ISO/IEC JTC 1/SC 22/WG 21 N 3803

Date: 2013-10-05

ISO/IEC PDTS 18822

ISO/IEC JTC1 SC22

Secretariat: ANSI

Programming Languages — C++ Standard Library — File System Technical Specification

Langages de programmation — Bibliothèque standard C++ — Spécification technique de système de fichiers

Warning

This document is not an ISO International Standard. It is distributed for review and comment. It is subject to change without notice and may not be referred to as an International Standard.

Recipients of this draft are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

Document type: Technical Specification
Document subtype:
Document stage: (30) Committee
Document language: E

Contents

Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of technical committees is to prepare International Standards. Draft International Standards adopted by the technical committees are circulated to the member bodies for voting. Publication as an International Standard requires approval by at least 75% of the member bodies casting a vote.

In other circumstances, particularly when there is an urgent market requirement for such documents, a technical committee may decide to publish other types of normative document:

— an ISO/IEC Publicly Available Specification (ISO/IEC PAS) represents an agreement between technical experts in an ISO working group and is accepted for publication if it is approved by more than 50% of the members of the parent committee casting a vote;

— an ISO/IEC Technical Specification (ISO/IEC TS) represents an agreement between the members of a technical committee and is accepted for publication if it is approved by 2/3 of the members of the committee casting a vote.

An ISO/PAS or ISO/TS is reviewed after three years in order to decide whether it will be confirmed for a further three years, revised to become an International Standard, or withdrawn. If the ISO/PAS or ISO/TS is confirmed, it is reviewed again after a further three years, at which time it must either be transformed into an International Standard or be withdrawn.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights.

ISO/IEC TS *18822* was prepared by ISO/IEC Joint Technical Committee 1, Subcommittee 22, Working Group 21.

1 Scope [fs.scope]

This Technical Specification specifies requirements for implementations of an interface that computer programs written in the C++ programming language may use to perform operations on file systems and their components, such as paths, regular files, and directories. This Technical Specification is applicable to information technology systems that can access hierarchical file systems, such as those with operating systems that conform to the POSIX ([fs.norm.ref]) interface. This Technical Specification is applicable only to vendors who wish to provide the interface it describes.

2 Conformance [fs.conformance]

Conformance is specified in terms of behavior. Ideal behavior is not always implementable, so the conformance sub-clauses take that into account.

2.1 POSIX conformance [fs.conform.9945]

Some behavior is specified by reference to POSIX ([fs.norm.ref]). How such behavior is actually implemented is unspecified.

> [*Note:* This constitutes an "as if" rule allowing implementations to call native operating system or other API's. --*end note*]

Implementations are encouraged to 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 are encouraged to 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 in an implementation-defined manner.

> [*Note:* Such errors might be reported by an #error directive, a `static_assert`, a `filesystem_error` exception, a special return value, or some other manner. --*end note*]

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 discs 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. -- *end example*]

The behavior of functions described in this Technical Specification may differ from their specification in the presence of file system races ([fs.def.race]). No diagnostic is required.

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*]

2.2 Operating system dependent behavior conformance [fs.conform.os]

Some behavior is specified as being operating system dependent ([fs.def.osdep]). The operating system an implementation is dependent upon is implementation defined.

It is permissible for an implementation to be dependent upon an operating system emulator rather than the actual underlying operating system.

3 Normative references [fs.norm.ref]

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

- ISO/IEC 14882, *Programming Language C++*
- ISO/IEC 9945, *Information Technology — Portable Operating System Interface (POSIX)*

[*Note:* The programming language and library described in ISO/IEC 14882 is herein called *the C++ Standard*. The operating system interface described in ISO/IEC 9945 is herein called *POSIX*. -- *end note*]

This Technical Specification mentions commercially available operating systems for purposes of exposition.[1]

Unless otherwise specified, the whole of the C++ Standard's Library introduction [lib.library] is included into this Technical Specification by reference.

4 Terms and definitions [fs.definitions]

For the purposes of this document, the terms and definitions given in the C++ Standard and the following apply.

4.1 absolute path [fs.def.absolute-path]

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.

---

1 POSIX® is a registered trademark of The IEEE. MAC OS® is a registered trademark of Apple Inc. 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.

4.2 canonical path [fs.def.cannonical-path]

An absolute path that has no elements that are symbolic links, and no dot or dot-dot elements ([path.generic]).

4.3 directory [fs.def.directory]

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.

4.4 file [fs.def.file]

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.

4.5 file system [fs.def.filesystem]

A collection of files and certain of their attributes.

4.6 file system race [fs.def.race]

The condition that occurs when multiple threads, processes, or computers interleave access and modification of the same object within a file system.

4.7 filename [fs.def.filename]

The name of a file. Filenames dot and dot-dot have special meaning. The following characteristics of filenames are operating system dependent:

- The permitted characters. [*Example*: Some operating systems prohibit the ASCII control characters (0x00-0x1F) in filenames. --*end example*].
- Filenames that are not permitted.
- Filenames that have special meaning.
- Case awareness and sensitivity during path resolution.
- Special rules that may apply to file types other than regular files, such as directories.

4.8 hard link [fs.def.hardlink]

A link ([fs.def.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]

4.9 link [fs.def.link]

A directory entry object that associates a filename with a file. A link is either a hard link ([fs.def.hardlink]) or a symbolic link ([fs.def.symlink]).

4.10 native encoding [fs.def.native.encode]

For narrow character strings, the operating system dependent current encoding for path names. For wide character strings, the implementation defined execution wide-character set encoding (C++ standard [lex.charset]).

4.11 native pathname format [fs.def.native]

The operating system dependent pathname format accepted by the host operating system.

4.12 NTCTS [fs.def.ntcts]

Acronym for "null-terminated character-type sequence". Describes a sequence of values of a given encoded character type terminated by that type's null character. If the encoded character type is `charT`, the null character can be constructed by `charT()`.

4.13 operating system dependent behavior [fs.def.osdep]

Behavior that is dependent upon the behavior and characteristics of an operating system. See [fs.conform.os].

4.14 parent directory [fs.def.parent]

When discussing a given directory, the directory that both contains a directory entry for the given directory and is represented by the pathname dot-dot in the given directory.

When discussing other types of files, a directory containing a directory entry for the file under discussion.

This concept does not apply to dot and dot-dot.

4.15 path [fs.def.path]

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 filenames.

4.16 pathname [fs.def.pathname]

A character string that represents the name of a path. Pathnames are formatted according to the

generic pathname format grammar ([path.generic]) or an operating system dependent native pathname format.

4.17 pathname resolution [fs.def.pathres]

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]*

4.18 relative path [fs.def.relative-path]

A path that is not absolute, and so 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:* Paths `"."` and `".."` are relative paths. *--end note*]

4.19 symbolic link [fs.def.symlink]

A link ([fs.def.link]) with the property that when the file is encountered during pathname resolution, 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*]

5 Requirements [fs.req]

Throughout this Technical Specification, `char`, `wchar_t`, `char16_t`, and `char32_t` are collectively called *encoded character types*.

Template parameters named `charT` shall be one of the encoded character types.

Template parameters named `InputIterator` shall meet the C++ Standard's library input iterator requirements ([input.iterators]) and shall have a value type that is one of the encoded character types.

> [*Note:* Use of an encoded character type implies an associated encoding. Since `signed char` and `unsigned char` have no implied encoding, they are not included as permitted types. *--end note*]

6 Header `<filesystem>` synopsis [fs.filesystem.synopsis]

```
namespace std { namespace tbd { namespace filesystem {

    class path;

    void swap(path& lhs, path& rhs) noexcept;
    size_t hash_value(const path& p) noexcept;
```

```
bool operator==(const path& lhs, const path& rhs) noexcept;
bool operator!=(const path& lhs, const path& rhs) noexcept;
bool operator< (const path& lhs, const path& rhs) noexcept;
bool operator<=(const path& lhs, const path& rhs) noexcept;
bool operator> (const path& lhs, const path& rhs) noexcept;
bool operator>=(const path& lhs, const path& rhs) noexcept;

path operator/ (const path& lhs, const path& rhs);

template <class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const path& p);

template <class charT, class traits>
basic_istream<charT, traits>&
operator>>(basic_istream<charT, traits>& is, path& p);

template <class Source>
  path u8path(Source const& source);
template <class InputIterator>
  path u8path(InputIterator begin, InputIterator end);

class filesystem_error;
class directory_entry;

class directory_iterator;

// enable directory_iterator range-based for statements
const directory_iterator& begin(const directory_iterator& iter) noexcept;
directory_iterator end(const directory_iterator&) noexcept;

class recursive_directory_iterator;

// enable recursive_directory_iterator range-based for statements
const recursive_directory_iterator& begin(
  const recursive_directory_iterator& iter) noexcept;
recursive_directory_iterator end(
  const recursive_directory_iterator&) noexcept;

enum class file_type;  // [enum.file_type]

enum class perms;  // [enum.perms]

class file_status;

struct space_info  // returned by space function
{
  uintmax_t capacity;
  uintmax_t free;
  uintmax_t available; // free space available to a non-privileged process
};

enum class copy_options;  // [enum.copy_options]

enum class directory_options
{
  none,
  follow_directory_symlink,
```

```
    skip_permission_denied
  };

  typedef chrono::time_point<unspecified-trivial-clock>  file_time_type;

  // operational functions

  path         absolute(const path& p, const path& base=current_path());

  path         canonical(const path& p, const path& base = current_path());
  path         canonical(const path& p, error_code& ec);
  path         canonical(const path& p, const path& base, error_code& ec);

  void         copy(const path& from, const path& to);
  void         copy(const path& from, const path& to,
                 error_code& ec) noexcept;
  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) noexcept;

  bool         copy_file(const path& from, const path& to);
  bool         copy_file(const path& from, const path& to,
                 error_code& ec) noexcept;
  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) noexcept;

  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) noexcept;

  bool         create_directory(const path& p);
  bool         create_directory(const path& p, error_code& ec) noexcept;

  void         create_directory(const path& p, const path& attributes);
  void         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          exists(file_status s) noexcept;
    bool          exists(const path& p);
    bool          exists(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;

    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) noexcept;

    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;

    path          read_symlink(const path& p);
    path          read_symlink(const path& p, 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) noexcept;

    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          system_complete(const path& p);
    path          system_complete(const path& p, error_code& ec);

    path          temp_directory_path();
    path          temp_directory_path(error_code& ec);

    path          unique_path(const path& model="%%%%-%%%%-%%%%-%%%%");
    path          unique_path(const path& model, error_code& ec);

} } }  // namespaces std::tbd::filesystem
```

