
**Programming languages — C++ — File
System Technical Specification**

*Languages de programmation — C++ — Spécification technique de
système de fichiers*

STANDARDSISO.COM: Click to view the full PDF of ISO/IEC TS 18822:2015



COPYRIGHT PROTECTED DOCUMENT

© ISO/IEC 2015

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized otherwise in any form or by any means, electronic or mechanical, including photocopying, or posting on the internet or an intranet, without prior written permission. Permission can be requested from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

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

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

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

- an ISO/IEC Publicly Available Specification (ISO/IEC PAS) represents an agreement between technical experts in an ISO/IEC 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 the joint 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/IEC PAS or ISO/IEC 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/IEC PAS or ISO/IEC 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 and IEC shall not be held responsible for identifying any or all such patent rights.

ISO/IEC TS 18822 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

Contents

Contents

1 Scope

2 Conformance

2.1 POSIX conformance

2.2 Operating system dependent behavior conformance

2.3 File system race behavior

3 Normative references

4 Terms and definitions

4.1 absolute path

4.2 canonical path

4.3 directory

4.4 file

4.5 file system

4.6 file system race

4.7 filename

4.8 hard link

4.9 link

4.10 native encoding

4.11 native pathname format

4.12 NTCTS

4.13 operating system dependent behavior

4.14 parent directory

4.15 path

4.16 pathname

4.17 pathname resolution

4.18 relative path

4.19 symbolic link

5 Requirements

5.1 Namespaces and headers

5.2 Feature test macros

6 Header `<experimental/filesystem>` synopsis

7 Error reporting

8 Class `path`

8.1 `path` generic pathname format grammar

8.2 `path` conversions

8.2.1 `path` argument format conversions

8.2.2 `path` type and encoding conversions

8.3 `path` requirements

8.4 `path` members

8.4.1 `path` constructors

8.4.2 `path` assignments

- 8.4.3 path appends
- 8.4.4 path concatenation
- 8.4.5 path modifiers
- 8.4.6 path native format observers
- 8.4.7 path generic format observers
- 8.4.8 path compare
- 8.4.9 path decomposition
- 8.4.10 path query
- 8.5 path iterators
- 8.6 path non-member functions
 - 8.6.1 path inserter and extractor
 - 8.6.2 path factory functions
- 9 Class filesystem_error
 - 9.1 filesystem_error members
- 10 Enumerations
 - 10.1 Enum class file_type
 - 10.2 Enum class copy_options
 - 10.3 Enum class perms
 - 10.4 Enum class directory_options
- 11 Class file_status
 - 11.1 file_status constructors
 - 11.2 file_status observers
 - 11.3 file_status modifiers
- 12 Class directory_entry
 - 12.1 directory_entry constructors
 - 12.2 directory_entry modifiers
 - 12.3 directory_entry observers
- 13 Class directory_iterator
 - 13.1 directory_iterator members
 - 13.2 directory_iterator non-member functions
- 14 Class recursive_directory_iterator
 - 14.1 recursive_directory_iterator members
 - 14.2 recursive_directory_iterator non-member functions
- 15 Operational functions
 - 15.1 Absolute
 - 15.2 Canonical
 - 15.3 Copy
 - 15.4 Copy file
 - 15.5 Copy symlink
 - 15.6 Create directories
 - 15.7 Create directory
 - 15.8 Create directory symlink
 - 15.9 Create hard link
 - 15.10 Create symlink

15.11 Current path
15.12 Exists
15.13 Equivalent
15.14 File size
15.15 Hard link count
15.16 Is block file
15.17 Is character file
15.18 Is directory
15.19 Is empty
15.20 Is fifo
15.21 Is other
15.22 Is regular file
15.23 Is socket
15.24 Is symlink
15.25 Last write time
15.26 Permissions
15.27 Read symlink
15.28 Remove
15.29 Remove all
15.30 Rename
15.31 Resize file
15.32 Space
15.33 Status
15.34 Status known
15.35 Symlink status
15.36 System complete
15.37 Temporary directory path

1 Scope [fs.scope]

- 1 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 (3) interface. This Technical Specification is applicable only to vendors who wish to provide the interface it describes.

2 Conformance [fs.conformance]

- 1 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]

- 1 Some behavior is specified by reference to POSIX (3). How such behavior is actually implemented is unspecified.
- 2 [Note: This constitutes an "as if" rule allowing implementations to call native operating system or other API's. —end note]
- 3 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 as specified in § 7.
- 4 [Note: This allows users to rely on an exception being thrown or an error code being set when an implementation cannot provide any reasonable behavior. — end note]
- 5 Implementations are not required to provide behavior that is not supported by a particular file system.
- 6 [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]

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

- 1 Some behavior is specified as being operating system dependent (4.13). The operating system an implementation is dependent upon is implementation defined.

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

2.3 File system race behavior [fs.race.behavior]

- 1 Behavior is undefined if calls to functions provided by this Technical Specification introduce a file system race (4.6).
- 2 If the possibility of a file system race would make it unreliable for a program to test for a precondition before calling a function described herein, *Requires* is not specified for the function.
- 3 [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]

3 Normative references [fs.norm.ref]

- 1 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.
- 2 • ISO/IEC 14882, *Programming Language C++*
 - 3 • ISO/IEC 9945, *Information Technology — Portable Operating System Interface (POSIX)*
- 4 [Note: The programming language and library described in ISO/IEC 14882 is herein called *the C++ Standard*. References to clauses within the C++ Standard are written as "C++14 §3.2". Section references are relative to N3936.
- 5 The operating system interface described in ISO/IEC 9945 is herein called *POSIX*. —end note]
- 6 This Technical Specification mentions commercially available operating systems for purposes of exposition. [footnote]
- 7 Unless otherwise specified, the whole of the C++ Standard's Library introduction (C++14 §17) is included into this Technical Specification by reference.
- 8 [footnote] 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 Terms and definitions [fs.definitions]

- ¹ 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]

- ¹ 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.

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

- ¹ An absolute path that has no elements that are symbolic links, and no dot or dot-dot elements (8.1).

4.3 directory [fs.def.directory]

- ¹ 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]

- ¹ 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]

- ¹ A collection of files and certain of their attributes.

