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

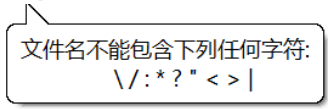
- File system
  - Path operations
    - Overview
    - std::filesystem::path
  - File system operations

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: 

[*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/`	`foo\`
`//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`
`././`	`.`	`.`

POSIX may or may not respect //.

(Due to UNC path)

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)
const string_type& native() const noexcept;        (2)
operator string_type() const;                      (3)
```

1) Equivalent to `native().c_str()`.
2) Returns the native-format representation of the pathname by reference.
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 `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:

```cpp
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;
```
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;
std::u32string generic_u32string() const;
```
(2) (since C++17)

```
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>
    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<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`

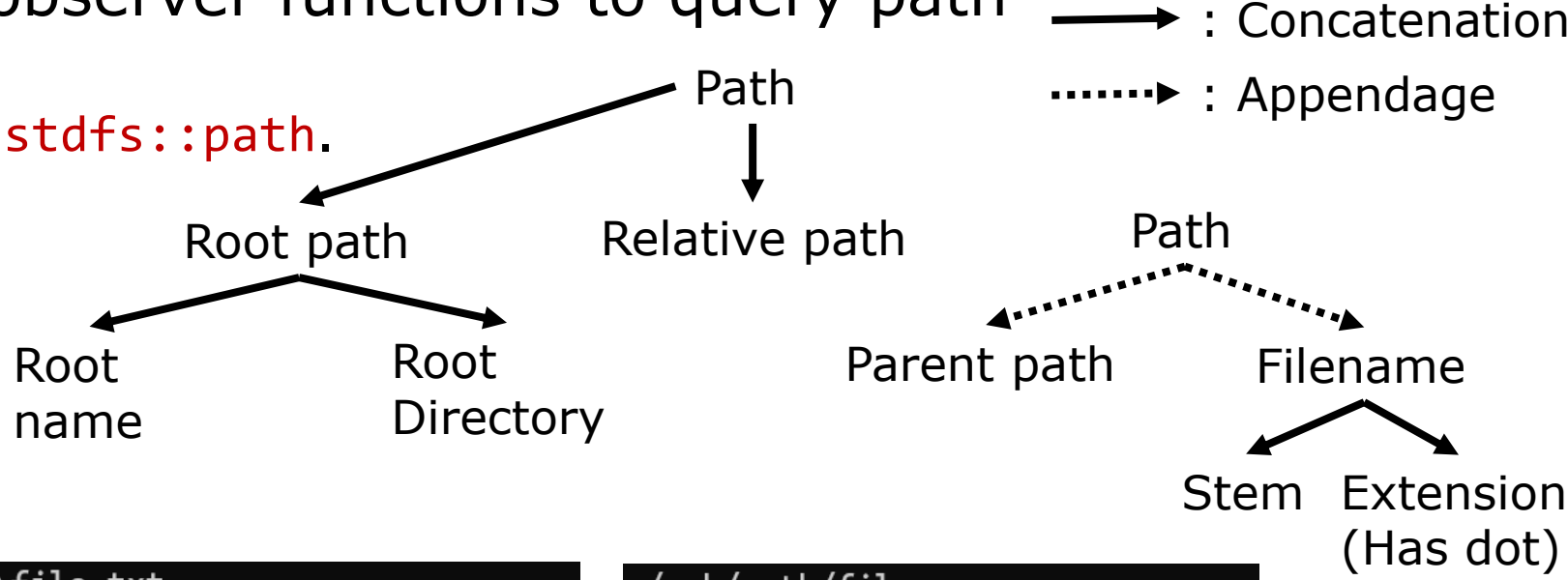
`relative_path`

`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:

Normalization of a generic format pathname means:

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

1. If the path is empty, stop.

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

```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.

```
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.

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 $[a_1, a_2) = [test.txt], [b_1, b_2) = [test2, test.txt, ..]$.
- 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"

```
// 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.
  - Comparable (by `<=>/==` or `.compare`); compare **components**. e.g. `"D:/Test" == "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.
    - Recap: `quoted` will escape the quote and the escape, so `\` is escaped to `\\`.
  - Formattable **since C++26**.

```
"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

```
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 */
};
```

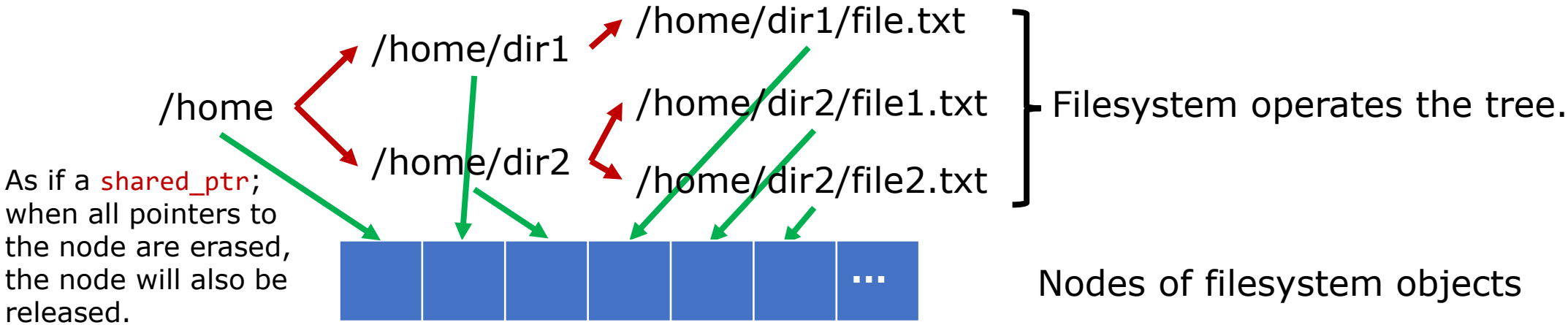
File types

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?

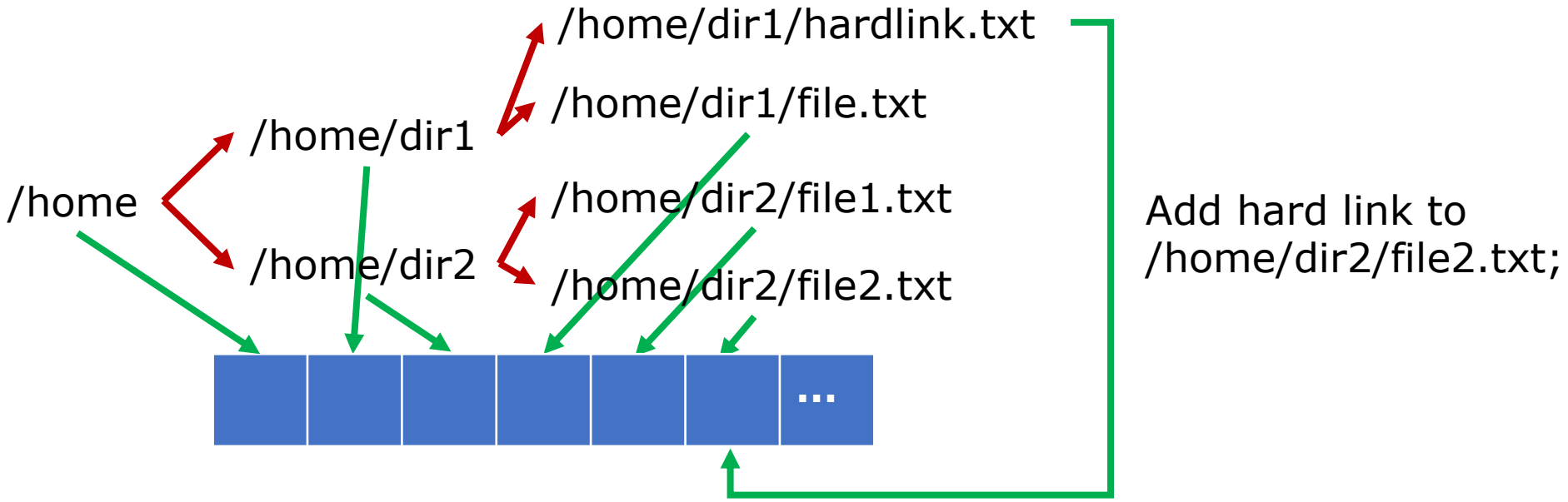
Link

- 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:



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).

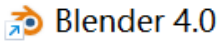
Junction

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:

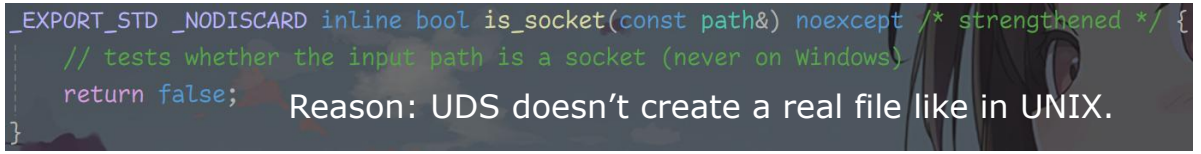
```
sockaddr_un sockAddr;
sockAddr.sun_family = AF_UNIX;
// Specify file path, less than 108 bytes in Linux.
std::strcpy(sockAddr.sun_path, "sockfile.sock");
listenfd = ::socket(AF_UNIX, SOCK_STREAM, 0);
::bind(listenfd, (sockaddr*)(&sockAddr), sizeof(sockaddr_un));
// For server: listen and accept
//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



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 `0`764, 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.

File status query

- To test file type, you can use these APIs:
  - Params: `(const path& p[, error_code])`.

is_block_file (C++17)	checks whether the given path refers to block device (function)
is_character_file (C++17)	checks whether the given path refers to a character device (function)
is_directory (C++17)	checks whether the given path refers to a directory (function)
is_fifo (C++17)	checks whether the given path refers to a named pipe (function)
is_other (C++17)	checks whether the argument refers to an *other* file (function)
is_regular_file (C++17)	checks whether the argument refers to a regular file (function)
is_socket (C++17)	checks whether the argument refers to a named IPC socket (function)
is_symlink (C++17)	checks whether the argument refers to a symbolic link (function)
status_known (C++17)	checks whether file status is known (function)

Other file: file exists but isn't regular file / directory / symlink.

File status query

- However, a more efficient way is to query type once and cache it, so every test will be a cheap and plain integer comparison.
  - Just by `stdfs::status`!
  - For example:

```
switch (fs::path p{argv[1]}; status(p).type()) {
  case fs::file_type::not_found:
    std::cout << "path \"" << p.string() << "\" does not exist\n";
    break;
  case fs::file_type::regular:
    std::cout << '"' << p.string() << "\" exists with "
              << file_size(p) << " bytes\n";
    break;
  case fs::file_type::directory:
    std::cout << '"' << p.string() << "\" is a directory containing:\n";
    for (const auto& e : std::filesystem::directory_iterator{p}) {
      std::cout << "  " << e.path().string() << '\n';
    }
    break;
  default:
    std::cout << '"' << p.string() << "\" is a special file\n";
    break;
}
```

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

File status query

- `status` essentially returns `file_status`, with `.type()->file_type` and `.permissions()->perms`:
  - Another example:

```
std::string asString(const std::filesystem::perms& pm)
{
  using perms = std::filesystem::perms;
  std::string s;
  s.resize(9);
  s[0] = (pm & perms::owner_read)   != perms::none ? 'r' : '-';
  s[1] = (pm & perms::owner_write)  != perms::none ? 'w' : '-';
  s[2] = (pm & perms::owner_exec)   != perms::none ? 'x' : '-';
  s[3] = (pm & perms::group_read)   != perms::none ? 'r' : '-';
  s[4] = (pm & perms::group_write)  != perms::none ? 'w' : '-';
  s[5] = (pm & perms::group_exec)   != perms::none ? 'x' : '-';
  s[6] = (pm & perms::others_read)  != perms::none ? 'r' : '-';
  s[7] = (pm & perms::others_write) != perms::none ? 'w' : '-';
  s[8] = (pm & perms::others_exec)  != perms::none ? 'x' : '-';
  return s;
}
```

```
std::cout << "permissions: " << asString(status(mypath).permissions())
          << '\n';
```

Member functions

(constructor)	constructs a file_status object (public member function)
operator=	assigns contents (public member function)
(destructor)	implicit destructor (public member function)
type	gets or sets the type of the file (public member function)
permissions	gets or sets the permissions of the file (public member function)

Non-member functions

operator== (C++20)	compares two file_status objects (function)

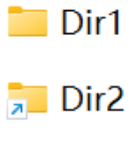
Note: `is_xxx` (e.g. `is_directory`) can also accept `file_status`, which is `noexcept` since it's just plain integer comparison.

File status query

- However, when you try to apply `status` to symbolic links:

```cpp
stdfs::path p{ R"(D:\Work\C++\ExpDir\Dir2)" };
auto status = stdfs::status(p);
std::cout << stdfs::is_directory(status) << " "
          << stdfs::is_symlink(status) << "\n";
```

```
1 0
```



  - Oops, the type is not `symlink`, but directory!
  - Reason: filesystem will automatically follow symbolic links in its APIs.
- To query the shallow status instead of following links, you should use `symlink_status`:

```cpp
stdfs::path p{ R"(D:\Work\C++\ExpDir\Dir2)" };
auto status = stdfs::symlink_status(p);
std::cout << stdfs::is_directory(status) << " "
          << (status.type() == stdfs::file_type::junction) << "\n";
```

```
0 1
```

Note 1: `is_symlink(p[, ec])` internally uses `symlink_status`.

Note 2, again: hard link cannot be distinguished.

Here we create `Dir2` as junction, which is listed as a new enumerator in MS-STL so **`is_symlink` is still `false`** (well, though it can be queried by `symlink_status`).

File status query

MS-STL: return 0;
libc++ & libstdc++: throw error ("is a directory");

- The last two status are size and last modification time, which can also be queried by path:
  - `stdfs::file_size(p[, ec]) -> std::uintmax_t;`
    - The result of directory is implementation-defined.
  - `stdfs::last_write_time(p[, ec]) -> stdfs::file_time_type;`
    - The specific utilities in `std::chrono` will be introduced later.
- Note: except for type, you can change any status directly.
  - `stdfs::resize_file(p, newSize[, ec]);`
    - If `newSize` is greater than current size, fill new space with 0; otherwise truncate the file.
    - Note: on filesystems that support sparse files, greater size of file may not decrease available storage in filesystem as long as "new space with 0" is not really written.

The non-throwing overload returns `static_cast<std::uintmax_t>(-1)` on errors.

File status query

- `stdfs::last_write_time(p, newTime[, ec]);`
  - But the grain of filesystem may be not same as program time, so there can be rounding error.
- `stdfs::permissions(p, newPerm[, opt[, ec]]);`
  - `opt` is specified as `enum class stdfs::perm_options`, also with bit operators.