unspecified-trivial-clock is an unspecified type provided by the implementation that satisfies the `TrivialClock` requirements ([time.clock.req]) and that is capable of representing and measuring file time values.

7 Error reporting [fs.err.report]

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:

- Uses where file system errors are truly exceptional and indicate a serious failure. Throwing an exception is the most appropriate response. This is the preferred default for most everyday programming.

- Uses where file system 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]

Functions **not** having an argument of type `error_code&` report errors as follows, unless otherwise specified:

- 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.

- Failure to allocate storage is reported by throwing an exception as described in the C++ standard, 17.6.4.10 [res.on.exception.handling].

- Destructors throw nothing.

Functions having an argument of type `error_code&` report errors as follows, unless otherwise specified:

- 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 appropriate for the specific error. Otherwise, `clear()` is called on the `error_code&` argument.

8 Class `path` [class.path]

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.

```
namespace std { namespace tbd { namespace filesystem {

      class path
      {
      public:
        typedef see below                value_type;
        typedef basic_string<value_type> string_type;
        static constexpr value_type   preferred_separator = see below;

        // constructors and destructor
        path();
        path(const path& p);
        path(path&& p) noexcept;
        template <class Source>
```

```
    path(Source const& source);
  template <class InputIterator>
    path(InputIterator begin, InputIterator end);
  template <class Source>
    path(Source const& source, const locale& loc);
  template <class InputIterator>
    path(InputIterator begin, InputIterator end,
      const locale& loc);
 ~path() = default;

  // assignments
  path& operator=(const path& p);
  path& operator=(path&& p) noexcept;
  template <class Source>
    path& operator=(Source const& source);
  template <class Source>
    path& assign(Source const& source)
  template <class InputIterator>
    path& assign(InputIterator begin, InputIterator end);

  // appends
  path& operator/=(const path& p);
  template <class Source>
    path& operator/=(Source const& source);
  template <class Source>
    path& append(Source const& source);
  template <class InputIterator>
    path& append(InputIterator begin, InputIterator end);

  // concatenation
  path& operator+=(const path& x);
  path& operator+=(const string_type& x);
  path& operator+=(const value_type* x);
  path& operator+=(value_type x);
  template <class Source>
    path& operator+=(Source const& x);
  template <class charT>
    path& operator+=(charT x);
  template <class Source>
    path& concat(Source const& x);
  template <class InputIterator>
    path& concat(InputIterator begin, InputIterator end);

  // 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;

  // native format observers
  const string_type&  native() const noexcept;
  const value_type*   c_str() const noexcept;
  operator string_type() const;

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

```
      string(const Allocator& a = Allocator()) const;
    string    string() const;
    wstring   wstring() const;
    string    u8string() const;
    u16string u16string() const;
    u32string u32string() const;

    // generic format observers
    template <class charT, class traits = char_traits<charT>,
              class Allocator = allocator<charT> >
    basic_string<charT, traits, Allocator>
      generic_string(const Allocator& a = Allocator()) const;
    string    generic_string() const;
    wstring   generic_wstring() const;
    string    generic_u8string() const;
    u16string generic_u16string() const;
    u32string generic_u32string() const;

    // compare
    int  compare(const path& p) const noexcept;
    int  compare(const string& s) const;
    int  compare(const value_type* s) const;

    // 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;

    // query
    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;

    // iterators
    class iterator;
    typedef iterator const_iterator;

    iterator begin() const;
    iterator end() const;

  private:
    string_type pathname;  // exposition only
  };

} } }  // namespaces std::tbd::filesystem
```

`value_type` is a `typedef` for the operating system dependent encoded character type used to represent pathnames.

The value of `preferred_separator` is the operating system dependent *preferred-separator* character ([path.generic]).

> [*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 (\). *--end example*]

8.1 `path` generic pathname format grammar [path.generic]

pathname:

root-name root-directory$_{opt}$ *relative-path*$_{opt}$
root-directory relative-path$_{opt}$
relative-path

root-name:

An operating system dependent name that identifies the starting location for absolute paths.

> [*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 disc drive. *--end note*]

root-directory:

directory-separator

relative-path:

filename
relative-path directory-separator
relative-path directory-separator filename

filename:

name
dot
dot-dot

name:

A sequence of characters other than *directory-separator* characters.

> [*Note:* Operating systems often place restrictions on the characters that may be used in a *filename*. 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]

dot:
The filename consisting solely of a single period character (.).

dot-dot:
The filename consisting solely of two period characters (..).

directory-separator:
slash
slash directory-separator
preferred-separator
preferred-separator directory-separator

preferred-separator:
An operating system dependent directory separator character. May be a synonym for *slash*.

slash:
The slash character (/).

Multiple successive *directory-separator* characters are considered to be the same as one *directory-separator* character.

The filename *dot* is treated as a reference to the current directory. The filename *dot-dot* is treated as a reference to the parent directory. Specific filenames may have special meanings for a particular operating system.

8.2 `path` conversions [path.cvt]

8.2.1 `path` argument format conversions [path.fmt.cvt]

[*Note:* The format conversions described in this section are not applied on POSIX or Windows based operating systems because on these systems:

- The generic format is acceptable as a native path.
- There is no need to distinguish between native format and generic format arguments.
- Paths for regular files and paths for directories share the same syntax.

-- *end note*]

Functions arguments that take character sequences representing paths may use the generic pathname format grammar ([path.generic]) or the native pathname format. If and only if such arguments are in the generic format and the generic format is not acceptable to the operating system as a native path, conversion to native format shall be performed during the processing of the argument.

[*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 as an extension to distinguish between the formats. -- *end note*]

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 last element is *directory-separator*, otherwise it shall be treated as a regular file path.

8.2.2 `path` type and encoding conversions [path.type.cvt]

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 differs from `path::value_type`. Encoding and method of conversion for the argument or return value to be converted to is determined by its value type:

- `char`: Encoding is the native narrow encoding ([fs.def.native.encode]). 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 returns is performed.

  For Windows based operating systems, the native narrow 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 narrow character strings to identify paths. Changing this behavior would be surprising and error prone. --*end note*]

- `wchar_t`: Encoding is the native wide encoding ([fs.def.native.encode]). Conversion method 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 returns is performed. -- *end note*]

- `char16_t`: Encoding is UTF-16. Conversion method is unspecified.

- `char32_t`: Encoding is UTF-32. Conversion method is unspecified.

8.3 `path` requirements [path.req]

In addition to the [fs.req] requirements, function template parameters named `Source` shall be one of:

- `basic_string<charT, traits, Allocator>`. The type `charT` shall be an encoded character type ([fs.req]). A function argument `Source const& source` shall have an effective range [`source.begin()`, `source.end()`).

- A type meeting the input iterator requirements that iterates over a NTCTS. The value type shall be an encoded character type. A function argument `Source const& 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()`.

- A character array that after array-to-pointer decay results in a pointer to a NTCTS. The value type shall be an encoded character type. A function argument `Source const& source` shall have an effective range [`source`, `end`) where `end` is the first iterator value with an element value equal to `iterator_traits<decay<Source>::type>::value_type()`.

[*Note:* See path conversions ([path.cvt]) for how these value types and their encodings convert to `path::value_type` and its encoding. -- *end note*]

Arguments of type `Source` shall not be null pointers.

8.4 `path` members [path.member]

8.4.1 `path` constructors [path.construct]

```
path();
```

Effects: Constructs an object of class `path`.

Postconditions: `empty()`.

```
path(const path& p);
path(path&& p) noexcept;
```

Effects: Constructs an object of class `path` with `pathname` having the original value of `p.pathname`. In the second form, `p` is left in a valid but unspecified state.

Postconditions: empty().

```
template <class Source>
  path(Source const& source);
template <class InputIterator>
  path(InputIterator begin, InputIterator end);
```

Effects: Constructs an object of class `path`, storing the effective range of `source` ([path.req]) or the range [`begin`,`end`) in `pathname`, converting format and encoding if required ([path.arg.convert]).