4.6 file system race [fs.def.race]

- ¹ 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]

- ¹ The name of a file. Filenames dot and dot-dot have special meaning. The following characteristics of filenames are operating system dependent:
- ² • The permitted characters. [*Example: Some operating systems prohibit the ASCII control characters (0x00-0x1F) in filenames. —end example*].
 - ³ • The maximum permitted length.
 - ⁴ • Filenames that are not permitted.

- ⁵ • Filenames that have special meaning.
- ⁶ • Case awareness and sensitivity during path resolution.
- ⁷ • Special rules that may apply to file types other than regular files, such as directories.

4.8 hard link [fs.def.hardlink]

- ¹ A link (4.9) 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.
- ² [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]

- ¹ A directory entry that associates a filename with a file. A link is either a hard link (4.8) or a symbolic link (4.19).

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

- ¹ 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++14 §2.3).

4.11 native pathname format [fs.def.native]

- ¹ The operating system dependent pathname format accepted by the host operating system.

4.12 NTCTS [fs.def.ntcts]

- ¹ 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 `EcharT`, the null character can be constructed by `EcharT()`.

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

- ¹ Behavior that is dependent upon the behavior and characteristics of an operating system. See [fs.conform.os].

4.14 parent directory [fs.def.parent]

- ¹ When discussing a given directory, the directory that both contains a directory entry for the given directory and is represented by the filename dot-dot in the given directory.
- ² When discussing other types of files, a directory containing a directory entry for the file under discussion.

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

4.15 path [fs.def.path]

- 1 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. The maximum number of elements in the sequence is operating system dependent.

4.16 pathname [fs.def.pathname]

- 1 A character string that represents the name of a path. Pathnames are formatted according to the generic pathname format grammar (8.1) or an operating system dependent native pathname format.

4.17 pathname resolution [fs.def.pathres]

- 1 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]

- 1 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: Pathnames "." and ".." are relative paths. —end note]

4.19 symbolic link [fs.def.symlink]

- 1 A type of file 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.
- 2 [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]

- 1 Throughout this Technical Specification, `char`, `wchar_t`, `char16_t`, and `char32_t` are collectively called *encoded character types*.
- 2 Template parameters named `EcharT` shall be one of the encoded character types.
- 3 Template parameters named `InputIterator` shall meet the C++ Standard's library input iterator requirements (C++14 §24.2.3) and shall have a value type that is one of the encoded character types.

- 4 [Note: Use of an encoded character type implies an associated encoding. Since `signed char` and `unsigned char` have no implied encoding, they are not included as permitted types. —end note]
- 5 Template parameters named `Allocator` shall meet the C++ Standard's library Allocator requirements (C++14 §17.6.3.5).

5.1 Namespaces and headers [fs.req.namespaces]

- 1 The components described in this technical specification are experimental and not part of the C++ standard library. All components described in this technical specification are declared in namespace `std::experimental::filesystem::v1` or a sub-namespace thereof unless otherwise specified. The header described in this technical specification shall import the contents of `std::experimental::filesystem::v1` into `std::experimental::filesystem` as if by

```
2 namespace std {
    namespace experimental {
        namespace filesystem {
            inline namespace v1 {}
        }
    }
}
```

- 3 Unless otherwise specified, references to other entities described in this technical specification are assumed to be qualified with `std::experimental::filesystem::v1::`, and references to entities described in the C++ standard are assumed to be qualified with `std::`.

5.2 Feature test macros [fs.req.macros]

- 1 This macro allows users to determine which version of this Technical Specification is supported by header `<experimental/filesystem>`.
- 2 Header `<experimental/filesystem>` shall supply the following macro definition:

```
3 #define __cpp_lib_experimental_filesystem 201406
```

- 4 [Note: The value of macro `__cpp_lib_experimental_filesystem` is `yyyymm` where `yyyy` is the year and `mm` the month when the version of the Technical Specification was completed. — end note]

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

```
1 namespace std { namespace experimental { namespace filesystem { inline namespace v1 {
    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(const Source& source);
template <class InputIterator>
    path u8path(InputIterator first, InputIterator last);

class filesystem_error;
class directory_entry;

class directory_iterator;

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

class recursive_directory_iterator;

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

class file_status;

struct space_info
{
    uintmax_t capacity;
    uintmax_t free;
    uintmax_t available;
};

enum class file_type;
enum class perms;
enum class copy_options;
enum class directory_options;

typedef chrono::time_point<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;

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

void      create_directory_symlink(const path& to, const path& new_symlink);
void      create_directory_symlink(const path& to, const path& new_symlink,
                                   error_code& ec) noexcept;

void      create_hard_link(const path& to, const path& new_hard_link);
void      create_hard_link(const path& to, const path& new_hard_link,
                          error_code& ec) noexcept;

void      create_symlink(const path& to, const path& new_symlink);
void      create_symlink(const path& to, const path& new_symlink,
                        error_code& ec) noexcept;

path      current_path();
path      current_path(error_code& ec);
void      current_path(const path& p);
void      current_path(const path& p, error_code& ec) noexcept;

bool      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;

void           permissions(const path& p, perms prms);
void           permissions(const path& p, perms prms, 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);

} } } } // namespaces std::experimental::filesystem::v1

```

- ² **trivial_clock** is an implementation-defined type that satisfies the `TrivialClock` requirements (C++14 §20.12.3) and that is capable of representing and measuring file time values. Implementations should ensure that the resolution and range of `file_time_type` reflect the operating system dependent resolution and range of file time values.

7 Error reporting [fs.err.report]

- ¹ Filesystem library functions often provide two overloads, one that throws an exception to report file system errors, and another that sets an `error_code`.
- ² [Note: This supports two common use cases:
- ³ • 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.
 - ⁴ • Uses where file system errors are routine and do not necessarily represent failure. Returning an error code is the most appropriate response. This allows application specific error handling, including simply ignoring the error.

