补充与总结

Supplementary and Summary

现代C++基础
Modern C++ Basics

Jiaming Liang, ~~undergraduate~~ from Peking University

Postgraduate from PKU since 2024.9 :-)

- **File system**
- **Time utilities (Chrono)**
- **Math utilities**
- **Summary and future prospect**

Supplementary and Summary

File system

Overview

- Files represent organized data in non-volatile storage to let programs share data across different runs.
  - Files are named collection of data.
- Directories are used to construct file hierarchy.
  - Directories are named collection of files and directories.
- File system is an abstraction layer provided by OS to enable users to use *path* to access files and directories.
  - It records metadata of files and directories (size, modification time, owner, etc.) to make them organized and hierarchical.
- C++17 includes related utilities in `<filesystem>`.

Supplementary

- File system
  - Path operations
    - Overview
    - std::filesystem::path
  - File system operations

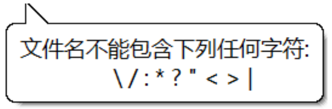
Path Overview

- Essentially, path is a string that represents location of a file.
- There are two different kinds of paths:
    1. Absolute path: always refer to the same location.
    2. Relative path: the location relative to *current working directory* (CWD) for the current process.
        - By changing CWD, the process can get different locations.
- A path consists of these components:
    1. Root name (optional): like drive name in Windows (C:, D:); or UNC (`//machine`), etc.
    2. Root directory (optional): a directory separator (`\` on Windows, `/` on Linux, `:` on classic MacOS).
    3. Relative path: a sequence of filenames separated by directory separator.

Path Overview

- Besides no separators, filenames have many other platform-dependent characteristics or restrictions.

(1.1) — The permitted characters. e.g. on Windows:



文件名不能包含下列任何字符:
\ / : * ? " < > |

[*Example 1*: Some operating systems prohibit the ASCII control characters (0x00 – 0x1F) in filenames. — *end example*]

[*Note 1*: Wider portability can be achieved by limiting *filename* characters to the POSIX Portable Filename Character Set:
`A B C D E F G H I J K L M N O P Q R S T U V W X Y Z`
`a b c d e f g h i j k l m n o p q r s t u v w x y z`
`0 1 2 3 4 5 6 7 8 9 . _ -` — *end note*]

(1.2) — The maximum permitted length. e.g. well-known 260 in some Windows functions.

(1.3) — Filenames that are not permitted.

(1.4) — Filenames that have special meaning.

e.g. CON on Windows.

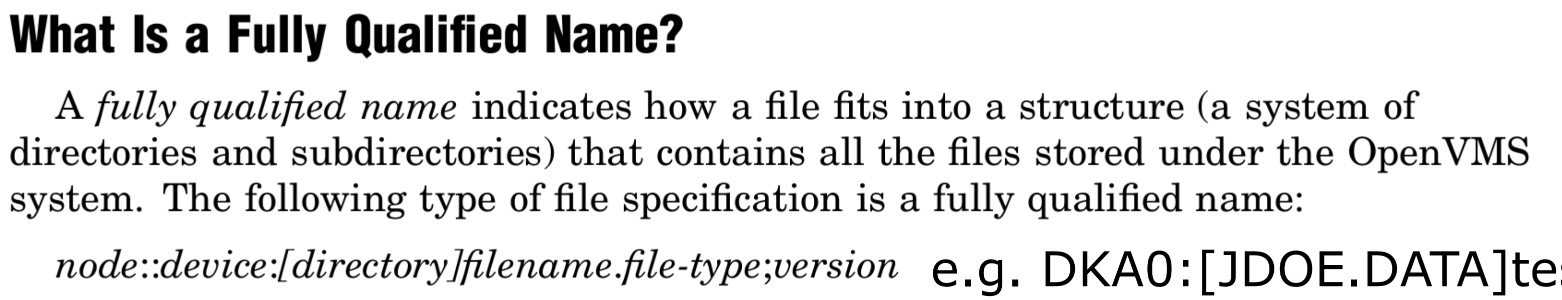
(1.5) — Case awareness and sensitivity during path resolution. e.g. Linux is case-sensitive while Windows not.

(1.6) — Special rules that may apply to file types other than regular files, such as directories.

Particularly, `.` and `..` means current directory and parent directory respectively.

Path Overview

- To make program cross-platform, C++ regulates a “generic format” that uses POSIX convention.
  - That is, these three components are just concatenated to form a path.
    - And / is considered as universal separator.
- Besides, C++ allows “native format” that depends on file system.
  - E.g. OpenVMS, a legacy system previously used in bank (well, if it really supports C++17 utilities).



What Is a Fully Qualified Name?

A *fully qualified name* indicates how a file fits into a structure (a system of directories and subdirectories) that contains all the files stored under the OpenVMS system. The following type of file specification is a fully qualified name:

node::*device*:[*directory*]*filename*.*file-type*;*version*

e.g. DKA0:[JDOE.DATA]test.txt

Path Overview

- Note 1: “generic” only means it’s a valid format in all systems; but its location is not always the same.
  - What is `D:\sub\path`?
    - Windows: an absolute path at D drive, with two components `sub` and `path`.
    - Linux: a relative path with name “`D:\sub\path`”, i.e. the whole string is a single component.
  - What is `/home/user`?
    - Linux: an absolute path with two components `home` and `user`.
    - Windows: a relative path at drive of CWD, with two components `home` and `user`.
- Note 2: it’s not very safe to rely on relative path since it’s just like a global variable, which can be changed by other threads and external library.
  - i.e. the location of a relative path can be modified arbitrarily.

Path Overview

- Note 3: “relative” or “absolute” only means whether it’s interfered by CWD; there can be many paths that refer to the same location.
  - E.g. `/home/user`, `/home/user/.`, `/home/user/dir/..`, ….
  - To unify all representations, we can do *path normalization*.
  - There are two kinds of normalization:
    - Lexical: a string-level substitution, which doesn’t change whether the path is relative or absolute.
      - So `/home/user`, `/home/user/.`, `/home/user/dir/..` are all normalized to `/home/user`, and paths `./user`, `./user/.`, `./user/dir/..` are all normalized to `user`.
    - Filesystem-dependent: paths are normalized to a unique absolute path.
      - Assuming CWD is `/home`, `./user`, `./user/.`, `./user/dir/..` are all normalized to `/home/user`.
      - C++ call such normalization as “canonical”.

Path Overview

- Specifically, a normalized path requires:

A path might be or can become normalized. In a normalized path:

- File names are separated only by a single preferred directory separator.
- The file name "." is not used unless the whole path is nothing but "." (representing the current directory).
- The file name does not contain ".." file names (we do not go down and then up again) unless they are at the beginning of a relative path.
- The path only ends with a directory separator if the trailing file name is a directory with a name other than "." or "..".

Path	**POSIX normalized**		**Windows normalized**	
`foo/.///bar/../`	`foo/`	POSIX may or may not respect //.	`foo\`	(Due to UNC path)
`//host/../foo.txt`	`//host/foo.txt`		`\\host\foo.txt`	
`./f/../.f/`	`.f/`		`.f\`	
`C:bar/../`	`.`		`C:`	
`C:/bar/..`	`C:/`		`C:\`	
`C:\bar\..`	`C:\bar\..`		`C:\`	
`/./../data.txt`	`/data.txt`		`\data.txt`	
`././`	`.`		`.`	

Credit: C++17 the Complete Guide, *Nicolai M. Josuttis*.

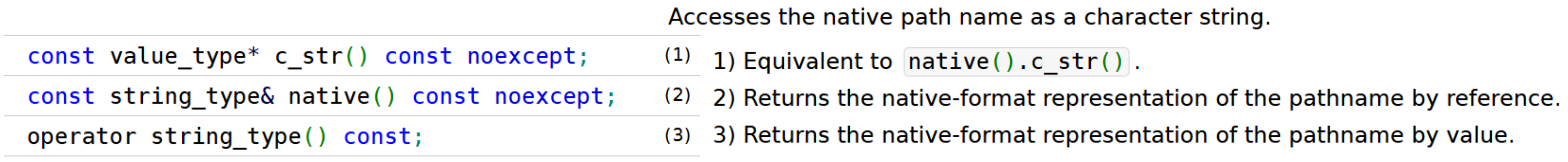
Supplementary

- File system
  - Path operations
    - Overview
    - std::filesystem::path
  - File system operations

For brevity, we use `namespace stdfs = std::filesystem`.

Path

- C++ uses `stdfs::path` to represent a path.
  - It is essentially a string of some *native encoding*, e.g. UTF-8 on Linux, UTF-16 on Windows*.
    - The underlying character can be checked by `stdfs::path::value_type`, normally `char` on Linux and `wchar_t` on Windows.
    - And `stdfs::path::string_type = std::basic_string<value_type>`.
  - And the string stores path in *native format*.
  - You can just access the underlying string in `const` way:

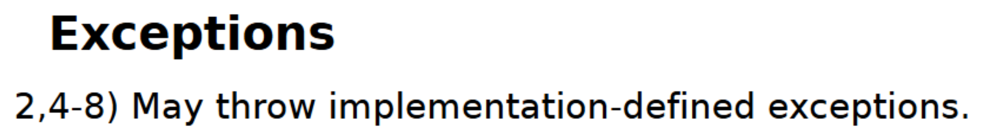


Accesses the native path name as a character string.

`const value_type* c_str() const noexcept;`	(1)	1) Equivalent to `native().c_str()`.
`const string_type& native() const noexcept;`	(2)	2) Returns the native-format representation of the pathname by reference.
`operator string_type() const;`	(3)	3) Returns the native-format representation of the pathname by value.

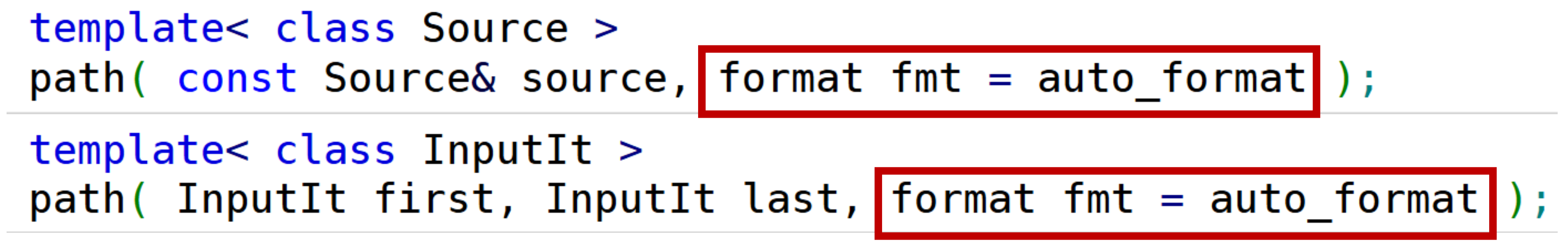
*: Strictly speaking, usually a filesystem doesn't really respect encoding; it just treats the path as some byte sequence (even if it's not a valid UTF-8 / 16). So "native encoding" essentially means "a string that you can pass into filesystem syscall directly", e.g. 2-byte-per-unit sequence on Windows.

Path

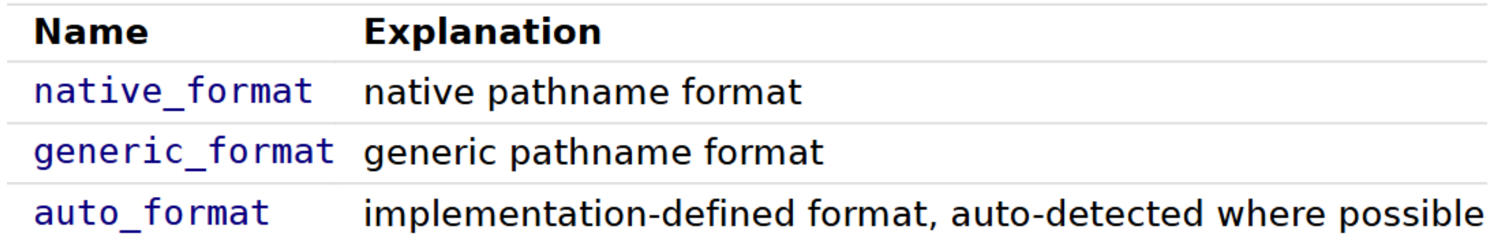

Exceptions

2,4-8) May throw implementation-defined exceptions.

- However, you can construct the path by any encoding and format.


```cpp
template< class Source >
path( const Source& source, format fmt = auto_format );
template< class InputIt >
path( InputIt first, InputIt last, format fmt = auto_format );
```

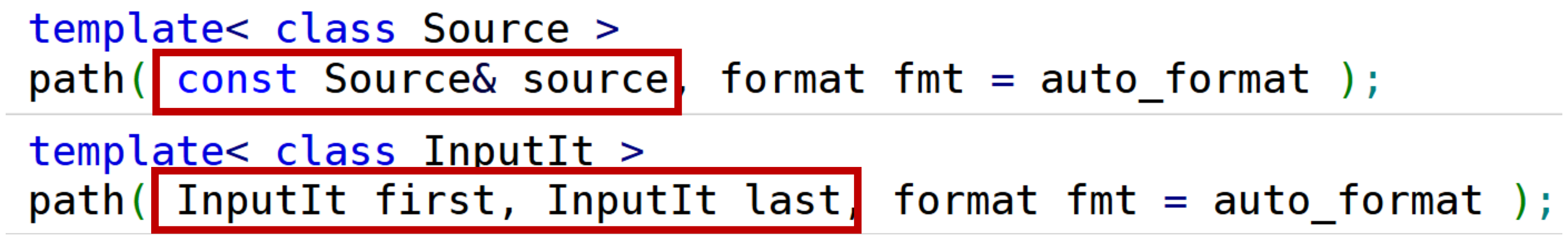
  - For format, it's essentially a scoped enumeration in stdfs::path with three enumerators:



Name	Explanation
native_format	native pathname format
generic_format	generic pathname format
auto_format	implementation-defined format, auto-detected where possible

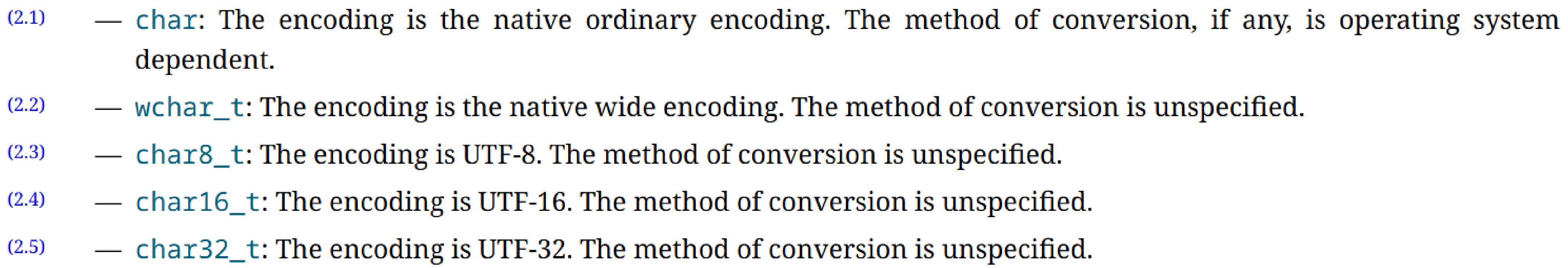
  - By default it uses auto_format, i.e. determine if the input format is native or generic automatically and convert if necessary.