```
template <class Source>
```

```
  path(Source const& source, const locale& loc);
template <class InputIterator>
  path(InputIterator begin, InputIterator end, const locale& loc);
```

Requires: The value type of `Source` and `InputIterator` is `char`.

Effects: Constructs an object of class `path`, storing the effective range of `source` or the range [`begin`,`end`) in `pathname`, after converting format if required and after converting the encoding as follows:

If `value_type` is `wchar_t`, converts to the native wide encoding ([fs.def.native.encode]) using the `codecvt<wchar_t, char>` facet of `loc`. Otherwise a conversion is performed using the `codecvt<wchar_t, char>` facet of `loc`, and then a second conversion to the current narrow encoding.

[*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::tbd::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 ([fs.def.native.encode]). The resulting wide string is then converted to a narrow character `pathname` string in the current native narrow encoding. If the native wide encoding is UTF-16 or UTF-32, and the current native narrow 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 narrow 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]

8.4.2 `path` assignments [path.assign]

```
path& operator=(const path& p);
```

Effects: If `*this` and `p` are the same object, has no effect. Otherwise, modifies `pathname` to have the original value of `p.pathname`.

Returns: `*this`

```
path& operator=(path&& p) noexcept;
```

Effects: If `*this` and `p` are the same object, has no effect. Otherwise, modifies `pathname` to have the original value of `p.pathname`. `p` is left in a valid but unspecified state. [*Note:* A valid implementation is `swap(p)`. —*end note*]

Returns: `*this`

```
template <class Source>
  path& operator=(Source const& source);
template <class Source>
  path& assign(Source const& source);
template <class InputIterator>
  path& assign(InputIterator begin, InputIterator end);
```

Effects: Stores the effective range of `source` ([path.req]) or the range [`begin`,`end`) in `pathname`, converting format and encoding if required ([path.arg.convert]).

Returns: `*this`

8.4.3 `path` appends [path.append]

The append operations use `operator/=` to denote their semantic effect of appending *preferred-separator* when needed.

```
path& operator/=(const path& p);
```

Effects:

Appends `path::preferred_separator` to `pathname` unless:

- an added separator would be redundant, or
- would change an relative path to an absolute path, or
- `p.empty()`, or
- `*p.native().cbegin()` is a directory separator.

Then appends `p.native()` to `pathname`.

Returns: `*this`

```
template <class Source>
  path& operator/=(Source const& source);
template <class Source>
  path& append(Source const& source);
```

```
template <class InputIterator>
  path& append(InputIterator begin, InputIterator end);
```

Effects:

Appends `path::preferred_separator` to `pathname`, converting format and encoding if required ([path.arg.convert]), unless:

- an added separator would be redundant, or
- would change an relative path to an absolute path, or
- `p.empty()`, or
- `*p.native().cbegin()` is a separator.

Appends the effective range of `source` ([path.req]) or the range [`begin`,`end`) to `pathname`, converting format and encoding if required ([path.arg.convert]).

Returns: `*this`

8.4.4 `path` concatenation [path.concat]

```
path& operator+=(const path& x);
path& operator+=(const string_type& x);
path& operator+=(const value_type* x);
path& operator+=(value_type x);
template <class Source>
  path& operator+=(Source const& x);
template <class charT>
  path& operator+=(charT x);
template <class Source>
  path& concat(Source const& x);
template <class InputIterator>
  path& concat(InputIterator begin, InputIterator end);
```

Postcondition: `native() == prior_native +` *`effective-argument`*, where `prior_native` is `native()` prior to the call to `operator+=`, and *`effective-argument`* is:

- `x.native()` if `x` is present and is `const path&`, otherwise
- the effective range `source` ([path.req]), if `source` is present, otherwise,
- the range [`begin`,`end`), if `begin` and `end` are present, otherwise,
- `x`.

If the value type of *`effective-argument`* would not be `path::value_type`, the actual argument or argument range is first converted ([path.arg.convert]) so that *`effective-argument`* has value type `path::value_type`.

Returns: `*this`

8.4.5 `path` modifiers [path.modifiers]

```
void clear() noexcept;
```

Postcondition: `this->empty()` is true.

```
path& make_preferred();
```

Effects: Each *directory-separator* is converted to *preferred-separator*.

Returns: `*this`

[*Example:*

```
path p("foo/bar");
std::cout << p << '\n';
p.make_preferred();
std::cout << p << '\n';
```

On an operating system where *preferred-separator* is the same as *directory-separator*, 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();
```

Postcondition: `!has_filename()`.

Returns: `*this`.

[*Example:*

```
std::cout << path("/foo").remove_filename();  // outputs "/"
std::cout << path("/").remove_filename();     // outputs ""
```

--end example]

```
path& replace_filename(const path& replacement);
```

Effects:

```
remove_filename();
```

```
operator/=(replacement);
```

Returns: `*this`.

[*Example:*

```
std::cout << path("/foo").replace_filename("bar");  //
outputs "/bar"
std::cout << path("/").replace_filename("bar");     //
outputs "bar"
```

--end example]

```
path& replace_extension(const path& replacement = path());
```

Effects:

- Any existing `extension()` is removed from the stored path, then
- If `replacement` is not empty and does not begin with a dot character, a dot character is appended to the stored path, then
- `replacement` is concatinated to the stored path.

Returns: `*this`

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

Effects: Swaps the contents of the two paths.

Complexity: constant time.

8.4.6 `path` native format observers [path.native.obs]

The string returned by all native format observers is in the native pathname format.

```
const string_type&  native() const noexcept;
```

Returns: `pathname`.

```
const value_type* c_str() const noexcept;
```

Returns: `pathname.c_str()`.

```
operator string_type() const;
```

Returns: `pathname`.

[*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. This provides basic interoperability without the need to modify existing

standard library classes or headers. --*end note*]

```
template <class charT, class traits = char_traits<charT>,
          class Allocator = allocator<charT> >
basic_string<charT, traits, Allocator>
  string(const Allocator& a = Allocator()) const;
```

Returns: `pathname`.

Remarks: All memory allocation, including for the return value, shall be performed by `a`. Conversion, if any, is specified by [path.cvt]).

```
string string() const;
wstring wstring() const;
string u8string() const;
u16string u16string() const;
u32string u32string() const;
```

Returns: `pathname`.

Remarks: Conversion, if any, is performed as specified by [path.cvt]. The encoding of the string returned by `u8string()` is always UTF-8.

8.4.7 `path` generic format observers [path.generic.obs]

Generic format observer functions return strings formatted according to the generic pathname format ([path.generic]). The forward slash (`'/'`) character is used as the *directory-separator* character.

[*Example:* On an operating system that uses backslash as its preferred-separator, `path("foo\\bar").generic_string()` returns `"foo/bar"`. -- *end example*]

```
template <class charT, class traits = char_traits<charT>,
          class Allocator = allocator<charT> >
basic_string<charT, traits, Allocator>
  generic_string(const Allocator& a = Allocator()) const;
```

Returns: `pathname`, reformatted according to the generic pathname format ([path.generic]).

Remarks: All memory allocation, including for the return value, shall be performed by `a`. Conversion, if any, is specified by [path.cvt].

```
string generic_string() const;
wstring generic_wstring() const;
string generic_u8string() const;
u16string generic_u16string() const;
u32string generic_u32string() const;
```

Returns: `pathname`, reformatted according to the generic pathname format ([path.generic]).

Remarks: Conversion, if any, is specified by [path.cvt]. The encoding of the string returned by `generic_u8string()` is always UTF-8.

8.4.8 `path` compare [path.compare]

```
int compare(const path& p) const noexcept;
```

Returns: A value less than 0 if `native()` for the elements of `*this` are lexicographically less than `native()` for the elements of `p`, otherwise a value greater than 0 if `native()` for the elements of `*this` are lexicographically greater than `native()` for the elements of `p`, otherwise 0.

Remark: The elements are determined as if by iteration over the half-open range [`begin()`, `end()`) for `*this` and `p`.

```
int compare(const string& s) const
```

Returns: `compare(path(s))`.

```
int compare(const value_type* s) const
```

Returns: `compare(path(s))`.

8.4.9 `path` decomposition [path.decompose]

```
path root_name() const;
```

Returns: root-name, if `pathname` includes *root-name*, otherwise `path()`.

```
path root_directory() const;
```

Returns: root-directory, if `pathname` includes *root-directory*, otherwise `path()`.

If *root-directory* is composed of *slash name*, *slash* is excluded from the returned string.

```
path root_path() const;
```

Returns: `root_name() / root_directory()`

```
path relative_path() const;
```

Returns: A `path` composed from `pathname`, if `!empty()`, beginning with the first *filename* after *root-path*. Otherwise, `path()`.

```
path parent_path() const;
```

Returns: `(empty() || begin() == --end()) ? path() : pp`, where *pp* is constructed as if by starting with an empty `path` and successively applying

`operator/=` for each element in the range `begin()`, `--end()`.

```
path filename() const;
```

Returns: `empty() ? path() : *--end()`

[*Example:*

```
std::cout << path("/foo/bar.txt").filename(); // outputs
"bar.txt"
std::cout << path("/").filename();            // outputs "/"
std::cout << path(".").filename();            // outputs "."
std::cout << path("..").filename();           // outputs ".."
```

--end example]

```
path stem() const;
```

Returns: if `filename()` contains a character but does not consist solely of one or to two periods, returns the substring of `filename()` starting at its beginning and ending with the character before the last period. Otherwise, returns `filename()`.

