

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. e.g. [CON on Windows](#).

(1.4) — Filenames that have special meaning.

(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/	foo\
//host/./foo.txt	//host/foo.txt	\\host\foo.txt (Due to UNC path)
./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.

```
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 **std::filesystem::path** with three enumerators:

Name	Explanation
<code>native_format</code>	native pathname format
<code>generic_format</code>	generic pathname format
<code>auto_format</code>	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:\璇瞭獬.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;    Default is just an empty string.  
path( const path& p );  
path( path&& p ) noexcept;  
path( string_type&& source, format fmt = auto_format );
```

# 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 with allocator:

---

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

---

(1)

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

(1)

- So to get Generic format + Native encoding, just use `.generic_string<std::filesystem::path::value_type>()`.
- Finally, you can also check the preferred separator by `static constexpr std::filesystem::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`

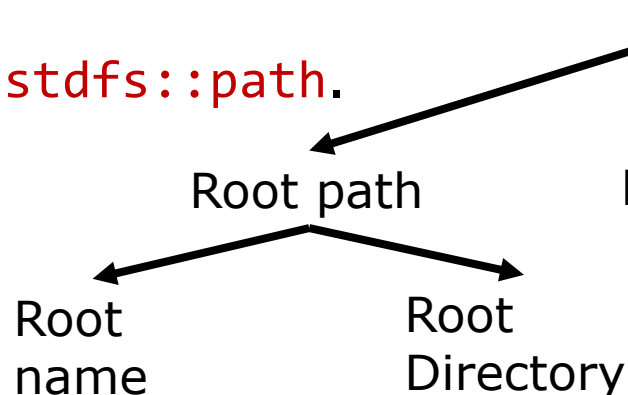
`relative_path`

`parent_path`

`filename`

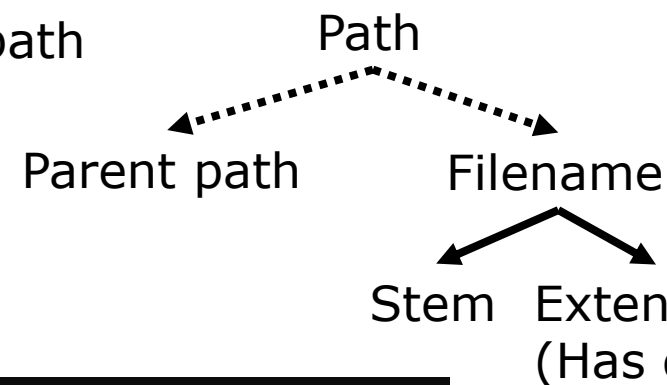
`stem`

`extension`



Relative path

→ : Concatenation  
.....→ : Appendage



```
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:

*Normalization* of a generic format pathname means:

1. If the path is empty, stop.

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

D:/. .\sub\.\path\.. \file.txt 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*.

[Note 4: The generic pathname grammar defines *directory-separator* as one or more slashes and *preferred-separators*. — end note]

D:/. .\sub\.\path\.. \file.txt 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\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]

7. If the last filename is dot-dot, remove any trailing *directory-separator*.
7. is applied to e.g. `.. \.. \.`

8. If the path is empty, add a dot.
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

```
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.

- 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";
```

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

```
"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.

```
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.

# Path relativization<sup>1</sup>

- 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.

# Path relativization

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

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  $[a_1, a_2) = [\text{test.txt}]$ ,  $[b_1, b_2) = [\text{test2}, \text{test.txt}, \dots]$ .
- 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"};          n = 1, m = 0
std::cout << p.lexically_relative("a/b/..") << "\n";
```

```
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_normalize()` to `*this`, `base`, or both. — end note]

- The second way is filesystem-dependent `std::filesystem::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 `std::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 );
```

# Path composition

```
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 );
```

- 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: 

```
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 `((std::fs::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.
  - $p / "a" / "b" \Leftrightarrow ((p / "a") / "b")$ , while
  - $p /= "a" /= "b" \Leftrightarrow (p /= ("a" /= "b"))$ .
    - And it's illegal for two string literals to  $/=$ .
  - You have to add a bunch of brackets; that's why we use  $((std::path\{a\} += b) += c)...$
- 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:

```
#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.



# Path iteration

- So instead, the standard regulates that:

<sup>2</sup> 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

Return reference  
to `*this`.

Modifiers
<code>clear</code>
<code>make_preferred</code>
<code>remove_filename</code>
<code>replace_filename</code>
<code>replace_extension</code>
<code>swap</code>

- 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"

```
// 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;
}
```

# Path

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.

- And finally some simple utilities, just list here. e.g. "D:/Test"  
• Comparable (by `<=>/==` or `.compare`); compare **components**. `==` "D://Test"  
• 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. `"C:\\foo\\bar"`  
• Recap: `quoted` will escape the quote and the escape, so `\` is escaped to `\\`.
- Formattable **since C++26**.

## 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
```

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

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

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 `std::filesystem::equivalent` to compare, which needs file system call to check equality of two paths. Covered later.