterator>
  path& concat(InputIterator first, InputIterator last);
```

3 *Effects:* Equivalent to: `return *this += path(first, last);`

29.11.7.4.5 Modifiers [fs.path.modifiers]

```
void clear() noexcept;
```

1 *Ensures:* `empty() == true`.

```
path& make_preferred();
```

2 *Effects:* Each *directory-separator* of the pathname in the generic format is converted to *preferred-separator*.

3 *Returns:* `*this`.

4 [*Example*:

```
path p("foo/bar");
std::cout << p << '\n';
p.make_preferred();
std::cout << p << '\n';
```

On an operating system where *preferred-separator* is a slash, the output is:

```
"foo/bar"
"foo/bar"
```

On an operating system where *preferred-separator* is a backslash, the output is:

```
"foo/bar"
"foo\bar"
```

— *end example*]

```
path& remove_filename();
```

Ensures: `!has_filename()`.

Effects: Remove the generic format pathname of `filename()` from the generic format pathname.

Ensures: `!has_filename()`.

Returns: `*this`.

[*Example*:

```
path("foo/bar").remove_filename(); // yields "foo/"
path("foo/").remove_filename();    // yields "foo/"
path("/foo").remove_filename();    // yields "/"
path("/").remove_filename();       // yields "/"
```

— *end example*]

```
path& replace_filename(const path& replacement);
```

Effects: Equivalent to:

```
remove_filename();
operator/=(replacement);
```

Returns: `*this`.

[*Example*:

```
path("/foo").replace_filename("bar");  // yields "/bar" on POSIX
path("/").replace_filename("bar");     // yields "/bar" on POSIX
```

— *end example*]

```
path& replace_extension(const path& replacement = path());
```

Effects:

(13.1) — Any existing `extension()`(29.11.7.4.9) is removed from the pathname in the generic format, then

(13.2) — If `replacement` is not empty and does not begin with a dot character, a dot character is appended to the pathname in the generic format, then

(13.3) — `operator+=(replacement);`.

Returns: `*this`.

```
void swap(path& rhs) noexcept;
```

Effects: Swaps the contents (in all formats) of the two paths.

Complexity: Constant time.

29.11.7.4.6 Native format observers [fs.path.native.obs]

The string returned by all native format observers is in the native pathname format (29.11.7).

```
const string_type& native() const noexcept;
```

Returns: The pathname in the native format.

```
const value_type* c_str() const noexcept;
```

Effects: Equivalent to: `return native().c_str();`

```
operator string_type() const;
```

Returns: `native()`.

[*Note*: Conversion to `string_type` is provided so that an object of class `path` can be given as an argument to existing standard library file stream constructors and open functions. — *end note*]

```
template<class EcharT, class traits = char_traits<EcharT>,
         class Allocator = allocator<EcharT>>
  basic_string<EcharT, traits, Allocator>
    string(const Allocator& a = Allocator()) const;
```

6 *Returns:* `native()`.

7 *Remarks:* All memory allocation, including for the return value, shall be performed by `a`. Conversion, if any, is specified by 29.11.7.2.

```
std::string string() const;
std::wstring wstring() const;
std::u8string u8string() const;
std::u16string u16string() const;
std::u32string u32string() const;
```

8 *Returns:* `native()`.

9 *Remarks:* Conversion, if any, is performed as specified by 29.11.7.2.

29.11.7.4.7 Generic format observers [fs.path.generic.obs]

1 Generic format observer functions return strings formatted according to the generic pathname format (29.11.7.1). A single slash (`'/'`) character is used as the *directory-separator*.

2 [*Example*: On an operating system that uses backslash as its *preferred-separator*,

```
path("foo\\bar").generic_string()
```

returns `"foo/bar"`. — *end example*]

```
template<class EcharT, class traits = char_traits<EcharT>,
         class Allocator = allocator<EcharT>>
  basic_string<EcharT, traits, Allocator>
    generic_string(const Allocator& a = Allocator()) const;
```

3 *Returns:* The pathname in the generic format.

4 *Remarks:* All memory allocation, including for the return value, shall be performed by `a`. Conversion, if any, is specified by 29.11.7.2.

```
std::string generic_string() const;
std::wstring generic_wstring() const;
std::u8string generic_u8string() const;
std::u16string generic_u16string() const;
std::u32string generic_u32string() const;
```

5 *Returns:* The pathname in the generic format.

6 *Remarks:* Conversion, if any, is specified by 29.11.7.2.

29.11.7.4.8 Compare [fs.path.compare]

```
int compare(const path& p) const noexcept;
```

1 *Returns:*

(1.1) — Let `rootNameComparison` be the result of `this->root_name().native().compare(p.root_name().native())`. If `rootNameComparison` is not `0`, `rootNameComparison`.

(1.2) — Otherwise, if `!this->has_root_directory()` and `p.has_root_directory()`, a value less than `0`.

(1.3) — Otherwise, if `this->has_root_directory()` and `!p.has_root_directory()`, a value greater than `0`.

(1.4) — Otherwise, if `native()` for the elements of `this->relative_path()` are lexicographically less than `native()` for the elements of `p.relative_path()`, a value less than `0`.

(1.5) — Otherwise, if `native()` for the elements of `this->relative_path()` are lexicographically greater than `native()` for the elements of `p.relative_path()`, a value greater than `0`.

(1.6) — Otherwise, `0`.

```
int compare(const string_type& s) const
int compare(basic_string_view<value_type> s) const;
int compare(const value_type* s) const
```

2 *Effects:* Equivalent to: `return compare(path(s));`

29.11.7.4.9 Decomposition [fs.path.decompose]

```
path root_name() const;
```

1 *Returns:* *root-name*, if the pathname in the generic format includes *root-name*, otherwise `path()`.

```
path root_directory() const;
```

2 *Returns:* *root-directory*, if the pathname in the generic format includes *root-directory*, otherwise `path()`.

```
path root_path() const;
```

3 *Returns:* `root_name() / root_directory()`.

```
path relative_path() const;
```

4 *Returns:* A `path` composed from the pathname in the generic format, if `empty()` is `false`, beginning with the first *filename* after `root_path()`. Otherwise, `path()`.

```
path parent_path() const;
```

5 *Returns:* `*this` if `has_relative_path()` is `false`, otherwise a path whose generic format pathname is the longest prefix of the generic format pathname of `*this` that produces one fewer element in its iteration.

```
path filename() const;
```

6 *Returns:* `relative_path().empty() ? path() : *--end()`.

7 [*Example*:

```
path("/foo/bar.txt").filename();   // yields "bar.txt"
path("/foo/bar").filename();       // yields "bar"
path("/foo/bar/").filename();      // yields ""
path("/").filename();              // yields ""
path("//host").filename();         // yields ""
path(".").filename();              // yields "."
path("..").filename();             // yields ".."
```

— *end example*]

```
path stem() const;
```

8 *Returns:* Let `f` be the generic format pathname of `filename()`. Returns a path whose pathname in the generic format is

(8.1) — `f`, if it contains no periods other than a leading period or consists solely of one or two periods;

(8.2) — otherwise, the prefix of `f` ending before its last period.

9 [*Example*:

```
std::cout << path("/foo/bar.txt").stem(); // outputs "bar"
path p = "foo.bar.baz.tar";
for (; !p.extension().empty(); p = p.stem())
  std::cout << p.extension() << '\n';
  // outputs: .tar
  // .baz
  // .bar
```

— *end example*]

```
path extension() const;
```

10 *Returns:* A path whose pathname in the generic format is the suffix of `filename()` not included in `stem()`.

11 [*Example*:

```
path("/foo/bar.txt").extension();  // yields ".txt" and stem() is "bar"
path("/foo/bar").extension();      // yields "" and stem() is "bar"
path("/foo/.profile").extension(); // yields "" and stem() is ".profile"
path(".bar").extension();          // yields "" and stem() is ".bar"
path("..bar").extension();         // yields ".bar" and stem() is "."
```

— *end example*]

12 [*Note*: The period is included in the return value so that it is possible to distinguish between no extension and an empty extension. — *end note*]

13 [*Note*: On non-POSIX operating systems, for a path `p`, it may not be the case that `p.stem() + p.extension() == p.filename()`, even though the generic format pathnames are the same. — *end note*]

29.11.7.4.10 Query [fs.path.query]

```
[[nodiscard]] bool empty() const noexcept;
```

1 *Returns:* `true` if the pathname in the generic format is empty, otherwise `false`.

```
bool has_root_path() const;
```

2 *Returns:* `!root_path().empty()`.

```
bool has_root_name() const;
```

3 *Returns:* `!root_name().empty()`.

```
bool has_root_directory() const;
```

4 *Returns:* `!root_directory().empty()`.

```
bool has_relative_path() const;
```

5 *Returns:* `!relative_path().empty()`.

```
bool has_parent_path() const;
```

6 *Returns:* `!parent_path().empty()`.

```
bool has_filename() const;
```

7 *Returns:* `!filename().empty()`.

```
bool has_stem() const;
```

8 *Returns:* `!stem().empty()`.

```
bool has_extension() const;
```

9 *Returns:* `!extension().empty()`.

```
bool is_absolute() const;
```

10 *Returns:* `true` if the pathname in the native format contains an absolute path (29.11.7), otherwise `false`.

11 [*Example*: `path("/").is_absolute()` is `true` for POSIX-based operating systems, and `false` for Windows-based operating systems. — *end example*]

```
bool is_relative() const;
```

12 *Returns:* `!is_absolute()`.

29.11.7.4.11 Generation [fs.path.gen]

```
path lexically_normal() const;
```

1 *Returns:* A path whose pathname in the generic format is the normal form (29.11.7.1) of the pathname in the generic format of `*this`.

2 [*Example*:

```
assert(path("foo/./bar/..").lexically_normal() == "foo/");
assert(path("foo/.///bar/../").lexically_normal() == "foo/");
```

The above assertions will succeed. On Windows, the returned path's *directory-separator* characters will be backslashes rather than slashes, but that does not affect `path` equality. — *end example*]

```
path lexically_relative(const path& base) const;
```

3 *Returns:* `*this` made relative to `base`. Does not resolve (29.11.7) symlinks. Does not first normalize (29.11.7.1) `*this` or `base`.

4 *Effects:* If `root_name() != base.root_name()` is `true` or `is_absolute() != base.is_absolute()` is `true` or `!has_root_directory() && base.has_root_directory()` is `true`, returns `path()`. Determines the first mismatched element of `*this` and `base` as if by:

```
auto [a, b] = mismatch(begin(), end(), base.begin(), base.end());
```

Then,

(4.1) — if `a == end()` and `b == base.end()`, returns `path(".")`; otherwise

(4.2) — let `n` be the number of *filename* elements in `[b, base.end())` that are not dot or dot-dot or empty, minus the number that are dot-dot. If `n<0`, returns `path()`; otherwise

(4.3) — if `n == 0` and `(a == end() || a->empty())`, returns `path(".")`; otherwise

(4.4) — returns an object of class `path` that is default-constructed, followed by

(4.4.1) — application of `operator/=(path(".."))` `n` times, and then

(4.4.2) — application of `operator/=` for each element in `[a, end())`.

5 [*Example*:

```
assert(path("/a/d").lexically_relative("/a/b/c") == "../../d");
assert(path("/a/b/c").lexically_relative("/a/d") == "../b/c");
assert(path("a/b/c").lexically_relative("a") == "b/c");
assert(path("a/b/c").lexically_relative("a/b/c/x/y") == "../..");
assert(path("a/b/c").lexically_relative("a/b/c") == ".");
assert(path("a/b").lexically_relative("c/d") == "../../a/b");
```

The above assertions will succeed. On Windows, the returned path's *directory-separator* characters will be backslashes rather than slashes, but that does not affect `path` equality. — *end example*]

6 [*Note*: If symlink following semantics are desired, use the operational function `relative()`. — *end note*]

7 [*Note*: If normalization (29.11.7.1) is needed to ensure consistent matching of elements, apply `lexically_normal()` to `*this`, `base`, or both. — *end note*]

```
path lexically_proximate(const path& base) const;
```

8 *Returns:* If the value of `lexically_relative(base)` is not an empty path, return it. Otherwise return `*this`.

9 [*Note*: If symlink following semantics are desired, use the operational function `proximate()`. — *end note*]

10 [*Note*: If normalization (29.11.7.1) is needed to ensure consistent matching of elements, apply `lexically_normal()` to `*this`, `base`, or both. — *end note*]

29.11.7.5 Iterators [fs.path.itr]

1 Path iterators iterate over the elements of the pathname in the generic format (29.11.7.1).

2 A `path::iterator` is a constant iterator satisfying all the requirements of a bidirectional iterator (**??**) except that, for dereferenceable iterators `a` and `b` of type `path::iterator` with `a == b`, there is no requirement that `*a` and `*b` are bound to the same object. Its `value_type` is `path`.

3 Calling any non-const member function of a `path` object invalidates all iterators referring to elements of that object.

4 For the elements of the pathname in the generic format, the forward traversal order is as follows:

(4.1) — The *root-name* element, if present.

(4.2) — The *root-directory* element, if present. [*Note*: The generic format is required to ensure lexicographical comparison works correctly. — *end note*]

(4.3) — Each successive *filename* element, if present.

(4.4) — An empty element, if a trailing non-root *directory-separator* is present.

5 The backward traversal order is the reverse of forward traversal.

```
iterator begin() const;
```

6 *Returns:* An iterator for the first present element in the traversal list above. If no elements are present, the end iterator.

```
iterator end() const;
```

7 *Returns:* The end iterator.

29.11.7.6 Inserter and extractor [fs.path.io]

```
template<class charT, class traits>
  friend basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const path& p);
```

1 *Effects:* Equivalent to `os << quoted(p.string<charT, traits>())`. [*Note*: The `quoted` function is described in 29.7.8. — *end note*]

2 *Returns:* `os`.

```
template<class charT, class traits>
  friend basic_istream<charT, traits>&
    operator>>(basic_istream<charT, traits>& is, path& p);
```

3 *Effects:* Equivalent to:

```
basic_string<charT, traits> tmp;
is >> quoted(tmp);
p = tmp;
```

4 *Returns:* `is`.

29.11.7.7 Non-member functions [fs.path.nonmember]

```
void swap(path& lhs, path& rhs) noexcept;
```

1 *Effects:* Equivalent to `lhs.swap(rhs)`.

```
size_t hash_value (const path& p) noexcept;
```

2 *Returns:* A hash value for the path `p`. If for two paths, `p1 == p2` then `hash_value(p1) == hash_value(p2)`.

```
friend bool operator==(const path& lhs, const path& rhs) noexcept;
```

3 *Returns:* `!(lhs < rhs) && !(rhs < lhs)`.

4 [*Note*: Path equality and path equivalence have different semantics.

(4.1) — Equality is determined by the `path` non-member `operator==`, which considers the two paths' lexical representations only. [*Example*: `path("foo") == "bar"` is never `true`. — *end example*]

(4.2) — Equivalence is determined by the `equivalent()` non-member function, which determines if two paths resolve (29.11.7) to the same file system entity. [*Example*: `equivalent("foo", "bar")` will be `true` when both paths resolve to the same file. — *end example*]

Programmers wishing to determine if two paths are "the same" must decide if "the same" means "the same representation" or "resolve to the same actual file", and choose the appropriate function accordingly. — *end note*]

```
friend bool operator!=(const path& lhs, const path& rhs) noexcept;
```

5 *Returns:* `!(lhs == rhs)`.

```
friend bool operator< (const path& lhs, const path& rhs) noexcept;
```

6 *Returns:* `lhs.compare(rhs) < 0`.

```
friend bool operator<=(const path& lhs, const path& rhs) noexcept;
```

7 *Returns:* `!(rhs < lhs)`.

```
friend bool operator> (const path& lhs, const path& rhs) noexcept;
```

8 *Returns:* `rhs < lhs`.

```
friend bool operator>=(const path& lhs, const path& rhs) noexcept;
```

9 *Returns:* `!(lhs < rhs)`.

```
friend path operator/ (const path& lhs, const path& rhs);
```

10 *Effects:* Equivalent to: `return path(lhs) /= rhs;`

29.11.8 Class `filesystem_error` [fs.class.filesystem.error]

```
namespace std::filesystem {
  class filesystem_error : public system_error {
  public:
    filesystem_error(const string& what_arg, error_code ec);
    filesystem_error(const string& what_arg,
                     const path& p1, error_code ec);
    filesystem_error(const string& what_arg,
                     const path& p1, const path& p2, error_code ec);

    const path& path1() const noexcept;
    const path& path2() const noexcept;
    const char* what() const noexcept override;
  };
}
```

1 The class `filesystem_error` defines the type of objects thrown as exceptions to report file system errors from functions described in this subclause.

29.11.8.1 Members [fs.filesystem.error.members]

1 Constructors are provided that store zero, one, or two paths associated with an error.

```
filesystem_error(const string& what_arg, error_code ec);
```

2 *Ensures:*

(2.1) — `code() == ec`,

(2.2) — `path1().empty() == true`,

(2.3) — `path2().empty() == true`, and

(2.4) — `string_view(what()).find(what_arg.c_str()) != string_view::npos`.

```
filesystem_error(const string& what_arg, const path& p1, error_code ec);
```

3 *Ensures:*

(3.1) — `code() == ec`,

(3.2) — `path1()` returns a reference to the stored copy of `p1`,

(3.3) — `path2().empty() == true`, and

(3.4) — `string_view(what()).find(what_arg.c_str()) != string_view::npos`.

```
filesystem_error(const string& what_arg, const path& p1, const path& p2, error_code ec);
```

4 *Ensures:*

(4.1) — `code() == ec`,

(4.2) — `path1()` returns a reference to the stored copy of `p1`,

(4.3) — `path2()` returns a reference to the stored copy of `p2`, and

(4.4) — `string_view(what()).find(what_arg.c_str()) != string_view::npos`.

```
const path& path1() const noexcept;
```

5 *Returns:* A reference to the copy of `p1` stored by the constructor, or, if none, an empty path.

```
const path& path2() const noexcept;
```

6 *Returns:* A reference to the copy of `p2` stored by the constructor, or, if none, an empty path.

```
const char* what() const noexcept override;
```

7 *Returns:* An NTBS that incorporates the `what_arg` argument supplied to the constructor. The exact format is unspecified. Implementations should include the `system_error::what()` string and the pathnames of `path1` and `path2` in the native format in the returned string.

29.11.9 Enumerations [fs.enum]

29.11.9.1 Enum `path::format` [fs.enum.path.format]

1 This enum specifies constants used to identify the format of the character sequence, with the meanings listed in Table 116.

Table 116 — Enum `path::format`

Name	Meaning
`native_format`	The native pathname format.
`generic_format`	The generic pathname format.
`auto_format`	The interpretation of the format of the character sequence is implementation-defined. The implementation may inspect the content of the character sequence to determine the format. [*Note*: For POSIX-based systems, native and generic formats are equivalent and the character sequence should always be interpreted in the same way. — *end note*]

29.11.9.2 Enum class `file_type` [fs.enum.file.type]

1 This enum class specifies constants used to identify file types, with the meanings listed in Table 117. The values of the constants are distinct.

Table 117 — Enum class `file_type`

Constant	Meaning
`none`	The type of the file has not been determined or an error occurred while trying to determine the type.
`not_found`	Pseudo-type indicating the file was not found. [*Note*: The file not being found is not considered an error while determining the type of a file. — *end note*]
`regular`	Regular file
`directory`	Directory file
`symlink`	Symbolic link file
`block`	Block special file
`character`	Character special file
`fifo`	FIFO or pipe file
`socket`	Socket file
`implementation-defined`	Implementations that support file systems having file types in addition to the above `file_type` types shall supply implementation-defined `file_type` constants to separately identify each of those additional file types
`unknown`	The file exists but the type could not be determined

29.11.9.3 Enum class `copy_options` [fs.enum.copy.opts]

1 The `enum class` type `copy_options` is a bitmask type (**??**) that specifies bitmask constants used to control the semantics of copy operations. The constants are specified in option groups with the meanings listed in Table 118. The constant `none` represents the empty bitmask, and is shown in each option group for purposes of exposition; implementations shall provide only a single definition. Every other constant in the table represents a distinct bitmask element.

Table 118 — Enum class `copy_options`

Option group controlling `copy_file` function effects for existing target files	
Constant	**Meaning**
`none`	(Default) Error; file already exists.
`skip_existing`	Do not overwrite existing file, do not report an error.
`overwrite_existing`	Overwrite the existing file.
`update_existing`	Overwrite the existing file if it is older than the replacement file.
Option group controlling `copy` function effects for sub-directories	
Constant	**Meaning**
`none`	(Default) Do not copy sub-directories.
`recursive`	Recursively copy sub-directories and their contents.
Option group controlling `copy` function effects for symbolic links	
Constant	**Meaning**
`none`	(Default) Follow symbolic links.
`copy_symlinks`	Copy symbolic links as symbolic links rather than copying the files that they point to.
`skip_symlinks`	Ignore symbolic links.
Option group controlling `copy` function effects for choosing the form of copying	
Constant	**Meaning**
`none`	(Default) Copy content.
`directories_only`	Copy directory structure only, do not copy non-directory files.
`create_symlinks`	Make symbolic links instead of copies of files. The source path shall be an absolute path unless the destination path is in the current directory.
`create_hard_links`	Make hard links instead of copies of files.

29.11.9.4 Enum class `perms` [fs.enum.perms]

1 The `enum class` type `perms` is a bitmask type (**??**) that specifies bitmask constants used to identify file permissions, with the meanings listed in Table 119.

29.11.9.5 Enum class `perm_options` [fs.enum.perm.opts]

1 The `enum class` type `perm_options` is a bitmask type (**??**) that specifies bitmask constants used to control the semantics of permissions operations, with the meanings listed in Table 120. The bitmask constants are bitmask elements. In Table 120 `perm` denotes a value of type `perms` passed to `permissions`.

29.11.9.6 Enum class `directory_options` [fs.enum.dir.opts]

1 The `enum class` type `directory_options` is a bitmask type (**??**) that specifies bitmask constants used to identify directory traversal options, with the meanings listed in Table 121. The constant `none` represents the empty bitmask; every other constant in the table represents a distinct bitmask element.

29.11.10 Class `file_status` [fs.class.file.status]