[*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;
```

Returns: if `filename()` contains a period but does not consist solely of one or to two periods, returns the substring of `filename()` starting at the rightmost period and for the remainder of the path. Otherwise, returns an empty `path` object.

Remarks: Implementations are permitted to define additional behavior for file systems which append additional elements to extensions, such as alternate data streams or partitioned dataset names.

[*Example:*

```
std::cout << path("/foo/bar.txt").extension(); // outputs
".txt"
```

--end example]

[*Note:* The period is included in the return value so that it is possible to distinguish between no extension and an empty extension. Also note that for a path `p`, `p.stem()` `+p.extension() == p`. -- *end note*]

8.4.10 `path` query [path.query]

```
bool empty() const noexcept;
```

Returns: `m_pathname.empty()`.

```
bool has_root_path() const;
```

Returns: `!root_path().empty()`

```
bool has_root_name() const;
```

Returns: `!root_name().empty()`

```
bool has_root_directory() const;
```

Returns: `!root_directory().empty()`

```
bool has_relative_path() const;
```

Returns: `!relative_path().empty()`

```
bool has_parent_path() const;
```

Returns: `!parent_path().empty()`

```
bool has_filename() const;
```

Returns: `!filename().empty()`

```
bool has_stem() const;
```

Returns: `!stem().empty()`

```
bool has_extension() const;
```

Returns: `!extension().empty()`

```
bool is_absolute() const;
```

Returns: `true` if the elements of `root_path()` uniquely identify a file system location, else `false`.

[*Example:* `path("/").is_absolute()` is `true` for POSIX based operating systems, and `false` for Windows based operating systems. --*end example*]

```
bool is_relative() const;
```

Returns: `!is_absolute()`.

8.5 `path` iterators [path.itr]

Path iterators iterate over the elements of the stored pathname.

A `path::iterator` is a constant iterator satisfying all the requirements of a bidirectional iterator (C++ Std, 24.1.4 Bidirectional iterators [lib.bidirectional.iterators]). Its `value_type` is `path`.

Calling any non-const member function of a `path` object invalidates all iterators referring to elements of that object.

The forward traversal order is as follows:

- The *root-name* element, if present.
- The *root-directory* element, if present, in the generic format. *[note:* the generic format is required to ensure lexicographical comparison works correctly. -- *end note*]
- Each successive *filename* element, if present.
- *Dot*, if one or more trailing non-root *slash* characters are present.

The backward traversal order is the reverse of forward traversal.

```
iterator begin() const;
```

Returns: An iterator for the first present element in the traversal list above. If no elements are present, the end iterator.

```
iterator end() const;
```

Returns: The end iterator.

8.6 `path` non-member functions [path.non-member]

```
void swap(path& lhs, path& rhs) noexcept;
```

Effects: `lhs.swap(rhs)`.

```
size_t hash_value (const path& p) noexcept;
```

Returns: A hash value for the path `p`. If for two paths, `p1 == p2` then `hash_value(p1) == hash_value(p2)`.

```
bool operator< (const path& lhs, const path& rhs) noexcept;
```

Returns: `return lhs.compare(rhs) < 0`.

```
bool operator<=(const path& lhs, const path& rhs) noexcept;
```

Returns: `!(rhs < lhs)`.

```
bool operator> (const path& lhs, const path& rhs) noexcept;
```

Returns: `rhs < lhs`.

```
bool operator>=(const path& lhs, const path& rhs) noexcept;
```

Returns: `!(lhs < rhs)`.

```
bool operator==(const path& lhs, const path& rhs) noexcept;
```

Returns: `!(lhs < rhs) && !(rhs < lhs)`.

[*Note:* Path equality and path equivalence have different semantics.

Equality is determined by the `path` non-member `operator==`, which considers the two path's lexical representations only. Thus `path("foo") == "bar"` is never `true`.

Equivalence is determined by the `equivalent()` non-member function, which determines if two paths resolve to the same file system entity. Thus `equivalent("foo", "bar")` will be `true` when both paths resolve to the same file.

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*]

```
bool operator!=(const path& lhs, const path& rhs) noexcept;
```

Returns: `!(lhs == rhs)`.

```
path operator/ (const path& lhs, const path& rhs);
```

Returns: `path(lhs) /= rhs`.

8.6.1 `path` inserter and extractor [path.io]

```
template <class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const path& p);
```

Effects: `os << p.string<charT, traits>()`.

[*Note:* Pathnames containing spaces require special handling by the user to avoid truncation when read by the extractor. --*end note*]

```
template <class charT, class traits>
basic_istream<charT, traits>&
operator>>(basic_istream<charT, traits>& is, path& p);
```

Effects:

```
basic_string<charT, traits> tmp;
is >> tmp;
p = tmp;
```

Returns: `is`

8.6.2 `path` factory functions [path.factory]

```
template <class Source>
  path u8path(Source const& source);
template <class InputIterator>
  path u8path(InputIterator begin, InputIterator end);
```

Requires: The `source` and [`begin`,`end`) sequences are UTF-8 encoded. The value type of `Source` and `InputIterator` is `char`.

Returns:

- If `value_type` is `char` and the current native narrow encoding ([fs.def.native.encode]) is UTF-8, `path(source)` or `path(begin, end)`, else
- if `value_type` is `wchar_t` and the native wide encoding is UTF-16, or if `value_type` is `char16_t` or `char32_t`, convert `source` or [`begin`,`end`) to a temporary, `tmp`, of type `string_type` and return `path(tmp)`, else
- convert `source` or [`begin`,`end`) to a temporary, `tmp`, of type `u32string` and return `path(tmp)`.

Remarks: Argument format conversion ([path.fmt.cvt]) applies to the arguments for these functions. How Unicode encoding conversions are performed is unspecified.

[*Example*:

A string is to be read from a database that is encoded in UTF-8, and used to create a directory using the native encoding for filenames:

```
namespace fs = std::tbd::filesystem;
std::string utf8_string = read_utf8_data();
fs::create_directory(fs::u8path(utf8_string));
```

For POSIX based operating systems with the native narrow encoding set to UTF-8, no encoding or type conversion occurs.

For POSIX based operating systems with the native narrow encoding not set to UTF-8, a conversion to UTF-32 occurs, followed by a conversion to the current native narrow encoding. Some Unicode characters may have no native character set representation.

For Windows based operating systems a conversion from UTF-8 to UTF-16 occurs.

--end example]

9 Class `filesystem_error` [class.filesystem_error]

```
namespace std { namespace tbd { namespace 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;
      };
} } }  // namespaces std::tbd::filesystem
```

The class template `filesystem_error` defines the type of objects thrown as exceptions to report file system errors from functions described in this Technical Specification.

9.1 `filesystem_error` members [filesystem_error.members]

Constructors are provided that store zero, one, or two paths associated with an error.

```
filesystem_error(const string& what_arg, error_code ec);
```

Postcondition:

Expression	**Value**
`runtime_error::what()`	`what_arg.c_str()`
`code()`	`ec`
`path1().empty()`	`true`

`path2().empty()`	`true`

```
filesystem_error(const string& what_arg, const path& p1, error_code ec);
```

Postcondition:

Expression	**Value**
`runtime_error::what()`	`what_arg.c_str()`
`code()`	`ec`
`path1()`	Reference to stored copy of `p1`
`path2().empty()`	`true`

```
filesystem_error(const string& what_arg, const path& p1, const path& p2,
error_code ec);
```

Postcondition:

Expression	**Value**
`runtime_error::what()`	`what_arg.c_str()`
`code()`	`ec`
`path1()`	Reference to stored copy of `p1`
`path2()`	Reference to stored copy of `p2`

```
const path& path1() const noexcept;
```

Returns: Reference to copy of `p1` stored by the constructor, or, if none, an empty path.

```
const path& path2() const noexcept;
```

Returns: Reference to copy of `p2` stored by the constructor, or, if none, an empty path.