Path



```
template< class Source >
path( const Source& source, format fmt = auto_format );
template< class InputIt >
path( InputIt first, InputIt last, format fmt = auto_format );
```

- The template allows you to use any character type, and ctor will convert to its native encoding.



(2.1) — char: The encoding is the native ordinary encoding. The method of conversion, if any, is operating system dependent.

(2.2) — wchar_t: The encoding is the native wide encoding. The method of conversion is unspecified.

(2.3) — char8_t: The encoding is UTF-8. The method of conversion is unspecified.

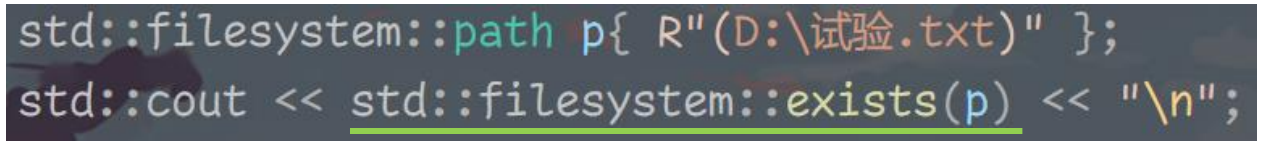
(2.4) — char16_t: The encoding is UTF-16. The method of conversion is unspecified.

(2.5) — char32_t: The encoding is UTF-32. The method of conversion is unspecified.

  - For `wchar_t`, Windows won't do any conversion but Linux needs to do so;
  - For `char`, Linux won't do any conversion but Windows needs to do so.
- Actually, Windows recognizes `char` for file API in ANSI (or Active) Code Page (ACP).
  - This will lead to complex behavior when interacting with compiler option…

Windows ACP

- Assuming that we have a file path “D:\试验.txt” on Windows.
  - And we use a default Chinese PC, i.e. ACP is GBK (id: 936).
- Given main.cpp as:

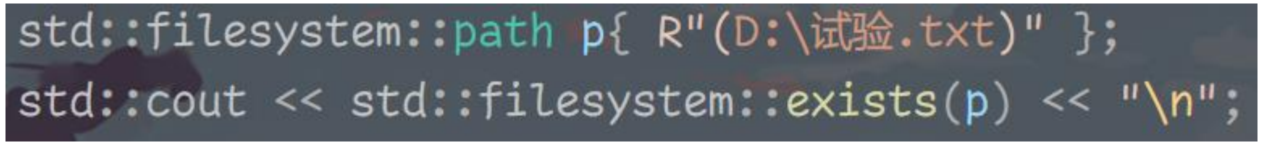


```
std::filesystem::path p{ R"(D:\试验.txt)" };
std::cout << std::filesystem::exists(p) << "\n";
```

Check whether some path exists, equiv. to `std::ifstream{p}.is_open()` if `p` is a file instead of directory.

- Case 1: msvc doesn’t add any option, and encoding of main.cpp is GBK.
  1. D:\试验.txt is in GBK, and msvc reads it as GBK correctly.
  2. The execution charset is GBK, so D:\试验.txt is still GBK in binary exe.
  3. Current ACP is GBK, so stdfs::path converts it from GBK to native encoding (UTF-16) and stores it;
  4. The path is correct so file system says it exists.

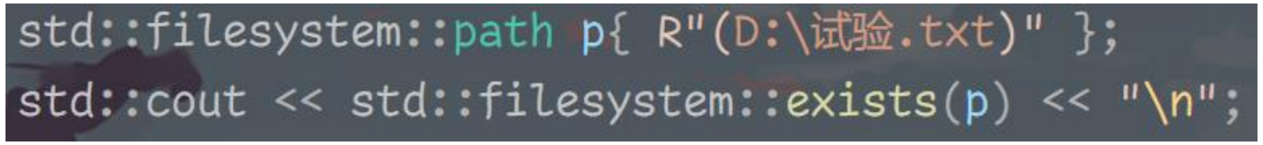
Windows ACP



```
std::filesystem::path p{ R"(D:\试验.txt)" };
std::cout << std::filesystem::exists(p) << "\n";
```

- Case 2: msvc adds `/utf-8`, and encoding of `main.cpp` is UTF-8.
  1. `D:\试验.txt` is in UTF-8, and msvc reads it as UTF-8 correctly.
  2. The execution charset is UTF-8, so `D:\试验.txt` **is UTF-8** in binary exe.
  3. Current ACP is GBK, so `stdfs::path` converts it **from GBK** to native encoding (UTF-16) and stores it;
     - However, UTF-8 string is not really a GBK string, and the corresponding binary leads to GBK as "`D:\璇曢獙.txt`".
  4. The path is not correct and file system says it doesn't exist.
- Case 3: msvc adds `/source-charset:utf-8`, and encoding of `main.cpp` is UTF-8.
  1. `D:\试验.txt` is in UTF-8, and msvc reads it as UTF-8 correctly.
  2. As default execution charset is ACP, `D:\试验.txt` **is GBK** in binary exe.
  3. So the path is still correct and file system says it exists.

Windows ACP

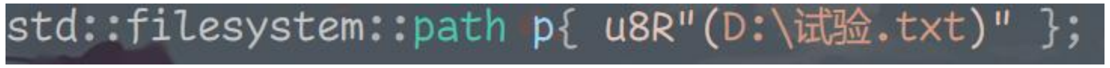


```
std::filesystem::path p{ R"(D:\试验.txt)" };
std::cout << std::filesystem::exists(p) << "\n";
```

- Case 4: msvc adds `/utf-8`, and encoding of `main.cpp` is GBK.
  1. `D:\试验.txt` in GBK is not valid UTF-8, so msvc warns C4828 (illegal character in UTF-8) and silently passes the original bytes as is.
     - So **accidentally**, it's still GBK in binary exe.
  2. Thus accidentally, the path is correct and file system says it exists.
- Case 5: assuming ACP is UTF-8 (65001), msvc doesn't add any option (equiv. to add `/utf-8`), and encoding of `main.cpp` is GBK.
  1. Same as case 4, i.e. the string is of GBK in binary exe.
  2. However, GBK is not valid UTF-8, so ctor of path throws an exception (`std::system_error` in MS-STL).

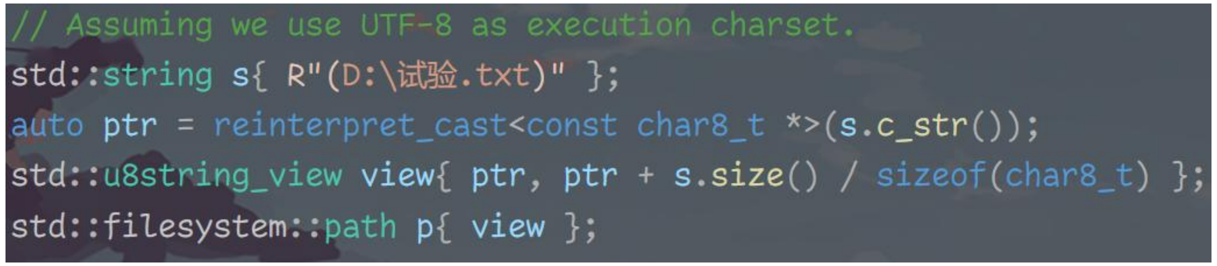
Path construction

- So to make a valid path, we need first:
  - Make sure the compiler knows how to read the file (string literals), i.e. the file encoding should be correctly specified in source charset.
- And then two ways:
  1. Make execution charset (for string literals) and string encoding (for other strings stored in e.g. `std::string`) same as ACP.
  2. Use `char8_t[] / char16_t[] / char32_t[]` instead.
     A. Any ACP is OK, since `char8_t` as a unique type will always be decoded as UTF-8 in ctor.
     B. Any execution charset is OK, since `char8_t` is always UTF-8 in binary exe.



```
std::filesystem::path p{ u8R"(D:\试验.txt)" };
```

  - And if you know your `std::string / …` is essentially UTF-8 while ACP is not, you can `reinterpret_cast` it to avoid conversion.



```
// Assuming we use UTF-8 as execution charset.
std::string s{ R"(D:\试验.txt)" };
auto ptr = reinterpret_cast<const char8_t *>(s.c_str());
std::u8string_view view{ ptr, ptr + s.size() / sizeof(char8_t) };
std::filesystem::path p{ view };
```

Path construction

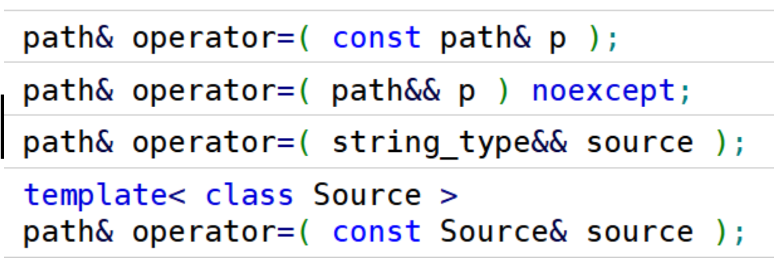
- On the other hand, on Linux + gcc, no matter what execution charset you use, `char[]` won't do any conversion.
  - As `char` is its native encoding, so libstdc++ assumes correct bytes.
  - Instead, `-fwide-exec-charset` will specify encoding of `wchar_t` and be converted to UTF-8 automatically.
- To specify encoding and conversion explicitly, you can use locale:

```
template< class Source >
path( const Source& source, const std::locale& loc, format fmt = auto_format );
```

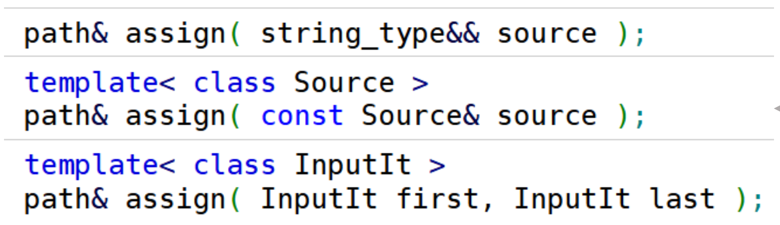
```
template< class InputIt >
path( InputIt first, InputIt last, const std::locale& loc, format fmt = auto_format );
```

  - As the conversion is explicitly specified in locale, the template only accepts a char sequence.

Path const

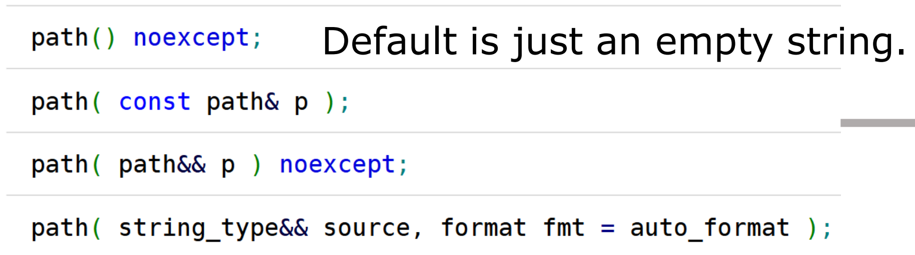


```
path& operator=( const path& p );
path& operator=( path&& p ) noexcept;
path& operator=( string_type&& source );
template< class Source >
path& operator=( const Source& source );
```



```
path& assign( string_type&& source );
template< class Source >
path& assign( const Source& source );
template< class InputIt >
path& assign( InputIt first, InputIt last );
```

- When native encoding is `wchar_t`:
  - Just use `codecvt<wchar_t, char, std::mbstate_t>` in locale to convert `char[]` to native encoding `wchar_t[]`.
    - E.g. For windows, GBK -> UTF-16.
- When native encoding is `char`:
  - First use `codecvt<wchar_t, char, std::mbstate_t>` in locale to convert `char[]` to `wchar_t[]`;
    - E.g. For Linux, GBK -> UTF-32
  - Then convert `wchar_t` back to native encoding `char` (e.g. UTF-8).
    - E.g. UTF-32 -> UTF-8, which is equiv. to construct from `wchar_t[]` directly.
- And finally some omitted overloads & `operator=`, just list here.

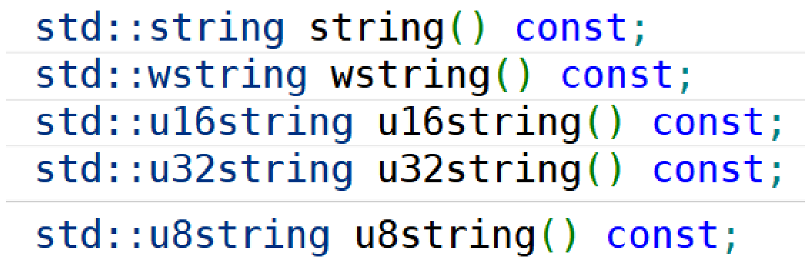


```
path() noexcept;
path( const path& p );
path( path&& p ) noexcept;
path( string_type&& source, format fmt = auto_format );
```

Default is just an empty string.