```
namespace std::filesystem {
  class file_status {
  public:
    // 29.11.10.1, constructors and destructor
    file_status() noexcept : file_status(file_type::none) {}
    explicit file_status(file_type ft,
                         perms prms = perms::unknown) noexcept;
```

Table 119 — Enum class `perms`

Name	Value (octal)	POSIX macro	Definition or notes
`none`	0		There are no permissions set for the file.
`owner_read`	0400	`S_IRUSR`	Read permission, owner
`owner_write`	0200	`S_IWUSR`	Write permission, owner
`owner_exec`	0100	`S_IXUSR`	Execute/search permission, owner
`owner_all`	0700	`S_IRWXU`	Read, write, execute/search by owner; `owner_read | owner_write | owner_exec`
`group_read`	040	`S_IRGRP`	Read permission, group
`group_write`	020	`S_IWGRP`	Write permission, group
`group_exec`	010	`S_IXGRP`	Execute/search permission, group
`group_all`	070	`S_IRWXG`	Read, write, execute/search by group; `group_read | group_write | group_exec`
`others_read`	04	`S_IROTH`	Read permission, others
`others_write`	02	`S_IWOTH`	Write permission, others
`others_exec`	01	`S_IXOTH`	Execute/search permission, others
`others_all`	07	`S_IRWXO`	Read, write, execute/search by others; `others_read | others_write | others_exec`
`all`	0777		`owner_all | group_all | others_all`
`set_uid`	04000	`S_ISUID`	Set-user-ID on execution
`set_gid`	02000	`S_ISGID`	Set-group-ID on execution
`sticky_bit`	01000	`S_ISVTX`	Operating system dependent.
`mask`	07777		`all | set_uid | set_gid | sticky_bit`
`unknown`	0xFFFF		The permissions are not known, such as when a `file_status` object is created without specifying the permissions

Table 120 — Enum class `perm_options`

Name	Meaning
`replace`	`permissions` shall replace the file's permission bits with `perm`
`add`	`permissions` shall replace the file's permission bits with the bitwise OR of `perm` and the file's current permission bits.
`remove`	`permissions` shall replace the file's permission bits with the bitwise AND of the complement of `perm` and the file's current permission bits.
`nofollow`	`permissions` shall change the permissions of a symbolic link itself rather than the permissions of the file the link resolves to.

Table 121 — Enum class `directory_options`

Name	Meaning
`none`	(Default) Skip directory symlinks, permission denied is an error.
`follow_directory_symlink`	Follow rather than skip directory symlinks.
`skip_permission_denied`	Skip directories that would otherwise result in permission denied.

```
    file_status(const file_status&) noexcept = default;
    file_status(file_status&&) noexcept = default;
    ~file_status();

    // assignments
    file_status& operator=(const file_status&) noexcept = default;
    file_status& operator=(file_status&&) noexcept = default;

    // 29.11.10.3, modifiers
    void       type(file_type ft) noexcept;
    void       permissions(perms prms) noexcept;

    // 29.11.10.2, observers
    file_type  type() const noexcept;
    perms      permissions() const noexcept;
  };
}
```

1 An object of type `file_status` stores information about the type and permissions of a file.

29.11.10.1 Constructors [fs.file.status.cons]

```
explicit file_status(file_type ft, perms prms = perms::unknown) noexcept;
```

1 *Ensures:* `type() == ft` and `permissions() == prms`.

29.11.10.2 Observers [fs.file.status.obs]

```
file_type type() const noexcept;
```

1 *Returns:* The value of `type()` specified by the postconditions of the most recent call to a constructor, `operator=`, or `type(file_type)` function.

```
perms permissions() const noexcept;
```

2 *Returns:* The value of `permissions()` specified by the postconditions of the most recent call to a constructor, `operator=`, or `permissions(perms)` function.

29.11.10.3 Modifiers [fs.file.status.mods]

```
void type(file_type ft) noexcept;
```

1 *Ensures:* `type() == ft`.

```
void permissions(perms prms) noexcept;
```

2 *Ensures:* `permissions() == prms`.

29.11.11 Class `directory_entry` [fs.class.directory.entry]

```
namespace std::filesystem {
  class directory_entry {
  public:
    // 29.11.11.1, constructors and destructor
    directory_entry() noexcept = default;
    directory_entry(const directory_entry&) = default;
    directory_entry(directory_entry&&) noexcept = default;
    explicit directory_entry(const filesystem::path& p);
    directory_entry(const filesystem::path& p, error_code& ec);
    ~directory_entry();

    // assignments
    directory_entry& operator=(const directory_entry&) = default;
    directory_entry& operator=(directory_entry&&) noexcept = default;

    // 29.11.11.2, modifiers
    void assign(const filesystem::path& p);
    void assign(const filesystem::path& p, error_code& ec);
    void replace_filename(const filesystem::path& p);
```

```
    void replace_filename(const filesystem::path& p, error_code& ec);
    void refresh();
    void refresh(error_code& ec) noexcept;

    // 29.11.11.3, observers
    const filesystem::path& path() const noexcept;
    operator const filesystem::path&() const noexcept;
    bool exists() const;
    bool exists(error_code& ec) const noexcept;
    bool is_block_file() const;
    bool is_block_file(error_code& ec) const noexcept;
    bool is_character_file() const;
    bool is_character_file(error_code& ec) const noexcept;
    bool is_directory() const;
    bool is_directory(error_code& ec) const noexcept;
    bool is_fifo() const;
    bool is_fifo(error_code& ec) const noexcept;
    bool is_other() const;
    bool is_other(error_code& ec) const noexcept;
    bool is_regular_file() const;
    bool is_regular_file(error_code& ec) const noexcept;
    bool is_socket() const;
    bool is_socket(error_code& ec) const noexcept;
    bool is_symlink() const;
    bool is_symlink(error_code& ec) const noexcept;
    uintmax_t file_size() const;
    uintmax_t file_size(error_code& ec) const noexcept;
    uintmax_t hard_link_count() const;
    uintmax_t hard_link_count(error_code& ec) const noexcept;
    file_time_type last_write_time() const;
    file_time_type last_write_time(error_code& ec) const noexcept;
    file_status status() const;
    file_status status(error_code& ec) const noexcept;
    file_status symlink_status() const;
    file_status symlink_status(error_code& ec) const noexcept;

    bool operator==(const directory_entry& rhs) const noexcept;
    bool operator!=(const directory_entry& rhs) const noexcept;
    bool operator< (const directory_entry& rhs) const noexcept;
    bool operator> (const directory_entry& rhs) const noexcept;
    bool operator<=(const directory_entry& rhs) const noexcept;
    bool operator>=(const directory_entry& rhs) const noexcept;

  private:
    filesystem::path pathobject;     // exposition only
    friend class directory_iterator; // exposition only
  };
}
```

1 A `directory_entry` object stores a `path` object and may store additional objects for file attributes such as hard link count, status, symlink status, file size, and last write time.

2 Implementations should store such additional file attributes during directory iteration if their values are available and storing the values would allow the implementation to eliminate file system accesses by `directory_entry` observer functions (29.11.14). Such stored file attribute values are said to be *cached*.

3 [*Note*: For purposes of exposition, class `directory_iterator` (29.11.12) is shown above as a friend of class `directory_entry`. Friendship allows the `directory_iterator` implementation to cache already available attribute values directly into a `directory_entry` object without the cost of an unneeded call to `refresh()`. — *end note*]

4 [*Example*:

```
using namespace std::filesystem;
```

```
// use possibly cached last write time to minimize disk accesses
for (auto&& x : directory_iterator("."))
{
  std::cout << x.path() << " " << x.last_write_time() << std::endl;
}

// call refresh() to refresh a stale cache
for (auto&& x : directory_iterator("."))
{
  lengthy_function(x.path());  // cache becomes stale
  x.refresh();
  std::cout << x.path() << " " << x.last_write_time() << std::endl;
}
```

On implementations that do not cache the last write time, both loops will result in a potentially expensive call to the `std::filesystem::last_write_time` function. On implementations that do cache the last write time, the first loop will use the cached value and so will not result in a potentially expensive call to the `std::filesystem::last_write_time` function. The code is portable to any implementation, regardless of whether or not it employs caching. — *end example*]

29.11.11.1 Constructors [fs.dir.entry.cons]

```
explicit directory_entry(const filesystem::path& p);
directory_entry(const filesystem::path& p, error_code& ec);
```

1 *Effects:* Constructs an object of type `directory_entry`, then `refresh()` or `refresh(ec)`, respectively.

2 *Ensures:* `path() == p` if no error occurs, otherwise `path() == filesystem::path()`.

3 *Throws:* As specified in 29.11.6.

29.11.11.2 Modifiers [fs.dir.entry.mods]

```
void assign(const filesystem::path& p);
void assign(const filesystem::path& p, error_code& ec);
```

1 *Effects:* Equivalent to `pathobject = p`, then `refresh()` or `refresh(ec)`, respectively. If an error occurs, the values of any cached attributes are unspecified.

2 *Throws:* As specified in 29.11.6.

```
void replace_filename(const filesystem::path& p);
void replace_filename(const filesystem::path& p, error_code& ec);
```

3 *Effects:* Equivalent to `pathobject.replace_filename(p)`, then `refresh()` or `refresh(ec)`, respectively. If an error occurs, the values of any cached attributes are unspecified.

Throws: As specified in 29.11.6.

```
void refresh();
void refresh(error_code& ec) noexcept;
```

4 *Effects:* Stores the current values of any cached attributes of the file `p` resolves to. If an error occurs, an error is reported (29.11.6) and the values of any cached attributes are unspecified.

5 *Throws:* As specified in 29.11.6.

6 [*Note*: Implementations of `directory_iterator` (29.11.12) are prohibited from directly or indirectly calling the `refresh` function since it must access the external file system, and the objective of caching is to avoid unnecessary file system accesses. — *end note*]

29.11.11.3 Observers [fs.dir.entry.obs]

1 Unqualified function names in the *Returns:* elements of the `directory_entry` observers described below refer to members of the `std::filesystem` namespace.

```
const filesystem::path& path() const noexcept;
operator const filesystem::path&() const noexcept;
```

2 *Returns:* `pathobject`.

```
bool exists() const;
bool exists(error_code& ec) const noexcept;
```

3 *Returns:* `exists(this->status())` or `exists(this->status(ec))`, respectively.

4 *Throws:* As specified in 29.11.6.

```
bool is_block_file() const;
bool is_block_file(error_code& ec) const noexcept;
```

5 *Returns:* `is_block_file(this->status())` or `is_block_file(this->status(ec))`, respectively.

6 *Throws:* As specified in 29.11.6.

```
bool is_character_file() const;
bool is_character_file(error_code& ec) const noexcept;
```

7 *Returns:* `is_character_file(this->status())` or `is_character_file(this->status(ec))`, respectively.

8 *Throws:* As specified in 29.11.6.

```
bool is_directory() const;
bool is_directory(error_code& ec) const noexcept;
```

9 *Returns:* `is_directory(this->status())` or `is_directory(this->status(ec))`, respectively.

10 *Throws:* As specified in 29.11.6.

```
bool is_fifo() const;
bool is_fifo(error_code& ec) const noexcept;
```

11 *Returns:* `is_fifo(this->status())` or `is_fifo(this->status(ec))`, respectively.

12 *Throws:* As specified in 29.11.6.

```
bool is_other() const;
bool is_other(error_code& ec) const noexcept;
```

13 *Returns:* `is_other(this->status())` or `is_other(this->status(ec))`, respectively.

14 *Throws:* As specified in 29.11.6.