```
const char* what() const noexcept;
```

Returns: A string containing `runtime_error::what()`. The exact format is unspecified. Implementations are encouraged but not required to include `path1.native_string()` if not empty, `path2.native_string()` if not empty, and `system_error::what()` strings in the returned string.

10 Enumerations [fs.enum]

10.1 Enum class `file_type` [enum.file_type]

This enum class specifies constants used to identify file types.

Constant Name	Value	Meaning
`none`	`0`	The type of the file has not been determined or an error occurred while trying to determine the type.
`not_found`	`-1`	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`	`1`	Regular file
`directory`	`2`	Directory file
`symlink`	`3`	Symbolic link file
`block`	`4`	Block special file
`character`	`5`	Character special file
`fifo`	`6`	FIFO or pipe file
`socket`	`7`	Socket file
`unknown`	`8`	The file does exist, but is of an operating system dependent type not covered by any of the other cases or the process does not have permission to query the file type

10.2 Enum class `copy_options` [enum.copy_options]

This enumeration specifies bitmask constants used to control the semantics of copy operations. The constants are specified in option groups. Constant `none` is shown in each option group for purposes of exposition; implementations shall provide only a single definition. Calling a Filesystem library function with more than a single constant for an option group results in undefined behavior.

Option group controlling `copy_file` effects for existing target files		
Constant	**Value**	**Meaning**
`none`	0	(Default) Error; file already exists.
`skip_existing`	1	Do not overwrite existing file, do not report an error.
`overwrite_existing`	2	Overwrite the existing file.
`update_existing`	4	Overwrite the existing file if it is older than the replacement file.
Option group controlling `copy` effects for sub-directories		
Constant	**Value**	**Meaning**
`none`	0	(Default) Do not copy sub-directories.
`recursive`	8	Recursively copy sub-directories and their contents.
Option group controlling `copy` effects symbolic link actions		
Constant	**Value**	**Meaning**
`none`	0	(Default) Follow symbolic links.
`copy_symlinks`	16	Copy symbolic links as symbolic links rather than copying the files that they point to.
`skip_symlinks`	32	Ignore symbolic links.
Option group controlling `copy` effects choosing the form of copying		

Constant	Value	Meaning
`none`	0	(Default) Copy content.
`directories_only`	64	Copy directory structure only, do not copy non-directory files.
`create_symlinks`	128	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`	256	Make hard links instead of copies of files.

10.3 Enum class `perms` [enum.perms]

This enumeration specifies bitmask constants used to identify file permissions.

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_exe`
`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_exe`
`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_exe`
`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
`add_perms`	`0x10000`		`permissions()` shall bitwise *or* the `perm` argument's permission bits to the file's current permission bits.
`remove_perms`	`0x20000`		`permissions()` shall bitwise *and* the complement of `perm` argument's permission bits to the file's current permission bits.
`resolve_symlinks`	`0x40000`		`permissions()` shall resolve symlinks

11 Class `file_status` [class.file_status]

```
namespace std { namespace tbd { namespace filesystem {

      class file_status
      {
      public:

        // constructors
        explicit file_status(file_type ft = file_type::none,
                             perms prms = perms::unknown) noexcept;
        file_status(const file_status&) noexcept = default;
        file_status(file_status&&) noexcept = default;
       ~file_status() = default;

        file_status& operator=(const file_status&) noexcept = default;
        file_status& operator=(file_status&&) noexcept = default;

        // observers
        file_type  type() const noexcept;
        perms      permissions() const noexcept;

        // modifiers
        void       type(file_type ft) noexcept;
        void       permissions(perms prms) noexcept;
      };
} } }  // namespaces std::tbd::filesystem
```

An object of type `file_status` stores information about the type and permissions of a file.

11.1 `file_status` constructors [file_status.cons]

```
explicit file_status() noexcept;
```

Postconditions: `type() == file_type::none`, `permissions() == perms_not_known`.

```
explicit file_status(file_type ft, perms prms = perms_not_known) noexcept;
```

Postconditions: `type() == ft`, `permissions() == prms`.

11.2 `file_status` observers [file_status.obs]

```
file_type type() const noexcept;
```

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;
```

Returns: The value of `permissions()` specified by the *postconditions* of the most recent call to a constructor, operator=, or `permissions(perms)` function.

11.3 `file_status` modifiers [file_status.mods]

```
void type(file_type ft) noexcept;
```

Postconditions: `type() == ft`.

```
void permissions(perms prms) noexcept;
```

Postconditions: `permissions() == prms`.

12 Class `directory_entry` [class.directory_entry]

```
namespace std { namespace tbd { namespace filesystem {

      class directory_entry
      {
      public:

        // constructors and destructor
        directory_entry() = default;
        directory_entry(const directory_entry&) = default;
        directory_entry(directory_entry&&) noexcept = default;
        explicit directory_entry(const path& p, file_status st=file_status(),
          file_status symlink_st=file_status());
       ~directory_entry() = default;

        // modifiers
        directory_entry& operator=(const directory_entry&) = default;
        directory_entry& operator=(directory_entry&&) noexcept = default;
        void assign(const path& p, file_status st=file_status(),
          file_status symlink_st=file_status());
        void replace_filename(const path& p, file_status st=file_status(),
          file_status symlink_st=file_status());

        // observers
        const path&  path() 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:
        path                m_path;       // for exposition only
        mutable file_status  m_status;     // for exposition only; stat()-like
        mutable file_status  m_symlink_status; // for exposition only;
                                               //  lstat()-like
      };

} } }  // namespaces std::tbd::filesystem
```

A `directory_entry` object stores a `path object`, a `file_status` object for non-symbolic link status, and a `file_status` object for symbolic link status. The `file_status` objects act as value caches.

> [*Note:* Because `status()` on a pathname may be a relatively expensive operation, some operating systems provide status information as a byproduct of directory iteration. Caching such status information can result in significant time savings. Cached and non-cached results may differ in the presence of file system races. -- *end note*]

12.1 `directory_entry` constructors [directory_entry.cons]

```
explicit directory_entry(const path& p, file_status st=file_status(),
                         file_status symlink_st=file_status());
```

Postcondition:

Expression	**Value**
`path()`	`p`
`status()`	`st`
`symlink_status()`	`symlink_st`

12.2 `directory_entry` modifiers [directory_entry.mods]

```
void assign(const path& p, file_status st=file_status(),
            file_status symlink_st=file_status());
```

Postcondition:

Expression	**Value**
`path()`	`p`
`status()`	`st`
`symlink_status()`	`symlink_st`

```
void replace_filename(const path& p, file_status st=file_status(),
                      file_status symlink_st=file_status());
```

Postcondition:

Expression	Value
`path()`	`path().parent_path() / s`
`status()`	`st`
`symlink_status()`	`symlink_st`

12.3 `directory_entry` observers [directory_entry.obs]

```
const path& path() const noexcept;
```

Returns: `m_path`

```
file_status status() const;
file_status status(error_code& ec) const noexcept;
```

Effects: As if,

```
if (!status_known(m_status))
{
  if (status_known(m_symlink_status) && !
is_symlink(m_symlink_status))
    { m_status = m_symlink_status; }
  else { m_status = status(m_path[, ec]); }
}
```

Returns: `m_status`

Throws: As specified in Error reporting.

```
file_status  symlink_status() const;
file_status  symlink_status(error_code& ec) const noexcept;
```

Effects: As if,

```
if (!status_known(m_symlink_status))
{
  m_symlink_status = symlink_status(m_path[, ec]);
}
```

Returns: `m_symlink_status`

Throws: As specified in Error reporting.

```
bool operator==(const directory_entry& rhs) const noexcept;
```

Returns: `m_path == rhs.m_path`.

```
bool operator!=(const directory_entry& rhs) const noexcept;
```

Returns: `m_path != rhs.m_path`.

```
bool operator< (const directory_entry& rhs) const noexcept;
```

Returns: `m_path < rhs.m_path`.

```
bool operator<=(const directory_entry& rhs) const noexcept;
```

Returns: `m_path <= rhs.m_path`.

```
bool operator> (const directory_entry& rhs) const noexcept;
```

Returns: `m_path > rhs.m_path`.

```
bool operator>=(const directory_entry& rhs) const noexcept;
```

Returns: `m_path >= rhs.m_path`.

13 Class `directory_iterator` [class.directory_iterator]

An object of type `directory_iterator` provides an iterator for a sequence of `directory_entry` elements representing the files in a directory. [*Note:* For iteration into sub-directories, see class `recursive_directory_iterator` ([class.rec.dir.itr]). --*end note*]

```
namespace std { namespace tbd { namespace filesystem {

      class directory_iterator :
        public iterator<input_iterator_tag, directory_entry>
      {
      public:
        // member functions
        directory_iterator() noexcept;
        explicit directory_iterator(const path& p);
        directory_iterator(const path& p, error_code& ec) noexcept;
        directory_iterator(const directory_iterator&) = default;
        directory_iterator(directory_iterator&&) = default;
       ~directory_iterator() = default;

        directory_iterator& operator=(const directory_iterator&) = default;
        directory_iterator& operator=(directory_iterator&&) = default;

        const directory_entry& operator*() const;
        const directory_entry* operator->() const;
        directory_iterator&    operator++();
        directory_iterator&    increment(error_code& ec) noexcept;

        // other members as required by
        //  C++ Std, 24.1.1 Input iterators [input.iterators]
      };

} } }  // namespaces std::tbd::filesystem
```

`directory_iterator` satisfies the requirements of an input iterator ([input.iterators]).

If an iterator of type `directory_iterator` 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.

The result of `operator*` on an end iterator is undefined behavior. For any other iterator value a `const directory_entry&` is returned. The result of `operator->` on an end iterator is undefined behavior. For any other iterator value a `const directory_entry*` is returned.

Two end iterators are always equal. An end iterator shall not be equal to a non-end iterator.

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/=`.

Directory iteration shall not yield directory entries for the current (*dot*) and parent (*dot-dot*) directories.

The order of directory entries obtained by dereferencing successive increments of a `directory_iterator` is unspecified.

> [*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.
>
> 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*]

13.1 `directory_iterator` members [directory_iterator.members]

```
directory_iterator() noexcept;
```

> *Effects:* Constructs the end iterator.

```
explicit directory_iterator(const path& p);
directory_iterator(const path& p, error_code& ec) noexcept;
```

> *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.

Throws: As specified in Error reporting.