At most one of add, remove, replace may be present, otherwise the behavior of the permissions function is undefined.

Enumerator	Meaning
replace	permissions will be completely replaced by the argument to `permissions()` (default behavior)
add	permissions will be replaced by the bitwise OR of the argument and the current permissions
remove	permissions will be replaced by the bitwise AND of the negated argument and current permissions
nofollow	permissions will be changed on the symlink itself, rather than on the file it resolves to

  - Default `opt` is `replace`.

Directory entry

- Umm, so is there a way to query all attributes once and just get them from cache?
  - By `stdfs::directory_entry`!
  - For example:

```cpp
auto entry = stdfs::directory_entry{ R"(D:\Work\C++\ExpDir\Dir2)" };
{
    std::cout << entry
              << " Last modification time = " << entry.last_write_time()
              << ", Hard link count = " << entry.hard_link_count() << "\n";

    if (entry.is_regular_file())
        std::cout << "File size = " << entry.file_size();
    else
    {
        std::cout << "Directory; is symlink : " << entry.is_symlink()
                  << " is junction: "
                  << (entry.symlink_status().type() ==
                      stdfs::file_type::junction);
    }
    std::cout << "\n----------" << std::endl;
```

```
"D:\\Work\\C++\\ExpDir\\Dir2" Last modification time = 2026-02-04 10:29:15.7225417, Hard link count = 1
Directory; is symlink : 0 is junction: 1
----------
```

Construct from path, and ctor will query all attributes if they exist.

Note: in libstdc++, you need to `#include<chrono>` to make time outputable (C++20, covered later).

Directory entry

- It has almost all query methods we covered before:
  - And some utilities (**only** applied to the underlying path!):
    - Comparable
    - `operator<<`

`.is_symlink` uses `symlink_status` while others use `status`. So `.is_symlink && .is_directory` can be `true`.

Observers

`path` `operator const path&`	returns the path the entry refers to (public member function)
`exists`	checks whether directory entry refers to existing file system object (public member function)
`is_block_file`	checks whether the directory entry refers to block device (public member function)
`is_character_file`	checks whether the directory entry refers to a character device (public member function)
`is_directory`	checks whether the directory entry refers to a directory (public member function)
`is_fifo`	checks whether the directory entry refers to a named pipe (public member function)
`is_other`	checks whether the directory entry refers to an *other* file (public member function)
`is_regular_file`	checks whether the directory entry refers to a regular file (public member function)
`is_socket`	checks whether the directory entry refers to a named IPC socket (public member function)
`is_symlink`	checks whether the directory entry refers to a symbolic link (public member function)
`file_size`	returns the size of the file to which the directory entry refers (public member function)
`hard_link_count`	returns the number of hard links referring to the file to which the directory entry refers (public member function)
`last_write_time`	gets the time of the last data modification of the file to which the directory entry refers (public member function)
`status` `symlink_status`	status of the file designated by this directory entry; status of the file/symlink designated by this directory entry (public member function)

Directory entry

- And also some simple modifiers:

`operator=`	assigns contents (public member function)
`assign`	assigns contents (public member function)
`replace_filename`	sets the filename (public member function)
`refresh`	updates the cached file attributes (public member function)

i.e. `path.replace_filename(…)` and `.refresh()`.

i.e. Re-query attributes by filesystem calls.

  - Applying these methods doesn’t e.g. rename the underlying file in the filesystem, but just re-query!
- Actually, `directory_entry` is value type of `directory_iterator`, which can be used to iterate over a path.

Directory iteration

1. Besides `operator++`, the iterator also defines `.increment(ec)` to accept error code.
2. Essentially, `begin` returns its own copy and `end` returns default constructed iterator.

- It has `friend begin` and `friend end` and naturally forms a range, making it iterable directly:

```
for (const auto& entry :
    stdfs::directory_iterator{ R"(D:\Work\C++\ExpDir)" })
```

Path will include root (passed in ctor) as prefix.

```
"D:\\Work\\C++\\ExpDir\\Dir1" Last modification time = 2026-02-04 10:29:15.7225417, Hard link count = 1
Directory; is symlink : 0 is junction: 0
----------
"D:\\Work\\C++\\ExpDir\\Dir2" Last modification time = 2026-02-04 10:29:15.7225417, Hard link count = 1
Directory; is symlink : 0 is junction: 1
----------
"D:\\Work\\C++\\ExpDir\\test.txt" Last modification time = 2026-02-05 09:34:49.0577604, Hard link count = 1
File size = 5
----------
```

- You can also add `stdfs::directory_options` as the second param in ctor to control iterator behavior:

Again, it's bit-operatable scoped enumeration.

Constants

Enumerator	Meaning
none	(default) skip directory symlinks, "permission denied" is error
follow_directory_symlink	follow rather than skip directory symlinks
skip_permission_denied	skip directories that would otherwise result in "permission denied" errors

Directory iteration

- To iterate recursively in the directory, you can also use `recursive_directory_iterator`.
  - which will then use implementation-defined way (usually DFS) to traverse all entries.
  - For example:

```
for (const auto &entry : stdfs::recursive_directory_iterator{
        R"(D:\Work\C++\ExpDir)",
        stdfs::directory_options::follow_directory_symlink })
```

```
"D:\\Work\\C++\\ExpDir\\Dir1" Last modification time = 2026-02-04 10:29:15.7225417, Hard link count = 1
Directory; is symlink : 0 is junction: 0
----------
"D:\\Work\\C++\\ExpDir\\Dir1\\test2.txt" Last modification time = 2026-02-04 10:29:14.7068836, Hard link count = 1
File size = 0
----------
"D:\\Work\\C++\\ExpDir\\Dir2" Last modification time = 2026-02-04 10:29:15.7225417, Hard link count = 1
Directory; is symlink : 0 is junction: 1
----------
"D:\\Work\\C++\\ExpDir\\Dir2\\test2.txt" Last modification time = 2026-02-04 10:29:14.7068836, Hard link count = 1
File size = 0
----------
"D:\\Work\\C++\\ExpDir\\test.txt" Last modification time = 2026-02-05 09:34:49.0577604, Hard link count = 1
File size = 5
----------
```

`Dir2` is junction to `Dir1`, and we follow symlink, so essentially contents in `Dir1` is printed again.

Directory iteration

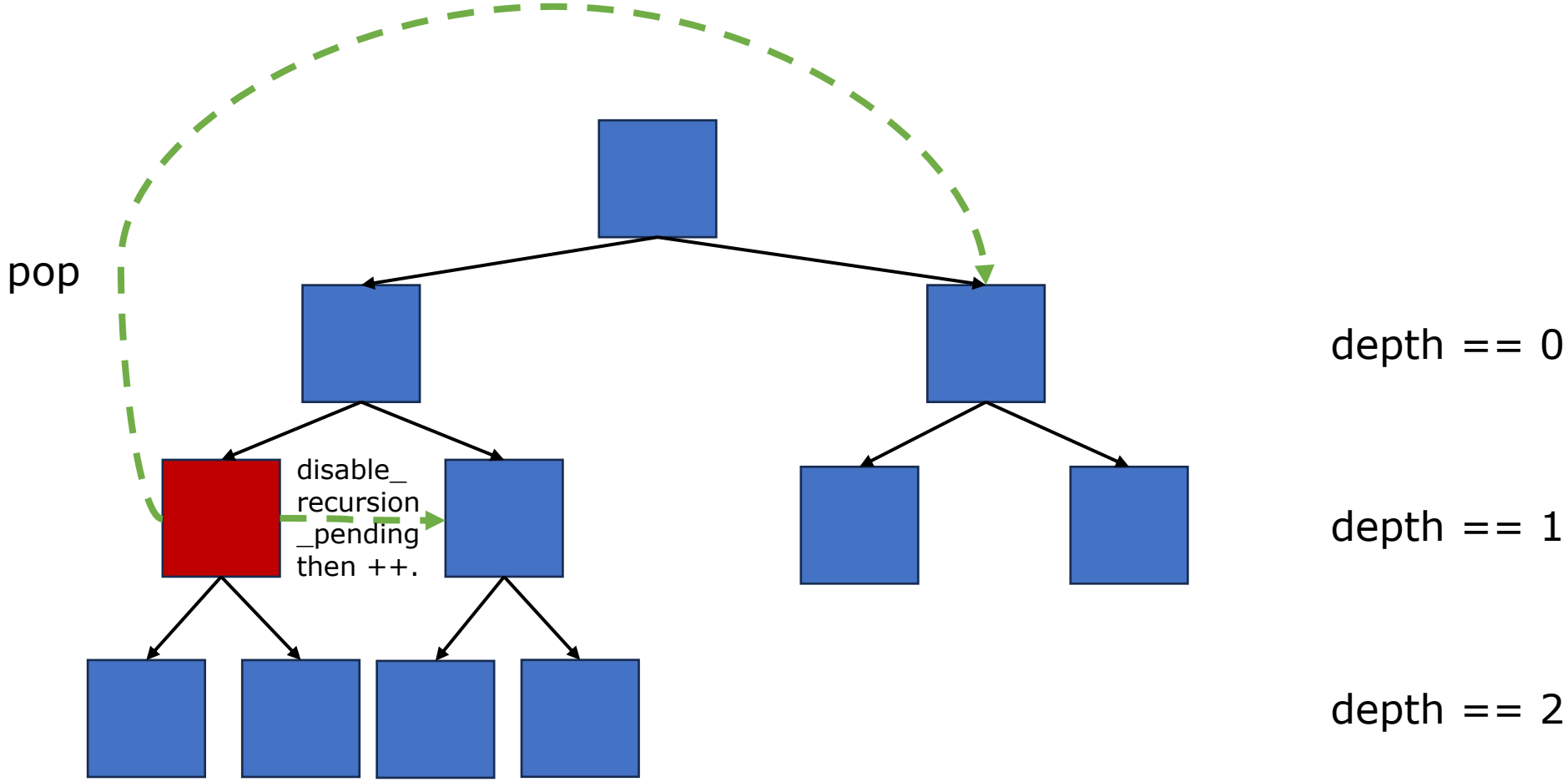
- The iterator has more member methods:

Observer:	`options ->directory_options`	returns the currently active options that affect the iteration (public member function)
	`depth ->int`	returns the current recursion depth (public member function) (Starting from 0)
	`recursion_pending ->bool`	checks whether the recursion is disabled for the current directory (public member function)
Modifier:	`pop`	moves the iterator one level up in the directory hierarchy (public member function)
	`disable_recursion_pending`	disables recursion until the next increment (public member function)

`pop([ec])`: discard following iterations in subtree and go to next entry in the last level. This operation will invalidate all previous copies; and when `.depth() == 0`, iterator will become `end()`.

`disable_recursion_pending`: when the current entry is directory, do not go into the directory for next `++`. `recursion_pending` will return `false` then. Increment will reset `recursion_pending` to `true`.

Directory iteration



Directory iteration

- Note 1: their `operator*` returns `const&`, so you cannot modify `directory_entry` without copying it.
- Note 2: they are input iterators.
  - Recap: input iterators are not forward iterators because they're one-pass; copying it and iterating the copy may lead to different results.
  - Here similarly, directory contents may be altered by other processes, so it's not multi-pass.
    - It's actually more complicated and **see our homework** for detailed discussion.
- Note 3: if new files and directories are added in directory `A` after creation of `(recursive_)directory_iterator{ "A" }`, it's unspecified whether they will be iterated or not.

Final Notes

- There exist some other minor read-only methods:
  1. Path operations:
     - `current_path() -> p` for CWD;
     - `temp_directory_path() -> p` (e.g. `/tmp` on Linux);
     - `equivalent(p1, p2) -> bool` for judging whether two paths are essentially same (following symlinks).
     - `read_symlink(p1) -> p2` for converting a symlink path to its target path (error if `p1` is not symlink).
  2. Filesystem operations:
     - `is_empty(p) -> bool`, return `true` if `p` is empty file or directory;
     - `space(p) -> space_info`, which determines space of mounted filesystem that `p` lies in.
       - On Linux, you can check filesystem of the path by `df -h`:
       - On Windows, each drive is seen as a mounted filesystem.

```
jiaming@jiaming-Legion:~/Self/test_2026$ df -h / /home /tmp /dev
Filesystem      Size  Used Avail Use% Mounted on
/dev/nvme0n1p7  182G  133G   40G  78% /
/dev/nvme0n1p7  182G  133G   40G  78% /
/dev/nvme0n1p7  182G  133G   40G  78% /
udev            7.8G     0  7.8G   0% /dev
```

Final Notes

- An example program from cppreference:

```
struct space_info {
    std::uintmax_t capacity;
    std::uintmax_t free;
    std::uintmax_t available;
};
```

capacity	total size of the filesystem, in bytes (public member object)
free	free space on the filesystem, in bytes (public member object)
available	free space available to a non-privileged process (may be equal or less than free) (public member object)

```
| Capacity      | Free          | Available     | Use% | Dir
| 8,298,082,304 | 8,298,082,304 | 8,298,082,304 | 0    | /dev/null
| 194,491,936,768 | 52,302,696,448 | 42,348,515,328 | 77 | /tmp
| 194,491,936,768 | 52,302,696,448 | 42,348,515,328 | 77 | /home
| 0             | 0             | 0             | 100  | /proc
| 194,491,936,768 | 52,302,696,448 | 42,348,515,328 | 77 | /
```

```
std::uintmax_t disk_usage_percent(const std::filesystem::space_info& si) noexcept
{
    if (constexpr std::uintmax_t X(-1);
        si.capacity == 0 || si.free == 0 || si.available == 0 ||
        si.capacity == X || si.free == X || si.available == X
    )
        return 100;

    std::uintmax_t unused_space = si.free, capacity = si.capacity;
    const std::uintmax_t used_space{capacity - unused_space};
    return 100 * used_space / capacity;
}

void print_disk_space_info(auto const& dirs, int width = 14)
{
    std::cout << std::left;
    for (const auto s : {"Capacity", "Free", "Available", "Use%", "Dir"})
        std::cout << "| " << std::setw(width) << s << ' ';

    for (std::cout << '\n'; auto const& dir : dirs)
    {
        std::error_code ec;
        const std::filesystem::space_info si = std::filesystem::space(dir, ec);
        for (auto x : {si.capacity, si.free, si.available, disk_usage_percent(si)})
            std::cout << "| " << std::setw(width) << static_cast<std::intmax_t>(x) << ' ';
        std::cout << "| " << dir << '\n';
        std::cout << ec.message() << "\n";
    }
}
```

Supplementary

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

Creation

- There are four kinds of modification operations:
  1. Create:

create_directory (C++17) **create_directories** (C++17)	creates new directory (function)
create_hard_link (C++17)	creates a hard link (function)
create_symlink (C++17) **create_directory_symlink** (C++17)	creates a symbolic link (function)

① directory:

- `create_directory(p[, existing_p[, ec]]) -> bool;`
  - When given `existing_p`, created `p` will copy OS-dependent attributes from another directory `existing_p` (Windows does nothing). Parent of `p` must exist (i.e. create only a single directory).
- `create_directories(p[, ec]) -> bool;`
  - Create every element that doesn't exist in directory path. So when parent of `p` doesn't exist, it will be created.

return `false` if directory already exists (not seen as error) or an error occurs.