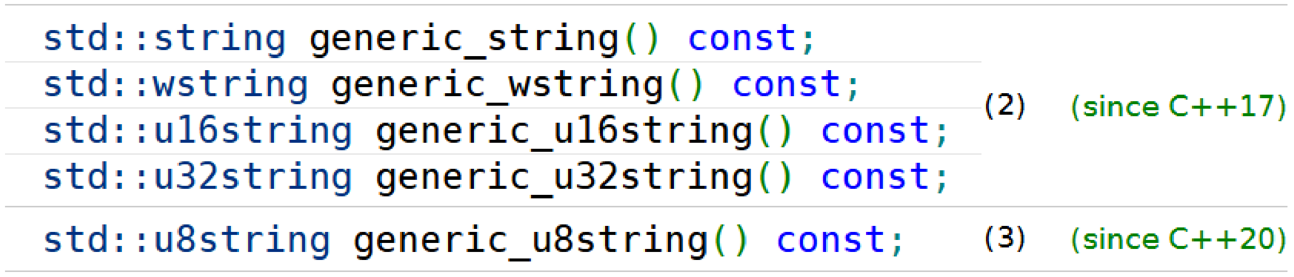
Path

- Besides construction, you can also get string of different formats and encodings.
  - Native format + Native encoding: just `.native()`, as we mentioned.
  - Native format + Converted encoding:



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

std::u8string u8string() const;
```

  - Generic format + Converted encoding:

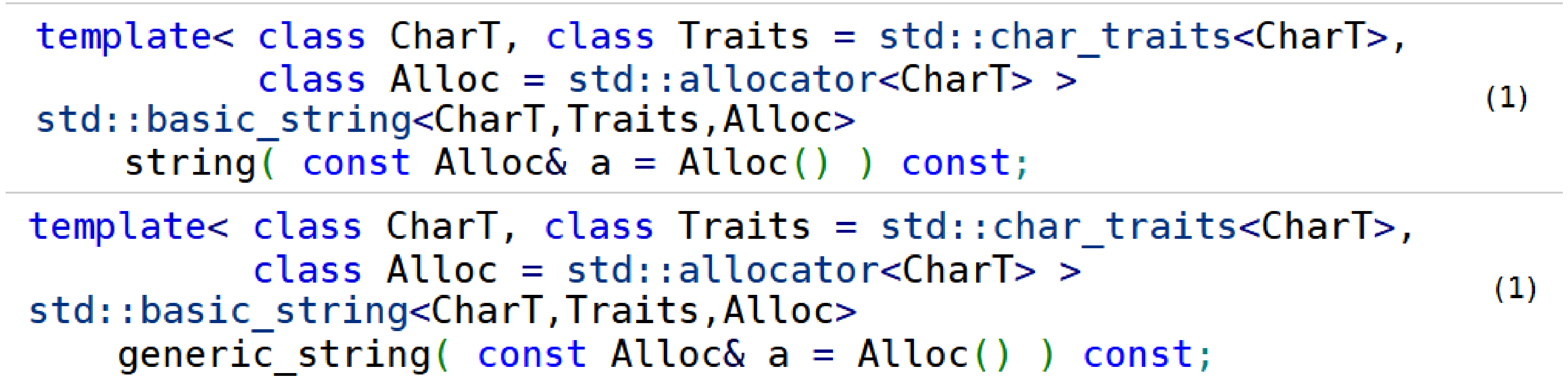


```
std::string generic_string() const;
std::wstring generic_wstring() const;
std::u16string generic_u16string() const;      (2)   (since C++17)
std::u32string generic_u32string() const;

std::u8string generic_u8string() const;        (3)   (since C++20)
```

    - Particularly, these functions are required to use / as directory separator.

Path

- And there are also template versions, together will allocator:



```
template< class CharT, class Traits = std::char_traits<CharT>,
          class Alloc = std::allocator<CharT> >
std::basic_string<CharT,Traits,Alloc>                              (1)
    string( const Alloc& a = Alloc() ) const;
```

```
template< class CharT, class Traits = std::char_traits<CharT>,
          class Alloc = std::allocator<CharT> >
std::basic_string<CharT,Traits,Alloc>                              (1)
    generic_string( const Alloc& a = Alloc() ) const;
```

  - So to get Generic format + Native encoding, just use `.generic_string<stdfs::path::value_type>()`.
- Finally, you can also check the preferred separator by `static constexpr stdfs::path::preferred_separator`.
  - `\` on Windows, `/` on Linux.

Path decomposition

Note: filename with pure extension (e.g. "D:\\.gitignore") is not considered as extension (i.e. stem = filename = ".gitignore", extension = "")

- There are also some observer functions to query path components:
  - which just return new stdfs::path.

root_name

root_directory

root_path

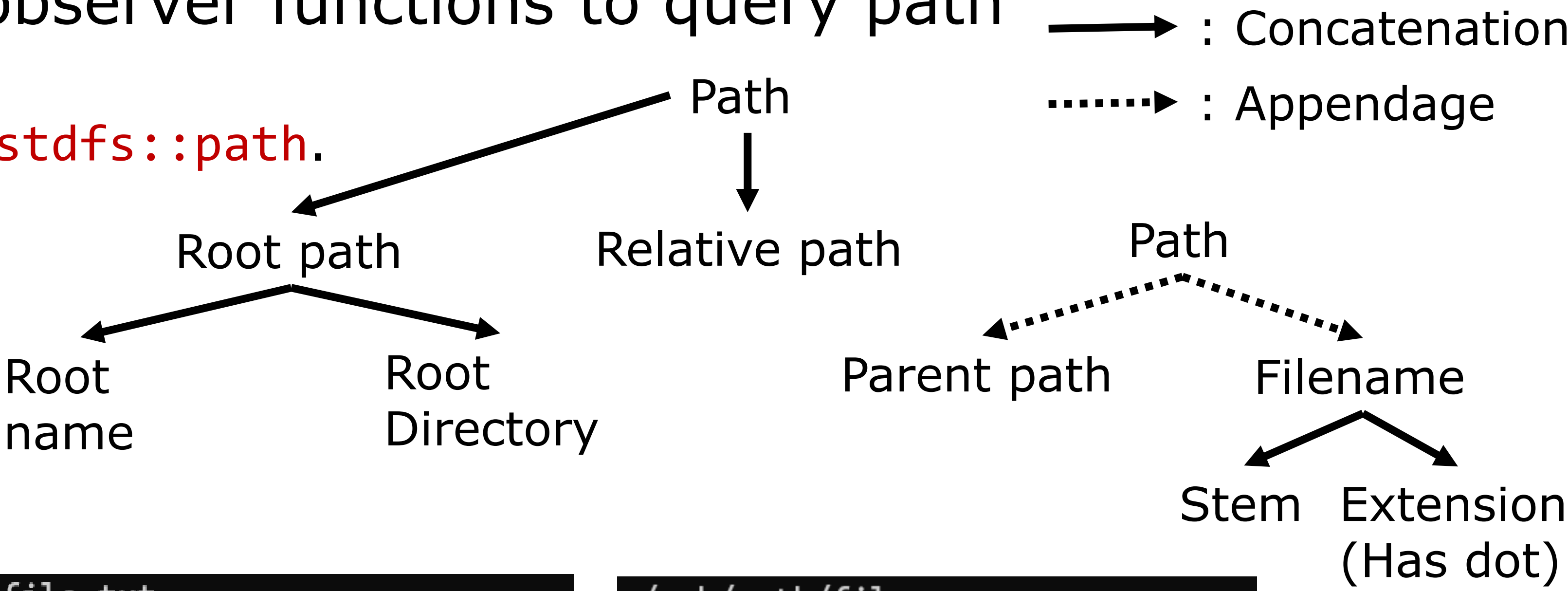
relative_path

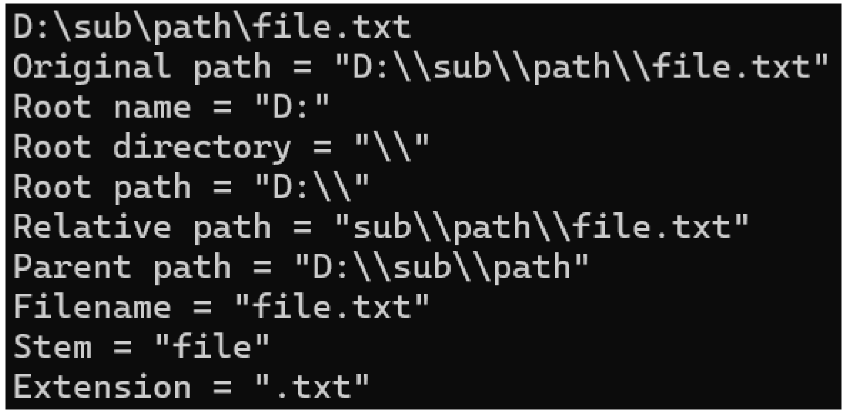
parent_path

filename

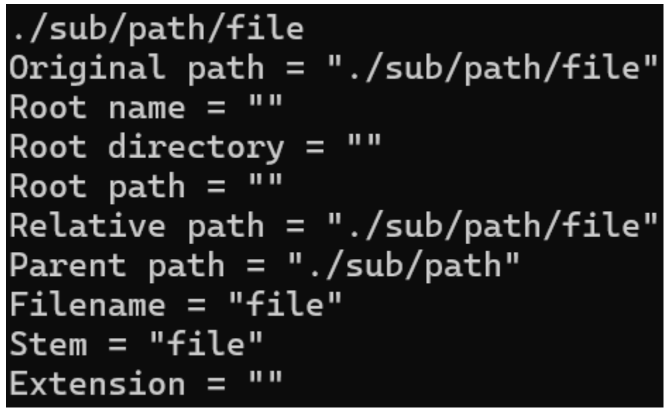
stem

extension





```
D:\sub\path\file.txt
Original path = "D:\\sub\\path\\file.txt"
Root name = "D:"
Root directory = "\\"
Root path = "D:\\"
Relative path = "sub\\path\\file.txt"
Parent path = "D:\\sub\\path"
Filename = "file.txt"
Stem = "file"
Extension = ".txt"
```

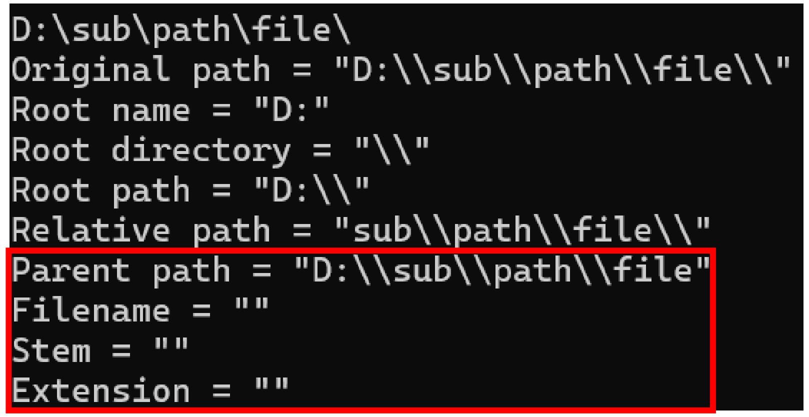


```
./sub/path/file
Original path = "./sub/path/file"
Root name = ""
Root directory = ""
Root path = ""
Relative path = "./sub/path/file"
Parent path = "./sub/path"
Filename = "file"
Stem = "file"
Extension = ""
```

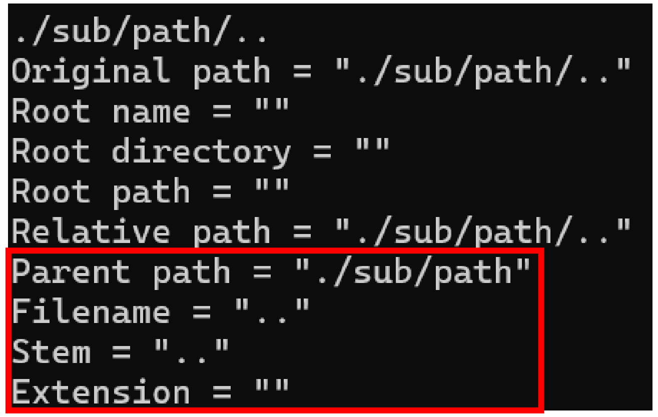
Path decomposition

- Particularly, such decomposition is **lexical**, which doesn’t even really interact with file system.
    - So “parent path” doesn’t really return path of parent directory, but just remove the last component.
        - And when a path ends with directory separator, the last component is just empty, so parent just removes the separator.

- For example:



```
D:\sub\path\file\
Original path = "D:\\sub\\path\\file\\"
Root name = "D:"
Root directory = "\\"
Root path = "D:\\"
Relative path = "sub\\path\\file\\"
Parent path = "D:\\sub\\path\\file"
Filename = ""
Stem = ""
Extension = ""
```



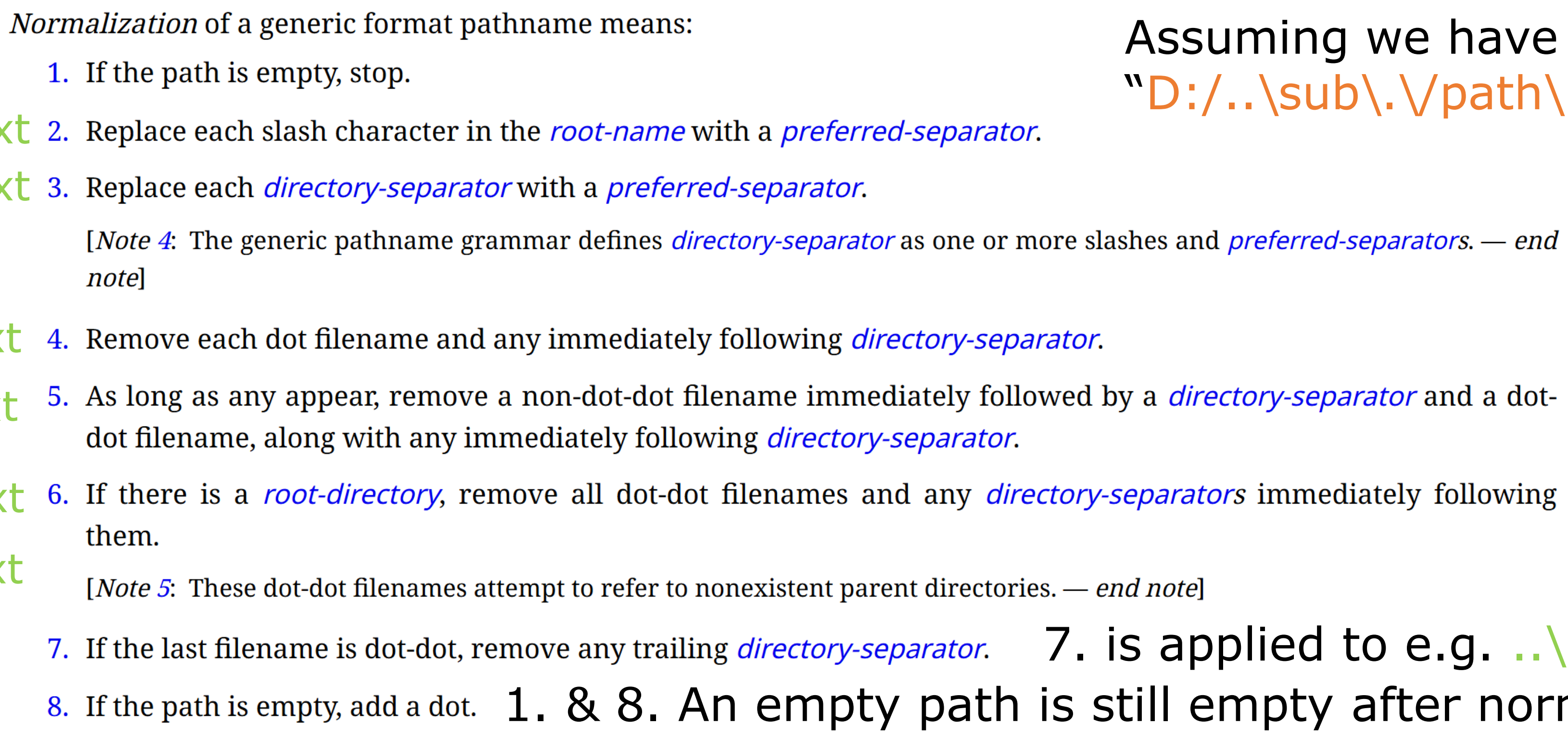
```
./sub/path/..
Original path = "./sub/path/.."
Root name = ""
Root directory = ""
Root path = ""
Relative path = "./sub/path/.."
Parent path = "./sub/path"
Filename = ".."
Stem = ".."
Extension = ""
```

Note: parent of root directory is still root directory (i.e. `"/"` -> `"/"`);
but parent of file name is empty (i.e. `"data.txt"` -> `""`).

Path normalization

- To find real parent, you need to do path normalization first.
  - And we say that there are two ways:
  - Lexical: by `.lexically_normal()`; the normalization process is:

Assuming we have a path as "D:/..\sub\.\/path\..\file.txt".



Normalization of a generic format pathname means:

1. If the path is empty, stop.
2. Replace each slash character in the *root-name* with a *preferred-separator*. — D:/..\sub\.\/path\..\file.txt
3. Replace each *directory-separator* with a *preferred-separator*. — D:\..\sub\.\/path\..\file.txt

   [*Note 4*: The generic pathname grammar defines *directory-separator* as one or more slashes and *preferred-separators*. — *end note*]
4. Remove each dot filename and any immediately following *directory-separator*. — D:\..\sub\.\path\..\file.txt
5. As long as any appear, remove a non-dot-dot filename immediately followed by a *directory-separator* and a dot-dot filename, along with any immediately following *directory-separator*. — D:\..\sub\path\..\file.txt
6. If there is a *root-directory*, remove all dot-dot filenames and any *directory-separators* immediately following them. — D:\..\sub\file.txt

   [*Note 5*: These dot-dot filenames attempt to refer to nonexistent parent directories. — *end note*]

   D:\sub\file.txt
7. If the last filename is dot-dot, remove any trailing *directory-separator*.
8. If the path is empty, add a dot.

7\. is applied to e.g. ..\..\.

1\. & 8. An empty path is still empty after normalization, but a non-empty but essentially empty is normalized to ..

Path normalization

- Filesystem-dependent: `stdfs::canonical / weakly_canonical`; paths are normalized to a unique absolute path.
  - Path is first converted to an absolute path by `stdfs::absolute(p)`;
  - Then perform lexical normalization.

  - `canonical` will check whether the path really exists, while `weakly_canonical` just normalizes it.
  - We'll go into details about two forms of global APIs in `stdfs` later.

```
path canonical( const std::filesystem::path& p );                    (1)
path canonical( const std::filesystem::path& p,
                std::error_code& ec );                               (2)
path weakly_canonical( const std::filesystem::path& p );             (3)
path weakly_canonical( const std::filesystem::path& p,
                       std::error_code& ec );                        (4)
```

Example

```cpp
std::filesystem::path p{ s };
std::cout << "Original path = " << p << "\n";
std::cout << "-------------- Lexical normal --------------\n";
OutputProperties(p.lexically_normal());
std::cout << "-------------- Weakly canonical --------------\n";
OutputProperties(std::filesystem::weakly_canonical(p));
```

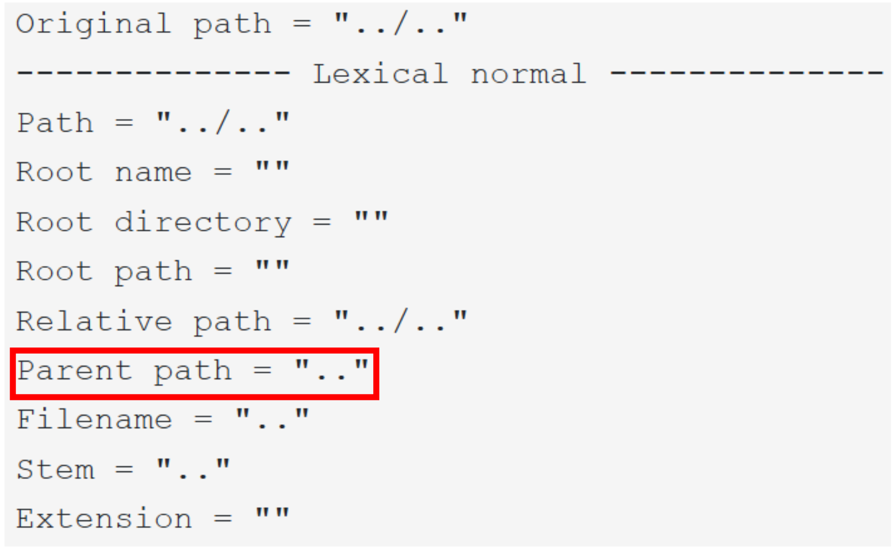
```
Original path = "./sub/path/.."
-------------- Lexical normal ---------------
Path = "sub/"
Root name = ""
Root directory = ""
Root path = ""
Relative path = "sub/"
Parent path = "sub"
Filename = ""
Stem = ""
Extension = ""
-------------- Weakly canonical --------------
Path = "/app/sub/"
Root name = ""
Root directory = "/"
Root path = "/"
Relative path = "app/sub/"
Parent path = "/app/sub"
Filename = ""
Stem = ""
Extension = ""
```

The last / is not stripped, even after normalization. .. only removes **trailing** /.

```
Original path = "./sub/path/"
-------------- Lexical normal --------------
Path = "sub/path/"
Root name = ""
Root directory = ""
Root path = ""
Relative path = "sub/path/"
Parent path = "sub/path"
Filename = ""
Stem = ""
Extension = ""
-------------- Weakly canonical --------------
Path = "/app/sub/path/"
Root name = ""
Root directory = "/"
Root path = "/"
Relative path = "app/sub/path/"
Parent path = "/app/sub/path"
Filename = ""
Stem = ""
Extension = ""
```

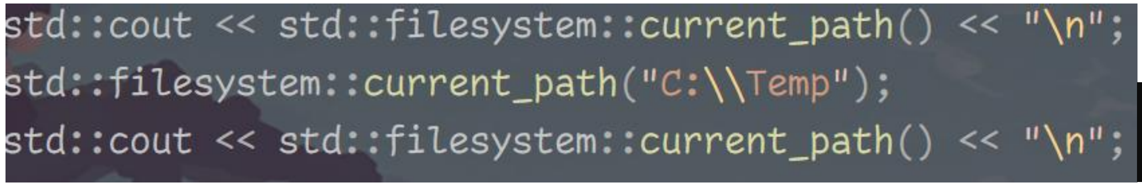
Path normalization

- Notice that "physical" parent of lexical normalization may still be wrong when normalized result still contains `..`.
  - Only `(weakly_)canonical` ensures a correct physical parent.

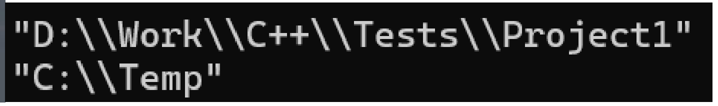


```
Original path = "../.."
-------------- Lexical normal --------------
Path = "../.."
Root name = ""
Root directory = ""
Root path = ""
Relative path = "../.."
Parent path = ".."
Filename = ".."
Stem = ".."
Extension = ""
```

- Finally, you can get or set CWD by `current_path()`:



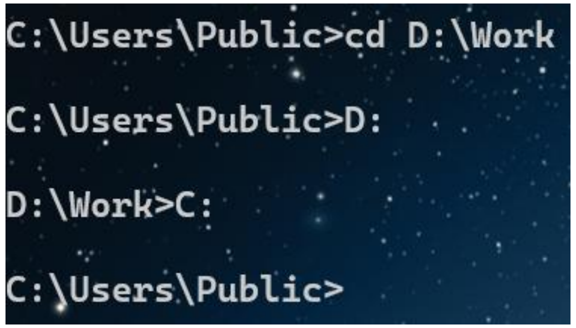
```
std::cout << std::filesystem::current_path() << "\n";
std::filesystem::current_path("C:\\Temp");
std::cout << std::filesystem::current_path() << "\n";
```



```
"D:\\Work\\C++\\Tests\\Project1"
"C:\\Temp"
```

  - So `absolute()` is essentially `current_path() / path` when `path` is relative.

DOS directory



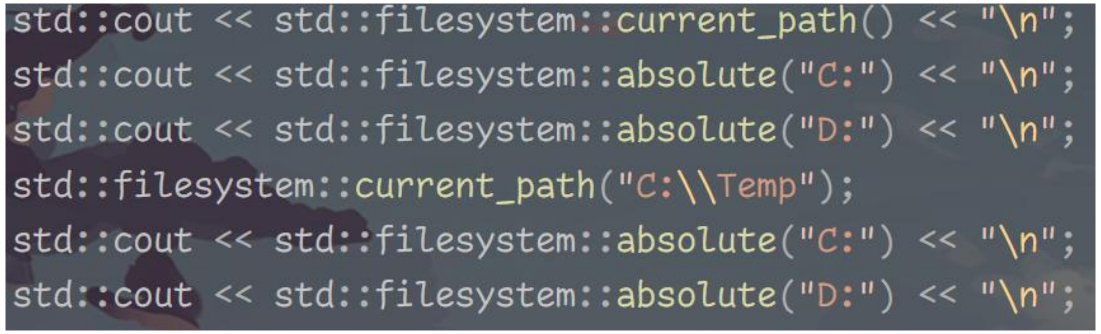
```
C:\Users\Public>cd D:\Work

C:\Users\Public>D:

D:\Work>C:

C:\Users\Public>
```

cd: change current directory

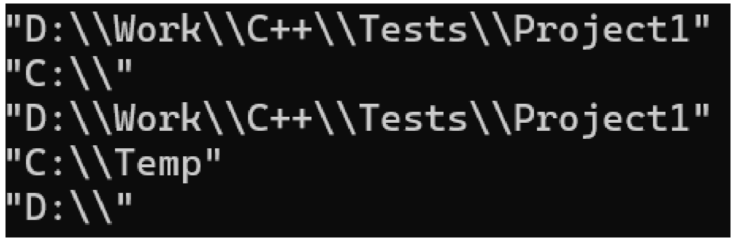
"D:" : switch to current directory of Drive D.

"C:" : switch to current directory of Drive C.

- It's also worth nothing that DOS maintains separate "current directory" for every drive.
  - So `C:` and `D:` are actually relative paths, while `C:\` and `D:\` are absolute paths.
- In Windows, current directory is unified as a single path, as known by CWD.
  - However, CMD pretends they are still there by storing them with "strange environment variables".
  - And Windows inherits the DOS behavior, regarding `C:` and `D:` as relative paths. But there exists only a single real CWD.



```cpp
std::cout << std::filesystem::current_path() << "\n";
std::cout << std::filesystem::absolute("C:") << "\n";
std::cout << std::filesystem::absolute("D:") << "\n";
std::filesystem::current_path("C:\\Temp");
std::cout << std::filesystem::absolute("C:") << "\n";
std::cout << std::filesystem::absolute("D:") << "\n";
```



```
"D:\\Work\\C++\\Tests\\Project1"
"C:\\"
"D:\\Work\\C++\\Tests\\Project1"
"C:\\Temp"
"D:\\"
```

Other non-CWD drive will return root.

1: term “relativization” is from Java.nio.

Path relativization[1]

- In contrast to normalization, path can also be “denormalized” by converting an absolute path to relative path.
  - More generally, given a path `b`, how can it be transformed to path `a` with shortest components.
  - For example, `path{ "/a/d" }.relative("/a/b/c")` would be `"../../d"`.
- Similarly, you can use two ways:
  - Lexical: by `a.lexically_relative(b)`; the process is:

    ① Check whether it’s possible to transform `b` to `a`.
    - If it’s impossible (i.e. conditions below), return empty path directly.

— `root_name() != base.root_name()` is `true`, or

E.g. two paths in different drives in Windows.

— `is_absolute() != base.is_absolute()` is `true`, or

Absolute path + relative path, not lexically transformable.

— `!has_root_directory() && base.has_root_directory()` is `true`, or

E.g. `bar` and `/foo` on Windows, i.e. you cannot cd from `/foo` to `bar` without CWD.

— any *filename* in `relative_path()` or `base.relative_path()` can be interpreted as a *root-name*,

E.g. for UNC path on Windows, e.g. `\\.\C:\Test` uses `C:\Test` as relative path.
That is, UNC doesn’t participate in lexical relative process.

```
a = D:\test\test.txt
b = D:\test\test2\test.txt\..
```

Path relativization

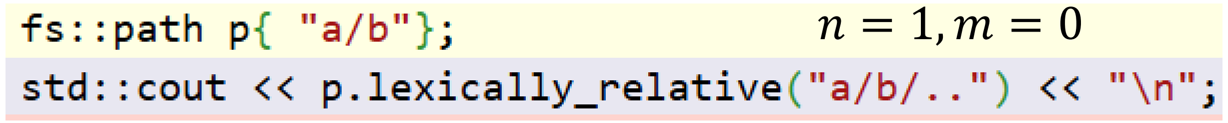
② Determine the first mismatched component of two paths (just like `std::mismatch`).

- Let remaining mismatched components of `a` be $[a_1, a_2)$ and `b` be $[b_1, b_2)$;
    - Example above is $[a_1, a_2) = [test.txt], [b_1, b_2) = [test2, test.txt, ..]$.
- If no mismatched component (i.e. $a_1 = a_2, b_1 = b_2$), return `path{ “.” }`;
- Otherwise, assuming that in $[b_1, b_2)$, n components are `..` and m components are not `..`, `.` and empty.
    - Example above is $n = 1, m = 2$.
    - If $n > m$ (that is, the lexically normalized form contains only `..`), then return empty path.
    - If $n = m$ (that is, the lexically normalized form is `.`), then:
        - If $a_1 = a_2$, return `path{ “.” }`;
        - Otherwise, return $[a_1, a_2)$.
    - If $n < m$, then return a path with 1. `".."` repeated for $m - n$ times; 2. $[a_1, a_2)$.