[*Note:* To iterate over the current directory, use `directory_iterator(".")` rather than `directory_iterator("")`. -- *end note*]

```
directory_iterator& operator++();
directory_iterator& increment(error_code& ec) noexcept;
```

Effects: As specified by the C++ Standard, 24.1.1 Input iterators [input.iterators]

Returns: `*this`.

Throws: As specified in Error reporting.

13.2 `directory_iterator` non-member functions [directory_iterator.nonmembers]

These functions enable use of `directory_iterator` with C++11 range-based for statements.

```
const directory_iterator& begin(const directory_iterator& iter) noexcept;
```

Returns: `iter`.

```
directory_iterator end(const directory_iterator&) noexcept;
```

Returns: `directory_iterator()`.

14 Class `recursive_directory_iterator` [class.rec.dir.itr]

An object of type `recursive_directory_iterator` provides an iterator for a sequence of `directory_entry` elements representing the files in a directory and its sub-directories.

```
namespace std { namespace tbd { namespace filesystem {

      class recursive_directory_iterator :
        public iterator<input_iterator_tag, directory_entry>
      {
      public:

        // constructors and destructor
        recursive_directory_iterator() noexcept;
        recursive_directory_iterator(const recursive_directory_iterator&)
          = default;
        explicit recursive_directory_iterator(const path& p,
          directory_options options = directory_options::none);
        recursive_directory_iterator(const path& p,
          directory_options options, error_code& ec) noexcept;
        recursive_directory_iterator(const path& p, error_code& ec) noexcept;
        recursive_directory_iterator(recursive_directory_iterator&&) = default;
```

```
      ~recursive_directory_iterator() = default;

       // observers
       directory_options  options() const;
       int                depth() const;
       bool               recursion_pending() const;

       // modifiers
       recursive_directory_iterator& operator=(
         const recursive_directory_iterator&) = default;
       recursive_directory_iterator& operator=(
         recursive_directory_iterator&&) = default;

       recursive_directory_iterator& operator++();
       recursive_directory_iterator& increment(error_code& ec);

       void pop();
       void disable_recursion_pending();

       // ... other members as required for Input iterators [input.iterators]
     };

} } }  // namespaces std::tbd::filesystem
```

The behavior of a `recursive_directory_iterator` is the same as a `directory_iterator` unless otherwise specified.

14.1 `recursive_directory_iterator` members [rec.dir.itr.members]

```
recursive_directory_iterator() noexcept;
```

Effects: Constructs the end iterator.

```
explicit recursive_directory_iterator(const path& p,
  directory_options options = directory_options::none);
recursive_directory_iterator(const path& p, directory_options options,
error_code& ec);
recursive_directory_iterator(const path& p, error_code& ec);
```

Effects: Constructs a iterator representing the first entry in the directory `p` resolves to, if any; otherwise, the end iterator.

Postcondition: `options() == options` for the signatures with a `directory_options` argument, otherwise `options() == directory_options::none`.

Throws: As specified in Error reporting.

[*Note:* To iterate over the current directory, use `recursive_directory_iterator(".")` rather than `recursive_directory_iterator("")`. -- *end note*]

[*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*]

```
directory_options options() const;
```

Requires: `*this != recursive_directory_iterator()`.

Returns: The value of the constructor `options` argument, if present, otherwise `directory_options::none`.

Throws: Nothing.

```
int depth() const;
```

Requires: `*this != recursive_directory_iterator()`.

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*]

Throws: Nothing.

```
bool recursion_pending() const;
```

Requires: `*this != recursive_directory_iterator()`.

Returns: `true` if `disable_recursion_pending()` has not been called subsequent to the prior construction or increment operation, otherwise `false`.

Throws: Nothing.

```
recursive_directory_iterator& operator++();
recursive_directory_iterator& increment(error_code& ec);
```

Requires: `*this != recursive_directory_iterator()`.

Effects: As specified by the C++ Standard, 24.1.1 Input iterators [input.iterators], except that:

- If there are no more entries at this depth, then if `depth() != 0` iteration over the parent directory resumes; otherwise `*this = recursive_directory_iterator()`.
- Otherwise if `recursion_pending() && is_directory(this->status()) && (!is_symlink(this->symlink_status()) || (options() & directory_options::follow_directory_symlink) != 0)` then directory `(*this)->path()` is recursively iterated into.

Returns: `*this`.

Throws: As specified in Error reporting.

```
void pop();
```

Requires: `*this != recursive_directory_iterator()`.

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.

```
void disable_recursion_pending();
```

Requires: `*this != recursive_directory_iterator()`.

Postcondition: `recursion_pending() == false`.

[*Note:* `disable_recursion_pending()` is used to prevent unwanted recursion into a directory. *--end note*]

14.2 `recursive_directory_iterator` non-member functions [rec.dir.itr.nonmembers]

These functions enable use of `recursive_directory_iterator` with C++11 range-based for statements.

```
const recursive_directory_iterator& begin(const recursive_directory_iterator&
iter) noexcept;
```

Returns: `iter`.

```
recursive_directory_iterator end(const recursive_directory_iterator&) noexcept;
```

Returns: `recursive_directory_iterator()`.

15 Operational functions [fs.op.funcs]

Operational functions query or modify files, including directories, in external storage.

[*Note:* Because hardware failures, network failures, file system races, and many other kinds of errors occur frequently in file system operations, users should be aware that any filesystem operational function, no matter how apparently innocuous, may encounter an error. See Error reporting. *-- end note*]

15.1 Absolute [fs.op.absolute]

```
path absolute(const path& p, const path& base=current_path());
```

Returns: An absolute path composed according to the following table

	p.has_root_directory()	**!p.has_root_directory()**
p.has_root_name()	`return p`	`return p.root_name() / absolute(base).root_directory() / absolute(base).relative_path() / p.relative_path()`
!p.has_root_name()	`return absolute(base).root_name() / p`	`return absolute(base) / p`

[*Note:* For the returned path, `rp, rp.is_absolute()` is true. -- *end note*]

Throws: If `base.is_absolute()` is true, throws only if memory allocation fails.

15.2 Canonical [fs.op.canonical]

```
path canonical(const path& p, const path& base = current_path());
path canonical(const path& p, error_code& ec);
path canonical(const path& p, const path& base, error_code& ec);
```

Overview: Converts `p`, which must exist, to an absolute path that has no symbolic link, "`.`", or "`..`" elements.

Returns: A path that refers to the same file system object as `absolute(p,base)`. For the overload without a `base` argument, `base` is `current_path()`. Signatures with argument `ec` return `path()` if an error occurs.

Throws: As specified in Error reporting.

Remarks: `!exists(p)` is an error.

[*Note:* Canonical pathnames allow security checking of a path (e.g. does this path live in /home/goodguy or /home/badguy?) -- *end note*]

15.3 Copy [fs.op.copy]

```
void copy(const path& from, const path& to);
void copy(const path& from, const path& to, error_code& ec) noexcept;
```

Effects: `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) noexcept;
```

Precondition: At most one constant from each option group ([enum.copy_options]) is present in `options`.

Effects:

Before the first use of `f` and `t`:

- If `(options & copy_options::create_symlinks) || (options & copy_options::skip_symlinks)`, then `auto f = symlink_status(from)` and if needed `auto t = symlink_status(to)`.
- Otherwise, `auto f = status(from)` and if needed `auto t = status(to)`.

Report an error as specified in Error reporting if:

- `!exists(f)`.
- `equivalent(f, t)`.
- `is_other(f) || is_other(t)`.
- `is_directory(f) && is_regular_file(t)`.

If `is_symlink(f)`, then:

- If `options & copy_options::skip_symlinks`, then return.
- Otherwise if `!exists(t)`, then `copy_symlink(from, to, options)`.
- Otherwise report an error as specified in Error reporting.

Otherwise if `is_regular_file(f)`, then:

- If `options & copy_options::directories_only`, then return.
- Otherwise if `options & copy_options::create_symlinks`, then create a symbolic link to the source file.

- Otherwise if `options & copy_options::create_hard_links`, then create a hard link to the source file.
- Otherwise:
  - If `is_directory(t)`, then `copy_file(from, to/from.filename(), options)`.
  - Otherwise, `copy_file(from, to, options)`.

Otherwise if `is_directory(f) && ((options & copy_options::recursive) || !(options))` then:

- If `!exists(t)`, then `create_directory(to, from)`.
- Then, iterate over the files in `from`, as if by `for (directory_entry& x : directory_iterator(from))`, and for each iteration `copy(x.path(), to/x.path().filename(), options | copy_options::`*`unspecified`*`)`.

Otherwise no effects.

Throws: As specified in Error reporting.

Remarks: For the signature with argument `ec`, any Filesystem library functions called by the implementation shall have an `error_code` argument if applicable.

[*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]

15.4 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) noexcept;
```

Returns: `copy_file(from, to, copy_options::none[, ec])`.

Throws: As specified in Error reporting.

```
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) noexcept;
```

Precondition: At most one constant from each `copy_options` option group ([enum.copy_options]) is present in `options`.

Effects:

If `exists(to) && !(options & (copy_options::skip_existing | copy_options::overwrite_existing | copy_options::update_existing))` report a file already exists error as specified in Error reporting.