```
bool is_regular_file() const;
bool is_regular_file(error_code& ec) const noexcept;
```

15 *Returns:* `is_regular_file(this->status())` or `is_regular_file(this->status(ec))`, respectively.

16 *Throws:* As specified in 29.11.6.

```
bool is_socket() const;
bool is_socket(error_code& ec) const noexcept;
```

17 *Returns:* `is_socket(this->status())` or `is_socket(this->status(ec))`, respectively.

18 *Throws:* As specified in 29.11.6.

```
bool is_symlink() const;
bool is_symlink(error_code& ec) const noexcept;
```

19 *Returns:* `is_symlink(this->symlink_status())` or `is_symlink(this->symlink_status(ec))`, respectively.

20 *Throws:* As specified in 29.11.6.

```
uintmax_t file_size() const;
uintmax_t file_size(error_code& ec) const noexcept;
```

21 *Returns:* If cached, the file size attribute value. Otherwise, `file_size(path())` or `file_size(path(), ec)`, respectively.

22 *Throws:* As specified in 29.11.6.

```
uintmax_t hard_link_count() const;
```

```
uintmax_t hard_link_count(error_code& ec) const noexcept;
```

23 *Returns:* If cached, the hard link count attribute value. Otherwise, `hard_link_count(path())` or `hard_link_count(path(), ec)`, respectively.

24 *Throws:* As specified in 29.11.6.

```
file_time_type last_write_time() const;
file_time_type last_write_time(error_code& ec) const noexcept;
```

25 *Returns:* If cached, the last write time attribute value. Otherwise, `last_write_time(path())` or `last_write_time(path(), ec)`, respectively.

26 *Throws:* As specified in 29.11.6.

```
file_status status() const;
file_status status(error_code& ec) const noexcept;
```

27 *Returns:* If cached, the status attribute value. Otherwise, `status(path())` or `status(path(), ec)`, respectively.

28 *Throws:* As specified in 29.11.6.

```
file_status symlink_status() const;
file_status symlink_status(error_code& ec) const noexcept;
```

29 *Returns:* If cached, the symlink status attribute value. Otherwise, `symlink_status(path())` or `symlink_status(path(), ec)`, respectively.

30 *Throws:* As specified in 29.11.6.

```
bool operator==(const directory_entry& rhs) const noexcept;
```

31 *Returns:* `pathobject == rhs.pathobject`.

```
bool operator!=(const directory_entry& rhs) const noexcept;
```

32 *Returns:* `pathobject != rhs.pathobject`.

```
bool operator< (const directory_entry& rhs) const noexcept;
```

33 *Returns:* `pathobject < rhs.pathobject`.

```
bool operator> (const directory_entry& rhs) const noexcept;
```

34 *Returns:* `pathobject > rhs.pathobject`.

```
bool operator<=(const directory_entry& rhs) const noexcept;
```

35 *Returns:* `pathobject <= rhs.pathobject`.

```
bool operator>=(const directory_entry& rhs) const noexcept;
```

36 *Returns:* `pathobject >= rhs.pathobject`.

29.11.12 Class `directory_iterator` [fs.class.directory.iterator]

1 An object of type `directory_iterator` provides an iterator for a sequence of `directory_entry` elements representing the path and any cached attribute values (29.11.11) for each file in a directory or in an implementation-defined directory-like file type. [*Note*: For iteration into sub-directories, see class `recursive_directory_iterator` (29.11.13). — *end note*]

```
namespace std::filesystem {
  class directory_iterator {
  public:
    using iterator_category = input_iterator_tag;
    using value_type        = directory_entry;
    using difference_type   = ptrdiff_t;
    using pointer           = const directory_entry*;
    using reference         = const directory_entry&;

    // 29.11.12.1, member functions
    directory_iterator() noexcept;
    explicit directory_iterator(const path& p);
```

```
    directory_iterator(const path& p, directory_options options);
    directory_iterator(const path& p, error_code& ec);
    directory_iterator(const path& p, directory_options options,
                       error_code& ec);
    directory_iterator(const directory_iterator& rhs);
    directory_iterator(directory_iterator&& rhs) noexcept;
    ~directory_iterator();

    directory_iterator& operator=(const directory_iterator& rhs);
    directory_iterator& operator=(directory_iterator&& rhs) noexcept;

    const directory_entry& operator*() const;
    const directory_entry* operator->() const;
    directory_iterator&    operator++();
    directory_iterator&    increment(error_code& ec);

    // other members as required by ??, input iterators
  };
}
```

2 `directory_iterator` ~~satisfies~~meets the *Cpp17InputIterator* requirements (**??**).

3 If an iterator of type `directory_iterator` reports an error or is advanced past the last directory element, that iterator shall become equal to the end iterator value. The `directory_iterator` default constructor shall create an iterator equal to the end iterator value, and this shall be the only valid iterator for the end condition.

4 The end iterator is not dereferenceable.

5 Two end iterators are always equal. An end iterator shall not be equal to a non-end iterator.

6 The result of calling the `path()` member of the `directory_entry` object obtained by dereferencing a `directory_iterator` is a reference to a `path` object composed of the directory argument from which the iterator was constructed with filename of the directory entry appended as if by `operator/=`.

7 Directory iteration shall not yield directory entries for the current (dot) and parent (dot-dot) directories.

8 The order of directory entries obtained by dereferencing successive increments of a `directory_iterator` is unspecified.

9 Constructors and non-const `directory_iterator` member functions store the values of any cached attributes (29.11.11) in the `directory_entry` element returned by `operator*()`. `directory_iterator` member functions shall not directly or indirectly call any `directory_entry` `refresh` function. [*Note*: The exact mechanism for storing cached attribute values is not exposed to users. For exposition, class `directory_iterator` is shown in 29.11.11 as a friend of class `directory_entry`. — *end note*]

10 [*Note*: Programs performing directory iteration may wish to test if the path obtained by dereferencing a directory iterator actually exists. It could be a symbolic link to a non-existent file. Programs recursively walking directory trees for purposes of removing and renaming entries may wish to avoid following symbolic links. — *end note*]

11 [*Note*: If a file is removed from or added to a directory after the construction of a `directory_iterator` for the directory, it is unspecified whether or not subsequently incrementing the iterator will ever result in an iterator referencing the removed or added directory entry. See POSIX `readdir_r`. — *end note*]

29.11.12.1 Members [fs.dir.itr.members]

```
directory_iterator() noexcept;
```

1 *Effects:* Constructs the end iterator.

```
explicit directory_iterator(const path& p);
directory_iterator(const path& p, directory_options options);
directory_iterator(const path& p, error_code& ec);
directory_iterator(const path& p, directory_options options, error_code& ec);
```

2 *Effects:* For the directory that `p` resolves to, constructs an iterator for the first element in a sequence of `directory_entry` elements representing the files in the directory, if any; otherwise the end iterator. However, if

`(options & directory_options::skip_permission_denied) != directory_options::none`

and construction encounters an error indicating that permission to access `p` is denied, constructs the end iterator and does not report an error.

3 *Throws:* As specified in 29.11.6.

4 [*Note*: To iterate over the current directory, use `directory_iterator(".")` rather than `directory_iterator("")`. — *end note*]

```
directory_iterator(const directory_iterator& rhs);
directory_iterator(directory_iterator&& rhs) noexcept;
```

5 *Effects:* Constructs an object of class `directory_iterator`.

6 *Ensures:* `*this` has the original value of `rhs`.

```
directory_iterator& operator=(const directory_iterator& rhs);
directory_iterator& operator=(directory_iterator&& rhs) noexcept;
```

7 *Effects:* If `*this` and `rhs` are the same object, the member has no effect.

8 *Ensures:* `*this` has the original value of `rhs`.

9 *Returns:* `*this`.

```
directory_iterator& operator++();
directory_iterator& increment(error_code& ec);
```

10 *Effects:* As specified for the prefix increment operation of Input iterators (**??**).

11 *Returns:* `*this`.

12 *Throws:* As specified in 29.11.6.

29.11.12.2 Non-member functions [fs.dir.itr.nonmembers]

1 These functions enable range access for `directory_iterator`.

```
directory_iterator begin(directory_iterator iter) noexcept;
```

2 *Returns:* `iter`.

```
directory_iterator end(const directory_iterator&) noexcept;
```

3 *Returns:* `directory_iterator()`.

29.11.13 Class `recursive_directory_iterator` [fs.class.rec.dir.itr]

1 An object of type `recursive_directory_iterator` provides an iterator for a sequence of `directory_entry` elements representing the files in a directory or in an implementation-defined directory-like file type, and its sub-directories.

```
namespace std::filesystem {
  class recursive_directory_iterator {
  public:
    using iterator_category = input_iterator_tag;
    using value_type        = directory_entry;
    using difference_type   = ptrdiff_t;
    using pointer           = const directory_entry*;
    using reference         = const directory_entry&;

    // 29.11.13.1, constructors and destructor
    recursive_directory_iterator() noexcept;
    explicit recursive_directory_iterator(const path& p);
    recursive_directory_iterator(const path& p, directory_options options);
    recursive_directory_iterator(const path& p, directory_options options,
                                 error_code& ec);
    recursive_directory_iterator(const path& p, error_code& ec);
    recursive_directory_iterator(const recursive_directory_iterator& rhs);
    recursive_directory_iterator(recursive_directory_iterator&& rhs) noexcept;
    ~recursive_directory_iterator();
```

```
    // 29.11.13.1, observers
    directory_options  options() const;
    int                depth() const;
    bool               recursion_pending() const;

    const directory_entry& operator*() const;
    const directory_entry* operator->() const;

    // 29.11.13.1, modifiers
    recursive_directory_iterator&
      operator=(const recursive_directory_iterator& rhs);
    recursive_directory_iterator&
      operator=(recursive_directory_iterator&& rhs) noexcept;

    recursive_directory_iterator& operator++();
    recursive_directory_iterator& increment(error_code& ec);

    void pop();
    void pop(error_code& ec);
    void disable_recursion_pending();

    // other members as required by ??, input iterators
  };
}
```

2 Calling `options`, `depth`, `recursion_pending`, `pop` or `disable_recursion_pending` on an iterator that is not dereferenceable results in undefined behavior.

3 The behavior of a `recursive_directory_iterator` is the same as a `directory_iterator` unless otherwise specified.

4 [*Note*: If the directory structure being iterated over contains cycles then the end iterator may be unreachable. — *end note*]

29.11.13.1 Members [fs.rec.dir.itr.members]

```
recursive_directory_iterator() noexcept;
```

1 *Effects:* Constructs the end iterator.