⁵ —end note]

- ⁶ Functions **not** having an argument of type `error_code&` report errors as follows, unless otherwise specified:
- ⁷ • When a call by the implementation to an operating system or other underlying API results in an error that prevents the function from meeting its specifications, an exception of type `filesystem_error` shall be thrown. For functions with a single path argument, that argument shall be passed to the `filesystem_error` constructor with a single path argument. For functions with two path arguments, the first of these arguments shall be passed to the `filesystem_error` constructor as the `path1` argument, and the second shall be passed as the `path2` argument. The `filesystem_error` constructor's `error_code` argument is set as appropriate for the specific operating system dependent error.
 - ⁸ • Failure to allocate storage is reported by throwing an exception as described in C++14 §17.6.5.12.
 - ⁹ • Destructors throw nothing.
- ¹⁰ Functions having an argument of type `error_code&` report errors as follows, unless otherwise specified:
- ¹¹ • If a call by the implementation to an operating system or other underlying API results in an error that prevents the function from meeting its specifications, the `error_code&` argument is set as appropriate for the specific operating system dependent error. Otherwise, `clear()` is called on the `error_code&` argument.

8 Class `path` [class.path]

- ¹ An object of class `path` represents a path (4.15) and contains a pathname (4.16). Such an object is concerned only with the lexical and syntactic aspects of a path. The path does not necessarily exist in external storage, and the pathname is not necessarily valid for the current operating system or for a particular file system.

²

```
namespace std { namespace experimental { namespace filesystem { inline namespace v1 {

    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() noexcept;
        path(const path& p);
        path(path&& p) noexcept;
        template <class Source>
```

```

    path(const Source& source);
template <class InputIterator>
    path(InputIterator first, InputIterator last);
template <class Source>
    path(const Source& source, const locale& loc);
template <class InputIterator>
    path(InputIterator first, InputIterator last, const locale& loc);
~path();

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

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

// 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+=(const Source& x);
template <class EcharT>
    path& operator+=(EcharT x);
template <class Source>
    path& concat(const Source& x);
template <class InputIterator>
    path& concat(InputIterator first, InputIterator last);

// 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 EcharT, class traits = char_traits<EcharT>,

```

```

        class Allocator = allocator<EcharT> >
basic_string<EcharT, traits, Allocator>
    string(const Allocator& a = Allocator()) const;
std::string    string() const;
std::wstring    wstring() const;
std::string    u8string() const;
std::u16string    u16string() const;
std::u32string    u32string() const;

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

// compare
int    compare(const path& p) const noexcept;
int    compare(const string_type& 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::experimental::filesystem::v1

```

- 3 value_type is a typedef for the operating system dependent encoded character type used to represent pathnames.
- 4 The value of preferred_separator is the operating system dependent *preferred-separator* character (8.1).
- 5 [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]

- 1 *pathname:*
 root-name root-directory_{opt} relative-path_{opt}
 root-directory relative-path_{opt}
 relative-path
- 2 *root-name:*
 An operating system dependent name that identifies the starting location for absolute paths.
- 3 [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]
- 4 *root-directory:*
 directory-separator
- 5 *relative-path:*
 filename
 relative-path directory-separator
 relative-path directory-separator filename
- 6 *filename:*
 name
 dot
 dot-dot

7 *name:*

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

- 8 [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:

9 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]

10 *dot:*

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

11 *dot-dot:*

The filename consisting solely of two period characters (..).

12 *directory-separator:*

slash

slash directory-separator

preferred-separator

preferred-separator directory-separator

13 *preferred-separator:*

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

14 *slash:*

The slash character (/).

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

16 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. What the filename *dot-dot* refers to relative to *root-directory* is implementation-defined. 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]

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

- ² • The generic format is acceptable as a native path.
- ³ • There is no need to distinguish between native format and generic format arguments.
- ⁴ • Paths for regular files and paths for directories share the same syntax.

⁵ —end note]

- ⁶ Functions arguments that take character sequences representing paths may use the generic pathname format grammar (8.1) or the native pathname format (4.11). 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.
- ⁷ [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]
- ⁸ 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]

- ¹ 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:
- ² • `char`: Encoding is the native narrow encoding (4.10). Conversion, if any, is operating system dependent.
 - ³ [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.
 - ⁴ For Windows based operating systems, the native narrow encoding is determined by calling a Windows API function. —end note]
 - ⁵ [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]
 - ⁶ • `wchar_t`: Encoding is the native wide encoding (4.10). Conversion method is unspecified.

- ⁷ [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]
- ⁸ • `char16_t`: Encoding is UTF-16. Conversion method is unspecified.
 - ⁹ • `char32_t`: Encoding is UTF-32. Conversion method is unspecified.

¹⁰ If the encoding being converted to has no representation for source characters, the resulting converted characters, if any, are unspecified.

8.3 path requirements [path.req]

- ¹ In addition to the requirements (5), function template parameters named `Source` shall be one of:
- ² • `basic_string<EcharT, traits, Allocator>`. A function argument `const Source& source` shall have an effective range `[source.begin(), source.end())`.
 - ³ • A type meeting the input iterator requirements that iterates over a NTCTS. The value type shall be an encoded character type. A function argument `const Source& source` shall have an effective range `[source, end)` where `end` is the first iterator value with an element value equal to `iterator_traits<Source>::value_type()`.
 - ⁴ • A character array that after array-to-pointer decay results in a pointer to the start of a NTCTS. The value type shall be an encoded character type. A function argument `const Source& source` shall have an effective range `[source, end)` where `end` is the first iterator value with an element value equal to `iterator_traits<decay<Source>::type>::value_type()`.
- ⁵ [Note: See path conversions (8.2) for how these value types and their encodings convert to `path::value_type` and its encoding. —end note]
- ⁶ Arguments of type `Source` shall not be null pointers.

8.4 path members [path.member]

8.4.1 path constructors [path.construct]

- ¹ `path() noexcept;`
- ² *Effects:* Constructs an object of class `path`.
- ³ *Postconditions:* `empty()`.
- ⁴ `path(const path& p);`
`path(path&& p) noexcept;`

- 5 *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.

```
6 template <class Source>
  path(const Source& source);
template <class InputIterator>
  path(InputIterator first, InputIterator last);
```

- 7 *Effects:* Constructs an object of class `path`, storing the effective range of `source` (8.3) or the range `[first,last)` in `pathname`, converting format and encoding if required (8.2.1).

```
8 template <class Source>
  path(const Source& source, const locale& loc);
template <class InputIterator>
  path(InputIterator first, InputIterator last, const locale& loc);
```

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

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

- 11 If `value_type` is `wchar_t`, converts to the native wide encoding (4.10) using the `codecvt<wchar_t, char, mbstate_t>` facet of `loc`. Otherwise a conversion is performed using the `codecvt<wchar_t, char, mbstate_t>` facet of `loc`, and then a second conversion to the current narrow encoding.