If `!exists(to) || (options & copy_options::overwrite_existing) || ((options & copy_options::update_existing) && last_write_time(from) > last_write_time(to)) || !(options & (copy_options::skip_existing | copy_options::overwrite_existing | copy_options::update_existing))` copy the contents and attributes of the file `from` resolves to the file `to` resolves to.

Returns: `true` if the `from` file was copied, otherwise `false`. The signature with argument `ec` return `false` if an error occurs.

Throws: As specified in Error reporting.

Complexity: At most one direct or indirect invocation of `status(to)`.

15.5 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;
```

Effects: *function*(read_symlink(existing_symlink*[, ec]*), new_symlink*[, ec]*), where *function* is create_symlink or create_directory_symlink, as appropriate.

Throws: As specified in Error reporting.

15.6 Create directories [fs.op.create_directories]

```
bool create_directories(const path& p);
bool create_directories(const path& p, error_code& ec) noexcept;
```

Effects: Establishes the postcondition by calling create_directory() for any element of p that does not exist.

Postcondition: is_directory(p)

Returns: true if a new directory was created, otherwise false. The signature with argument ec returns false if an error occurs.

Throws: As specified in Error reporting.

Complexity: *O(n+1)* where *n* is the number of elements of p that do not exist.

15.7 Create directory [fs.op.create_directory]

```
bool create_directory(const path& p);
bool create_directory(const path& p, error_code& ec) noexcept;
```

Effects: Establishes the postcondition by attempting to create the directory p resolves to, as if by POSIX mkdir() with a second argument of S_IRWXU|S_IRWXG|S_IRWXO. Creation failure because p resolves to an existing directory shall not be treated as an error.

Postcondition: is_directory(p)

Returns: true if a new directory was created, otherwise false. The signature with argument ec returns false if an error occurs.

Throws: As specified in Error reporting.

```
bool create_directory(const path& p, const path& existing_p);
bool create_directory(const path& p, const path& existing_p, error_code& ec)
```

```
noexcept;
```

Effects: Establishes the postcondition by attempting to create the directory `p` resolves to, with attributes copied from directory `existing_p`. The set of attributes copied is operating system dependent. Creation failure because `p` resolves to an existing directory shall not be treated as an error.

[*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*]

Postcondition: `is_directory(p)`

Returns: `true` if a new directory was created, otherwise `false`. The signature with argument `ec` returns false if an error occurs.

Throws: As specified in Error reporting.

15.8 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;
```

Effects: Establishes the postcondition, as if by POSIX `symlink()`.

Postcondition: `new_symlink` resolves to a symbolic link file that contains an unspecified representation of `to`.

Throws: As specified in Error reporting.

[*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*]

[*Note:* Some operating systems do not support symbolic links at all or support them only for regular files. Some file systems do not support symbolic links regardless of the operating system - the FAT file system used on memory cards and flash drives, for example. -- *end note*]

15.9 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;
```

Effects: Establishes the postcondition, as if by POSIX `link()`.

Postcondition:

- `exists(to) && exists(new_hard_link) && equivalent(to, new_hard_link)`
- The contents of the file or directory `to` resolves to are unchanged.

Throws: As specified in Error reporting.

[*Note:* Some operating systems do not support hard links at all or support them only for regular files. Some file systems do not support hard links regardless of the operating system - the FAT file system used on memory cards and flash drives, for example. Some file systems limit the number of links per file. -- *end note*]

15.10 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;
```

Effects: Establishes the postcondition, as if by POSIX `symlink()`.

Postcondition: `new_symlink` resolves to a symbolic link file that contains an unspecified representation of `to`.

Throws: As specified in Error reporting.

[*Note:* Some operating systems do not support symbolic links at all or support them only for regular files. Some file systems do not support symbolic links regardless of the operating system - the FAT system used on memory cards and flash drives, for example. -- *end note*]

15.11 Current path [fs.op.current_path]

```
path current_path();
path current_path(error_code& ec);
```

Returns: The absolute path of the current working directory, obtained as if by POSIX `getcwd()`. The signature with argument `ec` returns `path()` if an error occurs.

Throws: As specified in Error reporting.

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.

[*Note:* The `current_path()` name was chosen to emphasize that the return is a path, not just a single directory name.

The current path as returned by many operating systems is a dangerous global variable. It may be changed unexpectedly by a 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;
```

Effects: Establishes the postcondition, as if by POSIX `chdir()`.

Postcondition: `equivalent(p, current_path())`.

Throws: As specified in Error reporting.

[*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*]

15.12 Exists [fs.op.exists]

```
bool exists(file_status s) noexcept;
```

Returns: `status_known(s) && s.type() != file_type::not_found`

```
bool exists(const path& p);
bool exists(const path& p, error_code& ec) noexcept;
```

Returns: `exists(status(p))` or `exists(status(p, ec))`, respectively. The signature with argument `ec` returns `false` if an error occurs.

Throws: As specified in Error reporting.

15.13 Equivalent [fs.op.equivalent]

```
bool equivalent(const path& p1, const path& p2);
bool equivalent(const path& p1, const path& p2, error_code& ec) noexcept;
```

Effects: Determines `file_status s1` and `s2`, as if by `status(p1)` and `status(p2)`, respectively.

Returns: `true`, if `sf1 == sf2` and `p1` and `p2` resolve to the same file system entity, else `false`. The signature with argument `ec` returns `false` if an error occurs.

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. This is determined as if by the values of the POSIX `stat` structure, obtained as if

by `stat()` for the two paths, having equal `st_dev` values and equal `st_ino` values.

Throws: `filesystem_error` if `(!exists(s1) && !exists(s2)) || (is_other(s1) && is_other(s2))`, otherwise as specified in Error reporting.

15.14 File size [fs.op.file_size]

```
uintmax_t file_size(const path& p);
uintmax_t file_size(const path& p, error_code& ec) noexcept;
```

Returns: If `exists(p) && is_regular_file(p)`, the size in bytes of the file `p` resolves to, determined as if by the value of the POSIX `stat` structure member `st_size` obtained as if by POSIX `stat()`. Otherwise, `static_cast<uintmax_t>(-1)`. The signature with argument `ec` returns `static_cast<uintmax_t>(-1)` if an error occurs.

Throws: As specified in Error reporting.

15.15 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;
```

Returns: The number of hard links for `p`. The signature with argument `ec` returns `static_cast<uintmax_t>(-1)` if an error occurs.

Throws: As specified in Error reporting.

15.16 Is block file [fs.op.is_block_file]

```
bool is_block_file(file_status s) noexcept;
```

Returns: `s.type() == file_type::block`

```
bool is_block_file(const path& p);
bool is_block_file(const path& p, error_code& ec) noexcept;
```

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.

Throws: As specified in Error reporting.

15.17 Is character file [fs.op.is_char_file]

```
bool is_character_file(file_status s) noexcept;
```

Returns: `s.type() == file_type::character`

```
bool is_character_file(const path& p);
bool is_character_file(const path& p, error_code& ec) noexcept;
```

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.

Throws: As specified in Error reporting.

15.18 Is directory [fs.op.is_directory]

```
bool is_directory(file_status s) noexcept;
```

Returns: `s.type() == file_type::directory`

```
bool is_directory(const path& p);
bool is_directory(const path& p, error_code& ec) noexcept;
```

Returns: `is_directory(status(p))` or `is_directory(status(p, ec))`, respectively. The signature with argument `ec` returns `false` if an error occurs.

Throws: As specified in Error reporting.

15.19 Is empty [fs.op.is_empty]

```
bool is_empty(const path& p);
bool is_empty(const path& p, error_code& ec) noexcept;
```

Effects: Determines `file_status s`, as if by `status(p, ec)`.

Returns:
```
is_directory(s)
          ? directory_iterator(p) == directory_iterator()
          : file_size(p) == 0;
```

The signature with argument `ec` returns `false` if an error occurs.

Throws: As specified in Error reporting.

15.20 Is fifo [fs.op.is_fifo]

```
bool is_fifo(file_status s) noexcept;
```

Returns: `s.type() == file_type::fifo`

```
bool is_fifo(const path& p);
bool is_fifo(const path& p, error_code& ec) noexcept;
```

Returns: `is_fifo(status(p))` or `is_fifo(status(p, ec))`, respectively. The signature with argument `ec` returns `false` if an error occurs.

Throws: As specified in Error reporting.

15.21 Is other [fs.op.is_other]

```
bool is_other(file_status s) noexcept;
```

Returns: `return 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;
```

Returns: `is_other(status(p))` or `is_other(status(p, ec))`, respectively. The signature with argument `ec` returns `false` if an error occurs.

Throws: As specified in Error reporting.

15.22 Is regular file [fs.op.is_regular_file]

```
bool is_regular_file(file_status s) noexcept;
```

Returns: `s.type() == file_type::regular`.

```
bool is_regular_file(const path& p);
```

Returns: `is_regular_file(status(p))`.

Throws: `filesystem_error` if `status(p)` would throw `filesystem_error`.