```
explicit recursive_directory_iterator(const path& p);
recursive_directory_iterator(const path& p, directory_options options);
recursive_directory_iterator(const path& p, directory_options options, error_code& ec);
recursive_directory_iterator(const path& p, error_code& ec);
```

2 *Effects:* Constructs an iterator representing the first entry in the directory to which `p` resolves, if any; otherwise, the end iterator. However, if

```
(options & directory_options::skip_permission_denied) != directory_options::none
```

and construction encounters an error indicating that permission to access `p` is denied, constructs the end iterator and does not report an error.

3 *Ensures:* `options() == options` for the signatures with a `directory_options` argument, otherwise `options() == directory_options::none`.

4 *Throws:* As specified in 29.11.6.

5 [*Note*: To iterate over the current directory, use `recursive_directory_iterator(".")` rather than `recursive_directory_iterator("")`. — *end note*]

6 [*Note*: By default, `recursive_directory_iterator` does not follow directory symlinks. To follow directory symlinks, specify `options` as `directory_options::follow_directory_symlink` — *end note*]

```
recursive_directory_iterator(const recursive_directory_iterator& rhs);
```

7 *Effects:* Constructs an object of class `recursive_directory_iterator`.

8 *Ensures:*

(8.1) — `options() == rhs.options()`

(8.2) — `depth() == rhs.depth()`

(8.3) — `recursion_pending() == rhs.recursion_pending()`

```
recursive_directory_iterator(recursive_directory_iterator&& rhs) noexcept;
```

9 *Effects:* Constructs an object of class `recursive_directory_iterator`.

10 *Ensures:* `options()`, `depth()`, and `recursion_pending()` have the values that `rhs.options()`, `rhs.depth()`, and `rhs.recursion_pending()`, respectively, had before the function call.

```
recursive_directory_iterator& operator=(const recursive_directory_iterator& rhs);
```

11 *Effects:* If `*this` and `rhs` are the same object, the member has no effect.

12 *Ensures:*

(12.1) — `options() == rhs.options()`

(12.2) — `depth() == rhs.depth()`

(12.3) — `recursion_pending() == rhs.recursion_pending()`

13 *Returns:* `*this`.

```
recursive_directory_iterator& operator=(recursive_directory_iterator&& rhs) noexcept;
```

14 *Effects:* If `*this` and `rhs` are the same object, the member has no effect.

15 *Ensures:* `options()`, `depth()`, and `recursion_pending()` have the values that `rhs.options()`, `rhs.depth()`, and `rhs.recursion_pending()`, respectively, had before the function call.

16 *Returns:* `*this`.

```
directory_options options() const;
```

17 *Returns:* The value of the argument passed to the constructor for the `options` parameter, if present, otherwise `directory_options::none`.

18 *Throws:* Nothing.

```
int depth() const;
```

19 *Returns:* The current depth of the directory tree being traversed. [*Note*: The initial directory is depth `0`, its immediate subdirectories are depth `1`, and so forth. — *end note*]

20 *Throws:* Nothing.

```
bool recursion_pending() const;
```

21 *Returns:* `true` if `disable_recursion_pending()` has not been called subsequent to the prior construction or increment operation, otherwise `false`.

22 *Throws:* Nothing.

```
recursive_directory_iterator& operator++();
recursive_directory_iterator& increment(error_code& ec);
```

23 *Effects:* As specified for the prefix increment operation of Input iterators (**??**), except that:

(23.1) — If there are no more entries at the current depth, then if `depth() != 0` iteration over the parent directory resumes; otherwise `*this = recursive_directory_iterator()`.

(23.2) — Otherwise if

```
recursion_pending() && is_directory((*this)->status()) &&
(!is_symlink((*this)->symlink_status()) ||
 (options() & directory_options::follow_directory_symlink) != directory_options::none)
```

then either directory `(*this)->path()` is recursively iterated into or, if

```
(options() & directory_options::skip_permission_denied) != directory_options::none
```

and an error occurs indicating that permission to access directory `(*this)->path()` is denied, then directory `(*this)->path()` is treated as an empty directory and no error is reported.

24 *Returns:* `*this`.

25 *Throws:* As specified in 29.11.6.

```
void pop();
void pop(error_code& ec);
```

26 *Effects:* If `depth() == 0`, set `*this` to `recursive_directory_iterator()`. Otherwise, cease iteration of the directory currently being iterated over, and continue iteration over the parent directory.

27 *Throws:* As specified in 29.11.6.

28 *Remarks:* Any copies of the previous value of `*this` are no longer required to be dereferenceable nor to be in the domain of `==`.

```
void disable_recursion_pending();
```

29 *Ensures:* `recursion_pending() == false`.

30 [*Note*: `disable_recursion_pending()` is used to prevent unwanted recursion into a directory. — *end note*]

29.11.13.2 Non-member functions [fs.rec.dir.itr.nonmembers]

1 These functions enable use of `recursive_directory_iterator` with range-based for statements.

```
recursive_directory_iterator begin(recursive_directory_iterator iter) noexcept;
```

2 *Returns:* `iter`.

```
recursive_directory_iterator end(const recursive_directory_iterator&) noexcept;
```

3 *Returns:* `recursive_directory_iterator()`.

29.11.14 Filesystem operation functions [fs.op.funcs]

1 Filesystem operation functions query or modify files, including directories, in external storage.

2 [*Note*: Because hardware failures, network failures, file system races (29.11.2.3), and many other kinds of errors occur frequently in file system operations, users should be aware that any filesystem operation function, no matter how apparently innocuous, may encounter an error; see 29.11.6. — *end note*]

29.11.14.1 Absolute [fs.op.absolute]

```
path absolute(const path& p);
path absolute(const path& p, error_code& ec);
```

1 *Effects:* Composes an absolute path referencing the same file system location as `p` according to the operating system (29.11.2.2).

2 *Returns:* The composed path. The signature with argument `ec` returns `path()` if an error occurs.

3 [*Note*: For the returned path, `rp`, `rp.is_absolute()` is `true` unless an error occurs. — *end note*]

4 *Throws:* As specified in 29.11.6.

5 [*Note*: To resolve symlinks, or perform other sanitization which might require queries to secondary storage, such as hard disks, consider `canonical` (29.11.14.2). — *end note*]

6 [*Note*: Implementations are strongly encouraged to not query secondary storage, and not consider `!exists(p)` an error. — *end note*]

7 [*Example*: For POSIX-based operating systems, `absolute(p)` is simply `current_path()/p`. For Windows-based operating systems, `absolute` might have the same semantics as `GetFullPathNameW`. — *end example*]

29.11.14.2 Canonical [fs.op.canonical]

```
path canonical(const path& p);
path canonical(const path& p, error_code& ec);
```

1 *Effects:* Converts `p` to an absolute path that has no symbolic link, dot, or dot-dot elements in its pathname in the generic format.

2 *Returns:* A path that refers to the same file system object as `absolute(p)`. The signature with argument `ec` returns `path()` if an error occurs.

3 *Throws:* As specified in 29.11.6.

4 *Remarks:* `!exists(p)` is an error.

29.11.14.3 Copy [fs.op.copy]

```
void copy(const path& from, const path& to);
```

1 *Effects:* Equivalent to `copy(from, to, copy_options::none)`.

```
void copy(const path& from, const path& to, error_code& ec);
```

2 *Effects:* Equivalent to `copy(from, to, copy_options::none, ec)`.

```
void copy(const path& from, const path& to, copy_options options);
void copy(const path& from, const path& to, copy_options options,
     error_code& ec);
```

3 ~~*Requires:*~~ *Expects:* At most one element from each option group (29.11.9.3) is set in `options`.

4 *Effects:* Before the first use of `f` and `t`:

(4.1) — If

```
(options & copy_options::create_symlinks) != copy_options::none ||
(options & copy_options::skip_symlinks) != copy_options::none
```

then `auto f = symlink_status(from)` and if needed `auto t = symlink_status(to)`.

(4.2) — Otherwise, if

```
(options & copy_options::copy_symlinks) != copy_options::none
```

then `auto f = symlink_status(from)` and if needed `auto t = status(to)`.

(4.3) — Otherwise, `auto f = status(from)` and if needed `auto t = status(to)`.

Effects are then as follows:

(4.4) — If `f.type()` or `t.type()` is an implementation-defined file type (29.11.9.2), then the effects are implementation-defined.

(4.5) — Otherwise, an error is reported as specified in 29.11.6 if:

(4.5.1) — `exists(f)` is `false`, or

(4.5.2) — `equivalent(from, to)` is `true`, or

(4.5.3) — `is_other(f) || is_other(t)` is `true`, or

(4.5.4) — `is_directory(f) && is_regular_file(t)` is `true`.

(4.6) — Otherwise, if `is_symlink(f)`, then:

(4.6.1) — If `(options & copy_options::skip_symlinks) != copy_options::none` then return.

(4.6.2) — Otherwise if

```
!exists(t) && (options & copy_options::copy_symlinks) != copy_options::none
```

then `copy_symlink(from, to)`.

(4.6.3) — Otherwise report an error as specified in 29.11.6.

(4.7) — Otherwise, if `is_regular_file(f)`, then:

(4.7.1) — If `(options & copy_options::directories_only) != copy_options::none`, then return.

(4.7.2) — Otherwise, if `(options & copy_options::create_symlinks) != copy_options::none`, then create a symbolic link to the source file.

(4.7.3) — Otherwise, if `(options & copy_options::create_hard_links) != copy_options::none`, then create a hard link to the source file.

(4.7.4) — Otherwise, if `is_directory(t)`, then `copy_file(from, to/from.filename(), options)`.

(4.7.5) — Otherwise, `copy_file(from, to, options)`.

(4.8) — Otherwise, if

```
is_directory(f) &&
(options & copy_options::create_symlinks) != copy_options::none
```

then report an error with an `error_code` argument equal to `make_error_code(errc::is_a_directory)`.

(4.9) — Otherwise, if

```
is_directory(f) &&
((options & copy_options::recursive) != copy_options::none ||
 options == copy_options::none)
```

then:

(4.9.1) — If `exists(t)` is `false`, then `create_directory(to, from)`.

(4.9.2) — Then, iterate over the files in `from`, as if by

```
for (const directory_entry& x : directory_iterator(from))
  copy(x.path(), to/x.path().filename(),
       options | copy_options::in-recursive-copy);
```

where *`in-recursive-copy`* is a bitmask element of `copy_options` that is not one of the elements in 29.11.9.3.

(4.10) — Otherwise, for the signature with argument `ec`, `ec.clear()`.

(4.11) — Otherwise, no effects.

5 *Throws:* As specified in 29.11.6.

6 *Remarks:* For the signature with argument `ec`, any library functions called by the implementation shall have an `error_code` argument if applicable.

7 [*Example*: Given this directory structure:

```
/dir1
  file1
  file2
  dir2
    file3
```

Calling `copy("/dir1", "/dir3")` would result in:

```
/dir1
  file1
  file2
  dir2
    file3
/dir3
  file1
  file2
```

Alternatively, calling `copy("/dir1", "/dir3", copy_options::recursive)` would result in:

```
/dir1
  file1
  file2
  dir2
    file3
/dir3
  file1
  file2
  dir2
    file3
```

— *end example*]

29.11.14.4 Copy file [fs.op.copy.file]

```
bool copy_file(const path& from, const path& to);
bool copy_file(const path& from, const path& to, error_code& ec);
```

1 *Returns:* `copy_file(from, to, copy_options::none)` or `copy_file(from, to, copy_options::none, ec)`, respectively.

2 *Throws:* As specified in 29.11.6.