Creation

ATTENTION: what the path symlink refers to is from the view of symlink itself. (but hard link doesn't do so).

② links:
- `create_hard_link(original_path, link_path[, ec]);`
- `create_symlink(original_path, link_path[, ec]);`
- `create_directory_symlink(original_path, link_path[, ec]);`
  - Some OS may need distinct APIs to create symbolic links to regular files and directories, so it's recommended to use `create_directory_symlink` for directory. POSIX doesn't distinguish them.
  - Note: these two methods just create soft links, which then require root privilege in Windows. And, there is no way to create junction in Windows by standard library.

③ Regular files: not in `stdfs`; you can use `std::ofstream` to do so.

Example:

```
fs::create_directories("sandbox/subdir");
fs::create_symlink("target", "sandbox/sym1");
fs::create_directory_symlink("subdir", "sandbox/sym2");

for (auto it = fs::directory_iterator("sandbox"); it != fs::directory_iterator(); ++it)
    if (is_symlink(it->symlink_status()))
        std::cout << *it << "->" << read_symlink(*it) << '\n';

assert(std::filesystem::equivalent("sandbox/sym2", "sandbox/subdir"));
```

```
"sandbox/sym1"->"target"
"sandbox/sym2"->"subdir"
```

Copy

2. Copy:

copy (C++17)	copies files or directories (function)
copy_file (C++17)	copies file contents (function)
copy_symlink (C++17)	copies a symbolic link (function)

- Copy supports `copy_options` (again, bit-operatable scoped enumeration) to configure copy behavior.
- `copy(from, to[, opt[, ec]])`: copy according to `opt`.
- `copy_file(from, to[, opt[, ec]])`: copy regular file according to `opt`.
  - This can be cheaper than `copy` for less check, and doesn't incur TOC/TOU compared with `if(!is_directory(a)) copy(a, b)`.
  - Error conditions (after following symbolic links, don't cover OS error like permission):
    - `!is_regular_file(from)`; or
    - `exists(to) && (equivalent(from, to) || !is_regular_file(to) || opt == copy_options::none)`.

The specific behavior (including error conditions) for `copy` is too long and thus not covered here. Check details in cppreference.

Copy option enumerators

At most one copy option in each of the following options groups may be present, otherwise the behavior of the copy functions is undefined.

Member constant	Meaning
options controlling `copy_file()` when the file already exists	
`none`	Report an error (default behavior).
`skip_existing`	Keep the existing file, without reporting an error.
`overwrite_existing`	Replace the existing file.
`update_existing`	Replace the existing file only if it is older than the file being copied.
options controlling the effects of `copy()` on subdirectories	
`none`	Skip subdirectories (default behavior).
`recursive`	Recursively copy subdirectories and their content.
options controlling the effects of `copy()` on symbolic links	
`none`	Follow symlinks (default behavior).
`copy_symlinks`	Copy symlinks as symlinks, not as the files they point to.
`skip_symlinks`	Ignore symlinks.
options controlling the kind of copying `copy()` does	
`none`	Copy file content (default behavior).
`directories_only`	Copy the directory structure, but do not copy any non-directory files.
`create_symlinks`	Instead of creating copies of files, create symlinks pointing to the originals. Note: the source path must be an absolute path unless the destination path is in the current directory.
`create_hard_links`	Instead of creating copies of files, create hardlinks that resolve to the same files as the originals.

(Also applied to `copy` for files).

Copy

- `copy_symlink(from, to[, ec])`: copy symbolic link to another position, instead of following link and copy content like `copy`.
  - Equiv. to `create_(directory_)symlink(read_symlink(from), to)`.
  - Note: `copy_symlink` doesn't copy junction on Windows; you can only use `copy(from, to, copy_options::copy_symlinks)` to copy it.

- Example of copy:

```cpp
fs::create_directories("sandbox/dir/subdir");
std::ofstream("sandbox/file1.txt").put('a');
fs::copy("sandbox/file1.txt", "sandbox/file2.txt"); // copy file
fs::copy("sandbox/dir", "sandbox/dir2"); // copy directory (non-recursive)
const auto copyOptions = fs::copy_options::update_existing
                       | fs::copy_options::recursive
                       | fs::copy_options::directories_only
                       ;
fs::copy("sandbox", "sandbox_copy", copyOptions);
static_cast<void>(std::system("tree"));
fs::remove_all("sandbox");
fs::remove_all("sandbox_copy");
```

```
.
├── sandbox
│   ├── dir
│   │   └── subdir
│   ├── dir2
│   ├── file1.txt
│   └── file2.txt
└── sandbox_copy
    ├── dir
    │   └── subdir
    └── dir2

8 directories, 2 files
```

Remove and Rename

3. Remove:
   - `remove(p[, ec]) -> bool`: requires `p` to be regular file or empty directory, symlink is not followed; return `true` if `p` is removed.
   - `remove_all(p[, ec]) -> uintmax_t`: remove all contents in `p` (including `p`), symlink is not followed; return number of removed entries (-1 on error).
4. Rename:
   - `rename(old_p, new_p[, ec])`:
     - If `new_p` is hard link to `old_p`, nothing happens;
     - If `old_p` is regular file: remove `new_p` if it exists and is a regular file, and rename to `new_p`.
     - If `old_p` is directory: remove `new_p` if it exists and is a directory, and rename to `new_p`. Parent of `new_p` should exist and `new_p` shouldn't end with separator.
     - Note: other processes cannot observe such removal.

Supplementary and Summary

Chrono

All utilities are defined in `std::chrono`; for brevity, we use `namespace stdc = std::chrono`.

Overview

- We know most of physical quantities need *units*.
  - 1 minutes = 60 seconds; 1 second = 1000 milliseconds; etc.
  - And their ratio can be conveyed by *rational numbers* (有理数).
- In `<chrono>` library, such ratio is used for time units too.

Type	Definition
std::chrono::nanoseconds	std::chrono::duration</* *int64* */, std::nano>
std::chrono::microseconds	std::chrono::duration</* *int55* */, std::micro>
std::chrono::milliseconds	std::chrono::duration</* *int45* */, std::milli>
std::chrono::seconds	std::chrono::duration</* *int35* */>
std::chrono::minutes	std::chrono::duration</* *int29* */, std::ratio<60>>
std::chrono::hours	std::chrono::duration</* *int23* */, std::ratio<3600>>
std::chrono::days (since C++20)	std::chrono::duration</* *int25* */, std::ratio<86400>>
std::chrono::weeks (since C++20)	std::chrono::duration</* *int22* */, std::ratio<604800>>
std::chrono::months (since C++20)	std::chrono::duration</* *int20* */, std::ratio<2629746>>
std::chrono::years (since C++20)	std::chrono::duration</* *int17* */, std::ratio<31556952>>

`intXX`: Signed integer with at least `XX` bits.

Supplementary

- Chrono
  - Compile-time rational number
  - Time
  - Date and time zone

num: numerator, 分子
den: denominator, 分母

Rational number

- We know rational number means a real number that can be expressed as **ratio of two integers**.
  - Their computation is exact, i.e. doesn't incur rounding error.
- So is it enough to just use `struct R { int num; int den; }`?
  - Not really…

1. `3 / 2, 6 / 4, …` are essentially same, i.e. it needs to do reduction.
2. class NTTP is introduced in C++20, while `<ratio>` is in C++11.

- Instead, it introduces a new **type** `std::ratio<Num, Den>`.
  - which defines quantities after reduction:

e.g. `std::ratio<6, 4>` has `num = 3, den = 2, type = std::ratio<3, 2>`.

Nested types

Type	Definition
type	std::ratio<num, den> (the rational type after reduction)

Data members

In the definitions given below,

- sign(Denom) is -1 if Denom is negative, or 1 otherwise; and
- gcd(Num, Denom) is the greatest common divisor of std::abs(Num) and std::abs(Denom).

Member			Definition
constexpr	std::intmax_t	num [static]	sign(Denom) * Num / gcd(Num, Denom) (public static member constant)
constexpr	std::intmax_t	den [static]	std::abs(Denom) / gcd(Num, Denom) (public static member constant)

- Operations are defined on types:

```
using two_third = std::ratio<2, 3>;
using one_sixth = std::ratio<1, 6>;
using sum = std::ratio_add<two_third, one_sixth>;

std::cout << "2/3 + 1/6 = " << sum::num << '/' << sum::den << '\n';
```

Arithmetic

ratio_add (C++11)	adds two ratio objects at compile-time (alias template)
ratio_subtract (C++11)	subtracts two ratio objects at compile-time (alias template)
ratio_multiply (C++11)	multiplies two ratio objects at compile-time (alias template)
ratio_divide (C++11)	divides two ratio objects at compile-time (alias template)

These operations will reduce the computed type automatically (so no need to use `::type` explicitly).

Comparison

ratio_equal (C++11)	compares two ratio objects for equality at compile-time (class template)
ratio_not_equal (C++11)	compares two ratio objects for inequality at compile-time (class template)
ratio_less (C++11)	compares two ratio objects for *less than* at compile-time (class template)
ratio_less_equal (C++11)	compares two ratio objects for *less than or equal to* at compile-time (class template)
ratio_greater (C++11)	compares two ratio objects for *greater than* at compile-time (class template)
ratio_greater_equal (C++11)	compares two ratio objects for *greater than or equal to* at compile-time (class template)

```
template< class R1, class R2 >
struct ratio_equal : std::integral_constant<bool, /* see below */> { };
```
(since C++11)

If the ratios R1 and R2 are equal, provides the member constant `value` equal `true`. Otherwise, `value` is `false`.

Helper variable template

```
template< class R1, class R2 >
constexpr bool ratio_equal_v = ratio_equal<R1, R2>::value;
```
(since C++17)

Alias for SI units:

quecto (C++26)	std::ratio<1, 10000000000000000000000000000000>, if std::intmax_t can represent the denominator
ronto (C++26)	std::ratio<1, 10000000000000000000000000000>, if std::intmax_t can represent the denominator
yocto	std::ratio<1, 10000000000000000000000000>, if std::intmax_t can represent the denominator
zepto	std::ratio<1, 10000000000000000000000>, if std::intmax_t can represent the denominator
atto	std::ratio<1, 1000000000000000000>
femto	std::ratio<1, 1000000000000000>
pico	std::ratio<1, 1000000000000>
nano	std::ratio<1, 1000000000>
micro	std::ratio<1, 1000000>
milli	std::ratio<1, 1000>
centi	std::ratio<1, 100>
deci	std::ratio<1, 10>
deca	std::ratio<10, 1>
hecto	std::ratio<100, 1>
kilo	std::ratio<1000, 1>
mega	std::ratio<1000000, 1>
giga	std::ratio<1000000000, 1>
tera	std::ratio<1000000000000, 1>
peta	std::ratio<1000000000000000, 1>
exa	std::ratio<1000000000000000000, 1>
zetta	std::ratio<1000000000000000000000, 1>, if std::intmax_t can represent the numerator
yotta	std::ratio<1000000000000000000000000, 1>, if std::intmax_t can represent the numerator
ronna (C++26)	std::ratio<1000000000000000000000000000, 1>, if std::intmax_t can represent the numerator
quetta (C++26)	std::ratio<1000000000000000000000000000000, 1>, if std::intmax_t can represent the numerator

Supplementary

- Chrono
  - Compile-time rational number
  - Time
    - Duration
    - Time point
  - Date and time zone

Duration

To use these literals, `using namespace std::literals` or `std::chrono_literals`.

- In physics, we define *time point* (时刻) and *time* (时间).
  - Time is just interval between two time points.
- In C++, such interval is represented by `stdc::duration`:

`Rep`: how to represent the number, e.g. `long long` or `double`.
`Period`: number unit (w.r.t. second), e.g. `std::milli` (`std::ratio<1, 1000>`) for milliseconds. Default means second.

```
template<
    class Rep,
    class Period = std::ratio<1>
> class duration;
```

  - It essentially only stores a number of type `Rep`.
    - You can get it (in value type instead of reference) by `.count()`.
  - It also provides some user-defined literals whose `Period` are adjusted accordingly:

`operator""h` (C++14)	a `std::chrono::duration` literal representing hours (function)
`operator""min` (C++14)	a `std::chrono::duration` literal representing minutes (function)
`operator""s` (C++14)	a `std::chrono::duration` literal representing seconds (function)
`operator""ms` (C++14)	a `std::chrono::duration` literal representing milliseconds (function)
`operator""us` (C++14)	a `std::chrono::duration` literal representing microseconds (function)
`operator""ns` (C++14)	a `std::chrono::duration` literal representing nanoseconds (function)

```
auto m = 10ms;
```

`auto` is `stdc::milliseconds`, i.e. `stdc::duration<intXX, std::milli>`.
Guess what about `auto m = 10.0ms`?

Duration

- You can also convert between different representations & periods by `stdc::duration_cast`:

```
template< class ToDuration, class Rep, class Period >
constexpr ToDuration duration_cast( const std::chrono::duration<Rep, Period>& d );
```

  - For example:

```
auto m = 10ms;
auto n = stdc::duration_cast<stdc::duration<double>>(m);
std::cout << n.count() << "\n";
```

```
0.01
```

  - To keep numeric accuracy, it will find "largest" type for both types first.
    - Representation of `ToDuration` is converted finally, i.e. after all computations are done.

Let

- ToRep be `typename ToDuration::rep`,
- ToPeriod be `typename ToDuration::period`,
- CF be `std::ratio_divide<Period, ToPeriod>`,
- CR be `std::common_type<Rep, ToRep, std::intmax_t>::type`,
- `cr_count` be `static_cast<CR>(d.count())`,
- `cr_num` be `static_cast<CR>(CF::num)`, and
- `cr_den` be `static_cast<CR>(CF::den)`,

the result is:

		`CF::num`	
		`1`	not `1`
`CF::den`	`1`	`ToDuration(static_cast<ToRep>(d.count()))`	`ToDuration(static_cast<ToRep>(cr_count * cr_num))`
	not `1`	`ToDuration(static_cast<ToRep>(cr_count / cr_den))`	`ToDuration(static_cast<ToRep>(cr_count * cr_num / cr_den))`

*: after decay, if no specialization.