```
result = ..\test.txt
```

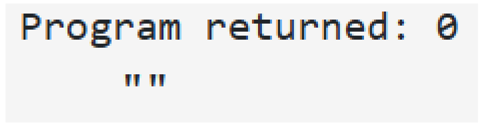
Path relativization

- Actually, this algorithm may falsely report empty path even when such transformation should be possible.
  - For example:



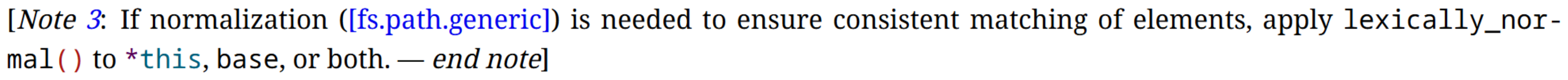
```
fs::path p{ "a/b"};
std::cout << p.lexically_relative("a/b/..") << "\n";
```

$n = 1, m = 0$



```
Program returned: 0
    ""
```

    - Though theoretically it can be “`..`”.
  - If you want an always-correct lexical transformation, you need to do lexical normalization first.

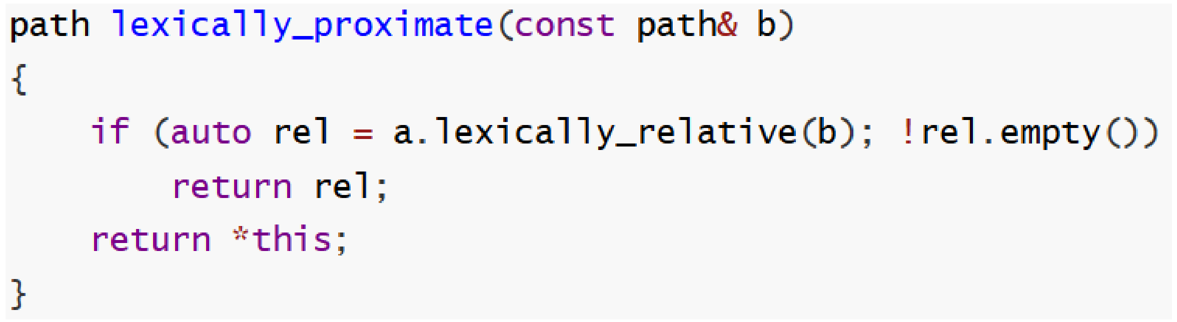


[*Note 3*: If normalization ([fs.path.generic]) is needed to ensure consistent matching of elements, apply `lexically_normal()` to `*this`, base, or both. — *end note*]

- The second way is filesystem-dependent `stdfs::relative(a, b)`, which will always ensure correct relative path in the file system.
  - It’s same as `lexically_relative` two `weakly_canonical` paths.

Path proximation

- Finally there also exists proximation, which means “relativization if possible, otherwise return original path”.
  - Effectively:



```
path lexically_proximate(const path& b)
{
    if (auto rel = a.lexically_relative(b); !rel.empty())
        return rel;
    return *this;
}
```

  - And `stdfs::proximate(a, b)` is also same as `lexically_proximate` two `weakly_canonical` paths.
  - BTW, when `b` is not provided in `relative` and `proximate`, `current_directory` will be used.

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

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

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

template< class Source >
path& append( const Source& source );

template< class InputIt >
path& append( InputIt first, InputIt last );
```

Path composition

- For a given path `./sub/path/file.txt`, it's essentially a combination of hierarchical components.
- C++ provides two utilities to combine components.
  1. Append: combine two components with a directory separator, if needed.
     - For example: (on Linux)

```
std::filesystem::path p = "/home";
//  p == /home/tux/.fonts
std::cout << p / "tux" / ".fonts" << '\n';
```

```
std::filesystem::path p = "/home";
//  p == /home/tux/.fonts
std::cout << p / "tux/" / ".fonts" << '\n';
```

     - However, there exist lots of corner cases…

     ① Subpath is absolute path: C++ chooses to overwrite (replace) LHS.
       - For example:

```
// On Windows,
path("foo") / "C:/bar";  // the result is "C:/bar" (replaces)
```

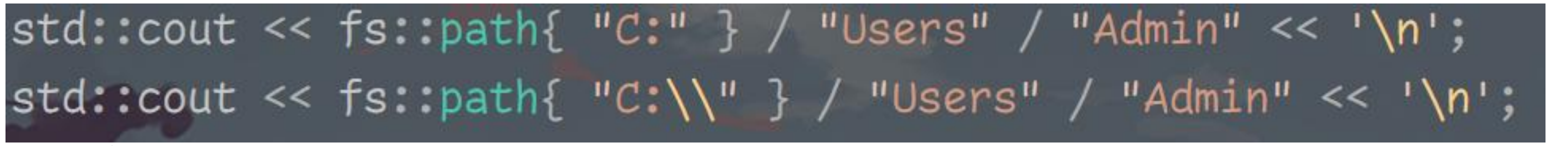
       - But this can be astonishing:

```
std::filesystem::path p = "/home";
//  p == /.fonts
std::cout << p / "tux" / "/.fonts" << '\n';
```

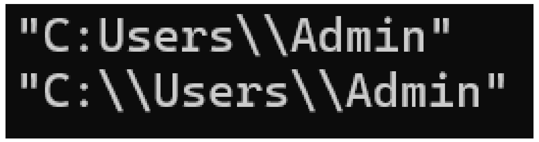
Reason: "`/.fonts`" is absolute path.

Path composition

- DOS-like behavior on Windows also causes surprising result even when subpath is relative.
  - ② No separator is inserted for a single drive; it's still a relative path.

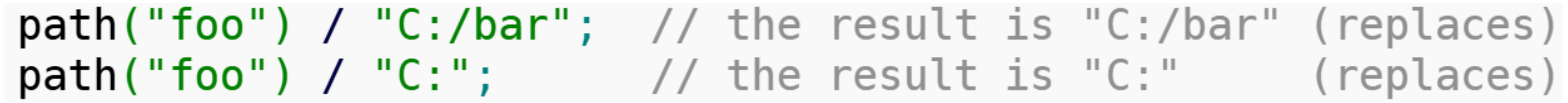


```
std::cout << fs::path{ "C:" } / "Users" / "Admin" << '\n';
std::cout << fs::path{ "C:\\" } / "Users" / "Admin" << '\n';
```



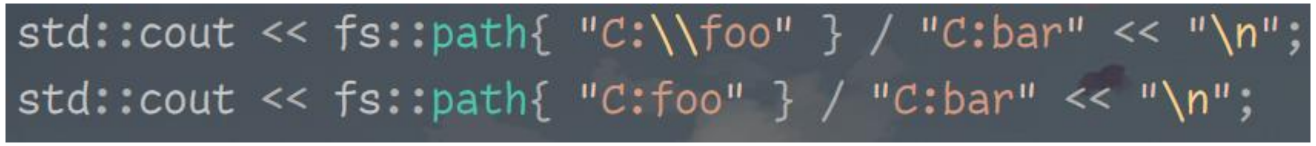
```
"C:Users\\Admin"
"C:\\Users\\Admin"
```

  - ③ Appending a relative path that has different drive will replace the whole path;

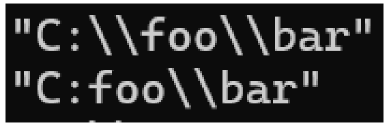


```
path("foo") / "C:/bar";   // the result is "C:/bar" (replaces)
path("foo") / "C:";       // the result is "C:"     (replaces)
```

  - ④ Appending a relative path that has same drive will append as if LHS is CWD.

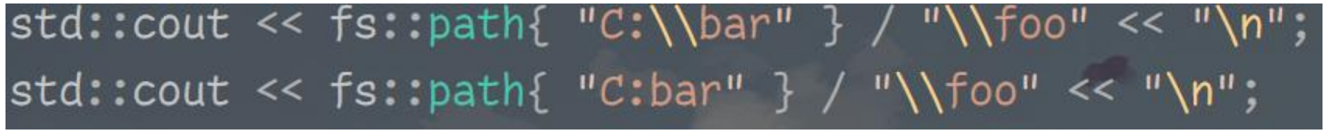


```
std::cout << fs::path{ "C:\\foo" } / "C:bar" << "\n";
std::cout << fs::path{ "C:foo" } / "C:bar" << "\n";
```

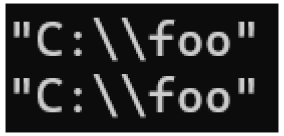


```
"C:\\foo\\bar"
"C:foo\\bar"
```

  - ⑤ Appending a relative path that has root directory but no drive, to another path with drive will reserve LHS drive.



```
std::cout << fs::path{ "C:\\bar" } / "\\foo" << "\n";
std::cout << fs::path{ "C:bar" } / "\\foo" << "\n";
```



```
"C:\\foo"
"C:\\foo"
```

Path composition

```
template< class Source >
path& concat( const Source& source );                                  (7)

template< class InputIt >
path& concat( InputIt first, InputIt last );                           (8)
```

2. Concatenate: combine two components as if concatenating underlying strings directly; no additional separator is introduced.

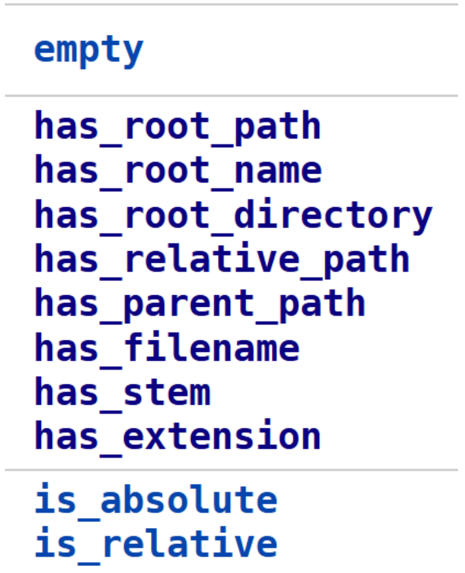
```
path& operator+=( const path& p );                                     (1)

path& operator+=( const string_type& str );
path& operator+=( std::basic_string_view<value_type> str );            (2)

path& operator+=( const value_type* ptr );                             (3)

path& operator+=( value_type x );                                      (4)

template< class CharT >
path& operator+=( CharT x );                                           (5)

template< class Source >
path& operator+=( const Source& source );                              (6)
```

These overloads are designed to mimic overloads of `std::string::operator+=`.

- Strangely, there is no `operator+`; but normally this operation is used to concatenate with a string, so you can just `operator+` all strings first.
  - Or you have to use either `((stdfs::path{a} += b) += c)`… or `.native()` to use `operator+` of `std::basic_string`.

Path composition

- Note 1: Due to associativity, `p / "a" / "b"` is legal while `p /= "a" /= "b"` is illegal.
  - `p / "a" / "b"` ⇔ `((p / "a") / "b")`, while
  - `p /= "a" /= "b"` ⇔ `(p /= ("a" /= "b"))`.
    - And it's illegal for two string literals to `/=`.
  - You have to add a bunch of brackets; that's why we use `((stdfs::path{a} += b) += c)…`.
- Note 2: there also exist some boolean observers to check existence.
  - "empty" means the underlying string contains nothing.
  - And `has_xxx` means whether `xxx` is empty or not.



```
empty

has_root_path
has_root_name
has_root_directory
has_relative_path
has_parent_path
has_filename
has_stem
has_extension

is_absolute
is_relative
```

Path iteration

- As a combination of many different components, path can also be iterated (grouped by separator) in generic format.
- It provides `.begin()` and `.end()` that return a path const iterator.
  - It just iterates through root name, root directory and filenames.
  - For example:

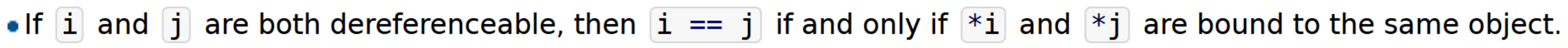
```cpp
#include <filesystem>
#include <iostream>
namespace fs = std::filesystem;

int main()
{
    const fs::path p =
#   ifdef _WIN32
        "C:\\users\\abcdef\\AppData\\Local\\Temp\\";
#   else
        "/home/user/.config/Cppcheck/Cppcheck-GUI.conf";
#   endif
    std::cout << "Examining the path " << p << " through iterators gives\n";
    for (auto it = p.begin(); it != p.end(); ++it)
        std::cout << *it << " │ ";
    std::cout << '\n';
}
```

```
--- Windows ---
Examining the path "C:\users\abcdef\AppData\Local\Temp\" through iterators gives
"C:" │ "/" │ "users" │ "abcdef" │ "AppData" │ "Local" │ "Temp" │ "" │

--- UNIX ---
Examining the path "/home/user/.config/Cppcheck/Cppcheck-GUI.conf" through iterators gives
"/" │ "home" │ "user" │ ".config" │ "Cppcheck" │ "Cppcheck-GUI.conf" │
```

Deferenced result (i.e. `iterator::value_type`) is another `path`.

Path iteration

- Though it seems to be bidirectional iterator, it’s actually only regulated to be input iterator.
  - Reason: before C++20, forward iterator has such a regulation:



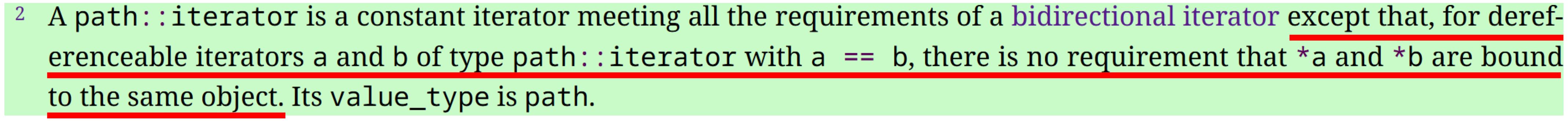
- If `i` and `j` are both dereferenceable, then `i == j` if and only if `*i` and `*j` are bound to the same object.

  - That is, every component should have a fixed source to make every iterator dereferenced to that source.
  - This requires `path` to store a container of components, making it expensive to construct any `path`…
    - And this is how libstdc++ implements it, making it bidirectional.
- However, path iterator is quite like a string `std::views::split` by separator!
  - Another way (as libc++ and MS-STL do) is to cache the range in the iterator, so only when iterator is used will parsing begins.