```
bool copy_file(const path& from, const path& to, copy_options options);
```

```
bool copy_file(const path& from, const path& to, copy_options options,
               error_code& ec);
```

3 ~~*Requires:*~~ *Expects:* At most one element from each option group (29.11.9.3) is set in `options`.

4 *Effects:* As follows:

(4.1) — Report an error as specified in 29.11.6 if:

(4.1.1) — `is_regular_file(from)` is `false`, or

(4.1.2) — `exists(to)` is `true` and `is_regular_file(to)` is `false`, or

(4.1.3) — `exists(to)` is `true` and `equivalent(from, to)` is `true`, or

(4.1.4) — `exists(to)` is `true` and

```
(options & (copy_options::skip_existing |
            copy_options::overwrite_existing |
            copy_options::update_existing)) == copy_options::none
```

(4.2) — Otherwise, copy the contents and attributes of the file `from` resolves to, to the file `to` resolves to, if:

(4.2.1) — `exists(to)` is `false`, or

(4.2.2) — `(options & copy_options::overwrite_existing) != copy_options::none`, or

(4.2.3) — `(options & copy_options::update_existing) != copy_options::none` and `from` is more recent than `to`, determined as if by use of the `last_write_time` function (29.11.14.25).

(4.3) — Otherwise, no effects.

5 *Returns:* `true` if the `from` file was copied, otherwise `false`. The signature with argument `ec` returns `false` if an error occurs.

6 *Throws:* As specified in 29.11.6.

7 *Complexity:* At most one direct or indirect invocation of `status(to)`.

29.11.14.5 Copy symlink [fs.op.copy.symlink]

```
void copy_symlink(const path& existing_symlink, const path& new_symlink);
void copy_symlink(const path& existing_symlink, const path& new_symlink,
                  error_code& ec) noexcept;
```

1 *Effects:* Equivalent to *`function`*`(read_symlink(existing_symlink), new_symlink)` or *`function`*`(read_symlink(existing_symlink, ec), new_symlink, ec)`, respectively, where in each case *`function`* is `create_symlink` or `create_directory_symlink` as appropriate.

2 *Throws:* As specified in 29.11.6.

29.11.14.6 Create directories [fs.op.create.directories]

```
bool create_directories(const path& p);
bool create_directories(const path& p, error_code& ec);
```

1 *Effects:* Calls `create_directory()` for each element of `p` that does not exist.

2 *Returns:* `true` if a new directory was created for the directory `p` resolves to, otherwise `false`.

3 *Throws:* As specified in 29.11.6.

4 *Complexity:* $\mathcal{O}(n)$ where n is the number of elements of `p`.

29.11.14.7 Create directory [fs.op.create.directory]

```
bool create_directory(const path& p);
bool create_directory(const path& p, error_code& ec) noexcept;
```

1 *Effects:* Creates the directory `p` resolves to, as if by POSIX `mkdir` with a second argument of `static_cast<int>(perms::all)`. If `mkdir` fails because `p` resolves to an existing directory, no error is reported. Otherwise on failure an error is reported.

2 *Returns:* `true` if a new directory was created, otherwise `false`.

3 *Throws:* As specified in 29.11.6.

```
bool create_directory(const path& p, const path& existing_p);
bool create_directory(const path& p, const path& existing_p, error_code& ec) noexcept;
```

4 *Effects:* Creates the directory `p` resolves to, with attributes copied from directory `existing_p`. The set of attributes copied is operating system dependent. If `mkdir` fails because `p` resolves to an existing directory, no error is reported. Otherwise on failure an error is reported. [*Note*: For POSIX-based operating systems, the attributes are those copied by native API `stat(existing_p.c_str(), &attributes_stat)` followed by `mkdir(p.c_str(), attributes_stat.st_mode)`. For Windows-based operating systems, the attributes are those copied by native API `CreateDirectoryExW(existing_p.c_str(), p.c_str(), 0)`. — *end note*]

5 *Returns:* `true` if a new directory was created with attributes copied from directory `existing_p`, otherwise `false`.

6 *Throws:* As specified in 29.11.6.

29.11.14.8 Create directory symlink [fs.op.create.dir.symlk]

```
void create_directory_symlink(const path& to, const path& new_symlink);
void create_directory_symlink(const path& to, const path& new_symlink,
                              error_code& ec) noexcept;
```

1 *Effects:* Establishes the postcondition, as if by POSIX `symlink()`.

2 *Ensures:* `new_symlink` resolves to a symbolic link file that contains an unspecified representation of `to`.

3 *Throws:* As specified in 29.11.6.

4 [*Note*: Some operating systems require symlink creation to identify that the link is to a directory. Portable code should use `create_directory_symlink()` to create directory symlinks rather than `create_symlink()` — *end note*]

5 [*Note*: Some operating systems do not support symbolic links at all or support them only for regular files. Some file systems (such as the FAT file system) do not support symbolic links regardless of the operating system. — *end note*]

29.11.14.9 Create hard link [fs.op.create.hard.lk]

```
void create_hard_link(const path& to, const path& new_hard_link);
void create_hard_link(const path& to, const path& new_hard_link,
                                      error_code& ec) noexcept;
```

1 *Effects:* Establishes the postcondition, as if by POSIX `link()`.

2 *Ensures:*

(2.1) — `exists(to) && exists(new_hard_link) && equivalent(to, new_hard_link)`

(2.2) — The contents of the file or directory `to` resolves to are unchanged.

3 *Throws:* As specified in 29.11.6.

4 [*Note*: Some operating systems do not support hard links at all or support them only for regular files. Some file systems (such as the FAT file system) do not support hard links regardless of the operating system. Some file systems limit the number of links per file. — *end note*]

29.11.14.10 Create symlink [fs.op.create.symlink]

```
void create_symlink(const path& to, const path& new_symlink);
void create_symlink(const path& to, const path& new_symlink,
                    error_code& ec) noexcept;
```

1 *Effects:* Establishes the postcondition, as if by POSIX `symlink()`.

2 *Ensures:* `new_symlink` resolves to a symbolic link file that contains an unspecified representation of `to`.

3 *Throws:* As specified in 29.11.6.

4 [*Note*: Some operating systems do not support symbolic links at all or support them only for regular files. Some file systems (such as the FAT file system) do not support symbolic links regardless of the operating system. — *end note*]

29.11.14.11 Current path [fs.op.current.path]

```
path current_path();
path current_path(error_code& ec);
```

1 *Returns:* The absolute path of the current working directory, whose pathname in the native format is obtained as if by POSIX `getcwd()`. The signature with argument `ec` returns `path()` if an error occurs.

2 *Throws:* As specified in 29.11.6.

3 *Remarks:* The current working directory is the directory, associated with the process, that is used as the starting location in pathname resolution for relative paths.

4 [*Note*: The `current_path()` name was chosen to emphasize that the returned value is a path, not just a single directory name. — *end note*]

5 [*Note*: The current path as returned by many operating systems is a dangerous global variable. It may be changed unexpectedly by third-party or system library functions, or by another thread. — *end note*]

```
void current_path(const path& p);
void current_path(const path& p, error_code& ec) noexcept;
```

6 *Effects:* Establishes the postcondition, as if by POSIX `chdir()`.

7 *Ensures:* `equivalent(p, current_path())`.

8 *Throws:* As specified in 29.11.6.

9 [*Note*: The current path for many operating systems is a dangerous global state. It may be changed unexpectedly by a third-party or system library functions, or by another thread. — *end note*]

29.11.14.12 Equivalent [fs.op.equivalent]

```
bool equivalent(const path& p1, const path& p2);
bool equivalent(const path& p1, const path& p2, error_code& ec) noexcept;
```

1 *Returns:* `true`, if `p1` and `p2` resolve to the same file system entity, otherwise `false`. The signature with argument `ec` returns `false` if an error occurs.

2 Two paths are considered to resolve to the same file system entity if two candidate entities reside on the same device at the same location. [*Note*: On POSIX platforms, this is determined as if by the values of the POSIX `stat` class, obtained as if by `stat()` for the two paths, having equal `st_dev` values and equal `st_ino` values. — *end note*]

3 *Remarks:* `!exists(p1) || !exists(p2)` is an error.

4 *Throws:* As specified in 29.11.6.

29.11.14.13 Exists [fs.op.exists]

```
bool exists(file_status s) noexcept;
```

1 *Returns:* `status_known(s) && s.type() != file_type::not_found`.

```
bool exists(const path& p);
bool exists(const path& p, error_code& ec) noexcept;
```

2 Let `s` be a `file_status`, determined as if by `status(p)` or `status(p, ec)`, respectively.

3 *Effects:* The signature with argument `ec` calls `ec.clear()` if `status_known(s)`.

4 *Returns:* `exists(s)`.

5 *Throws:* As specified in 29.11.6.

29.11.14.14 File size [fs.op.file.size]

```
uintmax_t file_size(const path& p);
uintmax_t file_size(const path& p, error_code& ec) noexcept;
```

1 *Effects:* If `exists(p)` is `false`, an error is reported (29.11.6).

2 *Returns:*

(2.1) — If `is_regular_file(p)`, the size in bytes of the file `p` resolves to, determined as if by the value of the POSIX `stat` class member `st_size` obtained as if by POSIX `stat()`.

(2.2) — Otherwise, the result is implementation-defined.

The signature with argument `ec` returns `static_cast<uintmax_t>(-1)` if an error occurs.

3 *Throws:* As specified in 29.11.6.

29.11.14.15 Hard link count [fs.op.hard.lk.ct]

```
uintmax_t hard_link_count(const path& p);
uintmax_t hard_link_count(const path& p, error_code& ec) noexcept;
```

1 *Returns:* The number of hard links for `p`. The signature with argument `ec` returns `static_cast<uintmax_t>(-1)` if an error occurs.

2 *Throws:* As specified in 29.11.6.

29.11.14.16 Is block file [fs.op.is.block.file]

```
bool is_block_file(file_status s) noexcept;
```

1 *Returns:* `s.type() == file_type::block`.

```
bool is_block_file(const path& p);
bool is_block_file(const path& p, error_code& ec) noexcept;
```

2 *Returns:* `is_block_file(status(p))` or `is_block_file(status(p, ec))`, respectively. The signature with argument `ec` returns `false` if an error occurs.

3 *Throws:* As specified in 29.11.6.

29.11.14.17 Is character file [fs.op.is.char.file]

```
bool is_character_file(file_status s) noexcept;
```

1 *Returns:* `s.type() == file_type::character`.

```
bool is_character_file(const path& p);
bool is_character_file(const path& p, error_code& ec) noexcept;
```

2 *Returns:* `is_character_file(status(p))` or `is_character_file(status(p, ec))`, respectively. The signature with argument `ec` returns `false` if an error occurs.

3 *Throws:* As specified in 29.11.6.

29.11.14.18 Is directory [fs.op.is.directory]

```
bool is_directory(file_status s) noexcept;
```

1 *Returns:* `s.type() == file_type::directory`.

```
bool is_directory(const path& p);
bool is_directory(const path& p, error_code& ec) noexcept;
```

2 *Returns:* `is_directory(status(p))` or `is_directory(status(p, ec))`, respectively. The signature with argument `ec` returns `false` if an error occurs.

3 *Throws:* As specified in 29.11.6.

29.11.14.19 Is empty [fs.op.is.empty]

```
bool is_empty(const path& p);
bool is_empty(const path& p, error_code& ec);
```

1 *Effects:*

(1.1) — Determine `file_status s`, as if by `status(p)` or `status(p, ec)`, respectively.

(1.2) — For the signature with argument `ec`, return `false` if an error occurred.

(1.3) — Otherwise, if `is_directory(s)`:

(1.3.1) — Create a variable `itr`, as if by `directory_iterator itr(p)` or `directory_iterator itr(p, ec)`, respectively.

(1.3.2) — For the signature with argument `ec`, return `false` if an error occurred.

(1.3.3) — Otherwise, return `itr == directory_iterator()`.

(1.4) — Otherwise:

(1.4.1) — Determine `uintmax_t sz`, as if by `file_size(p)` or `file_size(p, ec)`, respectively.

(1.4.2) — For the signature with argument `ec`, return `false` if an error occurred.

(1.4.3) — Otherwise, return `sz == 0`.

2 *Throws:* As specified in 29.11.6.

29.11.14.20 Is fifo [fs.op.is.fifo]

```
bool is_fifo(file_status s) noexcept;
```

1 *Returns:* `s.type() == file_type::fifo`.

