TECHNICAL SPECIFICATION

ISO/IEC TS 18822

First edition
2015-07-01

Programming languages — C++ — File System Technical Specification

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

Reference number
ISO/IEC TS 18822:2015(E)

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order to decide whether it will be confirmed for a further three years, revised to become an International Standard, or withdrawn.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again after a further three years, at which time it must either be transformed into an International Standard or be withdrawn.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

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.namespace]

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);
```