- 12 *[Example:*

- 13 A string is to be read from a database that is encoded in ISO/IEC 8859-1, and used to create a directory:

```
14 namespace fs = std::experimental::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));
```

- 15 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 (4.10). 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.

- 16 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.

17 —end example]

8.4.2 path assignments [path.assign]

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

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

3 *Returns:* `*this`

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

- 5 *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]

6 *Returns:* `*this`

7 `template <class Source>`
`path& operator=(const Source& source);`
`template <class Source>`
`path& assign(const Source& source);`
`template <class InputIterator>`
`path& assign(InputIterator first, InputIterator last);`

- 8 *Effects:* Stores the effective range of `source` (8.3) or the range `[first,last)` in `pathname`, converting format and encoding if required (8.2.1).

9 *Returns:* `*this`

8.4.3 path appends [path.append]

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

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

3 *Effects:*

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

- 4 • an added separator would be redundant, or
- 5 • would change a relative path to an absolute path [*Note: An empty path is relative. — end note*], or
- 6 • `p.empty()`, or
- 7 • `*p.native().cbegin()` is a directory separator.

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

9 *Returns:* `*this`

```
10 template <class Source>
    path& operator/=(const Source& source);
template <class Source>
    path& append(const Source& source);
template <class InputIterator>
    path& append(InputIterator first, InputIterator last);
```

11 *Effects:*

12 Appends `path::preferred_separator` to `pathname`, converting format and encoding if required (8.2.1), unless:

- 13 • an added separator would be redundant, or
- 14 • would change an relative path to an absolute path, or
- 15 • `source.empty()`, or
- 16 • `*source.native().cbegin()` is a separator.

17 Appends the effective range of `source` (8.3) or the range `[first,last)` to `pathname`, converting format and encoding if required (8.2.1).

18 *Returns:* `*this`

8.4.4 path concatenation [`path.concat`]

```
1 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+=(const Source& x);
  template <class EcharT>
      path& operator+=(EcharT x);
  template <class Source>
      path& concat(const Source& x);
  template <class InputIterator>
      path& concat(InputIterator first, InputIterator last);
```

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

- 3 • `x.native()` if `x` is present and is `const path&`, otherwise
- 4 • the effective range `source` (8.3), if `source` is present, otherwise,
- 5 • the range `[first,last)`, if `first` and `last` are present, otherwise,
- 6 • `x`.

7 If the value type of *effective-argument* would not be `path::value_type`, the actual argument or argument range is first converted (8.2.1) so that *effective-argument* has value type `path::value_type`.

8 *Returns:* `*this`

8.4.5 path modifiers [path.modifiers]

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

1 *Postcondition:* `empty()`

2 `path& make_preferred();`

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

4 *Returns:* `*this`

5 [Example:

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

7 On an operating system where *preferred-separator* is the same as *directory-separator*, the output is:

```
8 "foo/bar"
  "foo/bar"
```

9 On an operating system where *preferred-separator* is a backslash, the output is:

```
10 "foo/bar"
   "foo\bar"
```

11 —end example]

12 `path& remove_filename();`

13 *Postcondition:* !has_filename().

14 *Returns:* *this.

15 *[Example:*

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

17 *—end example]*

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

19 *Effects:*

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

21 *Returns:* *this.

22 *[Example:*

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

24 *—end example]*

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

26 *Effects:*

- 27 • Any existing `extension()` (8.4.9) is removed from the stored path, then
- 28 • If `replacement` is not empty and does not begin with a dot character, a dot character is appended to the stored path, then
- 29 • `replacement` is concatenated to the stored path.

30 *Returns:* *this

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

32 *Effects:* Swaps the contents of the two paths.

33 *Complexity:* constant time.

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

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

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

3 *Returns:* pathname.

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

5 *Returns:* pathname.c_str().

6 `operator string_type() const;`

7 *Returns:* pathname.

8 [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]

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

10 *Returns:* pathname.

11 *Remarks:* All memory allocation, including for the return value, shall be performed by `a`. Conversion, if any, is specified by 8.2.

12 `std::string string() const;
std::wstring wstring() const;
std::string u8string() const;
std::u16string u16string() const;
std::u32string u32string() const;`

13 *Returns:* pathname.

14 *Remarks:* Conversion, if any, is performed as specified by 8.2. The encoding of the string returned by `u8string()` is always UTF-8.

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

1 Generic format observer functions return strings formatted according to the generic pathname format (8.1). The forward slash ('/') character is used as the *directory-separator* character.

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

3 `template <class EcharT, class traits = char_traits<EcharT>,
class Allocator = allocator<EcharT> >`

```
basic_string<EcharT, traits, Allocator>
generic_string(const Allocator& a = Allocator()) const;
```

4 *Returns:* pathname, reformatted according to the generic pathname format (8.1).

5 *Remarks:* All memory allocation, including for the return value, shall be performed by a. Conversion, if any, is specified by 8.2.

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

7 *Returns:* pathname, reformatted according to the generic pathname format (8.1).

8 *Remarks:* Conversion, if any, is specified by 8.2. The encoding of the string returned by generic_u8string() is always UTF-8.

8.4.8 path compare [path.compare]

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

2 *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.

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

```
4 int compare(const string_type& s) const
```

5 *Returns:* compare(path(s)).

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

7 *Returns:* compare(path(s)).

8.4.9 path decomposition [path.decompose]

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

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

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

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

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

6 `path root_path() const;`

7 *Returns:* *root_name()* / *root_directory()*

8 `path relative_path() const;`

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

10 `path parent_path() const;`

11 *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()*).