```
bool is_fifo(const path& p);
bool is_fifo(const path& p, error_code& ec) noexcept;
```

2 *Returns:* `is_fifo(status(p))` or `is_fifo(status(p, ec))`, respectively. The signature with argument `ec` returns `false` if an error occurs.

3 *Throws:* As specified in 29.11.6.

29.11.14.21 Is other [fs.op.is.other]

```
bool is_other(file_status s) noexcept;
```

1 *Returns:* `exists(s) && !is_regular_file(s) && !is_directory(s) && !is_symlink(s)`.

```
bool is_other(const path& p);
bool is_other(const path& p, error_code& ec) noexcept;
```

2 *Returns:* `is_other(status(p))` or `is_other(status(p, ec))`, respectively. The signature with argument `ec` returns `false` if an error occurs.

3 *Throws:* As specified in 29.11.6.

29.11.14.22 Is regular file [fs.op.is.regular.file]

```
bool is_regular_file(file_status s) noexcept;
```

1 *Returns:* `s.type() == file_type::regular`.

```
bool is_regular_file(const path& p);
```

2 *Returns:* `is_regular_file(status(p))`.

3 *Throws:* `filesystem_error` if `status(p)` would throw `filesystem_error`.

```
bool is_regular_file(const path& p, error_code& ec) noexcept;
```

4 *Effects:* Sets `ec` as if by `status(p, ec)`. [*Note*: `file_type::none`, `file_type::not_found` and `file_type::unknown` cases set `ec` to error values. To distinguish between cases, call the `status` function directly. — *end note*]

5 *Returns:* `is_regular_file(status(p, ec))`. Returns `false` if an error occurs.

29.11.14.23 Is socket [fs.op.is.socket]

```
bool is_socket(file_status s) noexcept;
```

1 *Returns:* `s.type() == file_type::socket`.

```
bool is_socket(const path& p);
bool is_socket(const path& p, error_code& ec) noexcept;
```

2 *Returns:* `is_socket(status(p))` or `is_socket(status(p, ec))`, respectively. The signature with argument `ec` returns `false` if an error occurs.

3 *Throws:* As specified in 29.11.6.

29.11.14.24 Is symlink [fs.op.is.symlink]

```
bool is_symlink(file_status s) noexcept;
```

1 *Returns:* `s.type() == file_type::symlink`.

```
bool is_symlink(const path& p);
bool is_symlink(const path& p, error_code& ec) noexcept;
```

2 *Returns:* `is_symlink(symlink_status(p))` or `is_symlink(symlink_status(p, ec))`, respectively. The signature with argument `ec` returns `false` if an error occurs.

3 *Throws:* As specified in 29.11.6.

29.11.14.25 Last write time [fs.op.last.write.time]

```
file_time_type last_write_time(const path& p);
file_time_type last_write_time(const path& p, error_code& ec) noexcept;
```

1 *Returns:* The time of last data modification of `p`, determined as if by the value of the POSIX `stat` class member `st_mtime` obtained as if by POSIX `stat()`. The signature with argument `ec` returns `file_time_type::min()` if an error occurs.

2 *Throws:* As specified in 29.11.6.

```
void last_write_time(const path& p, file_time_type new_time);
void last_write_time(const path& p, file_time_type new_time,
                     error_code& ec) noexcept;
```

3 *Effects:* Sets the time of last data modification of the file resolved to by `p` to `new_time`, as if by POSIX `futimens()`.

4 *Throws:* As specified in 29.11.6.

5 [*Note*: A postcondition of `last_write_time(p) == new_time` is not specified since it might not hold for file systems with coarse time granularity. — *end note*]

29.11.14.26 Permissions [fs.op.permissions]

```
void permissions(const path& p, perms prms, perm_options opts=perm_options::replace);
void permissions(const path& p, perms prms, error_code& ec) noexcept;
void permissions(const path& p, perms prms, perm_options opts, error_code& ec);
```

1 ~~*Requires:*~~ *Expects:* Exactly one of the `perm_options` constants `replace`, `add`, or `remove` is present in `opts`.

2 *Remarks:* The second signature behaves as if it had an additional parameter `perm_options opts` with an argument of `perm_options::replace`.

3 *Effects:* Applies the action specified by `opts` to the file `p` resolves to, or to file `p` itself if `p` is a symbolic link and `perm_options::nofollow` is set in `opts`. The action is applied as if by POSIX `fchmodat()`.

4 [*Note*: Conceptually permissions are viewed as bits, but the actual implementation may use some other mechanism. — *end note*]

5 *Throws:* As specified in 29.11.6.

29.11.14.27 Proximate [fs.op.proximate]

```
path proximate(const path& p, error_code& ec);
```

1 *Returns:* `proximate(p, current_path(), ec)`.

2 *Throws:* As specified in 29.11.6.

```
path proximate(const path& p, const path& base = current_path());
path proximate(const path& p, const path& base, error_code& ec);
```

3 *Returns:* For the first form:

```
weakly_canonical(p).lexically_proximate(weakly_canonical(base));
```

For the second form:

```
weakly_canonical(p, ec).lexically_proximate(weakly_canonical(base, ec));
```

or `path()` at the first error occurrence, if any.

4 *Throws:* As specified in 29.11.6.

29.11.14.28 Read symlink [fs.op.read.symlink]

```
path read_symlink(const path& p);
path read_symlink(const path& p, error_code& ec);
```

1 *Returns:* If `p` resolves to a symbolic link, a `path` object containing the contents of that symbolic link. The signature with argument `ec` returns `path()` if an error occurs.

2 *Throws:* As specified in 29.11.6. [*Note*: It is an error if `p` does not resolve to a symbolic link. — *end note*]

29.11.14.29 Relative [fs.op.relative]

```
path relative(const path& p, error_code& ec);
```

1 *Returns:* `relative(p, current_path(), ec)`.

2 *Throws:* As specified in 29.11.6.

```
path relative(const path& p, const path& base = current_path());
path relative(const path& p, const path& base, error_code& ec);
```

3 *Returns:* For the first form:

```
weakly_canonical(p).lexically_relative(weakly_canonical(base));
```

For the second form:

```
weakly_canonical(p, ec).lexically_relative(weakly_canonical(base, ec));
```

or `path()` at the first error occurrence, if any.

4 *Throws:* As specified in 29.11.6.

29.11.14.30 Remove [fs.op.remove]

```
bool remove(const path& p);
bool remove(const path& p, error_code& ec) noexcept;
```

1 *Effects:* If `exists(symlink_status(p, ec))`, the file `p` is removed as if by POSIX `remove()`. [*Note*: A symbolic link is itself removed, rather than the file it resolves to. — *end note*]

2 *Ensures:* `exists(symlink_status(p))` is `false`.

3 *Returns:* `false` if `p` did not exist, otherwise `true`. The signature with argument `ec` returns `false` if an error occurs.

4 *Throws:* As specified in 29.11.6.

29.11.14.31 Remove all [fs.op.remove.all]

```
uintmax_t remove_all(const path& p);
uintmax_t remove_all(const path& p, error_code& ec);
```

1 *Effects:* Recursively deletes the contents of `p` if it exists, then deletes file `p` itself, as if by POSIX `remove()`. [*Note*: A symbolic link is itself removed, rather than the file it resolves to. — *end note*]

2 *Ensures:* `exists(symlink_status(p))` is `false`.

3 *Returns:* The number of files removed. The signature with argument `ec` returns `static_cast<uintmax_t>(-1)` if an error occurs.

4 *Throws:* As specified in 29.11.6.

29.11.14.32 Rename [fs.op.rename]

```
void rename(const path& old_p, const path& new_p);
void rename(const path& old_p, const path& new_p, error_code& ec) noexcept;
```

1 *Effects:* Renames `old_p` to `new_p`, as if by POSIX `rename()`.

[*Note*:

(1.1) — If `old_p` and `new_p` resolve to the same existing file, no action is taken.

(1.2) — Otherwise, the rename may include the following effects:

(1.2.1) — if `new_p` resolves to an existing non-directory file, `new_p` is removed; otherwise,

(1.2.2) — if `new_p` resolves to an existing directory, `new_p` is removed if empty on POSIX compliant operating systems but may be an error on other operating systems.

A symbolic link is itself renamed, rather than the file it resolves to. — *end note*]

2 *Throws:* As specified in 29.11.6.

29.11.14.33 Resize file [fs.op.resize.file]

```
void resize_file(const path& p, uintmax_t new_size);
void resize_file(const path& p, uintmax_t new_size, error_code& ec) noexcept;
```

1 *Effects:* Causes the size that would be returned by `file_size(p)` to be equal to `new_size`, as if by POSIX `truncate()`.

2 *Throws:* As specified in 29.11.6.

29.11.14.34 Space [fs.op.space]

```
space_info space(const path& p);
space_info space(const path& p, error_code& ec) noexcept;
```

1 *Returns:* An object of type `space_info`. The value of the `space_info` object is determined as if by using POSIX `statvfs` to obtain a POSIX `struct statvfs`, and then multiplying its `f_blocks`, `f_bfree`, and `f_bavail` members by its `f_frsize` member, and assigning the results to the `capacity`, `free`, and `available` members respectively. Any members for which the value cannot be determined shall be set to `static_cast<uintmax_t>(-1)`. For the signature with argument `ec`, all members are set to `static_cast<uintmax_t>(-1)` if an error occurs.

2 *Throws:* As specified in 29.11.6.

3 *Remarks:* The value of member `space_info::available` is operating system dependent. [*Note*: `available` may be less than `free`. — *end note*]

29.11.14.35 Status [fs.op.status]

```
file_status status(const path& p);
```

1 *Effects:* As if:

```
error_code ec;
file_status result = status(p, ec);
if (result.type() == file_type::none)
  throw filesystem_error(implementation-supplied-message, p, ec);
return result;
```

2 *Returns:* See above.

3 *Throws:* `filesystem_error`. [*Note*: `result` values of `file_status(file_type::not_found)` and `file_status(file_type::unknown)` are not considered failures and do not cause an exception to be thrown. — *end note*]