Common type

- Basically*, `std::common_type<A, B>` is just type of conditional operator (i.e. `cond ? std::declval<A>() : std::declval<B>()`).
  - The detailed procedures are quite complex; generally speaking, it chooses either `A` or `B` that can be implicitly converted from another.
    - If none of or both of conversions can happen, then compilation error.
  - But for arithmetic types, since all types can do implicit conversion, it checks usual arithmetic conversions.
    - i.e. Floating-point > integers; more bits > less bits.
- `std::common_type<T0, T1, …>` just finds common type iteratively.
  - i.e. `std::common_type<std::common_type<T0, T1>, …>`.
- So for `duration_cast` between two integers, `intmax_t` will be used during computation.

- CR be `std::common_type<Rep, ToRep, std::intmax_t>::type`,
- `cr_count` be `static_cast<CR>(d.count())`,
- `cr_num` be `static_cast<CR>(CF::num)`, and
- `cr_den` be `static_cast<CR>(CF::den)`,

Duration

- For ctor:

```
constexpr duration() = default;                                   (1)
duration( const duration& ) = default;                            (2)
template< class Rep2 >
constexpr explicit duration( const Rep2& r );                     (3)
template< class Rep2, class Period2 >
constexpr duration( const duration<Rep2, Period2>& d );           (4)
```

3) Constructs a duration with `r` ticks.

This overload participates in overload resolution only if all following conditions are satisfied:

- `is_convertible<const Rep2&, Rep>::value` is `true`.
- Any of the following conditions is satisfied:[1]
  - `std::chrono::treat_as_floating_point<Rep>::value` is `true`.
  - `std::chrono::treat_as_floating_point<Rep2>::value` is `false`.

i.e. `Float` cannot be used to construct `stdc::duration<Int>` (so conversion is lossless).

4) Constructs a duration by converting `d` to an appropriate period and tick count, as if by `std::chrono::duration_cast<duration>(d).count()`.

This overload participates in overload resolution only if no overflow is induced in the conversion, and any of the following conditions is satisfied:[2]

- `std::chrono::treat_as_floating_point<Rep>::value` is `true`.
- All following conditions are satisfied:
  - `std::ratio_divide<Period2, Period>::den` is `1`.
  - `std::chrono::treat_as_floating_point<Rep2>::value` is `false`.

i.e. either the duration uses floating-point, or `Period2` is exactly divisible by `Period` (so conversion is lossless).

Duration

```
duration& operator++();
duration operator++( int );
duration& operator--();
duration operator--( int );
```

```
duration& operator+=( const duration& d );
duration& operator-=( const duration& d );
duration& operator*=( const rep& rhs );
duration& operator/=( const rep& rhs );
duration& operator%=( const rep& rhs );
duration& operator%=( const duration& rhs );
```

- It also defines many arithmetic operations.
- Note 1: `operator++/--` can only be used when the underlying representation can do so.
- Note 2: Pay attention to implicit conversion.
  - For example:

```
auto m = std::chrono::milliseconds{10};
m *= 1.5f;
std::cout << "150% of 10min: " << m.count() << "min" << std::endl;
```

    - Output:

```
150% of 10min: 10min
```

    - Reason: these types are integers!

```
std::chrono::milliseconds      std::chrono::duration</* int45 */, std::milli>
```

      - So `1.5f` is implicitly converted to integer 1.
  - A similar one: `auto m = 10ms; m *= 1.5f;`
  - Solution: make representation a floating-point number.
    - E.g. `auto m = 10.0ms` instead, or specify representation explicitly.

Duration

- Note 3: other operators use common type of two durations.
  - which is specialized to be common type of underlying representation.

```
template< class Rep1, class Period1, class Rep2, class Period2 >
typename std::common_type<duration<Rep1,Period1>, duration<Rep2,Period2>>::type
    constexpr operator+( const duration<Rep1,Period1>& lhs,
                         const duration<Rep2,Period2>& rhs );                    (1)

template< class Rep1, class Period1, class Rep2, class Period2 >
typename std::common_type<duration<Rep1,Period1>, duration<Rep2,Period2>>::type
    constexpr operator-( const duration<Rep1,Period1>& lhs,
                         const duration<Rep2,Period2>& rhs );                    (2)

template< class Rep1, class Period, class Rep2 >
duration<typename std::common_type<Rep1,Rep2>::type, Period>
    constexpr operator*( const duration<Rep1,Period>& d,
                         const Rep2& s );                                        (3)

template< class Rep1, class Rep2, class Period >
duration<typename std::common_type<Rep1,Rep2>::type, Period>
    constexpr operator*( const Rep1& s,
                         const duration<Rep2,Period>& d );                       (4)

template< class Rep1, class Period, class Rep2 >
duration<typename std::common_type<Rep1,Rep2>::type, Period>
    constexpr operator/( const duration<Rep1,Period>& d,
                         const Rep2& s );                                        (5)

ss Rep1, class Period1, class Rep2, class Period2 >
:common_type<Rep1,Rep2>::type
 operator/( const duration<Rep1,Period1>& lhs,
            const duration<Rep2,Period2>& rhs );                                 (6)

ss Rep1, class Period, class Rep2 >
name std::common_type<Rep1,Rep2>::type, Period>
 operator%( const duration<Rep1, Period>& d,
            const Rep2& s );                                                     (7)

ss Rep1, class Period1, class Rep2, class Period2 >
:common_type<duration<Rep1,Period1>, duration<Rep2,Period2>>::type
 operator%( const duration<Rep1,Period1>& lhs,
            const duration<Rep2,Period2>& rhs );                                 (8)
```

std::common_type<std::chrono::duration>

Defined in header <chrono>

```
template< class Rep1, class Period1, class Rep2, class Period2 >
struct common_type<std::chrono::duration<Rep1, Period1>,          (since C++11)
                   std::chrono::duration<Rep2, Period2>>;
```

Exposes the type named type, which is the common type of two std::chrono::durations, whose period is the greatest common divisor of Period1 and Period2.

Member types

Member type	Definition
type	std::chrono::duration<typename std::common_type<Rep1, Rep2>::type, /* see note */>

Note

The period of the resulting duration can be computed by forming a ratio of the greatest common divisor of `Period1::num` and `Period2::num` and the least common multiple of `Period1::den` and `Period2::den`.

Duration

```
template< class Rep1, class Period1, class Rep2, class Period2 >
    requires std::three_way_comparable<std::common_type_t<Rep1, Rep2>>
constexpr auto operator<=>( const std::chrono::duration<Rep1, Period1>& lhs,
                            const std::chrono::duration<Rep2, Period2>& rhs );
```

- Note 4: comparable:
  - which will also convert to their common type first and then compare `.count()`.

- Note 5: C++17 adds some simple math functions:
  - For example:

```
auto m = 10.5s; // To integer seconds
std::cout << stdc::ceil<stdc::seconds>(m).count() << "\n";
```

floor(std::chrono::duration) (C++17)

ceil(std::chrono::duration) (C++17)

round(std::chrono::duration) (C++17)

abs(std::chrono::duration) (C++17)

Duration

- C++20 adds more utilities:
  1. Days, weeks, months and years: since every month and year have different length, it uses the average number.
     - Accurate: 1 days = 86400s, 1 weeks = 7 days;
     - Average: 1 years = 365.2425 days, 1 months = $\frac{1}{12}$ years.
     - And also their integer aliases (after rounding):

std::chrono::days (since C++20)	`std::chrono::duration</* int25 */, std::ratio<86400>>`
std::chrono::weeks (since C++20)	`std::chrono::duration</* int22 */, std::ratio<604800>>`
std::chrono::months (since C++20)	`std::chrono::duration</* int20 */, std::ratio<2629746>>`
std::chrono::years (since C++20)	`std::chrono::duration</* int17 */, std::ratio<31556952>>`

     - Note: they do **NOT** have literals; `y` and `d` are used to convey a `year` and a `day`, which is date (time point) instead of duration.
       - Duration is `years` and `days`, not `year` and `day`.

BTW: since `months` isn't multiple of days, `days + months` doesn't result in a duration with unit `days`; the common type has a unit of 54 seconds. `days + years` is 216 seconds.

Duration

2. I/O and formatter: **operator<<** (C++20) **std::formatter**<std::chrono::duration> (C++20) **from_stream** (C++20)
   ① operator<<: output .count() with additional unit, depending on Period type.
   - For non-special types:

None of the above, and `Period::type::den == 1`	[*num*]s
None of the above	[*num*/*den*]s

   ② formatter: default format is equiv. to operator<<.

fill-and-align (optional) *width* (optional) *precision* (optional) **L** (optional) *chrono-spec* (optional)

   - The specific formatting specifiers *chrono-spec* are same as strftime in C and time locale.
     - It's rather complicated so we just give a brief introduction; check manual for detailed usage.
   - A simple example first:

```
auto m = 10000.5s;
std::println("{:%H hours, %M minutes, %S seconds}", m);
```

```
02 hours, 46 minutes, 40 seconds
```

Period::type	Suffix
std::atto	as
std::femto	fs
std::pico	ps
std::nano	ns
std::micro	µs (U+00B5) or us
std::milli	ms
std::centi	cs
std::deci	ds
std::ratio<1>	s
std::deca	das
std::hecto	hs
std::kilo	ks
std::mega	Ms
std::giga	Gs
std::tera	Ts
std::peta	Ps
std::exa	Es
std::ratio<60>	min
std::ratio<3600>	h
std::ratio<86400>	d

Conversion specifier	Explanation
%%	Writes a literal % character.
%n	Writes a newline character.
%t	Writes a horizontal tab character.
Time of day	
%H %OH	Writes the hour (24-hour clock) as a decimal number. If the result is a single digit, it is prefixed with 0. The modified command %0H writes the locale's alternative representation.
%I %OI	Writes the hour (12-hour clock) as a decimal number. If the result is a single digit, it is prefixed with 0. The modified command %0I writes the locale's alternative representation.
%M %OM	Writes the minute as a decimal number. If the result is a single digit, it is prefixed with 0. The modified command %0M writes the locale's alternative representation.
%S %OS	Writes the second as a decimal number. If the number of seconds is less than 10, the result is prefixed with 0. If the precision of the input cannot be exactly represented with seconds, then the format is a decimal floating-point number with a fixed format and a precision matching that of the precision of the input (or to a microseconds precision if the conversion to floating-point decimal seconds cannot be made within 18 fractional digits). The character for the decimal point is localized according to the locale. The modified command %0S writes the locale's alternative representation.
%p	Writes the locale's equivalent of the AM/PM designations associated with a 12-hour clock.
%R	Equivalent to **"%H:%M"**.
%T	Equivalent to **"%H:%M:%S"**.
%r	Writes the locale's 12-hour clock time.
%X %EX	Writes the locale's time representation. The modified command %EX writes the locale's alternate time representation.
Duration count	
%Q	Writes the count of ticks of the duration, i.e. the value obtained via `count()`.
%q	Writes the unit suffix of the duration, as specified in `operator<<()`.

Historical reason:
convenient for shell input.

%O?: locale's alternative numeric symbols.

%E?: locale's alternative era-based representation (纪元/年号纪年法) .

Duration format

```
auto m = 10000.5s;
std::println("{:%H hours, %M minutes, %S seconds}", m);
```

```
02 hours, 46 minutes, 40 seconds
```

- Note 1: the "precision" in seconds is not precision in format spec.
  - As example above shows, 0.5 second is not output.
  - But if we change it to milliseconds…
    - Wow, it has three fractional digits now.

```
auto m = 10000.5ms * 1000;
std::println("{:%H hours, %M minutes, %S seconds}", m);
```

```
02 hours, 46 minutes, 40.500 seconds
```

  - So assuming that the period is P;
    - If there exist integers K, M that satisfy $P = K \times 10^M$, then precision is defined as $\max(-M, 0)$.
      - For example, $1 = 1 \times 10^0$, so precision is 0 and thus floating-point is output as integer.
      - $1/1000 = 1 \times 10^{-3}$, so precision is 3.
    - Otherwise, precision is 6.
      - For example, $1/3$ cannot be expressed in form above and uses 6.

```
auto m = std::chrono::duration<double, std::ratio<1, 3>>{ 60.0 };
std::println("{:%H hours, %M minutes, %S seconds}", m);
```

```
00 hours, 00 minutes, 20.000000 seconds
```

$\max(-M, 0)$.

- For example, $1 = 1 \times 10^0$, so precision is 0 and thus floating-point is output as integer.
- $1/1000 = 1 \times 10^{-3}$, so precision is 3.

We'll cover details of `stdc::hh_mm_ss` later.

Duration format

- If you want a more sophisticated format, you can use `stdc::hh_mm_ss`.
  - It's just a utility class that equivalently converts duration to hour-minute-second integers, with subsecond for fractions.
  - For example:

```
auto m = 10000.5s;
auto hms = stdc::hh_mm_ss{ m };
std::println("{} {} {} {}", hms.hours(), hms.minutes(),
                            hms.seconds(), hms.subseconds());
```

```
2h 46min 40s 0.5s
```

  - It is also essentially where precision is defined:

Member constants

`constexpr unsigned` **`fractional_width`** [static]	the smallest possible integer in the range `[0, 18]` such that `precision` (see below) will exactly represent the value of `Duration{1}`, or `6` if there's no such integer (public static member constant)

Member types

Member type	Definition
`precision`	`std::chrono::duration<std::common_type_t<Duration::rep, std::chrono::seconds::rep>, std::ratio<1, 10`$^{fractional\_width}$`>>`

Duration format

- Note 2: currently effects of precision spec. in formatter are not clear for chrono (e.g. `{:.3%Q}` is valid but `.3` has no effect).
  - After asking proposal author of chrono library, I think this is a defect and may be solved by DR later.
- Note 3: 12-hour representation example (Windows):

Print functions don't accept locale (you need to change global locale), so we use `format` explicitly.

```
void Output(const std::locale& loc)
{ // Assuming loc gives UTF-8.
    using namespace std::literals;
    auto amTime = 5000s, pmTime = 50000s;
    constexpr std::string_view fmtStr = "%I: {0:L%I}, %OI: {0:L%OI}, %r: {0:L%r}, %p: {0:L%p}";
    std::println("Locale: {}", loc.name());
    std::println("AM {:%T}: {}", amTime, std::format(loc, fmtStr, amTime));
    std::println("PM {:%T}: {}", pmTime, std::format(loc, fmtStr, pmTime));
}
Output(std::locale::classic());
Output(std::locale{ "Chinese-Simplified.utf8" });
Output(std::locale{ "Japanese.utf8" });
Output(std::locale{ "el_GR.utf8" }); // Greek
```

```
Locale: C
AM 01:23:20: %I: 01, %OI: 01, %r: 01:23:20 AM, %p: AM
PM 13:53:20: %I: 01, %OI: 01, %r: 01:53:20 PM, %p: PM
Locale: Chinese-Simplified.utf8
AM 01:23:20: %I: 01, %OI: 01, %r: 1:23:20, %p: 上午
PM 13:53:20: %I: 01, %OI: 01, %r: 13:53:20, %p: 下午
Locale: Japanese.utf8
AM 01:23:20: %I: 01, %OI: 01, %r: 1:23:20, %p: 午前
PM 13:53:20: %I: 01, %OI: 01, %r: 13:53:20, %p: 午後
Locale: el_GR.utf8
AM 01:23:20: %I: 01, %OI: 01, %r: 1:23:20 πμ, %p: πμ
PM 13:53:20: %I: 01, %OI: 01, %r: 1:53:20 μμ, %p: μμ
```

Windows locales generally don't handle `%O` and `%E`.