12 `path filename() const;`

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

14 [Example:

```
15  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 ".."
```

16 —end example]

17 `path stem() const;`

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

19 [Example:

```
20  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
```

21 —end example]

22 `path extension() const;`

23 *Returns:* if `filename()` contains a period but does not consist solely of one or 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.

24 *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.

25 *[Example:*

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

27 *—end example]*

28 *[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.filename()`. *—end note]*

8.4.10 `path` query [`path.query`]

1 `bool empty() const noexcept;`

2 *Returns:* `pathname.empty()`.

3 `bool has_root_path() const;`

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

5 `bool has_root_name() const;`

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

7 `bool has_root_directory() const;`

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

9 `bool has_relative_path() const;`

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

11 `bool has_parent_path() const;`

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

13 `bool has_filename() const;`

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

15 `bool has_stem() const;`

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

17 `bool has_extension() const;`

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

19 `bool is_absolute() const;`

20 *Returns:* true if `pathname` contains an absolute path (4.1), else false.

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

22 `bool is_relative() const;`

23 *Returns:* `!is_absolute()`.

8.5 path iterators [path.itr]

1 Path iterators iterate over the elements of the stored `pathname`.

2 A `path::iterator` is a constant iterator satisfying all the requirements of a bidirectional iterator (C++14 §24.1.4 Bidirectional iterators). Its `value_type` is `path`.

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

4 The forward traversal order is as follows:

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

9 The backward traversal order is the reverse of forward traversal.

10 `iterator begin() const;`

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

12 `iterator end() const;`

13 *Returns:* The end iterator.

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

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

2 *Effects:* `lhs.swap(rhs)`.

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

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

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

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

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

8 *Returns:* `!(rhs < lhs)`.

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

10 *Returns:* `rhs < lhs`.

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

12 *Returns:* `!(lhs < rhs)`.

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

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

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

16 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.

17 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.

18 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]

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

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

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

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

8.6.1 path inserter and extractor [path.io]

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

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

3 [Note: The `quoted` function is described in C++14 §27.7.6. — end note]

4 *Returns:* `os`

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

6 *Effects:*

7 `basic_string<charT, traits> tmp;`
`is >> quoted(tmp);`
`p = tmp;`

8 *Returns:* `is`

8.6.2 path factory functions [path.factory]

1 `template <class Source>`
`path u8path(const Source& source);`
`template <class InputIterator>`
`path u8path(InputIterator first, InputIterator last);`

2 *Requires:* The `source` and `[first,last)` sequences are UTF-8 encoded. The value type of `Source` and `InputIterator` is `char`.

3 *Returns:*

- 4 • If `value_type` is `char` and the current native narrow encoding (4.11) is UTF-8, `path(source)` or `path(first, last)`, else
- 5 • 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 `[first,last)` to a temporary, `tmp`, of type `string_type` and return `path(tmp)`, else

- 6 • convert source or [first,last) to a temporary, tmp, of type u32string and return path(tmp).

7 *Remarks:* Argument format conversion (8.2.1) applies to the arguments for these functions. How Unicode encoding conversions are performed is unspecified.

8 *[Example:*

- 9 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:

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

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

- 12 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.

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

14 *—end example]*

9 Class `filesystem_error` [class.filesystem_error]

```
1 namespace std { namespace experimental { namespace filesystem { inline namespace v1 {
    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::experimental::filesystem::v1
```

- 2 The class `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]

- ¹ Constructors are provided that store zero, one, or two paths associated with an error.

² `filesystem_error(const string& what_arg, error_code ec);`

³ *Postcondition:*

Expression	Value
<code>runtime_error::what()</code>	<code>what_arg.c_str()</code>
<code>code()</code>	<code>ec</code>
<code>path1().empty()</code>	<code>true</code>
<code>path2().empty()</code>	<code>true</code>

⁴ `filesystem_error(const string& what_arg, const path& p1, error_code ec);`

⁵ *Postcondition:*

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

⁶ `filesystem_error(const string& what_arg, const path& p1, const path& p2, error_code ec);`

⁷ *Postcondition:*

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

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

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

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

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

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

13 *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]

1 This enum class specifies constants used to identify file types.

Constant Name	Value	Meaning
<code>none</code>	0	The type of the file has not been determined or an error occurred while trying to determine the type.
<code>not_found</code>	-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]
<code>regular</code>	1	Regular file
<code>directory</code>	2	Directory file
<code>symlink</code>	3	Symbolic link file
<code>block</code>	4	Block special file
<code>character</code>	5	Character special file
<code>fifo</code>	6	FIFO or pipe file
<code>socket</code>	7	Socket file
<code>unknown</code>	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`]

- ¹ The enum class type `copy_options` is a bitmask type (C++14 §17.5.2.1.3) that 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 <code>copy_file</code> function effects for existing target files		
Constant	Value	Meaning
<code>none</code>	0	(Default) Error; file already exists.
<code>skip_existing</code>	1	Do not overwrite existing file, do not report an error.
<code>overwrite_existing</code>	2	Overwrite the existing file.
<code>update_existing</code>	4	Overwrite the existing file if it is older than the replacement file.
Option group controlling <code>copy</code> function effects for sub-directories		
Constant	Value	Meaning
<code>none</code>	0	(Default) Do not copy sub-directories.
<code>recursive</code>	8	Recursively copy sub-directories and their contents.
Option group controlling <code>copy</code> function effects for symbolic links		
Constant	Value	Meaning
<code>none</code>	0	(Default) Follow symbolic links.
<code>copy_symlinks</code>	16	Copy symbolic links as symbolic links rather than copying the files that they point to.
<code>skip_symlinks</code>	32	Ignore symbolic links.
Option group controlling <code>copy</code> function effects for choosing the form of copying		
Constant	Value	Meaning
<code>none</code>	0	(Default) Copy content.
<code>directories_only</code>	64	Copy directory structure only, do not copy non-directory files.
<code>create_symlinks</code>	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.
<code>create_hard_links</code>	256	Make hard links instead of copies of files.