```
file_status status(const path& p, error_code& ec) noexcept;
```

4 *Effects:* If possible, determines the attributes of the file `p` resolves to, as if by using POSIX `stat()` to obtain a POSIX `struct stat`. If, during attribute determination, the underlying file system API reports an error, sets `ec` to indicate the specific error reported. Otherwise, `ec.clear()`. [*Note*: This allows users to inspect the specifics of underlying API errors even when the value returned by `status()` is not `file_status(file_type::none)`. — *end note*]

5 Let `prms` denote the result of `(m & perms::mask)`, where `m` is determined as if by converting the `st_mode` member of the obtained `struct stat` to the type `perms`.

6 *Returns:*

(6.1) — If `ec != error_code()`:

(6.1.1) — If the specific error indicates that `p` cannot be resolved because some element of the path does not exist, returns `file_status(file_type::not_found)`.

(6.1.2) — Otherwise, if the specific error indicates that `p` can be resolved but the attributes cannot be determined, returns `file_status(file_type::unknown)`.

(6.1.3) — Otherwise, returns `file_status(file_type::none)`.

[*Note*: These semantics distinguish between `p` being known not to exist, `p` existing but not being able to determine its attributes, and there being an error that prevents even knowing if `p` exists. These distinctions are important to some use cases. — *end note*]

(6.2) — Otherwise,

(6.2.1) — If the attributes indicate a regular file, as if by POSIX `S_ISREG`, returns `file_status(file_type::regular, prms)`. [*Note*: `file_type::regular` implies appropriate `<fstream>` operations would succeed, assuming no hardware, permission, access, or file system race errors. Lack of `file_type::regular` does not necessarily imply `<fstream>` operations would fail on a directory. — *end note*]

(6.2.2) — Otherwise, if the attributes indicate a directory, as if by POSIX `S_ISDIR`, returns `file_status(file_type::directory, prms)`. [*Note*: `file_type::directory` implies that calling `directory_iterator(p)` would succeed. — *end note*]

(6.2.3) — Otherwise, if the attributes indicate a block special file, as if by POSIX `S_ISBLK`, returns `file_status(file_type::block, prms)`.

(6.2.4) — Otherwise, if the attributes indicate a character special file, as if by POSIX `S_ISCHR`, returns `file_status(file_type::character, prms)`.

(6.2.5) — Otherwise, if the attributes indicate a fifo or pipe file, as if by POSIX `S_ISFIFO`, returns `file_status(file_type::fifo, prms)`.

(6.2.6) — Otherwise, if the attributes indicate a socket, as if by POSIX `S_ISSOCK`, returns `file_status(file_type::socket, prms)`.

(6.2.7) — Otherwise, if the attributes indicate an implementation-defined file type (29.11.9.2), returns `file_status(file_type::`*`A`*`, prms)`, where *`A`* is the constant for the implementation-defined file type.

(6.2.8) — Otherwise, returns `file_status(file_type::unknown, prms)`.

7 *Remarks:* If a symbolic link is encountered during pathname resolution, pathname resolution continues using the contents of the symbolic link.

29.11.14.36 Status known [fs.op.status.known]

```
bool status_known(file_status s) noexcept;
```

1 *Returns:* `s.type() != file_type::none`.

29.11.14.37 Symlink status [fs.op.symlink.status]

```
file_status symlink_status(const path& p);
file_status symlink_status(const path& p, error_code& ec) noexcept;
```

1 *Effects:* Same as `status()`, above, except that the attributes of `p` are determined as if by using POSIX `lstat()` to obtain a POSIX `struct stat`.

2 Let `prms` denote the result of `(m & perms::mask)`, where `m` is determined as if by converting the `st_mode` member of the obtained `struct stat` to the type `perms`.

3 *Returns:* Same as `status()`, above, except that if the attributes indicate a symbolic link, as if by POSIX `S_ISLNK`, returns `file_status(file_type::symlink, prms)`. The signature with argument `ec` returns `file_status(file_type::none)` if an error occurs.

4 *Remarks:* Pathname resolution terminates if `p` names a symbolic link.

5 *Throws:* As specified in 29.11.6.

29.11.14.38 Temporary directory path [fs.op.temp.dir.path]

```
path temp_directory_path();
```

```
path temp_directory_path(error_code& ec);
```

1 Let `p` be an unspecified directory path suitable for temporary files.

2 *Effects:* If `exists(p)` is `false` or `is_directory(p)` is `false`, an error is reported (29.11.6).

3 *Returns:* The path `p`. The signature with argument `ec` returns `path()` if an error occurs.

4 *Throws:* As specified in 29.11.6.

5 [*Example*: For POSIX-based operating systems, an implementation might return the path supplied by the first environment variable found in the list TMPDIR, TMP, TEMP, TEMPDIR, or if none of these are found, `"/tmp"`.

For Windows-based operating systems, an implementation might return the path reported by the Windows `GetTempPath` API function. — *end example*]

29.11.14.39 Weakly canonical [fs.op.weakly.canonical]

```
path weakly_canonical(const path& p);
path weakly_canonical(const path& p, error_code& ec);
```

1 *Returns:* `p` with symlinks resolved and the result normalized (29.11.7.1).

2 *Effects:* Using `status(p)` or `status(p, ec)`, respectively, to determine existence, return a path composed by `operator/=` from the result of calling `canonical()` with a path argument composed of the leading elements of `p` that exist, if any, followed by the elements of `p` that do not exist, if any. For the first form, `canonical()` is called without an `error_code` argument. For the second form, `canonical()` is called with `ec` as an `error_code` argument, and `path()` is returned at the first error occurrence, if any.

3 *Ensures:* The returned path is in normal form (29.11.7.1).

4 *Remarks:* Implementations should avoid unnecessary normalization such as when `canonical` has already been called on the entirety of `p`.

5 *Throws:* As specified in 29.11.6.

29.12 C library files [c.files]

29.12.1 Header <cstdio> synopsis [cstdio.syn]

```
namespace std {
  using size_t = see ??;
  using FILE = see below;
  using fpos_t = see below;
}

#define NULL see ??
#define _IOFBF see below
#define _IOLBF see below
#define _IONBF see below
#define BUFSIZ see below
#define EOF see below
#define FOPEN_MAX see below
#define FILENAME_MAX see below
#define L_tmpnam see below
#define SEEK_CUR see below
#define SEEK_END see below
#define SEEK_SET see below
#define TMP_MAX see below
#define stderr see below
#define stdin see below
#define stdout see below

namespace std {
  int remove(const char* filename);
  int rename(const char* old_p, const char* new_p);
  FILE* tmpfile();
  char* tmpnam(char* s);
```

```
  int fclose(FILE* stream);
  int fflush(FILE* stream);
  FILE* fopen(const char* filename, const char* mode);
  FILE* freopen(const char* filename, const char* mode, FILE* stream);
  void setbuf(FILE* stream, char* buf);
  int setvbuf(FILE* stream, char* buf, int mode, size_t size);
  int fprintf(FILE* stream, const char* format, ...);
  int fscanf(FILE* stream, const char* format, ...);
  int printf(const char* format, ...);
  int scanf(const char* format, ...);
  int snprintf(char* s, size_t n, const char* format, ...);
  int sprintf(char* s, const char* format, ...);
  int sscanf(const char* s, const char* format, ...);
  int vfprintf(FILE* stream, const char* format, va_list arg);
  int vfscanf(FILE* stream, const char* format, va_list arg);
  int vprintf(const char* format, va_list arg);
  int vscanf(const char* format, va_list arg);
  int vsnprintf(char* s, size_t n, const char* format, va_list arg);
  int vsprintf(char* s, const char* format, va_list arg);
  int vsscanf(const char* s, const char* format, va_list arg);
  int fgetc(FILE* stream);
  char* fgets(char* s, int n, FILE* stream);
  int fputc(int c, FILE* stream);
  int fputs(const char* s, FILE* stream);
  int getc(FILE* stream);
  int getchar();
  int putc(int c, FILE* stream);
  int putchar(int c);
  int puts(const char* s);
  int ungetc(int c, FILE* stream);
  size_t fread(void* ptr, size_t size, size_t nmemb, FILE* stream);
  size_t fwrite(const void* ptr, size_t size, size_t nmemb, FILE* stream);
  int fgetpos(FILE* stream, fpos_t* pos);
  int fseek(FILE* stream, long int offset, int whence);
  int fsetpos(FILE* stream, const fpos_t* pos);
  long int ftell(FILE* stream);
  void rewind(FILE* stream);
  void clearerr(FILE* stream);
  int feof(FILE* stream);
  int ferror(FILE* stream);
  void perror(const char* s);
}
```

1 The contents and meaning of the header `<cstdio>` are the same as the C standard library header `<stdio.h>`.

2 Calls to the function `tmpnam` with an argument that is a null pointer value may introduce a data race (**??**) with other calls to `tmpnam` with an argument that is a null pointer value.

See also: ISO C 7.21

29.12.2 Header `<cinttypes>` synopsis [cinttypes.syn]

```
#include <cstdint>  // see ??

namespace std {
  using imaxdiv_t = see below;

  intmax_t imaxabs(intmax_t j);
  imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);
  intmax_t strtoimax(const char* nptr, char** endptr, int base);
  uintmax_t strtoumax(const char* nptr, char** endptr, int base);
  intmax_t wcstoimax(const wchar_t* nptr, wchar_t** endptr, int base);
  uintmax_t wcstoumax(const wchar_t* nptr, wchar_t** endptr, int base);

  intmax_t abs(intmax_t);  // optional, see below
```

```
  imaxdiv_t div(intmax_t, intmax_t);  // optional, see below
}

#define PRIdN see below
#define PRIiN see below
#define PRIoN see below
#define PRIuN see below
#define PRIxN see below
#define PRIXN see below
#define SCNdN see below
#define SCNiN see below
#define SCNoN see below
#define SCNuN see below
#define SCNxN see below
#define PRIdLEASTN see below
#define PRIiLEASTN see below
#define PRIoLEASTN see below
#define PRIuLEASTN see below
#define PRIxLEASTN see below
#define PRIXLEASTN see below
#define SCNdLEASTN see below
#define SCNiLEASTN see below
#define SCNoLEASTN see below
#define SCNuLEASTN see below
#define SCNxLEASTN see below
#define PRIdFASTN see below
#define PRIiFASTN see below
#define PRIoFASTN see below
#define PRIuFASTN see below
#define PRIxFASTN see below
#define PRIXFASTN see below
#define SCNdFASTN see below
#define SCNiFASTN see below
#define SCNoFASTN see below
#define SCNuFASTN see below
#define SCNxFASTN see below
#define PRIdMAX see below
#define PRIiMAX see below
#define PRIoMAX see below
#define PRIuMAX see below
#define PRIxMAX see below
#define PRIXMAX see below
#define SCNdMAX see below
#define SCNiMAX see below
#define SCNoMAX see below
#define SCNuMAX see below
#define SCNxMAX see below
#define PRIdPTR see below
#define PRIiPTR see below
#define PRIoPTR see below
#define PRIuPTR see below
#define PRIxPTR see below
#define PRIXPTR see below
#define SCNdPTR see below
#define SCNiPTR see below
#define SCNoPTR see below
#define SCNuPTR see below
#define SCNxPTR see below
```

1 The contents and meaning of the header `<cinttypes>` are the same as the C standard library header `<inttypes.h>`, with the following changes:

(1.1) — The header `<cinttypes>` includes the header `<cstdint>` instead of `<stdint.h>`, and

(1.2) — if and only if the type `intmax_t` designates an extended integer type (**??**), the following function signatures are added:

```
intmax_t abs(intmax_t);
imaxdiv_t div(intmax_t, intmax_t);
```

which shall have the same semantics as the function signatures `intmax_t imaxabs(intmax_t)` and `imaxdiv_t imaxdiv(intmax_t, intmax_t)`, respectively.

See also: ISO C 7.8