Recap: `L` is needed for format to use locale.

Some locale may not respect `%r` (platform-dependent).

Duration

- To ensure 12-hour representation, C++20 also adds some functions for judging and conversion.
  - Result of out-of-bound time for `makeXX` is unspecified.

```
constexpr bool is_am( const std::chrono::hours& h ) noexcept;
constexpr bool is_pm( const std::chrono::hours& h ) noexcept;
constexpr std::chrono::hours make12( const std::chrono::hours& h ) noexcept;
constexpr std::chrono::hours make24( const std::chrono::hours& h,
                                     bool is_pm ) noexcept;
```

$h \in [0, 11]$

$h \in [12, 23]$

$result \in [1, 12]$
(0 is 12 a.m.)

③ `from_stream`: scan a string with a format to get a time.

- i.e. inversion of formatting, quite like std::scan (scan is not yet accepted).

```
template< class CharT, class Traits, class Rep, class Period,
          class Alloc = std::allocator<CharT> >
std::basic_istream<CharT, Traits>&
    from_stream( std::basic_istream<CharT, Traits>& is, const CharT* fmt,
                 std::chrono::duration<Rep, Period>& d,
                 std::basic_string<CharT, Traits, Alloc>* abbrev = nullptr,
                 std::chrono::minutes* offset = nullptr );
```

The last two parameters are for time zone and will be covered later.

Except for %Q/%q, it just accepts same chrono spec as format.

Conversion specifier	Explanation
%%	Matches a literal % character.
%n	Matches one whitespace character.
%t	Matches zero or one whitespace character.
	Time of day
%H %*N*H %OH	Parses the hour (24-hour clock) as a decimal number. The width *N* specifies the maximum number of characters to read. The default width is 2. Leading zeroes are permitted but not required. The modified command %OH interprets the locale's alternative representation.
%I %*N*I %OI	Parses the hour (12-hour clock) as a decimal number. The width *N* specifies the maximum number of characters to read. The default width is 2. Leading zeroes are permitted but not required. The modified command %OI interprets the locale's alternative representation.
%M %*N*M %OM	Parses the minute as a decimal number. The width *N* specifies the maximum number of characters to read. The default width is 2. Leading zeroes are permitted but not required. The modified command %OM interprets the locale's alternative representation.
%S %*N*S %OS	Parses the second as a decimal number. The width *N* specifies the maximum number of characters to read. The default width is 2. Leading zeroes are permitted but not required. The modified command %OS interprets the locale's alternative representation.
%p	Parses the locale's equivalent of the AM/PM designations associated with a 12-hour clock.
%R	Equivalent to **"%H:%M"**.
%T	Equivalent to **"%H:%M:%S"**.
%r	Parses the locale's 12-hour clock time.
%X %EX	Parses the locale's time representation. The modified command %EX interprets the locale's alternate time representation.

Duration input

- For example:

```
std::string buffer{ "18:14:21" };
std::ispanstream is{ std::span{ buffer.c_str(), buffer.size() } };
stdc::seconds secs;
stdc::from_stream(is, "%H:%M:%S", secs);
std::println("{:%r}", secs);
```

```
Microsoft Visual Studio 调试控
06:14:21 PM
```

  - Failed read will set stream `failbit` without modifying time passed in.
- A manipulator `stdc::parse` is also introduced for help:

```
chr::sys_days tp;
chr::hours h;
chr::minutes m;
// parse date into tp, hour into h and minute into m:
std::istringstream{"12/24/21 18:00"} >> chr::parse("%D", tp)
                                     >> chr::parse(" %H", h)
                                     >> chr::parse(":%M", m);
std::cout << tp << " at " << h << ' ' << m << '\n';
```

Duration

3. Details of `stdc::hh_mm_ss<Duration>`:

Member functions

(constructor)	constructs a hh_mm_ss (public member function)
is_negative **hours** **minutes** **seconds** **subseconds**	obtains components of the broken-down time (public member function)
operator precision **to_duration**	obtains the stored `std::chrono::duration` (public member function)

Convert back to duration

Non-member functions

operator<< (C++20)	outputs a hh_mm_ss into a stream (function template)

As if "{:L%T}" for equivalent duration.

Helper classes

std::formatter<std::chrono::hh_mm_ss> (C++20)	formatting support for hh_mm_ss (class template specialization)

As if formatter for the equivalent duration; default format is still same as `operator<<` (i.e. as if "{:L%T}").

```
constexpr hh_mm_ss() noexcept : hh_mm_ss{Duration::zero()} {}   (1)
constexpr explicit hh_mm_ss( Duration d );                      (2)
```

Constructs a hh_mm_ss object.

1) Constructs a hh_mm_ss object corresponding to `Duration::zero()`.
2) Constructs a hh_mm_ss object corresponding to `d`:
   - `is_negative()` returns `d < Duration::zero()`.
   - `hours()` returns `std::chrono::duration_cast<std::chrono::hours>(abs(d))`.
   - `minutes()` returns `std::chrono::duration_cast<std::chrono::minutes>(abs(d) - hours())`.
   - `seconds()` returns `std::chrono::duration_cast<std::chrono::seconds>(abs(d) - hours() - minutes())`.
   - `subseconds()` returns `abs(d) - hours() - minutes() - seconds()` if `std::chrono::treat_as_floating_point_v<precision::rep>` is `true`; otherwise it returns `std::chrono::duration_cast<precision>(abs(d) - hours() - minutes() - seconds())`.

```
constexpr bool is_negative() const noexcept;
constexpr std::chrono::hours hours() const noexcept;
constexpr std::chrono::minutes minutes() const noexcept;
constexpr std::chrono::seconds seconds() const noexcept;
constexpr precision subseconds() const noexcept;
```

Duration

- Finally some rarely-used notes:
  - Note 1: You can specialize `stdc::treating_as_floating_point` and `stdc::duration_values` for underlying representation.
    - `treating_as_floating_point`: as its name.
    - `duration_values`: need to define three functions.
      - which will be exposed as static functions of `stdc::duration` too.
      - Default: `Rep(0)`; `std::numeric_limits<Rep>::lowest()`; `std::numeric_limits<Rep>::max()`.
  - Note 2: hashable since C++26, as if hashing underlying representation.
  - Note 3: some member type aliases:

zero [static]

min [static]

max [static]

Member types

Member type	Definition
rep	Rep, an arithmetic type, or a class emulating an arithmetic type, representing the number of ticks
period	`Period`(until C++17) `typename Period::type`(since C++17), a `std::ratio` representing the tick period (i.e. the number of second's fractions per tick)

Supplementary

- Chrono
  - Compile-time rational number
  - Time
    - Duration
    - Time point
  - Date and time zone

Time point

- Essentially, time point is a relative quantity w.r.t. a reference origin.
  - That is, we can define it as an *epoch* (as reference) plus a duration.
  - In C++, epoch is defined in *clock*, which also has a *tick resolution* (which is still a duration).

Type	Meaning	Epoch
`system_clock`	Associated with the clock of the system (since C++11)	UTC time
`utc_clock`	Clock for UTC time values	UTC time
`gps_clock`	Clock for GPS values	GPS time
`tai_clock`	Clock for international atomic time values	TAI time
`file_clock`	Clock for timepoints of the filesystem library	impl.spec.
`local_t`	Pseudo clock for *local timepoints*	open
`steady_clock`	Clock for measurements (since C++11)	impl.spec.
`high_resolution_clock`	(see text)	

Since C++20

Table 11.2. Standard clock types since C++20

Time point

```
template<
    class Clock,
    class Duration = typename Clock::duration
> class time_point;
```

- We'll discuss detailed differences of clocks later; let's go through time point now.
  - Ctor:

`time_point();`	(1)	(since C++11) (constexpr since C++14)
`explicit time_point( const duration& d );`	(2)	(since C++11) (constexpr since C++14)
`template< class Duration2 >` `time_point( const time_point<Clock, Duration2>& t );`	(3)	(since C++11) (constexpr since C++14)

Constructs a new `time_point` from one of several optional data sources.

1) Default constructor, creates a `time_point` representing the `Clock`'s epoch (i.e., `time_since_epoch()` is zero).
2) Constructs a `time_point` at `Clock`'s epoch plus `d`.
3) Constructs a `time_point` by converting `t` to `duration`. This constructor only participates in overload resolution if `Duration2` is implicitly convertible to `duration`.

  - You can check the duration since epoch by: `duration time_since_epoch() const;`

Time point

- And limited arithmetic operations:
  - `operator+=/-=` a duration;
  - `operator+/-` a duration to get time point; or `operator-` another time point to get a duration. Duration type is still common type.

```
time_point& operator+=( const duration& d );
time_point& operator-=( const duration& d );
```

```
template< class C, class D1, class R2, class P2 >
time_point<C, typename std::common_type<D1, duration<R2,P2>>::type>
    operator+( const time_point<C,D1>& pt,
               const duration<R2,P2>& d );
```

```
template< class C, class D1, class D2 >
typename std::common_type<D1,D2>::type
    operator-( const time_point<C,D1>& pt_lhs,
               const time_point<C,D2>& pt_rhs );
```

    - Common type of `time_point` is also specialized, same as that of `duration`.
  - `operator++/--`;
  - Comparable;
  - And:

floor(std::chrono::time_point) (C++17)

ceil(std::chrono::time_point) (C++17)

round(std::chrono::time_point) (C++17)

which use `ToDuration` instead of time point as type template.
(e.g. `std::chrono::floor<std::chrono::days>(SysTime)` ✅

`std::chrono::floor<std::chrono::time_point<std::system_clock, std::chrono::days>>(SysTime)`). ❌

Time point

```
template< class ToDuration, class Clock, class Duration >
constexpr std::chrono::time_point<Clock, ToDuration>
    time_point_cast( const std::chrono::time_point<Clock, Duration> &t );
```

- Explicit cast: must be the same clock.
  - As if converting underlying duration by `duration_cast`.
- And some minor members:
  - `min/max()`: return time point with `.time_since_epoch()` that is same as duration's `min/max()`.
  - Hashable since C++26, as if hashing duration.
  - And some member types:

Type	Description
`Clock` **clock**	the clock on which this time point is measured (typedef)
`Duration` **duration**	a `std::chrono::duration` type used to measure the time since epoch (typedef)
`duration::rep` **rep**	an arithmetic type representing the number of ticks of the duration (typedef)
`duration::period` **period**	a `std::ratio` type representing the tick period of the duration (typedef)

Time point

- For example:

`time_point` is not printable and formattable by default except for specializations; `steady_clock` doesn't specialize so you can only print its duration.

```cpp
using Clock = std::chrono::steady_clock;
using TimePoint = std::chrono::time_point<Clock>;

void print_ms(const TimePoint& point)
{
    using Ms = std::chrono::milliseconds;
    const Clock::duration since_epoch = point.time_since_epoch();
    std::cout << std::chrono::duration_cast<Ms>(since_epoch) << '\n';
}

int main()
{
    const TimePoint default_value = TimePoint(); // (1)
    print_ms(default_value); // 0ms

    const Clock::duration duration_4_seconds = std::chrono::seconds(4);
    const TimePoint time_point_4_seconds(duration_4_seconds); // (2)
    // 4 seconds from start of epoch
    print_ms(time_point_4_seconds); // 4000ms
```

- So roughly speaking, `time_point` is a `duration` with limited utilities.

Clock

- Most of clocks define static method `now()` to get current time point.
  - Thus a simple profiling segment would be:

```
auto t1 = stdc::steady_clock::now();
SomeComplexComputation();
auto t2 = stdc::steady_clock::now();
stdc::duration<double, std::milli> msDelta = t2 - t1;
std::cout << msDelta;
```

- Then what's the difference of all clocks?
  - Epoch;
  - Tick resolution, which is implementation-defined;
  - **Steady** or not;
  - How to handle leap second (闰秒), if time is printable.
    - Current leap seconds are always positive since earth generally spins slower and slower.
    - Note: leap second may be cancelled since 2035 (well, I'm not sure), but the previous leap seconds still exist.

Clock

- Steady means that external adjustment cannot affect the time.
  - For example, `auto t1 = now(); auto t2 = now();` may lead to `t1 > t2`, if user changes system clock of OS.
  - Only `stdc::steady_clock` guarantees steady time.
- `stdc::system_clock`: clock of operating system.
  - Epoch: implementation-defined; regulated to be Unix Time since C++20.
    - i.e. 1970/1/1, 00:00:00 UTC (Thursday)
  - Leap second: every minute has at most 60 seconds; if a positive leap second happens, then previous second will take longer.
    - So every second unevenly shares the time (example later).
  - Note: You can get `time_t` in C (not covered here) from `stdc::system_clock`.

to_time_t [static]	converts a system clock time point to `std::time_t` (public static member function)
from_time_t [static]	converts `std::time_t` to a system clock time point (public static member function)

Clock

- `stdc::steady_clock`: monotonic clock with constant tick time.
  - Epoch: implementation-defined;
  - Steady: yes;
  - Not printable, every second evenly shares the time.

- `stdc::high_resolution_clock`: clock with smallest resolution.
  - In current implementations, it's an alias of either `steady_clock` or `system_clock`.

- These three clocks are in C++11; and new clocks are introduced since C++20.

Clock

```
template< class Duration >
static std::chrono::sys_time<std::common_type_t<Duration, std::chrono::seconds>>
    to_sys( const std::chrono::utc_time<Duration>& t );
template< class Duration >
static std::chrono::utc_time<std::common_type_t<Duration, std::chrono::seconds>>
    from_sys( const std::chrono::sys_time<Duration>& t );
```

- `stdc::utc_clock`: *Coordinated Universal Time* (UTC, 协调世界时), a.k.a. *Greenwich Mean Time* (GMT, 格林尼治平均时间).
  - This is the time we use in daily life (after offset by time zone).
    - For example, Beijing standard time is just UTC/GMT+8.
  - Epoch: Unix Time, i.e. 1970/1/1, 00:00:00 UTC (Thursday).
    - Note: the official UTC epoch is 1972/1/1, which isn't used in `utc_clock`.
  - Leap second: a minute may have 59 seconds or 61 seconds.
    - So every second evenly shares the time.
    - For example, 2017/1/1 Beijing time has `07:59:60`.
  - You can convert `utc_clock` to `system_clock` to cancel handling of leap second, or vice versa.
    - `sys_time` knows leap second (since its inner duration increases), but it doesn't tell users. So `from_sys` can get a `utc_time` with leap second.