10.3 Enum class perms [enum.perms]

- ¹ The enum class type perms is a bitmask type (C++14 §17.5.2.1.3) that 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_exec
group_read	040	S_IRGRP	Read permission, group
group_write	020	S_IWGRP	Write permission, group
group_exec	010	S_IXGRP	Execute/search permission, group
group_all	070	S_IRWXG	Read, write, execute/search by group; group_read group_write group_exec
others_read	04	S_IROTH	Read permission, others
others_write	02	S_IWOTH	Write permission, others
others_exec	01	S_IXOTH	Execute/search permission, others
others_all	07	S_IRWXO	Read, write, execute/search by others; others_read others_write others_exec
all	0777		owner_all group_all others_all
set_uid	04000	S_ISUID	Set-user-ID on execution
set_gid	02000	S_ISGID	Set-group-ID on execution
sticky_bit	01000	S_ISVTX	Operating system dependent.
mask	07777		all set_uid set_gid sticky_bit
unknown	0xFFFF		The permissions are not known, such as when a file_status object is created without specifying the permissions

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

10.4 Enum class `directory_options` [`enum.directory_options`]

- ¹ The enum class type `directory_options` is a bitmask type (C++14 §17.5.2.1.3) that specifies bitmask constants used to identify directory traversal options.

Name	Value	Meaning
<code>none</code>	0	(Default) Skip directory symlinks, permission denied is an error.
<code>follow_directory_symlink</code>	1	Follow rather than skip directory symlinks.
<code>skip_permission_denied</code>	2	Skip directories that would otherwise result in permission denied errors.

11 Class `file_status` [`class.file_status`]

¹

```

namespace std { namespace experimental { namespace filesystem { inline namespace v1 {

    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();

        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::experimental::filesystem::v1
```

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

11.1 `file_status` constructors [`file_status.cons`]

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

2 *Postconditions:* `type() == file_type::none`, `permissions() == perms::unknown`.

```
3 explicit file_status(file_type ft, perms prms = perms::unknown) noexcept;
```

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

11.2 `file_status` observers [`file_status.obs`]

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

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

```
3 perms permissions() const noexcept;
```

4 *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`]

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

2 *Postconditions:* `type() == ft`.

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

4 *Postconditions:* `permissions() == prms`.

12 Class `directory_entry` [`class.directory_entry`]

```
1 namespace std { namespace experimental { namespace filesystem { inline namespace v1 {
    class directory_entry
    {
    public:
        // constructors and destructor
        directory_entry() noexcept = default;
```

```

    directory_entry(const directory_entry&) = default;
    directory_entry(directory_entry&&) noexcept = default;
    explicit directory_entry(const path& p);
    ~directory_entry();

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

    // observers
    const path& path() const noexcept;
    operator const 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
};

} } } } // namespaces std::experimental::filesystem::v1

```

- 2 A `directory_entry` object stores a path object.

12.1 `directory_entry` constructors [`directory_entry.cons`]

1 `explicit directory_entry(const path& p);`

- 2 *Effects:* Constructs an object of type `directory_entry`.

- 3 *Postcondition:* `path() == p`.

12.2 `directory_entry` modifiers [`directory_entry.mods`]

1 `void assign(const path& p);`

- 2 *Postcondition:* `path() == p`.

3 `void replace_filename(const path& p);`

- 4 *Postcondition:* `path() == x.parent_path() / p` where `x` is the value of `path()` before the function is called.

12.3 directory_entry observers [directory_entry.obs]

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

2 *Returns:* `m_path`

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

4 *Returns:* `status(path() [, ec]).`

5 *Throws:* As specified in [Error reporting \(7\)](#).

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

7 *Returns:* `symlink_status(path() [, ec]).`

8 *Throws:* As specified in [Error reporting \(7\)](#).

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

10 *Returns:* `m_path == rhs.m_path.`

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

12 *Returns:* `m_path != rhs.m_path.`

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

14 *Returns:* `m_path < rhs.m_path.`

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

16 *Returns:* `m_path <= rhs.m_path.`

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

18 *Returns:* `m_path > rhs.m_path.`

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

20 *Returns:* `m_path >= rhs.m_path.`

13 Class `directory_iterator` [class.directory_iterator]

- ¹ 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` (14). —end note]

²

```
namespace std { namespace experimental { namespace filesystem { inline namespace v1 {

    class directory_iterator
    {
    public:
        typedef directory_entry      value_type;
        typedef ptrdiff_t            difference_type;
        typedef const directory_entry* pointer;
        typedef const directory_entry& reference;
        typedef input_iterator_tag    iterator_category;

        // member functions
        directory_iterator() noexcept;
        explicit directory_iterator(const path& p);
        directory_iterator(const path& p, directory_options options);
        directory_iterator(const path& p, error_code& ec) noexcept;
        directory_iterator(const path& p,
            directory_options options, error_code& ec) noexcept;
        directory_iterator(const directory_iterator& rhs);
        directory_iterator(directory_iterator& rhs) noexcept;
        ~directory_iterator();

        directory_iterator& operator=(const directory_iterator& rhs);
        directory_iterator& operator=(directory_iterator&& rhs) noexcept;

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

        // other members as required by C++14 §24.1.1 Input iterators
    };

} } } } // namespaces std::experimental::filesystem::v1
```

- ³ `directory_iterator` satisfies the requirements of an input iterator C++14 §24.2.3).
- ⁴ 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.
- ⁵ 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.

- 6 Two end iterators are always equal. An end iterator shall not be equal to a non-end iterator.
- 7 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 `/=`.
- 8 Directory iteration shall not yield directory entries for the current (*dot*) and parent (*dot-dot*) directories.
- 9 The order of directory entries obtained by dereferencing successive increments of a `directory_iterator` is unspecified.
- 10 [Note: Programs performing directory iteration may wish to test if the path obtained by dereferencing a directory iterator actually exists. It could be a symbolic link to a non-existent file. Programs recursively walking directory trees for purposes of removing and renaming entries may wish to avoid following symbolic links.
- 11 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`]

- 1 `directory_iterator()` `noexcept`;
- 2 *Effects*: Constructs the end iterator.
- 3

```
explicit directory_iterator(const path& p);
directory_iterator(const path& p, directory_options options);
directory_iterator(const path& p, error_code& ec) noexcept;
directory_iterator(const path& p,
    directory_options options, error_code& ec) noexcept;
```
- 4 *Effects*: For the directory that `p` resolves to, constructs an iterator for the first element in a sequence of `directory_entry` elements representing the files in the directory, if any; otherwise the end iterator. However, if
`(options & directory_options::skip_permissions_denied) != directory_options::none`
and construction encounters an error indicating that permission to access `p` is denied, constructs the end iterator and does not report an error.
- 5 *Throws*: As specified in [Error reporting \(7\)](#).
- 6 [Note: To iterate over the current directory, use `directory_iterator(".")` rather than `directory_iterator("")`. —end note]

```

7  directory_iterator(const directory_iterator& rhs);
   directory_iterator(directory_iterator&& rhs) noexcept;

```

8 *Effects:* Constructs an object of class `directory_iterator`.