Issue 2674: Bidirectional iterator requirement on path::iterator is very expensive

Path iteration

- So instead, the standard regulates that:



2 A `path::iterator` is a constant iterator meeting all the requirements of a bidirectional iterator except that, for dereferenceable iterators a and b of type `path::iterator` with `a == b`, there is no requirement that `*a` and `*b` are bound to the same object. Its `value_type` is `path`.

  - which makes it only satisfies input iterator, and thus it's impossible to apply some functions in `<algorithm>`.

- BUT, such requirement is not part of `bidirectional_iterator` in C++20!
  - So theoretically it should be able to utilize constrained algorithms, i.e. `std::ranges::xxx`.
  - Well, this problem is much more complicated… See our homework for discussion.
    - Anyway, libc++ already adds `iterator_concept` for it, while MS-STL cannot do so.

Path modification

Modifiers

- clear
- make_preferred
- remove_filename
- replace_filename
- replace_extension
- swap

Return reference to *this.

- There also exist some simple non-const methods:
  1. `.make_preferred()`: for path whose native format is also generic format, convert current separators to preferred separators.
     - For example:

```
fs::path p{ "C:/Test\\Test2" };
std::cout << p << "\n" << p.make_preferred() << "\n";
```

```
"C:/Test\\Test2"
"C:\\Test\\Test2"
```

  2. `.remove_filename()`: Remove the last component (if it exists) so `.has_filename()` returns `false`.
     - So after removal, the path is either empty or ends with a separator.

```
std::cout << std::boolalpha
          << (p = "foo/bar").remove_filename()
          << (p = "foo/").remove_filename() <<
          << (p = "/foo").remove_filename() <<
          << (p = "/").remove_filename() << '\
          << (p = "").remove_filename() << '\t
```

```
"foo/"
"foo/"
"/"
"/"
""
```

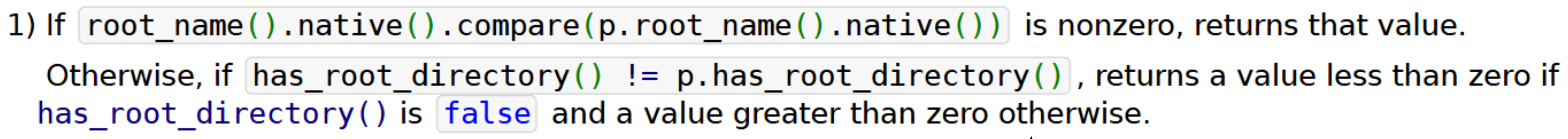
"foo" will also be converted to "".

Path modification

3. `.replace_filename(const path& rep)`: equivalent to 1. `this->remove_filename()`; 2. `(*this) /= rep`.
4. `.replace_extension(const path& rep = {})`: equivalent to code below:
    - For example:

Path:	Ext:	Result:
"/foo/bar.jpg"	".png"	"/foo/bar.png"
"/foo/bar.jpg"	"png"	"/foo/bar.png"
"/foo/bar.jpg"	"."	"/foo/bar."
"/foo/bar.jpg"	""	"/foo/bar"
"/foo/bar."	"png"	"/foo/bar.png"
"/foo/bar"	".png"	"/foo/bar.png"
"/foo/bar"	"png"	"/foo/bar.png"
"/foo/bar"	"."	"/foo/bar."
"/foo/bar"	""	"/foo/bar"
"/foo/."	".png"	"/foo/..png"
"/foo/."	"png"	"/foo/..png"
"/foo/."	"."	"/foo/.."
"/foo/."	""	"/foo/."
"/foo/"	".png"	"/foo/.png"
"/foo/"	"png"	"/foo/.png"

```cpp
// Assume we have a function TryRemoveExtension,
// which will remove extension if it exists.
path& ReplaceExtension(const path& rep = {})
{
    TryRemoveExtension();
    if (rep.empty)
        return *this;
    // Add '.' if necessary
    if (rep.native()[0] != DOT)
        this->concat(DOT);
    return *this += rep;
}
```


1) If `root_name().native().compare(p.root_name().native())` is nonzero, returns that value.

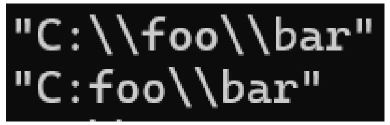
Otherwise, if `has_root_directory() != p.has_root_directory()`, returns a value less than zero if `has_root_directory()` is `false` and a value greater than zero otherwise.

Before element-wise comparison, it needs to judge these conditions first.

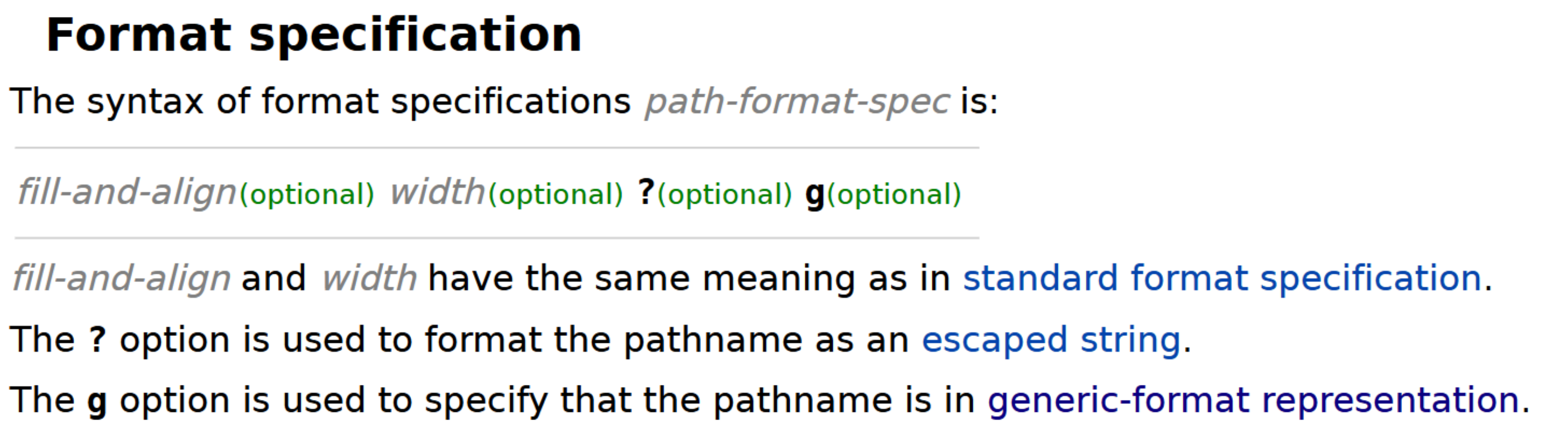
Path

- And finally some simple utilities, just list here.
  - Comparable (by `<=>`/`==` or `.compare`); compare **components**.
  - Hashable (by `std::hash` or `friend hash_value`); hash components.
  - Input / Output by >> / <<;
    - For `i/ostream<CharT, Traits>`, equiv. to input / output the `.string<CharT, Traits>()` with `std::quoted` so space will not interrupt input.
    - Recap: `quoted` will escape the quote and the escape, so `\` is escaped to `\\`.
  - Formattable **since C++26**.

e.g. `"D:/Test"` `== "D://Test"`



```
"C:\\foo\\bar"
"C:foo\\bar"
```



Format specification

The syntax of format specifications *path-format-spec* is:

fill-and-align(optional) *width*(optional) **?**(optional) **g**(optional)

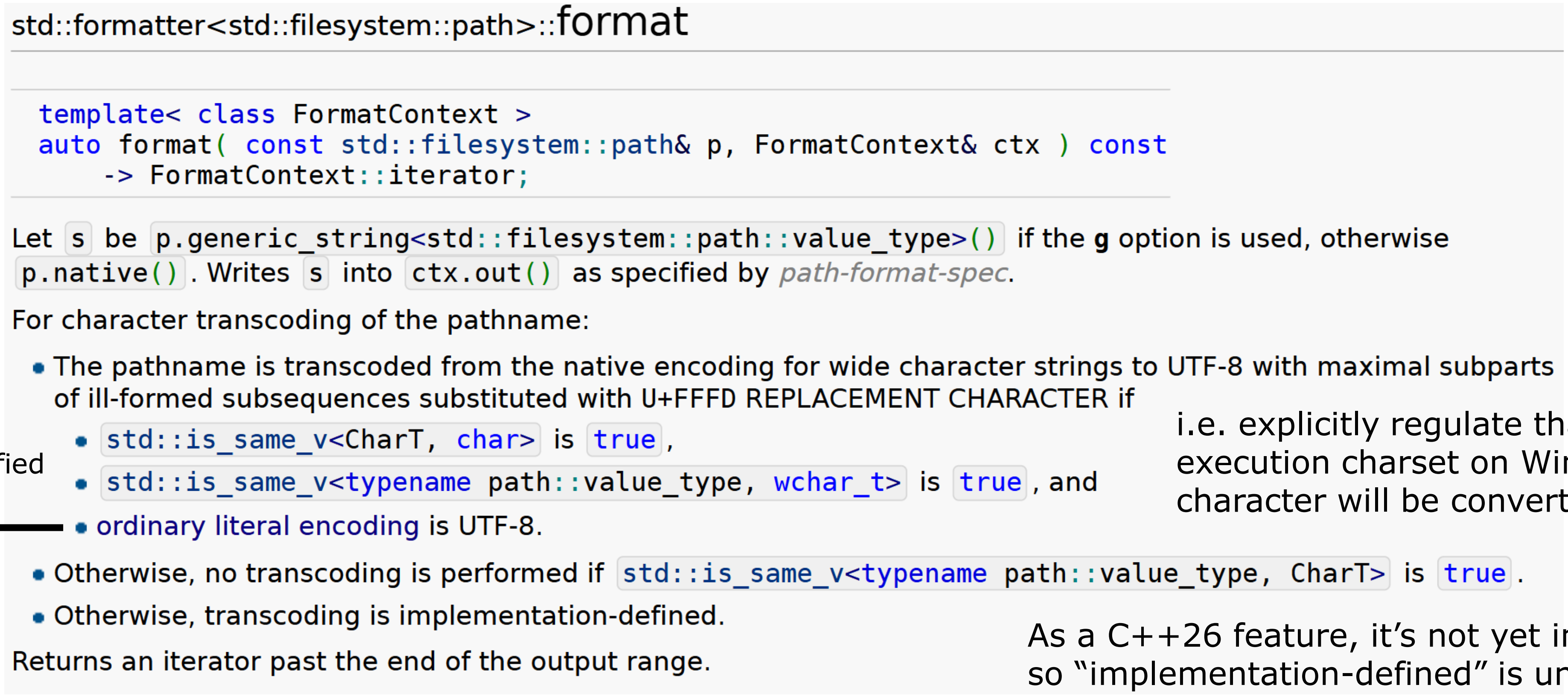
fill-and-align and *width* have the same meaning as in standard format specification.

The **?** option is used to format the pathname as an escaped string.

The **g** option is used to specify that the pathname is in generic-format representation.

Path formatting

- Its regulation is slightly different from `.string()`.



std::formatter<std::filesystem::path>::format

```cpp
template< class FormatContext >
auto format( const std::filesystem::path& p, FormatContext& ctx ) const
    -> FormatContext::iterator;
```

Let `s` be `p.generic_string<std::filesystem::path::value_type>()` if the **g** option is used, otherwise `p.native()`. Writes `s` into `ctx.out()` as specified by *path-format-spec*.

For character transcoding of the pathname:

- The pathname is transcoded from the native encoding for wide character strings to UTF-8 with maximal subparts of ill-formed subsequences substituted with U+FFFD REPLACEMENT CHARACTER if
  - `std::is_same_v<CharT, char>` is `true`,
  - `std::is_same_v<typename path::value_type, wchar_t>` is `true`, and
  - ordinary literal encoding is UTF-8.
- Otherwise, no transcoding is performed if `std::is_same_v<typename path::value_type, CharT>` is `true`.
- Otherwise, transcoding is implementation-defined.

Returns an iterator past the end of the output range.

i.e. encoding of char string literal, as specified in compiler execution charset option.

i.e. explicitly regulate that in UTF-8 execution charset on Windows, illegal character will be converted to U+FFFD.�

As a C++26 feature, it's not yet implemented so "implementation-defined" is unknown (but likely to be printable with `std::print`).

Final Notes

- Note 1: there also exists `.u8path()` to construct from a UTF-8 string in C++17, which is then deprecated in C++20.
  - Reason: C++20 introduces `char8_t`, which distinguishes UTF-8 sequence from `char` by template so there is no need to introduce a new method.
  - For the same reason, `.(generic_)u8string` returns `std::string` in C++17 and `std::u8string` in C++20.
- Note 2: to use `path` as key in map, you usually need to normalize it first by `canonical`.
  - Reason: comparison and hashing of `path` are performed **lexically** for the underlying components.
  - Without normalization, two equivalent paths may be seen as two keys.
  - On Windows, you may even need to `to_lower` all case-insensitive paths.
  - A more expensive but always-correct way is by `stdfs::equivalent` to compare, which needs file system call to check equality of two paths.

Covered later.

Supplementary

- File system
    - Path operations
    - File system operations
        - Overview
        - File status query and directory iteration
        - Modification operations

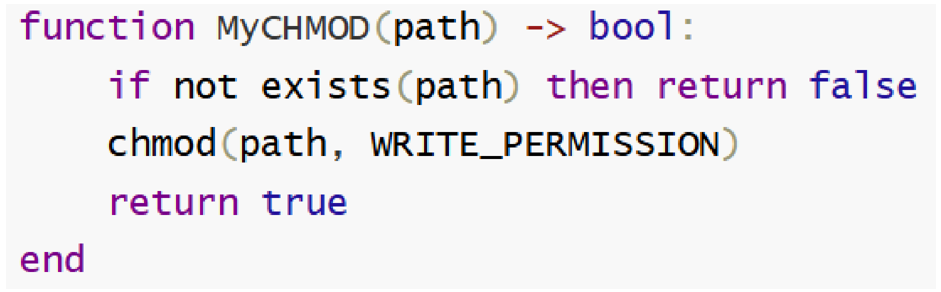
Overview

- In this section, most of the functions interact with the underlying filesystem, which are in the global scope `stdfs::`.
  - By contrast, functions we taught in the last section are purely lexical (and thus cheaper), which are member functions of `stdfs::path`.
- Almost every function provides two versions:
  1. Error code version: add `std::error_code&` as the last parameter to return filesystem error (`noexcept` if only filesystem error is possible);
  2. Exception version: throw `stdfs::filesystem_error` to represent error.