Clock

- `stdc::gps_clock`: time used by GPS.
  - Epoch: 1980/1/6, 00:00:00 UTC.
  - Leap second: every minute has **exactly** 60 seconds; if a positive leap second happens, then it's just seen as next second in the next minute.
    - So every second evenly shares the time.
    - Now (as of 2026), GPS clock is 18 seconds ahead of UTC clock.
- `stdc::tai_clock`: *International Atomic Time*.
  - Same as GPS time except for epoch (since GPS also uses atomic clock).
  - Epoch: 1958/1/1 00:00:00 TAI (1957/12/31 23:59:50 UTC).
    - Now (as of 2026), TAI clock is 37 seconds ahead of UTC clock.

  - You can convert `gps_time/tai_time` to `utc_time` or vice versa.

to_utc [static]	converts a gps_time to utc_time (public static member function)
from_utc [static]	converts a utc_time to gps_time (public static member function)

to_utc [static]	converts tai_time to utc_time (public static member function)
from_utc [static]	converts utc_time to tai_time (public static member function)

Clock

- For example:

  (Details covered later)

```cpp
int main()
{
  using namespace std::literals;
  namespace chr = std::chrono;

  auto tpUtc = chr::clock_cast<chr::utc_clock>(chr::sys_days{2017y/1/1} - 1000ms);
  for (auto end = tpUtc + 2500ms; tpUtc <= end; tpUtc += 200ms) {
    auto tpSys = chr::clock_cast<chr::system_clock>(tpUtc);
    auto tpGps = chr::clock_cast<chr::gps_clock>(tpUtc);
    auto tpTai = chr::clock_cast<chr::tai_clock>(tpUtc);
    std::cout << std::format("{:%F %T} SYS  ", tpSys);
    std::cout << std::format("{:%F %T %Z}  ",  tpUtc);
    std::cout << std::format("{:%F %T %Z}  ",  tpGps);
    std::cout << std::format("{:%F %T %Z}\n",  tpTai);
  }
}
```

```
2016-12-31 23:59:59.000 SYS  2016-12-31 23:59:59.000 UTC  2017-01-01 00:00:16.000 GPS  2017-01-01 00:00:35.000 TAI
2016-12-31 23:59:59.200 SYS  2016-12-31 23:59:59.200 UTC  2017-01-01 00:00:16.200 GPS  2017-01-01 00:00:35.200 TAI
2016-12-31 23:59:59.400 SYS  2016-12-31 23:59:59.400 UTC  2017-01-01 00:00:16.400 GPS  2017-01-01 00:00:35.400 TAI
2016-12-31 23:59:59.600 SYS  2016-12-31 23:59:59.600 UTC  2017-01-01 00:00:16.600 GPS  2017-01-01 00:00:35.600 TAI
2016-12-31 23:59:59.800 SYS  2016-12-31 23:59:59.800 UTC  2017-01-01 00:00:16.800 GPS  2017-01-01 00:00:35.800 TAI
2016-12-31 23:59:59.999 SYS  2016-12-31 23:59:60.000 UTC  2017-01-01 00:00:17.000 GPS  2017-01-01 00:00:36.000 TAI
2016-12-31 23:59:59.999 SYS  2016-12-31 23:59:60.200 UTC  2017-01-01 00:00:17.200 GPS  2017-01-01 00:00:36.200 TAI
2016-12-31 23:59:59.999 SYS  2016-12-31 23:59:60.400 UTC  2017-01-01 00:00:17.400 GPS  2017-01-01 00:00:36.400 TAI
2016-12-31 23:59:59.999 SYS  2016-12-31 23:59:60.600 UTC  2017-01-01 00:00:17.600 GPS  2017-01-01 00:00:36.600 TAI
2016-12-31 23:59:59.999 SYS  2016-12-31 23:59:60.800 UTC  2017-01-01 00:00:17.800 GPS  2017-01-01 00:00:36.800 TAI
2017-01-01 00:00:00.000 SYS  2017-01-01 00:00:00.000 UTC  2017-01-01 00:00:18.000 GPS  2017-01-01 00:00:37.000 TAI
2017-01-01 00:00:00.200 SYS  2017-01-01 00:00:00.200 UTC  2017-01-01 00:00:18.200 GPS  2017-01-01 00:00:37.200 TAI
2017-01-01 00:00:00.400 SYS  2017-01-01 00:00:00.400 UTC  2017-01-01 00:00:18.400 GPS  2017-01-01 00:00:37.400 TAI
```

Clock

```
std::string asString(const std::filesystem::file_time_type& tp)
{
  using system_clock = std::chrono::system_clock;
  auto t = system_clock::to_time_t(tp + (system_clock::now()
                                         - decltype(tp)::clock::now()));
  // convert to calendar time (including skipping trailing newline):
  std::string ts = std::ctime(&t);
  ts.resize(ts.size()-1);
  return ts;
}
```

- `stdc::file_clock`: time used by filesystem (for last modification time).
  - It's completely implementation-defined *TrivialClock*; C++20 adds some utilities like converter and formatter.

    `file_clock` provides exactly one of the following two pairs of static member functions:
    - `to_utc` and `from_utc`; or
    - `to_sys` and `from_sys`.
  - Before that (i.e. in C++17), it's not easy to print its time point.
    - You have to rely on `time_t`, and tolerate error between two `now()`.
- Finally, if you just want to convey a "pure time point" without any reference, you can use `stdc::local_t` as clock.
  - Like we do in high-school physics…
  - It also acts as a pseudo clock when its reference is not determined, as we'll show in time zone.

Clock

- More utilities added in C++20 include:

1. Type aliases:

Type	Meaning	Defined as
local_time<*Dur*>	Local timepoint	time_point<*LocalTime*, *Dur*>
local_seconds	Local timepoint in seconds	time_point<*LocalTime*, seconds>
local_days	Local timepoint in days	time_point<*LocalTime*, days>
sys_time<*Dur*>	System timepoint	time_point<system_clock, *Dur*>
sys_seconds	System timepoint in seconds	time_point<system_clock, seconds>
sys_days	System timepoint in days	time_point<system_clock, days>
utc_time<*Dur*>	UTC timepoint	time_point<utc_clock, *Dur*>
utc_seconds	UTC timepoint in seconds	time_point<utc_clock, seconds>
tai_time<*Dur*>	TAI timepoint	time_point<tai_clock, *Dur*>
tai_seconds	TAI timepoint in seconds	time_point<tai_clock, seconds>
gps_time<*Dur*>	GPS timepoint	time_point<gps_clock, *Dur*>
gps_seconds	GPS timepoint in seconds	time_point<gps_clock, seconds>
file_time<*Dur*>	Filesystem timepoint	time_point<file_clock, *Dur*>

Table 11.3. Standard timepoint types since C++20

Clock

2. I/O for time point: except for `stdc::steady_clock / high_resolution_clock / local_t`.
   - Formatter: you can use all formats except for `%Q/q` in duration for time in a day; Besides, it provides more formats for **date**:

Year	
%C %EC	Writes the year divided by 100 using floored division. If the result is a single decimal digit, it is prefixed with 0. The modified command %EC writes the locale's alternative representation of the century.
%y %Oy %Ey	Writes the last two decimal digits of the year. If the result is a single digit it is prefixed by 0. The modified command %0y writes the locale's alternative representation. The modified command %Ey writes the locale's alternative representation of offset from %EC (year only).
%Y %EY	Writes the year as a decimal number. If the result is less than four digits it is left-padded with 0 to four digits. The modified command %EY writes the locale's alternative full year representation.
Month	
%b %h	Writes the locale's abbreviated month name.
%B	Writes the locale's full month name.
%m %Om	Writes the month as a decimal number (January is 01). If the result is a single digit, it is prefixed with 0. The modified command %Om writes the locale's alternative representation.

Example: Japanese (since it still keeps era) 2011/9/25

Windows	Linux
20	20
20	平成
11	11
11	十一
11	23
2011	2011
2011	平成23年
9	9月
9	9月
9月	9月
09	09
09	九

Week-based year is like syllabus in University; e.g. you will take exam in Week 17 and 18.

2011/9/25 is in 38/39th week in 2011 (depending on how to count a week), 268th day in 2011.

		Windows	Linux
Day			
%d	Writes the day of month as a decimal number. If the result is a single decimal digit, it is prefixed with 0.	25	25
%0d	The modified command %0d writes the locale's alternative representation.	25	二十五
%e	Writes the day of month as a decimal number. If the result is a single decimal digit, it is prefixed with a space.	25	25
%0e	The modified command %0e writes the locale's alternative representation.	25	二十五
Day of the week			
%a	Writes the locale's abbreviated weekday name.	日	日
%A	Writes the locale's full weekday name.	日曜日	日曜日
%u	Writes the ISO weekday as a decimal number (1-7), where Monday is 1.	7	7
%0u	The modified command %0u writes the locale's alternative representation.	7	七
%w	Writes the weekday as a decimal number (0-6), where Sunday is 0.	0	0
%0w	The modified command %0w writes the locale's alternative representation.	0	〇
ISO 8601 week-based year			
In ISO 8601 weeks begin with Monday and the first week of the year must satisfy the following requirements: • Includes January 4 • Includes first Thursday of the year			
%g	Writes the last two decimal digits of the ISO 8601 week-based year. If the result is a single digit it is prefixed by 0.	11	11
%G	Writes the ISO 8601 week-based year as a decimal number. If the result is less than four digits it is left-padded with 0 to four digits.	2011	2011
%V	Writes the ISO 8601 week of the year as a decimal number. If the result is a single digit, it is prefixed with 0.	38	38
%0V	The modified command %0V writes the locale's alternative representation.	38	三十八
Week/day of the year			
%j	Writes the day of the year as a decimal number (January 1 is 001). If the result is less than three digits, it is left-padded with 0 to three digits.	268	268
%U	Writes the week number of the year as a decimal number. The first Sunday of the year is the first day of week 01. Days of the same year prior to that are in week 00. If the result is a single digit, it is prefixed with 0.	39	39
%0U	The modified command %0U writes the locale's alternative representation.	39	三十九
%W	Writes the week number of the year as a decimal number. The first Monday of the year is the first day of week 01. Days of the same year prior to that are in week 00. If the result is a single digit, it is prefixed with 0.	38	38
%0W	The modified command %0W writes the locale's alternative representation.	38	三十八
Date			
%D	Equivalent to "%m/%d/%y".	09/25/11	09/25/11
%F	Equivalent to "%Y-%m-%d".	2011-09-25	2011-09-25
%x	Writes the locale's date representation.	2011/09/25	2011年09月25日
%Ex	The modified command %Ex produces the locale's alternate date representation.	2011/09/25	平成23年09月25日

Clock

Special note: `operator<<(os, sys_time)` also has special constraints (require integer representation, unit no longer than 1 day):

This overload participates in overload resolution only if `std::chrono::treat_as_floating_point_v<typename Duration::rep>` is `false` and `Duration(1) < std::chrono::days(1)`.

- Different clocks have different abbr. and can be printed by `%Z`:
  - System clock / UTC clock / file clock: UTC;
  - TAI clock: TAI;
  - GPS clock: GPS;
  - You can also use `%z/Oz/Ez` to print time zone offset, which will be covered later.
- And the whole time point can also be printed by `%c`:

%c	Writes the locale's date and time representation.
%Ec	The modified command %Ec writes the locale's alternative date and time representation.

- `operator<<` is just equiv. to `{:L%F %T}`, i.e. `year-month-day hour:minute:second`.
  - `operator<<` adds an overload for `sys_days` so that it only prints `{:L%F}`.
  - For example:

```
std::cout << date << "\n";
std::cout << stdc::sys_seconds{} + date.time_since_epoch() << "\n";
```

```
2011-09-25
2011-09-25 00:00:00
```

Specifier	C	Chinese-Simplified.utf8	Japanese.utf8	el_GR.utf8
%C	20	20	20	20
%EC	20	20	20	20
%y	11	11	11	11
%Oy	11	11	11	11
%Ey	11	11	11	11
%Y	2011	2011	2011	2011
%EY	2011	2011	2011	2011
%b	Sep	9月	9	Σεπ
%h	Sep	9月	9	Σεπ
%B	September	九月	9月	Σεπτέμβριος
%m	09	09	09	09
%Om	09	09	09	09
%d	25	25	25	25
%Od	25	25	25	25
%e	25	25	25	25
%Oe	25	25	25	25
%a	Sun	周日	日	Κυρ
%A	Sunday	星期日	日曜日	Κυριακή

Full table on Windows, 2011/9/25

Specifier	C	zh_CN.utf8	ja_JP.utf8	el_GR.utf8
%C	20	20	20	20
%EC	20	20	平成	20
%y	11	11	11	11
%Oy	11	11	十一	11
%Ey	11	11	23	11
%Y	2011	2011	2011	2011
%EY	2011	2011	平成23年	2011
%b	Sep	9月	9月	Σεπ
%h	Sep	9月	9月	Σεπ
%B	September	九月	9月	Σεπτεμβρίου
%m	09	09	09	09
%Om	09	09	九	09
%d	25	25	25	25
%Od	25	25	二十五	25
%e	25	25	25	25
%Oe	25	25	二十五	25
%a	Sun	日	日	Κυρ
%A	Sunday	星期日	日曜日	Κυριακή

Full table on Linux, 2011/9/25

Difference only lies in %O, %E and locale-dependent output

Yes, Greek has two names for September.

%u	7	7	7	7
%Ou	7	7	7	7
%w	0	0	0	0
%Ow	0	0	0	0
%g	11	11	11	11
%G	2011	2011	2011	2011
%V	38	38	38	38
%OV	38	38	38	38
%j	268	268	268	268
%U	39	39	39	39
%OU	39	39	39	39
%W	38	38	38	38
%OW	38	38	38	38
%D	09/25/11	09/25/11	09/25/11	09/25/11
%F	2011-09-25	2011-09-25	2011-09-25	2011-09-25
%x	09/25/11	2011/9/25	2011/09/25	25/9/2011
%Ex	09/25/11	2011/9/25	2011/09/25	25/9/2011
%c	09/25/11 00:00:00	2011/9/25 00:00:00	2011/09/25 00:00:00	25/9/2011 12:00:00 πμ
%Ec	09/25/11 00:00:00	2011/9/25 00:00:00	2011/09/25 00:00:00	25/9/2011 12:00:00 πμ

%u	7	7	7	7
%Ou	7	7	七	7
%w	0	0	0	0
%Ow	0	0	〇	0
%g	11	11	11	11
%G	2011	2011	2011	2011
%V	38	38	38	38
%OV	38	38	三十八	38
%j	268	268	268	268
%U	39	39	39	39
%OU	39	39	三十九	39
%W	38	38	38	38
%OW	38	38	三十八	38
%D	09/25/11	09/25/11	09/25/11	09/25/11
%F	2011-09-25	2011-09-25	2011-09-25	2011-09-25
%x	09/25/11	2011年09月25日	2011年09月25日	25/09/2011
%Ex	09/25/11	2011年09月25日	平成23年09月25日	25/09/2011
%c	Sun Sep 25 10:30:59 2011	2011年09月25日 星期日 10时30分59秒	2011年09月25日 10時30分59秒	Κυρ 25 Σεπ 2011 10:30:59 πμ CST
%Ec	Sun Sep 25 10:30:59 2011	2011年09月25日 星期日 10时30分59秒	平成23年09月25日 10時30分59秒	Κυρ 25 Σεπ 2011 10:30:59 πμ CST

We use 10:30:59 in Linux.