```
bool is_regular_file(const path& p, error_code& ec) noexcept;
```

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*]

Returns: `is_regular_file(status(p, ec))`. Returns `false` if an error occurs.

15.23 Is socket [fs.op.is_socket]

```
bool is_socket(file_status s) noexcept;
```

Returns: `s.type() == file_type::socket`

```
bool is_socket(const path& p);
bool is_socket(const path& p, error_code& ec) noexcept;
```

Returns: `is_socket(status(p))` or `is_socket(status(p, ec))`, respectively. The signature with argument `ec` returns `false` if an error occurs.

Throws: As specified in Error reporting.

15.24 Is symlink [fs.op.is_symlink]

```
bool is_symlink(file_status s) noexcept;
```

Returns: `s.type() == file_type::symlink`

```
bool is_symlink(const path& p);
bool is_symlink(const path& p, error_code& ec) noexcept;
```

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.

Throws: As specified in Error reporting.

15.25 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;
```

Returns: The time of last data modification of `p`, determined as if by the value of the POSIX `stat` structure member `st_mtime` obtained as if by POSIX `stat()`. The signature with argument `ec` returns `static_cast<file_time_type>(-1)` if an error occurs.

Throws: As specified in Error reporting.

```
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;
```

Effects: Sets the time of last data modification of the file resolved to by `p` to `new_time`, as if by POSIX `stat()` followed by POSIX `utime()`.

Throws: As specified in Error reporting.

[*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*]

15.26 Permissions [fs.op.permissions]

```
void permissions(const path& p, perms prms);
void permissions(const path& p, perms prms, error_code& ec) noexcept;
```

Requires: `!((prms & add_perms) && (prms & remove_perms))`.

Effects: Applies the effective permissions bits from `prms` to the file `p` resolves to, as if by POSIX `fchmodat()`. The effective permission bits are determined as specified by the following table.

bits present in `prms`	**Effective bits applied**
Neither `add_perms` nor `remove_perms`	`prms & perms_mask`
`add_perms`	`status(p).permissions() | (prms & perms_mask)`
`remove_perms`	`status(p).permissions() & ~(prms & perms_mask)`

[*Note:* Conceptually permissions are viewed as bits, but the actual implementation may use some other mechanism. -- *end note*]

Throws: As specified in Error reporting.

15.27 Read symlink [fs.op.read_symlink]

```
path read_symlink(const path& p);
path read_symlink(const path& p, error_code& ec);
```

Returns: If `p` resolves to a symbolic link, a `path` object containing the contents of that symbolic link. Otherwise `path()`. The signature with argument `ec` returns `path()` if an error occurs.

Throws: As specified in Error reporting. [*Note:* It is an error if `p` does not resolve to a symbolic link. -- *end note*]

15.28 Remove [fs.op.remove]

```
bool remove(const path& p);
bool remove(const path& p, error_code& ec) noexcept;
```

Effects: If `exists(symlink_status(p,ec))`, it is removed as if by POSIX `remove()`.

[*Note:* A symbolic link is itself removed, rather than the file it resolves to being removed. -- *end note*]

Postcondition: `!exists(symlink_status(p))`.

Returns: `false` if p did not exist in the first place, otherwise `true`. The signature with argument `ec` returns `false` if an error occurs.

Throws: As specified in Error reporting.

15.29 Remove all [fs.op.remove_all]

```
uintmax_t remove_all(const path& p);
uintmax_t remove_all(const path& p, error_code& ec) noexcept;
```

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 being removed. -- *end note*]

Postcondition: `!exists(p)`

Returns: The number of files removed. The signature with argument `ec` returns `static_cast<uintmax_t>(-1)` if an error occurs.

Throws: As specified in Error reporting.

15.30 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;
```

Effects: Renames `old_p` to `new_p`, as if by POSIX `rename()`.

[*Note:* If `old_p` and `new_p` resolve to the same existing file, no action is taken. Otherwise, if `new_p` resolves to an existing non-directory file, it is removed, while if `new_p` resolves to an existing directory, it is removed if empty on POSIX compliant operating systems but is an error on some other operating systems. A symbolic link is itself renamed, rather than the file it resolves to being renamed. -- *end note*]

Throws: As specified in Error reporting.

15.31 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;
```

Postcondition: `file_size() == new_size`.

Throws: As specified in Error reporting.

Remarks: Achieves its postconditions as if by POSIX `truncate()`.

15.32 Space [fs.op.space]

```
space_info space(const path& p);
space_info space(const path& p, error_code& ec) noexcept;
```

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.

Throws: As specified in Error reporting.

15.33 Status [fs.op.status]

```
file_status status(const path& p);
```

Effects: As if:

```
error_code ec;
file_status result = status(p, ec);
if (result == file_type::none)
  throw filesystem_error(implementation-supplied-message, p,
ec);
return result;
```

Returns: See above.

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;
```

Effects:

If possible, determines the attributes of the file `p` resolves to, as if by POSIX `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*]

Returns:

If `ec != error_code()`:

- If the specific error indicates that `p` cannot be resolved because some element of the path does not exist, return `file_status(file_type::not_found)`.
- Otherwise, if the specific error indicates that `p` can be resolved but the attributes cannot be determined, return `file_status(file_type::unknown)`.
- Otherwise, return `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*]

Otherwise,

- If the attributes indicate a regular file, as if by POSIX S_ISREG(), return `file_status(file_type::regular)`. [*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*]
- Otherwise, if the attributes indicate a directory, as if by POSIX S_ISDIR(), return `file_status(file_type::directory)`. [*Note:* `file_type::directory` implies `directory_iterator(p)` would succeed. -- *end note*]
- Otherwise, if the attributes indicate a block special file, as if by POSIX S_ISBLK(), return `file_status(file_type::block)`.
- Otherwise, if the attributes indicate a character special file, as if by POSIX S_ISCHR(), return `file_status(file_type::character)`.

- Otherwise, if the attributes indicate a fifo or pipe file, as if by POSIX S_ISFIFO(), return `file_status(file_type::fifo)`.

- Otherwise, if the attributes indicate a socket, as if by POSIX S_ISSOCK(), return `file_status(file_type::socket)`.

- Otherwise, return `file_status(file_type::unknown)`.

Remarks: If a symbolic link is encountered during pathname resolution, pathname resolution continues using the contents of the symbolic link.

15.34 Status known [fs.op.status_known]

```
bool status_known(file_status s) noexcept;
```

Returns: `s.type() != file_type::none`

15.35 Symlink status [fs.op.symlink_status]

```
file_status symlink_status(const path& p);
file_status symlink_status(const path& p, error_code& ec) noexcept;
```

Effects: Same as status(), above, except that the attributes of `p` are determined as if by POSIX `lstat()`.

Returns: Same as status(), above, except that if the attributes indicate a symbolic link, as if by POSIX S_ISLNK(), return `file_status(file_type::symlink)`. The signature with argument `ec` returns `file_status(file_type::none)` if an error occurs.

Remarks: Pathname resolution terminates if `p` names a symbolic link.

Throws: As specified in Error reporting.

15.36 System complete [fs.op.system_complete]

```
path system_complete(const path& p);
path system_complete(const path& p, error_code& ec);
```

Effects: Composes an absolute path from `p`, using the same rules used by the operating system to resolve a path passed as the filename argument to standard library open functions.

Returns: The composed path. The signature with argument `ec` returns `path()` if an error occurs.

Postcondition: For the returned path, `rp, rp.is_absolute()` is true.

Throws: As specified in Error reporting.

[*Example:* For POSIX based operating systems, `system_complete(p)` has the same semantics as `complete(p, current_path())`.

For Windows based operating systems, `system_complete(p)` has the same semantics as `absolute(ph, current_path())` if `p.is_absolute() || !p.has_root_name()` or `p` and `base` have the same `root_name()`. Otherwise it acts like `absolute(p, kinky)`, where `kinky` is the current directory for the `p.root_name()` drive. This will be the current directory for that drive the last time it was set, and thus may be **residue left over from a prior program** run by the command processor! Although these semantics are useful, they are very error-prone. -- *end example*]

15.37 Temporary directory path [fs.op.temp_dir_path]

```
path temp_directory_path();
path temp_directory_path(error_code& ec);
```

Returns: An unspecifed directory path suitable for temporary files. An error shall be reported if `!exists(p) || !is_directory(p)`, where `p` is the path to be returned. The signature with argument `ec` returns `path()` if an error occurs.

Throws: As specified in Error reporting.

[*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*]

15.38 Unique path [fs.op.unique_path]

```
path unique_path(const path& model="%%%%-%%%%-%%%%-%%%%");
path unique_path(const path& model, error_code& ec);
```

The `unique_path` function generates a name suitable for temporary files, including directories. The name is based on a model that uses the percent sign character to specify replacement by a random hexadecimal digit.

[*Note:* The more bits of randomness in the generated name, the less likelihood of prior existence or being guessed. Each replacement hexadecimal digit in the model adds four bits of randomness. The default model thus provides 64 bits of randomness. --*end note*]

Returns: A path identical to `model`, except that each occurrence of the percent sign

character is replaced by a random hexadecimal digit character in the range 0-9, a-f. The signature with argument `ec` returns `path()` if an error occurs.

Throws: As specified in Error reporting.

Remarks: Implementations are encouraged to obtain the required randomness via a cryptographically secure pseudo-random number generator, such as one provided by the operating system. [*Note*: Such generators may block until sufficient entropy develops. --*end note*]

[*Example*:

```
cout << unique_path("test-%%%%%%%%%%.txt") << endl;
```

Typical output would be `"test-0db7f2bf57a.txt"`. Because 11 hexadecimal output characters are specified, 44 bits of randomness are supplied. -- *end example*]