  - Reason: filesystem operations can easily incur TOC/TOU problem, so pre-check cannot prevent error.
    - Thus, it’s hard to predict whether error occurrence is in hot path or not, making exceptions sometimes not proper.

Overview

- For example, if we want to write a “chmod when it exists”.
    - So pseudocode can be:



```
function MyCHMOD(path) -> bool:
    if not exists(path) then return false
    chmod(path, WRITE_PERMISSION)
    return true
end
```

- But filesystem is **cross-process**, so the events can be:
    - We check that file indeed exists (TOC);
    - Another process removes this file;
    - We change permission of the file, which doesn’t exist and causes error (TOU).
- When multiple processes access the same filesystem object (i.e. *race*), the specific behavior is implementation-defined.

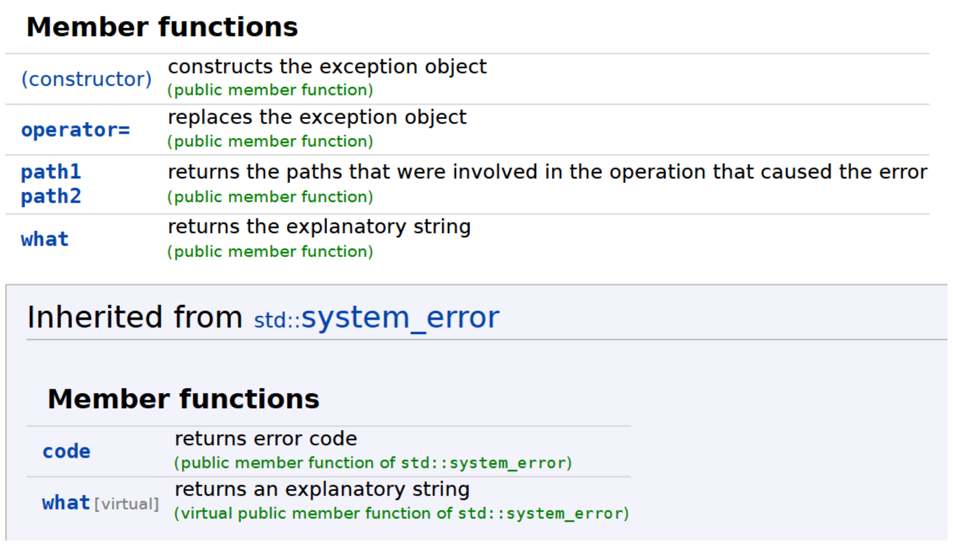
- Even worse, attackers can easily leverage TOC/TOU.
  - A well-known one is `stdfs::remove_all` (also in Rust!), which removes all files under the directory recursively but skip removing files in "symbolic links".
    - We'll cover symbolic links later; but generally it means an object that refers to another directory.
    - Such skip can prevent users from accidentally removing files in other folders.
  - And all three standard libraries initially implement it as:
    1. Check whether the object is a symbolic link;
    2. If it is not, recursively remove its files.
  - Say hackers want to remove `sensitive/` but they don't have permission; but current system runs a privileged program with `stdfs::remove_all`, which periodically removes `recyclebin/` that don't add permission.
    1. Hackers create a directory called `temp` in `recyclebin/` first;
    2. `remove_all` checks that it's not a symbolic link;
    3. Hackers delete `temp` and create a symbolic link `temp` to `sensitive/`.
    4. `remove_all` removes all files in `temp`, i.e. all files in `sensitive`. Hackers win!

Overview

- Similarly, user code can become vulnerable for TOC/TOU…
- Core problem: "path name" is a mutable property; a more robust way to always refer to the same filesystem object should be some handle.
  - Which is proposed in [P1883](#) (i.e. [the low-level file i/o library](#));
  - However, this can make filesystem APIs very obscure to use.

-------- Back to standard ---------

- APIs of `stdfs::filesystem_error` are quite simple:
  - We don't dig into `std::error_code` here; see our homework for details.



Member functions

(constructor)	constructs the exception object (public member function)
`operator=`	replaces the exception object (public member function)
`path1` `path2`	returns the paths that were involved in the operation that caused the error (public member function)
`what`	returns the explanatory string (public member function)

Inherited from std::system_error

Member functions

`code`	returns error code (public member function of `std::system_error`)
`what` [virtual]	returns an explanatory string (virtual public member function of `std::system_error`)

Supplementary

- File system
  - Path operations
  - File system operations
    - Overview
    - File status query and directory iteration
    - Modification operations

File status

- A file has the following attributes:
  - Name, as used in path;
  - Type;
  - Permission;
  - Size;
  - Last modification time.

  } C++ uses POSIX conventions, but allows slight customization for different systems (e.g. Windows).

- And POSIX regulates the following file types (as reflected in `stdfs::file_type`):
  - Implementations are allowed to add new types (e.g. `junction` in MS-STL).

```
enum class file_type {
    none = /* unspecified */,
    not_found = /* unspecified */,
    regular = /* unspecified */,
    directory = /* unspecified */,
    symlink = /* unspecified */,
    block = /* unspecified */,
    character = /* unspecified */,
    fifo = /* unspecified */,
    socket = /* unspecified */,
    unknown = /* unspecified */,
    /* implementation-defined */
};
```

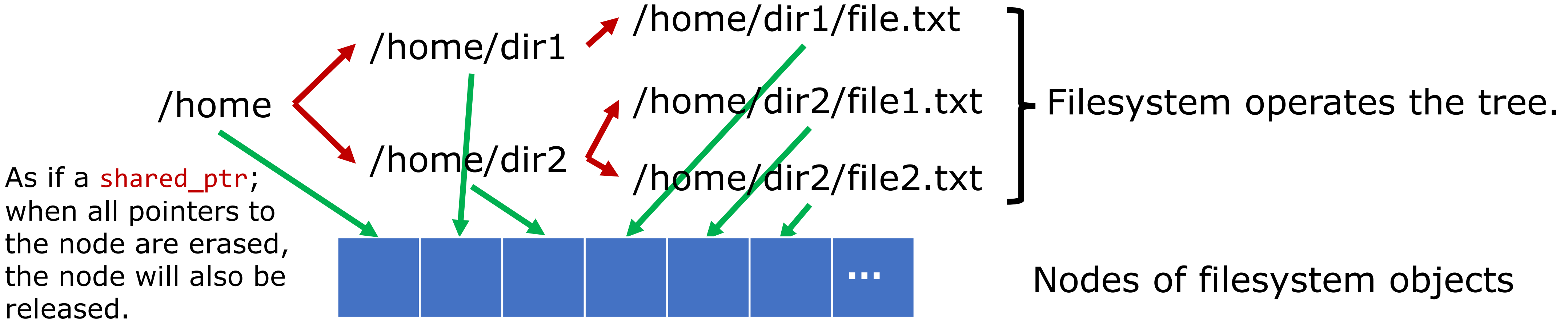
File types

Enumerator	Meaning
`none`	indicates that the file status has not been evaluated yet, or an error occurred when evaluating it
`not_found`	indicates that the file was not found (this is not considered an error)
`regular`	a regular file
`directory`	a directory
`symlink`	a symbolic link
`block`	a block special file
`character`	a character special file
`fifo`	a FIFO (also known as pipe) file
`socket`	a socket file
`unknown`	the file exists but its type could not be determined

What are these types?

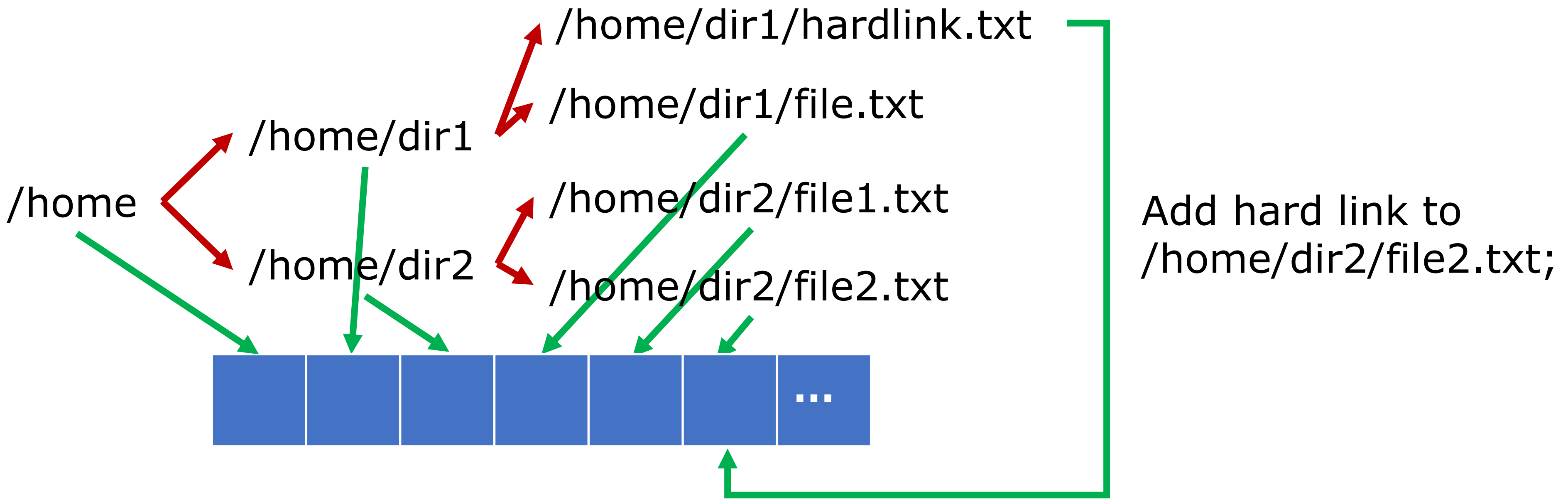
Link

- There are many kinds of links in the filesystem.
  - Essentially, links are just objects that *redirect* to other objects;
  - Different links are just redirection at different levels.
- A simple illusion of how filesystem handles objects:


/home
/home/dir1
/home/dir2
/home/dir1/file.txt
/home/dir2/file1.txt
/home/dir2/file2.txt
Filesystem operates the tree.
…
Nodes of filesystem objects
As if a shared_ptr; when all pointers to the node are erased, the node will also be released.

Hard Link

1. Hard link: as if adding reference count to underlying data nodes.
    - Filesystem cannot distinguish a hard link with the original object (so there is no type called “hard link”!).
    - When you delete `/home/dir2/file2.txt`, the content isn’t really deleted; `/home/dir1/hardlink.txt` still keeps the node.
    - Hard link count of an object can be checked by `stdfs::hard_link_count(p)`.



Hard Link

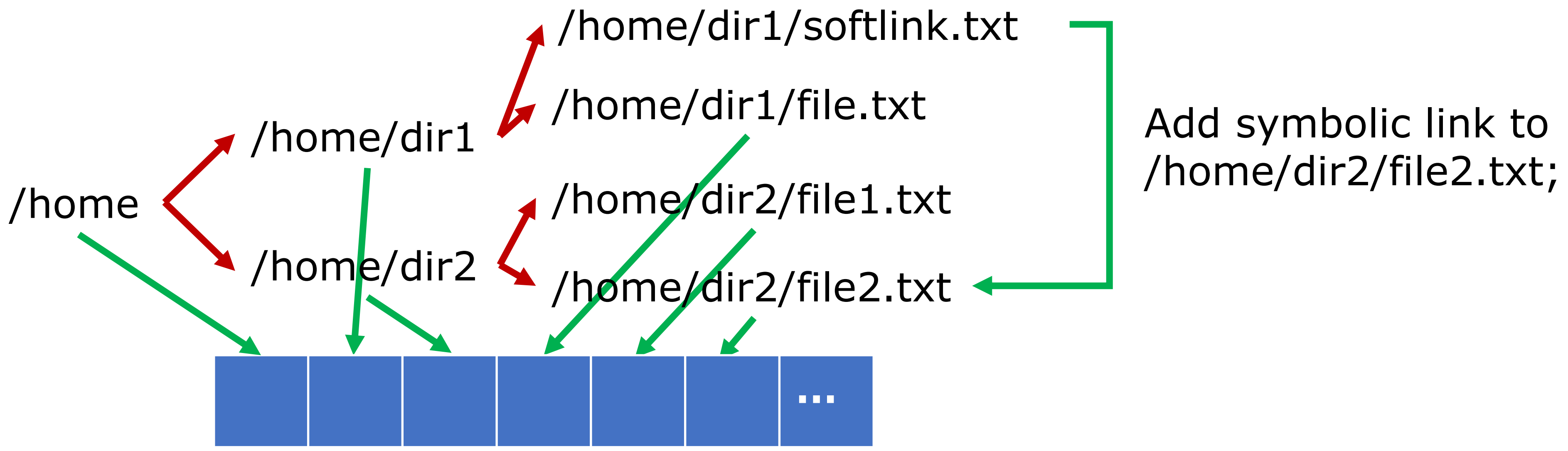
- Most of the filesystems don’t allow users to create hard link to directories easily.
  - Core reason: filesystem is usually assumed to form a **tree**.
    - However, creating hard link of some directory in its descendent will make a cycle in the graph, making it not a tree.
    - Some system programs may traverse the filesystem without special check; a circle will cause infinite loop.
  - Linux: `ln hd.txt a.txt`; do not allow hard link to directory.
  - MacOS: `ln hd.txt a.txt`; no native command for directory but can use POSIX `link()`;
    - MacOS `link` implementation checks whether new hard link to directory causes cycle; allow if not (but possibly need root privilege).
  - Windows: `mklink /H hd.txt a.txt`; do not allow hard link to directory.

Hard Link

- Note 1: `hard_link_count` for directory is implementation-defined.
  - Unix: "." and ".." are all seen as hard link; so a new directory will have `hard_link_count` as `2` and `++hard_link_count` of its parent directory.
  - Windows: always return `1`.
  - Return type: `uintmax_t`. The non-throwing overload returns `static_cast<uintmax_t>(-1)` on errors.
- Note 2: hard link has some other restrictions:
  1. Some filesystems don't support hard link (notably FAT file system; most of USB flash drives (U盘) use it).
  2. Hard link is not cross-filesystem since different filesystems may have different data structures. Hard link exists only in the same filesystem.
     - Particularly on Windows, they must be in the same volume (notably drive).
  3. Some filesystems may have limits for hard links per file.

Symbolic Link

2. Symbolic link (soft link): as if pointing to filesystem entry.
    - Or, you can think it as a `weak_ptr` to the underlying node.
    - When the real entry `/home/dir2/file2.txt` is deleted, its content will also be deleted though symbolic link `/home/dir1/softlink.txt` exists.
        - Any redirection operation of the soft link later (e.g. file I/O) will fail.