Clock

- `from_stream` accepts similar formats (just sometimes be case-insensitive, and add some `%N_` for "at most N characters"), so we don't cover it again.
  - And you can still use `parse` as manipulator.

3. Uniform clock conversion: since C++20 introduces many different clocks, it's necessary to introduce a converter.
   - And we say that you can just use e.g. `stdc::from_sys`, `stdc::to_utc`, etc.
     - However, it doesn't directly define arbitrary conversion between two clocks.
   - Instead, you can use `stdc::clock_cast`!
     - As we show previously:

```
auto tpSys = chr::clock_cast<chr::system_clock>(tpUtc);
auto tpGps = chr::clock_cast<chr::gps_clock>(tpUtc);
auto tpTai = chr::clock_cast<chr::tai_clock>(tpUtc);
```

     - Principle: direct conversion if possible, otherwise use UTC and system clock as hub.

Clock

- If both types can be converted from & to UTC or system clock:
  - Then `to_sys` and then `from_sys`, or `to_utc` and then `from_utc`.
  - If both two ways are legal, then compilation error for ambiguity.
- If source type can be converted to UTC and destination type can be converted from system clock (or vice versa, i.e. to system and from UTC):
  - Then `to_utc`, then UTC clock is converted to system clock (by UTC `to_sys`), then `from_sys`;
  - Or `to_sys`, then system clock is converted to UTC clock (by UTC `from_sys`), then `from_utc`;
  - If both two ways are legal, then compilation error for ambiguity.

- The underlying conversion relies on `stdc::clock_time_conversion`, which has already been specialized on UTC clock and system clock.
  - We don’t cover it in detail here. Check manual if necessary.

Clock

- And some final notes:

1. Do NOT use non-steady clock for waiting events.
   - Reason: Non-monotonicity can make waiting much longer than expected.
   - For example: `std::condition_variable::wait_for/until()`;
     - Initially, libstdc++ implements it by `system_clock`, so `wait_for(10s)` can wait for 10 minutes if external users adjust system clock.
     - This is fixed in gcc10 by introducing `pthread_cond_clockwait` in glibc and using `steady_clock`.
2. Given a UTC time point, you can check whether it's a leap second by `stdc::get_leap_second_info(utc)`.
   - Which returns

```
struct leap_second_info {
    bool is_leap_second;
    std::chrono::seconds elapsed;
};
```

`elapsed`: sum of all leap seconds since epoch.

Supplementary

- Chrono
  - Compile-time rational number
  - Time
  - Date and time zone
    - Date
    - Time zone

Date

- Though we provide many utilities for time point, we don’t tell a way to make a date.
  - Well, you can use duration since epoch to construct a time point, but then you need manual calculation (which is also inaccurate since years and months are average time).

- C++20 provides lots of utilities to help you to do so.
  - For example, if we want to meet on the 5th day of every month in 2021:

```
using namespace std::literals;
auto first = 2021y / 1 / 5;

for (auto d = first; d.year() == first.year(); d += stdc::months{ 1 })
{
    std::println("We'll meet on {}", d);
}
```

```
We'll meet on 2021-01-05
We'll meet on 2021-02-05
We'll meet on 2021-03-05
We'll meet on 2021-04-05
We'll meet on 2021-05-05
We'll meet on 2021-06-05
We'll meet on 2021-07-05
We'll meet on 2021-08-05
We'll meet on 2021-09-05
We'll meet on 2021-10-05
We'll meet on 2021-11-05
We'll meet on 2021-12-05
```

Date

```cpp
using namespace std::literals;
auto first = 2021y / 1 / 5;

for (auto d = first; d.year() == first.year(); d += stdc::months{ 1 })
{
    std::println("We'll meet on {}", d);
}
```

- Remember our note?

- And also their integer aliases (after rounding):

std::chrono::days (since C++20)	std::chrono::duration</* int25 */, std::ratio<86400>>
std::chrono::weeks (since C++20)	std::chrono::duration</* int22 */, std::ratio<604800>>
std::chrono::months (since C++20)	std::chrono::duration</* int20 */, std::ratio<2629746>>
std::chrono::years (since C++20)	std::chrono::duration</* int17 */, std::ratio<31556952>>

- Note: they do **NOT** have literals; `y` and `d` are used to convey a `year` and a `day`, which is time point instead of duration.
  - Duration is `years` and `days`, not `year` and `day`.

- So the whole process is:
  1. `2021y` constructs a `class year` (instead of a duration like `2021ms`).
  2. Use `operator/` of `year` and get `class year_month`.
  3. Use `operator/` of `year_month` and get `class year_month_day`.
  4. Use `operator+=(const months&)` of `year_month_day` to get date of next month.

Date

- You can also put day first: `auto d = 5d / 1 / 2021;`
  - i.e. `day -> month_day -> year_month_day`.
- Or month first: `auto d = stdc::month{ 1 } / 5 / 2021;`
  - i.e. `month -> month_day -> year_month_day`.
  - You can also use alias names of months: `auto d = stdc::January / 5 / 2021;`
- Besides normal day, there also exist three ways to denote a date:
  - Last day: `year_month_day_last`, count backward instead of forward; e.g. 3/31 is the last day of March.
  - Weekday: `weekday`, i.e. Monday ~ Sunday. The i^{th} weekday is denoted by `weekday_indexed`, and forms `(year_)month_weekday`.
  - Last weekday: `weekday_last`, count backward instead of forward; e.g. 2026/2/23 is the last Monday of 2026/2. Form `(year_)month_weekday_last`.

Date

1. `year_month / last -> year_month_day_last`
2. `weekday[index] -> weekday_indexed`
   `year_month / weekday_indexed -> year_month_weekday`
3. `weekday[last] -> weekday_last`
   `year_month / weekday_last -> year_month_weekday_last`

- You can just use them like a normal day:

Index starts from 1 instead of 0

```
// auto first = 2021y / 1 / stdc::last;              // Last day
// auto first = 2021y / 1 / stdc::Monday[1];         // *First* monday
auto first = 2021y / 1 / stdc::Monday[stdc::last];  // Last monday
for (auto d = first; d.year() == first.year(); d += stdc::months{ 1 })
{
    std::println("We'll meet on {}", d);
}
```

```
We'll meet on 2021/Jan/Mon[last]
We'll meet on 2021/Feb/Mon[last]
We'll meet on 2021/Mar/Mon[last]
We'll meet on 2021/Apr/Mon[last]
We'll meet on 2021/May/Mon[last]
We'll meet on 2021/Jun/Mon[last]
We'll meet on 2021/Jul/Mon[last]
We'll meet on 2021/Aug/Mon[last]
We'll meet on 2021/Sep/Mon[last]
We'll meet on 2021/Oct/Mon[last]
We'll meet on 2021/Nov/Mon[last]
We'll meet on 2021/Dec/Mon[last]
```

- Such output is not interesting, so you can use special formats:

```
std::println("We'll meet on {:%D}", d);
```

Last day of all months.

```
We'll meet on 01/31/21
We'll meet on 02/28/21
We'll meet on 03/31/21
We'll meet on 04/30/21
We'll meet on 05/31/21
We'll meet on 06/30/21
We'll meet on 07/31/21
We'll meet on 08/31/21
We'll meet on 09/30/21
We'll meet on 10/31/21
We'll meet on 11/30/21
We'll meet on 12/31/21
```

First Monday of all months.

```
We'll meet on 01/04/21
We'll meet on 02/01/21
We'll meet on 03/01/21
We'll meet on 04/05/21
We'll meet on 05/03/21
We'll meet on 06/07/21
We'll meet on 07/05/21
We'll meet on 08/02/21
We'll meet on 09/06/21
We'll meet on 10/04/21
We'll meet on 11/01/21
We'll meet on 12/06/21
```

Last Monday of all months.

```
We'll meet on 01/25/21
We'll meet on 02/22/21
We'll meet on 03/29/21
We'll meet on 04/26/21
We'll meet on 05/31/21
We'll meet on 06/28/21
We'll meet on 07/26/21
We'll meet on 08/30/21
We'll meet on 09/27/21
We'll meet on 10/25/21
We'll meet on 11/29/21
We'll meet on 12/27/21
```

So different classes have different explanations of `operator+=(months)`!

Date

- So now you can understand all the classes with different combinations:

1. Fundamental types share similar methods:
   - Ctor: constructed from an `unsigned int`.
   - `explicit operator unsigned int` (except for `weekday`).
     - `year` is `int` for the above two, for existence of BC.
   - `.ok()`: judge whether it exists (details covered later).
   - `operator==/<=>` (with the same class).
     - `weekday` doesn't have `operator<=>`.
   - `operator++/--/+=/-=/+/-`.
     - Defined on proper duration, e.g. `day += days`, `month += months`, `year += years`.
   - `formatter/operator<</from_stream`.
     - Format spec. is same as ones covered in `time_point`, except that you cannot use absent fields (e.g. illegal to use `%Y` for `day`).

last_spec (C++20) Tag type for stdc::last.

day (C++20)

month (C++20)

year (C++20)

weekday (C++20)

weekday_indexed (C++20)

weekday_last (C++20)

month_day (C++20)

month_day_last (C++20)

month_weekday (C++20)

month_weekday_last (C++20)

year_month (C++20)

year_month_day (C++20)

year_month_day_last (C++20)

year_month_weekday (C++20)

year_month_weekday_last (C++20)

Date

- So why is `weekday` special?
  - Order of `weekday` is not really uniformly defined…
  - Some cultures use Sunday as the last day, while others use it as the first day.
    - So you cannot really define a uniform `operator<=>`.
  1. C++ provides two getters:

     ```
     constexpr unsigned c_encoding() const noexcept;
     constexpr unsigned iso_encoding() const noexcept;
     ```

     [0, 6], Sunday is 0.
     [1, 7], Sunday is 7.
     - 0 and 7 are also same in ctor.
  2. Its arithmetic operations are defined on $(\mathrm{mod}\ 7)$.
     - For example, `Friday += 2` is `Sunday`; `++Sunday` is `Monday`; `Sunday - Friday == 2`; `Monday - Friday == 3`.
       - These integers are actually duration `days`.
  3. Finally, you can construct `weekday` by time point `sys_days` or `local_days`.
     - So you can know weekday of that calendar day.

Date

2. Index `weekday` by `operator[]` gives `weekday_indexed/last`:
   - The methods are pretty easy:

   - `operator<<` and default format of `weekday` are equiv. to `%a` (i.e. locale-dependent abbr.), and `weekday_indexed/last` is equiv. to `%a[index]`.
     - E.g. `Mon[1]`, `Fri[last]`.

Date

Attribute	Internal Type	Valid Values
Day	`unsigned char`	1 to 31
Month	`unsigned char`	1 to 12
Year	`short`	−32767 to 32767
Weekday	`unsigned char`	0 (Sunday) to 6 (Saturday) and 7 (again Sunday), which is converted to 0
Weekday index	`unsigned char`	1 to 5

- Note 1: regulations of `.ok()`:
  - Particularly, **internal types** constrain the range of values. Behavior of out-of-range values is undefined.
    - For example, `day{1}` is valid; `day{32}` is invalid but well-defined; `day{256}` is unspecified.
    - And consequently, result of `day{1} + days{255}` is unspecified (instead of definitely invalid).
- Note 2: for an invalid date (i.e. `.ok()` returns `false`), `operator<<` and empty format output `" is not a valid XX"` additionally.
  - `XX` can be `day/month/year/weekday/index`, depending on the type.
- Note 3: arithmetic operations of `month` are defined on $(\mathrm{mod}\ 12)$.
  - But stored months will +1 (i.e. in [1, 12]).

Date

3. Composite types also share similar methods:
   - Getters: get properties from stored fields.
     - E.g. for `year_month_weekday_last`:
     - The types are just fundamental types.
   - `.ok()`: judge whether the date **can** exist.
     - E.g. 2021/2/29 (as `year_month_day`) can't exist; but 2/29 can (as `month_day`); 2/31 can't exist, but 31 can (as `day`).
   - `operator==/!=` (with the same class).
     - `(year_)month_day(_last)` also support `operator<=>`.
   - `formatter/operator<<`; `from_stream` only for `(year_)month(_day)`.
     - Similarly, you cannot use absent fields for formatters (e.g. illegal to use `%D` for `year_month`).
   - `operator+/-/+=/-=(years/months)`, only for `year_xxx`.
     - With different interpretations in different classes, as we mentioned before.

year
month
weekday
weekday_last

month_day (C++20)
month_day_last (C++20)
month_weekday (C++20)
month_weekday_last (C++20)
year_month (C++20)
year_month_day (C++20)
year_month_day_last (C++20)
year_month_weekday (C++20)
year_month_weekday_last (C++20)

Date

- Note 1: operator<< and empty format come done to output fundamental types connected by /.
  - As we show in the example before:
  - So invalid indications will also be connected:

```
auto first = stdc::month{ 13 } / 32;
std::cout << first;
```

```
13 is not a valid month/32 is not a valid day
```

```
We'll meet on 2021/Jan/Mon[last]
We'll meet on 2021/Feb/Mon[last]
We'll meet on 2021/Mar/Mon[last]
We'll meet on 2021/Apr/Mon[last]
We'll meet on 2021/May/Mon[last]
We'll meet on 2021/Jun/Mon[last]
We'll meet on 2021/Jul/Mon[last]
We'll meet on 2021/Aug/Mon[last]
We'll meet on 2021/Sep/Mon[last]
We'll meet on 2021/Oct/Mon[last]
We'll meet on 2021/Nov/Mon[last]
We'll meet on 2021/Dec/Mon[last]
```

- Note 2: the result date of operator+/-/+=/-= may be invalid, e.g. 2021y/1/31 + months{1} == 2021y/2/31.
- Note 3: the four complete date types can be converted to sys_days or local_days.

```
constexpr operator std::chrono::sys_days() const noexcept;
constexpr explicit operator std::chrono::local_days() const noexcept;
```

- year_month_day (C++20)
- year_month_day_last (C++20)
- year_month_weekday (C++20)
- year_month_weekday_last (C++20)

Date

- Adjusted example:

```
2021/Jan/last:
 We meet on Sunday 01/31/21 at 18:30
2021/Feb/last:
 We meet on Sunday 02/28/21 at 18:30
2021/Mar/last:
 We meet on Wednesday 03/31/21 at 18:30
2021/Apr/last:
 We meet on Friday 04/30/21 at 18:30
2021/May/last:
 We meet on Monday 05/31/21 at 18:30
…
2021/Dec/last:
 We meet on Friday 12/31/21 at 18:30
```

```
// for each last of all months of 2021:
auto first = 2021y / 1 / chr::last;
for (auto d = first; d.year() == first.year(); d += chr::months{1}) {
  // print out the date:
  std::cout << d << ":\n";

  // init and print 18:30 UTC of those days:
  auto tp{chr::sys_days{d} + 18h + 30min};
  std::cout << std::format(" We meet on {:%A %D at %R}\n", tp);
}
```

- NOTE AGAIN: `first + months` is not same as `sys_days{ first } + months`; durations treat `months` as average.