9 *Postconditions:* `*this` has the original value of `rhs`.

```

10 directory_iterator& operator=(const directory_iterator& rhs);
   directory_iterator& operator=(directory_iterator&& rhs) noexcept;

```

11 *Effects:* If `*this` and `rhs` are the same object, the member has no effect.

12 *Postconditions:* `*this` has the original value of `rhs`.

13 *Returns:* `*this`.

```

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

```

15 *Effects:* As specified by C++14 §24.1.1 Input iterators.

16 *Returns:* `*this`.

17 *Throws:* As specified in [Error reporting \(7\)](#).

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

1 These functions enable use of `directory_iterator` with range-based for statements.

```

2  directory_iterator begin(directory_iterator iter) noexcept;

```

3 *Returns:* `iter`.

```

4  directory_iterator end(const directory_iterator&) noexcept;

```

5 *Returns:* `directory_iterator()`.

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

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

```

2  namespace std { namespace experimental { namespace filesystem { inline namespace v1 {
   class recursive_directory_iterator
   {
   public:

```

```

typedef directory_entry      value_type;
typedef ptrdiff_t           difference_type;
typedef const directory_entry* pointer;
typedef const directory_entry& reference;
typedef input_iterator_tag   iterator_category;

// constructors and destructor
recursive_directory_iterator() noexcept;
explicit recursive_directory_iterator(const path& p);
recursive_directory_iterator(const path& p, directory_options options);
recursive_directory_iterator(const path& p,
    directory_options options, error_code& ec) noexcept;
recursive_directory_iterator(const path& p, error_code& ec) noexcept;
recursive_directory_iterator(const recursive_directory_iterator& rhs);
recursive_directory_iterator(recursive_directory_iterator&& rhs) noexcept;
~recursive_directory_iterator();

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

const directory_entry& operator*() const;
const directory_entry* operator->() const;

// modifiers
recursive_directory_iterator&
    operator=(const recursive_directory_iterator& rhs);
recursive_directory_iterator&
    operator=(recursive_directory_iterator&& rhs) noexcept;

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

void pop();
void disable_recursion_pending();

// other members as required by C++14 §24.1.1 Input iterators
};

} } } } // namespaces std::experimental::filesystem::v1

```

- 3 The behavior of a `recursive_directory_iterator` is the same as a `directory_iterator` unless otherwise specified.
- 4 [Note: If the directory structure being iterated over contains cycles then the end iterator may be unreachable. —end note]

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

1 `recursive_directory_iterator()` noexcept;

2 *Effects:* Constructs the end iterator.


```

3 explicit recursive_directory_iterator(const path& p);
  recursive_directory_iterator(const path& p, directory_options options);
  recursive_directory_iterator(const path& p,
    directory_options options, error_code& ec) noexcept;
  recursive_directory_iterator(const path& p, error_code& ec) noexcept;

```

4 *Effects:* Constructs an iterator representing the first entry in the directory `p` resolves to, if any; otherwise, the end iterator. However, if
 (options & directory_options::skip_permissions_denied) != directory_options::none
 and construction encounters an error indicating that permission to access `p` is denied,
 constructs the end iterator and does not report an error.

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

6 *Throws:* As specified in [Error reporting \(7\)](#).

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

8 [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]

```

9 recursive_directory_iterator(const recursive_directory_iterator& rhs);

```

10 *Effects:* Constructs an object of class recursive_directory_iterator.

11 *Postconditions:*

```

this->options() == rhs.options() && this->depth() == rhs.depth()
&& this->recursion_pending() == rhs.recursion_pending().

```

```

12 recursive_directory_iterator(recursive_directory_iterator&& rhs) noexcept;

```

13 *Effects:* Constructs an object of class recursive_directory_iterator.

14 *Postconditions:* this->options(), this->depth(), and this->recursion_pending()
 return the values that rhs.options(), rhs.depth(), and rhs.recursion_pending(),
 respectively, had before the function call.

```

15 recursive_directory_iterator& operator=(const recursive_directory_iterator& rhs);

```

16 *Effects:* If *this and rhs are the same object, the member has no effect.

17 *Postconditions:*

```

this->options() == rhs.options() && this->depth() == rhs.depth()
&& this->recursion_pending() == rhs.recursion_pending().

```

18 *Returns:* *this.

19 `recursive_directory_iterator& operator=(recursive_directory_iterator&& rhs) noexcept;`

20 *Effects:* If *this and rhs are the same object, the member has no effect.

21 *Postconditions:* this->options(), this->depth(), and this->recursion_pending() return the values that rhs.options(), rhs.depth(), and rhs.recursion_pending(), respectively, had before the function call.

22 *Returns:* *this.

23 `directory_options options() const;`

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

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

26 *Throws:* Nothing.

27 `int depth() const;`

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

29 *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]

30 *Throws:* Nothing.

31 `bool recursion_pending() const;`

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

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

34 *Throws:* Nothing.

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

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

37 *Effects:* As specified by C++14 §24.2.3 Input iterators, except that:

- 38 • If there are no more entries at this depth, then if `depth() != 0` iteration over the parent directory resumes; otherwise `*this = recursive_directory_iterator()`.
- 39 • Otherwise if `recursion_pending() && is_directory(this->status())`
 && (!`is_symlink(this->symlink_status())`)
 || (`options() & directory_options::follow_directory_symlink`) !=
 `directory_options::none`) then either directory (`*this`)->`path()` is recursively
 iterated into or, if
 (`options() & directory_options::skip_permissions_denied`)
 != `directory_options::none` and an error occurs indicating that permission to
 access directory (`*this`)->`path()` is denied, then directory (`*this`)->`path()` is
 treated as an empty directory and no error is reported.