Filesystem functionality can be checked in MS Doc.
Notice that UDF is widely used in optical discs (光盘).

Symbolic Link

- Filesystem treats it as a distinct file type, but most native APIs will automatically redirect it to the real entry.
  - That is, users (instead of filesystem!) cannot distinguish symbolic link with a normal object, unless they use a few special functions.
- Therefore, filesystem allows symbolic link to directory.
- Unix: `ln –s sym a`;
- Windows: `mklink /D sym a`; **Particularly**, creating symbolic link on Windows requires root privilege.

- Note: Some filesystems don’t support soft link (notably FAT).
  - But as it points to filesystem entry, it’s cross-filesystem.
  - Therefore, it’s okay to create soft link to FAT on NTFS since NTFS supports it (but not vice versa).

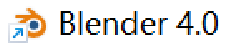
Junction

3. For non-root privilege, Windows allows link to directory by a new type called “junction”.
    - `mklink /J junc a`.
    - Like symbolic link, when the real node is deleted, junction still exists but becomes invalid.
    - Though junction is cross-filesystem, it only allows to link directory in local computer.
        - i.e. no support for network path and UNC.

- Most of `stdfs` APIs will follow all links above and resolve to the final real entry.
    - For example, `stdfs::exists`; `stdfs::absolute`; and thus `stdfs::canonical`; `stdfs::relative`; `stdfs::proximate`.

Check this blog for link details on Windows.

Shortcut



- BTW, there also exists “shortcut” (快捷方式) on Windows, which is actually a “user-space symbolic link”.
    - It’s a file with “.lnk” extension; filesystem just treats it as a normal file (and it is!).
    - Its content is the real path of the filesystem object, how it should be launched, etc.
    - So guess: when you click shortcut, how are you redirected to the real entry?
        - Yes, Windows Explorer (文件资源管理器) UI helps you!
        - The Explorer program automatically redirects your click as if you click the real entry; other programs that don’t do special process **cannot redirect**.
        - By contrast, links will be redirected by filesystem automatically even if programs don’t do special process.

Character and Block File

- When you plug in a USB (mouse, disk, etc.), how can OS identify the way to interact with the connected device?
  - By **drive** program, if you've learnt *Embedded System*.
  - When you want to support a new external device, you can write your own drive with a bunch of functions (e.g. `open`, `write`, etc.) to make OS know how to interact with it.
  - And finally, your drive needs to expose a *device file* to make users able to `open`, `write`, etc. by OS APIs.
- Character file and block file are just such device files.
  - When you read from / write to them, it's as if you get data from / input data to your drive and thus successfully interact with your device.
  - Character file means "I/O is passed directly without buffering"; e.g. terminal: `/dev/tty`
  - Block file means "Buffering I/O and pass until proper time". e.g. SSD: `/dev/sda`

Windows also supports character file (e.g. CON) but MS-STL always returns `false` for these two types since it's expensive to detect such file type.

FIFO

- FIFO is just named pipe.
  - In Linux, we can pass output of a program as input to another program by `|` (e.g. `cat data.txt | grep "info"`).
    - Pipe allows the data to keep in memory instead of writing into real files (and thus real storage) for inter-process communication.
  - Sometimes, it's inconvenient to create anonymous pipes and pass them to other programs.
    - Instead, you can create a named pipe in the filesystem, with some programs writing and other programs reading it (again, happen in memory without really going down to real storage).
    - E.g. in command line: `mkfifo test; cat data.txt > test &; grep "info" test`; or by `mkfifo` in POSIX API.
  - So `test` just has fifo type, which is specially interpreted in POSIX system.

Socket

```
// We omit error code handling here.
sockaddr_in sockAddr{};
sockAddr.sin_family = AF_INET;
::inet_pton(AF_INET, argv[1], &sockAddr.sin_addr);
sockAddr.sin_port = ::htons(port);
listenfd = ::socket(AF_INET, SOCK_STREAM, 0);
::bind(listenfd, (sockaddr*)(&sockAddr), sizeof(sockaddr_in));
// For server: listen and accept
// For client: connect by sockAddr
```

Fill network location.

- Similarly, sometimes it's inconvenient to use network socket to communicate across processes.
  - Instead, you can create a socket file (a.k.a. Unix domain socket, UDS).
  - We know how to create a network server socket in Linux.
  - For UDS, just change flag and location to local socket file:

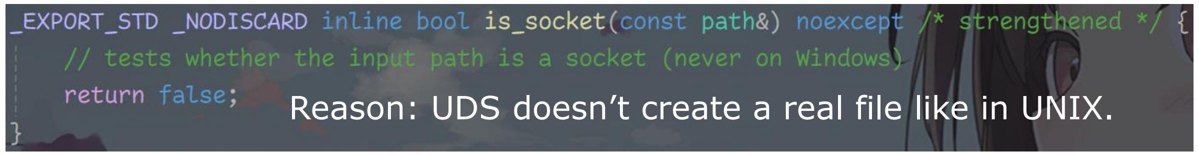
```
sockaddr_un sockAddr;
sockAddr.sun_family = AF_UNIX;
// Specify file path, less than 108 bytes in Linux.
std::strcpy(sockAddr.sun_path, "sockfile.sock");
listenfd = ::socket(AF_UNIX, SOCK_STREAM, 0);
::bind(listenfd, (sockaddr*)(&sockAddr), sizeof(sockaddr_un));
// For server: listen and accept
// For client: connect by sockAddr
```

After bind, process will create sockfile.sock in CWD, which has file type as socket.

- However, bind requires the address to be not already used, so the socket file shouldn't exist before bind (thus, it's usually a temporary file).

FIFO and Socket

- Note 1: difference between fifo and socket:
    1. Socket file can use `send` or `recv`, as network socket does; fifo can only use `write` or `read`.
    2. Socket can use datagram instead of byte stream (by `SOCK_DGRAM`), while fifo only uses byte stream.
    3. Socket is bidirectional, i.e. server and client can send or recv the other freely; fifo is unidirectional, i.e. sender always send and receiver always recv.
    4. Typically, fifo is used for two users, while socket can be used for multiple clients.
- Note 2: Windows also supports fifo (since Windows 2000 professional) and UDS (since Windows 10 17063 (2017/12)).
    - However, MS-STL always returns `false` for `is_fifo` and `is_socket`.



```
_EXPORT_STD _NODISCARD inline bool is_socket(const path&) noexcept /* strengthened */ {
    // tests whether the input path is a socket (never on Windows)
    return false;
}
```

Reason: UDS doesn't create a real file like in UNIX.

Permission

- For permission, there are three levels in POSIX: user / group / all.
  - Each level can have read / write / execution right.
  - In Linux bash, we can change permission by `chmod`; e.g. chmod 764 means that owner can read, write or execute, current group can read and write, while all others can only read.
- As each level can be expressed in three bits, you can use octal number to form the permission in C++.
  - E.g. literal 0764, since leading 0 means octal literal.
- However, Windows permission system (Active Control List, ACL) is not compatible with POSIX. It thus uses naïve DOS permission:
  1. All users can read, write and execute the file (777);
  2. All users can read and execute the file (555).

Permission

- In POSIX (again, not in Windows), there also exist three special permissions: `setuid`, `setgid` and `sticky bit`.
  - `setuid`: when executing the binary, use the owner's user id (so that it can execute code that only owner can execute).
    - Typical example: `/bin/passwd`, which will write `/etc/shadow`. Users can change password by `/bin/passwd`, but don't have privilege to write `/etc/shadow` (otherwise they can change anyone's password!).
      - Thus, `/bin/passwd` has permission `setuid`, i.e. it has root privilege to write `/etc/shadow` when executing `/bin/passwd`.
      - Since `/bin/passwd` controls what it accesses, it's still safe.
  - `setgid`: when executing the binary, use the owner's group id.
    - Typical example: `/usr/bin/wall`, which needs group privilege to write to others' `tty`.

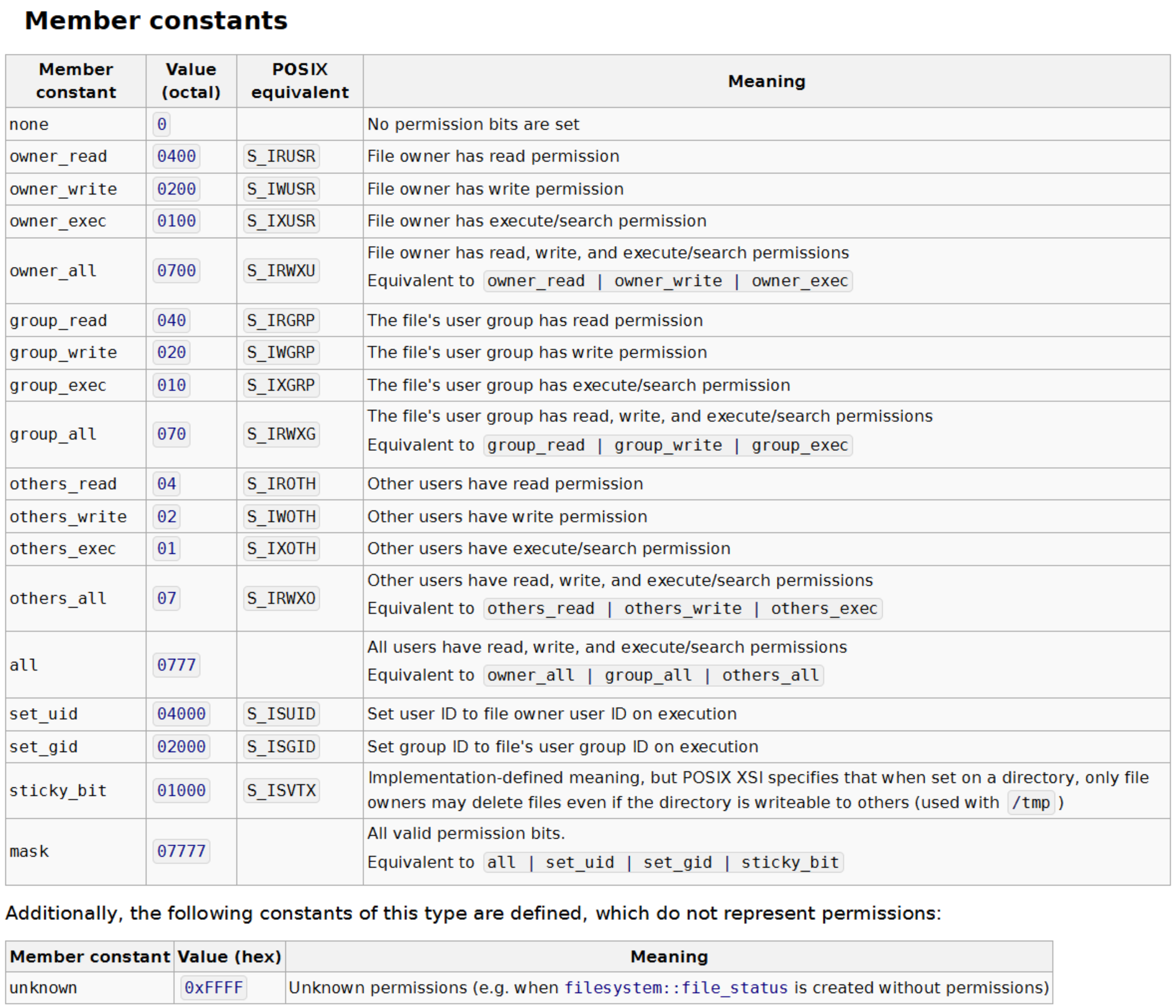
Permission

- `sticky bit`: typically means "when applying to a directory, only file owner can delete his/her file even if he/she can write the directory".
  - Typical example: `/tmp`; all users have write permission to `/tmp` to keep temporary files. They may delete their cache files later.
    - However, this then allows the user to delete others' file, which may randomly crash others' applications.
  - Solution 1: every user protects his/her file in `/tmp` by e.g. `chmod 007`.
    - Well, that's too troublesome…
  - Solution 2: apply sticky bit to `/tmp`!
    - Then only file owner can delete his/her file.

- C++ defines these permission specifications with enumeration.

Permission

- Permission values are defined as `enum class perms` with these enumerators:
  - With overloaded bit operators.



Member constants

Member constant	Value (octal)	POSIX equivalent	Meaning
`none`	`0`		No permission bits are set
`owner_read`	`0400`	`S_IRUSR`	File owner has read permission
`owner_write`	`0200`	`S_IWUSR`	File owner has write permission
`owner_exec`	`0100`	`S_IXUSR`	File owner has execute/search permission
`owner_all`	`0700`	`S_IRWXU`	File owner has read, write, and execute/search permissions Equivalent to `owner_read | owner_write | owner_exec`
`group_read`	`040`	`S_IRGRP`	The file's user group has read permission
`group_write`	`020`	`S_IWGRP`	The file's user group has write permission
`group_exec`	`010`	`S_IXGRP`	The file's user group has execute/search permission
`group_all`	`070`	`S_IRWXG`	The file's user group has read, write, and execute/search permissions Equivalent to `group_read | group_write | group_exec`
`others_read`	`04`	`S_IROTH`	Other users have read permission
`others_write`	`02`	`S_IWOTH`	Other users have write permission
`others_exec`	`01`	`S_IXOTH`	Other users have execute/search permission
`others_all`	`07`	`S_IRWXO`	Other users have read, write, and execute/search permissions Equivalent to `others_read | others_write | others_exec`
`all`	`0777`		All users have read, write, and execute/search permissions Equivalent to `owner_all | group_all | others_all`
`set_uid`	`04000`	`S_ISUID`	Set user ID to file owner user ID on execution
`set_gid`	`02000`	`S_ISGID`	Set group ID to file's user group ID on execution
`sticky_bit`	`01000`	`S_ISVTX`	Implementation-defined meaning, but POSIX XSI specifies that when set on a directory, only file owners may delete files even if the directory is writeable to others (used with `/tmp`)
`mask`	`07777`		All valid permission bits. Equivalent to `all | set_uid | set_gid | sticky_bit`

Additionally, the following constants of this type are defined, which do not represent permissions:

Member constant	Value (hex)	Meaning
`unknown`	`0xFFFF`	Unknown permissions (e.g. when `filesystem::file_status` is created without permissions)

`perms` satisfies the requirements of *BitmaskType* (which means the bitwise operators `operator&`, `operator|`, `operator^`, `operator~`, `operator&=`, `operator|=`, and `operator^=` are defined for this type). `none` represents the empty bitmask; every other enumerator represents a distinct bitmask element.