40 *Returns:* `*this`.

41 *Throws:* As specified in [Error reporting \(7\)](#).

42 `void pop();`

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

44 *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.

45 `void disable_recursion_pending();`

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

47 *Postcondition:* `recursion_pending() == false`.

48 [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`]

1 These functions enable use of `recursive_directory_iterator` with range-based for statements.

2 `recursive_directory_iterator begin(recursive_directory_iterator iter) noexcept;`

3 *Returns:* `iter`.

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

5 *Returns:* `recursive_directory_iterator()`.

15 Operational functions [fs.op.funcs]

- ¹ Operational functions query or modify files, including directories, in external storage.
- ² [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 \(7\)](#). —end note]

15.1 Absolute [fs.op.absolute]

¹ `path absolute(const path& p, const path& base=current_path());`

- ² *Returns:* An [absolute path](#) composed according to the following table

	<code>p.has_root_directory()</code>	<code>!p.has_root_directory()</code>
<code>p.has_root_name()</code>	return p	return p.root_name() / absolute(base).root_directory() / absolute(base).relative_path() / p.relative_path()
<code>!p.has_root_name()</code>	return absolute(base).root_name() / p	return absolute(base) / p

- ³ [Note: For the returned path, `rp`, `rp.is_absolute()` is true. —end note]

- ⁴ *Throws:* As specified in [Error reporting \(7\)](#).

15.2 Canonical [fs.op.canonical]

¹ `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);`

- ² *Overview:* Converts `p`, which must exist, to an absolute path that has no symbolic link, ".", or ".." elements.
- ³ *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.
- ⁴ *Throws:* As specified in [Error reporting \(7\)](#).

- 5 *Remarks:* `!exists(p)` is an error.
- 6 [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]

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

2 *Effects:* `copy(from, to, copy_options::none[, ec])`.

```
3 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;
```

4 *Precondition:* At most one constant from each option group (10.2) is present in `options`.

5 *Effects:*

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

7 • If

```
(options & copy_options::create_symlinks) != copy_options::none
|| (options & copy_options::skip_symlinks) != copy_options::none,
then auto f = symlink_status(from) and if needed
auto t = symlink_status(to).
```

8 • Otherwise, auto f = `status(from)` and if needed auto t = `status(to)`.

9 Report an error as specified in [Error reporting \(7\)](#) if:

- 10 • `!exists(f)`, or
- 11 • `equivalent(from, to)`, or
- 12 • `is_other(f) || is_other(t)`, or
- 13 • `is_directory(f) && is_regular_file(t)`.

14 If `is_symlink(f)`, then:

15 • If

```
(options & copy_options::skip_symlinks) != copy_options::none,
then return.
```

16 • Otherwise if `!exists(t)`

```
&& (options & copy_options::copy_symlinks) != copy_options::none,
then copy_symlink(from, to, options).
```

17 • Otherwise report an error as specified in [Error reporting \(7\)](#).

18 Otherwise if `is_regular_file(f)`, then:

- 19 • If `(options & copy_options::directories_only)`
`!= copy_options::none`, then return.
- 20 • Otherwise if `(options & copy_options::create_symlinks)`
`!= copy_options::none`, then create a symbolic link to the source file.
- 21 • Otherwise if `(options & copy_options::create_hard_links)`
`!= copy_options::none`, then create a hard link to the source file.
- 22 • Otherwise if `is_directory(t)`, then
`copy_file(from, to/from.filename(), options)`.
- 23 • Otherwise, `copy_file(from, to, options)`.

24 Otherwise if `is_directory(f) && ((options & copy_options::recursive)`
`!= copy_options::none || options == copy_options::none)` then:

- 25 • If `!exists(t)`, then `create_directory(to, from)`.
- 26 • 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)`.

27 Otherwise no effects.

28 *Throws:* As specified in [Error reporting \(7\)](#).

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

30 *[Example:* Given this directory structure:

```
31 /dir1
   file1
   file2
   dir2
     file3
```

32 Calling `copy("/dir1", "/dir3")` would result in:

```
33 /dir1
   file1
   file2
   dir2
     file3
/dir3
   file1
   file2
```

34 Alternatively, calling `copy("/dir1", "/dir3", copy_options::recursive)` would result in:

```
35 /dir1
    file1
    file2
    dir2
        file3
/dir3
    file1
    file2
    dir2
        file3
```

36 —end example]

15.4 Copy file [fs.op.copy_file]

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

2 *Returns:* `copy_file(from, to, copy_options::none, ec)`.

3 *Throws:* As specified in [Error reporting \(7\)](#).

```
4 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;
```

5 *Precondition:* At most one constant from each `copy_options` option group ([10.2](#)) is present in `options`.

6 *Effects:*

7 Report a file already exists error as specified in [Error reporting \(7\)](#) if:

- 8 • `exists(to)` and `equivalent(from, to)`, or
- 9 • `exists(to)` and `(options & (copy_options::skip_existing | copy_options::overwrite_existing | copy_options::update_existing))`
`== copy_options::none`.

10 Otherwise copy the contents and attributes of the file `from` resolves to to the file `to` resolves to if:

- 11 • `!exists(to)`, or
- 12 • `exists(to)` and
`(options & copy_options::overwrite_existing) != copy_options::none`,
or

- 13 • `exists(to)` and
`(options & copy_options::update_existing) != copy_options::none`
and `from` is more recent than `to`, determined as if by use of the
`last_write_time` function.

14 Otherwise no effects.

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

16 *Throws:* As specified in [Error reporting \(7\)](#).

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

15.5 Copy symlink [fs.op.copy_symlink]

```
1 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;
```

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

3 *Throws:* As specified in [Error reporting \(7\)](#).

15.6 Create directories [fs.op.create_directories]

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

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

3 *Postcondition:* `is_directory(p)`

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

5 *Throws:* As specified in [Error reporting \(7\)](#).

6 *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]

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