Date

- Another example:

```
constexpr std::chrono::weekday friday{5}; // uses overload (2)
static_assert(friday == std::chrono::Friday);

for (int y{2020}; y <= 2024; ++y)
{
    const std::chrono::year cur_year{y};
    for (int cur_month{1}; cur_month != 13; ++cur_month)
    {
        const std::chrono::year_month_day ymd{cur_year/cur_month/13};
        const std::chrono::weekday cur_weekday{std::chrono::sys_days(ymd)};
        if (cur_weekday == friday)
            std::cout << ymd << " is " << friday << '\n';
    }
}
```

We've said that `weekday` can be constructed from `sys_days`.

- Effects: print Friday that is 13th day in a month between 2020 and 2024.

Date

- Note 4: `year_month_day` is a special type.
  1. It can be constructed from `year_month_day_last`.
  2. It can be constructed from `sys_days` and `local_days`.
     - By contrast, other types can only be *converted to* `sys_days` and `local_days`.
     - So a uniform way to convert other types to `year_month_day` is by (1) converting to `sys_days` (2) constructing `year_month_day`.
  3. `operator<<` and empty format print `%F` (i.e. `%Y-%m-%d`) instead of components connected by `/`.
     - When it's invalid, the printed content doesn't come done to invalid messages of components either; it just prints " `is not a valid` **`date`**" additionally.

```
auto first = stdc::month{ 13 } / 32;
std::cout << first;
```

```
13 is not a valid month/32 is not a valid day
```

```
auto first = 2021y / 13 / 32;
std::cout << first;
```

```
2021-13-32 is not a valid date
```

Date

- Note 5: `year_month_day_last` defines `.day()` additionally, so you can get the last day directly.

- Note 6: `year_month` defines `operator-(year_month, year_month) -> months`, so you can get difference in months between them.

```
auto first = 2021y / 11, last = 2022y / 3;
std::cout << (last - first).count() << " months";
```

```
4 months
```

  - Note again: the specific value is not accurate as duration.

- Note 7: all types are hashable since C++26.

Supplementary

- Chrono
  - Compile-time rational number
  - Time
  - Date and time zone
    - Date
    - Time zone

Time zone

- Different regions have different time zones.
  - It’s just conveyed by an offset w.r.t. clock, e.g. UTC+8 for Beijing time.
    - So 13:00 Asia/Shanghai is same as 5:00 UTC, or 23:00 Ameria/Chicago (in the last day).
  - Some cultures also have summer/winter time (夏令时/冬令时), also called daylight-saving/standard time (日光节约时/标准时).
- But time zones have special characteristics…
  1. The offset isn’t necessarily multiple of hours; 15/30/45 minutes are often used too.
  2. Abbr. can be ambiguous, e.g. CST can refer to *Central Standard Time* (for Chicago), *China Standard Time* or *Cuba Standard Time*, etc.
  3. Time zone can change when countries decide to do so (e.g. when they introduce summer/winter time).

Daylight saving time

- BTW, a little bit knowledge about daylight saving time (DST).
  - Generally, it aims to match time with daylight.
    - For example, if daylight appears after 7:00 in winter but after 6:00 in summer, then applying DST will make daylight always appear after 7:00.
  - When it goes to summer, time will jump to later hours;
  - When it goes to winter, time will jump back to previous hours.

  - For example, in some states of America, the second Sunday on March will apply DST, and cancel it in the first Sunday on November.
    - E.g. in 2016, 2016-03-13 02:00:00 will jump to 2016-03-13 03:00:00, so 2016-03-13 02:30:00 **doesn't exist** locally.
    - And 2016-11-06 02:00:00 will jump back to 2016-11-06 01:00:00, so 2016-11-06 01:30:00 correspond to **two** UTC time points.
  - So a local time can be either **absent** or **ambiguous**!

Time zone

- There are four different kinds of time zone names, which are regulated by IANA (Internet Assigned Numbers Authority):
  1. City or country.
     - e.g. America/Chicago, Asia/Hong_Kong, Europe/Berlin, Pacific/Honolulu, etc.
  2. Offset: starts with Etc/, plus GMT+/-N.
     - e.g. Etc/GMT-8 is Beijing time (Yes, Etc/GMT**-8** is UTC**+8**, it needs negation for Unix convention on Etc/GMT).
  3. Abbreviations.
     - e.g. UTC, GMT, CST, PST.
  4. Deprecated entries (no deterministic forms).
     - e.g. PST8PDT, US/Hawaii, Canada/Central, Japan.

  - The unique names can be used to locate a time zone.

Time zone

- In C++, `stdc::time_zone` represents a time zone and `stdc::zoned_time` represents time attached to a time zone.
  - C++ provides a database for you to query a `const time_zone*`;
    1. `current_zone()`: get current time zone;
    2. `locate_zone(name)`: query time zone from name;
       - When no entry is located, `std::runtime_error` will be thrown.

```
auto tzHere = std::chrono::current_zone();
auto tzThere = std::chrono::locate_zone("Etc/GMT-8");
std::cout << tzHere->name() << '\n' << tzThere->name() << '\n';
```

```
Etc/UTC
Etc/GMT-8
```

  - And there are two ways to construct a `zoned_time`:
    1. Attach a local time to its zone;
    2. Convert a zoned time or system time to a zone.

Time zone

- For example, if we want to meet every Friday at 18:30 (Beijing time), but there are participants in Los Angeles.
  - The time schedule would be:

```cpp
auto begin = stdc::local_days{ 2026y / 1 / stdc::Friday[1] },
     end = stdc::local_days{ 2026y / 12 / stdc::Friday[stdc::last] };
auto zoneBeiJing = stdc::locate_zone("Asia/Shanghai"),
     zoneLA = stdc::locate_zone("America/Los_Angeles");
auto time = 18h + 30min;

for (; begin <= end; begin += stdc::weeks{ 1 })
{
    stdc::zoned_time timeBeiJing{ zoneBeiJing, begin + time },
                     timeLA{ zoneLA, timeBeiJing };
    std::println("We'll meet at:\n{}\n{}", timeBeiJing, timeLA);
}
```

1. Attach local time with a time zone; 2. Convert zoned time to another zone;

Windows

```
We'll meet at:
2026-01-02 18:30:00 GMT+8
2026-01-02 02:30:00 GMT-8
We'll meet at:
2026-01-09 18:30:00 GMT+8
2026-01-09 02:30:00 GMT-8
We'll meet at:
2026-01-16 18:30:00 GMT+8
2026-01-16 02:30:00 GMT-8
```

Linux

```
We'll meet at:
2026-01-02 18:30:00 CST
2026-01-02 02:30:00 PST
We'll meet at:
2026-01-09 18:30:00 CST
2026-01-09 02:30:00 PST
We'll meet at:
2026-01-16 18:30:00 CST
2026-01-16 02:30:00 PST
```

Time zone

BTW you can also use `stdc::floor<stdc::seconds>(…)` instead of `stdc::time_point_cast`.

- Another example: get current time in current time zone.

Convert system time to zoned time

```
auto currTime = stdc::time_point_cast<stdc::seconds>(stdc::system_clock::now());
auto currLocalTime = stdc::zoned_time{ stdc::current_zone(), currTime };
std::println("Current time: {0} {0:%Z}, i.e. {1} in Beijing", currTime, currLocalTime);
```

```
Current time: 2026-03-21 07:15:06 UTC, i.e. 2026-03-21 15:15:06 GMT+8 in Beijing
```

- And you can also convert a `zoned_time` back to `local_time` and `sys_time`:

operator local_time **get_local_time**	obtains the stored time point as a local_time (public member function)
operator sys_time **get_sys_time**	obtains the stored time point as a sys_time (public member function)

Time zone

- Specifically, for ctor of `zoned_time`:
  - Each variant provides two overloads, by name (effectively call `locate_zone`) or `const time_zone*`.

  1. Attach local time to a time zone:

```
zoned_time( TimeZonePtr z, const std::chrono::local_time<Duration>& tp );
```

```
zoned_time( std::string_view name, const std::chrono::local_time<Duration>& tp );
```

```
zoned_time( TimeZonePtr z, const std::chrono::local_time<Duration>& tp,
            std::chrono::choose c );
```

```
zoned_time( std::string_view name,
            const std::chrono::local_time<Duration>& tp, std::chrono::choose c );
```

What's this?

  - We've said that a local time can be ambiguous or absent in DST.
  - `enum class choose` has `earliest` and `latest` options, which selects the earliest one or the latest one for an ambiguous time.
  - Ambiguous time without `choose` param, or absent time will throw exception.

nonexistent_local_time (C++20)	exception thrown to report that a local time is nonexistent (class)
ambiguous_local_time (C++20)	exception thrown to report that a local time is ambiguous (class)

Time zone

2. Convert from sys time:

```
zoned_time( const std::chrono::sys_time<Duration>& st );
zoned_time( TimeZonePtr z, const std::chrono::sys_time<Duration>& st );
```

When time zone is not provided, UTC will be used as default.

```
zoned_time( std::string_view name, const std::chrono::sys_time<Duration>& st );
```

3. Convert from zoned time:

```
template< class Duration2, class TimeZonePtr2 >
zoned_time( TimeZonePtr z,
            const std::chrono::zoned_time<Duration2, TimeZonePtr2>& zt );
```

```
template< class Duration2, class TimeZonePtr2 >
zoned_time( std::string_view name,
            const std::chrono::zoned_time<Duration2, TimeZonePtr2>& zt );
```

- Actually it also provides variants with `stdc::choose` as the last param, but such param has no effect.

Time zone

```
template< class Duration2 >
zoned_time( const std::chrono::zoned_time<Duration2, TimeZonePtr>& other );
```

- And finally some naïve ctors:
  - Equiv. to default constructed `time_point`, i.e. `.time_since_epoch()` is zero. Again, absent time zone is UTC.

```
zoned_time();
zoned_time( const zoned_time& other ) = default;
explicit zoned_time( TimeZonePtr z );
explicit zoned_time( std::string_view name );
```

- For assignment:

```
zoned_time& operator=( const zoned_time& other ) = default;
zoned_time& operator=( const std::chrono::sys_time<Duration>& other );
zoned_time& operator=( const std::chrono::local_time<Duration>& other );
```

  - The last two will keep the original time zone.
  - And since `local_time` doesn't have variant with `choose`, it may throw exception.

Time zone

- And some other methods for `zoned_time`:
  - `.get_time_zone() -> const time_zone*;`
  - `.get_info() -> stdc::sys_info`: get information of time zone for current time.
    - It contains:

Member object	Type
begin, end	std::chrono::sys_seconds
offset	std::chrono::seconds
save	std::chrono::minutes
abbrev	std::string

1. In what range this offset is effective.
2. Offset with system time.
3. Possible DST offset.
4. Abbr. of time zone.

    - It can be printed by `operator<</format` directly.
    - For example:

Reason: China used DST from 1986 to 1991.

Los Angeles, in DST.
```
begin: 2026-03-08 10:00:00, end: 2026-11-01 09:00:00, offset: -25200s, save: 60min, abbrev: GMT-7
```
LA, out of DST.
```
begin: 2026-11-01 09:00:00, end: 2027-03-14 10:00:00, offset: -28800s, save: 0min, abbrev: GMT-8
```
Beijing, no DST.
```
begin: 1991-09-14 17:00:00, end: 32767-12-31 23:59:59, offset: 28800s, save: 0min, abbrev: GMT+8
```
Beijing, with DST. (should be GMT+9, but Windows seems wrong).
```
begin: 1988-04-16 18:00:00, end: 1988-09-10 17:00:00, offset: 28800s, save: 0min, abbrev: GMT+8
```

While Linux is correct: `[1988-04-16 18:00:00,1988-09-10 17:00:00,09:00:00,60min,CDT]`

Time zone

- Equal comparable: compare both time point and time zone pointer;
- Hashable since C++26;
- `operator<<` and `format`: same as outputting the local time, except that:
  1. `%Z` will output abbr. of time zone, which is system-dependent;
  2. `%z` will output offset of time zone in [+/-]HH[MM];
     - E.g. Beijing is +0800 (or equivalently 08, +08); in previous sections it's just +0000 (or 00, +00, -00) for UTC clock, TAI clock, etc.
  3. `%Oz, %Ez` will use [+/-]H[H][:MM], plus other locale-dependent transformation.
     - E.g. Beijing is +08:00 (or +8, +08, 8, 08); in previous sections it's +00:00.
  4. The default format is `%F %T %Z`.
     - E.g. 2026-03-21 15:15:06 GMT+8.
- And finally a member type:
  And a type alias:

Member type	Definition
duration	`std::common_type_t<Duration, std::chrono::seconds>`

```
using zoned_seconds = std::chrono::zoned_time<std::chrono::seconds>;
```

Time zone

- Generally, `zoned_time` isn't directly input; but you can input a local time by `from_stream` and construct `zoned_time`.

```
template< class CharT, class Traits, class Duration, class Alloc = std::allocator<CharT> >
std::basic_istream<CharT, Traits>&
    from_stream( std::basic_istream<CharT, Traits>& is, const CharT* fmt,
                 std::chrono::local_time<Duration>& tp,
                 std::basic_string<CharT, Traits, Alloc>* abbrev = nullptr,
                 std::chrono::minutes* offset = nullptr );
```

  - When `%Z` is provided, `abbrev` will be filled; when `%z/Oz/Ez` is provided, `offset` will be filled.
    - `%Z` takes the longest sequence of characters in regex `[\w|-|+|/]` (i.e. A-Z, a-z, 0-9, _, -, +, /).
  - For example:

```
stdc::local_seconds localTime;
std::string timezoneName;
stdc::minutes offset;

std::istringstream is{ "2025-1-21 15:30:05 Asia/Shanghai +0800"};
is >> stdc::parse("%F %T %Z %z", localTime, timezoneName, offset);
std::println("{} {} {}", localTime, timezoneName, offset);
```

```
2025-01-21 15:30:05 Asia/Shanghai 480min
```

Note: `stdc::parse` accepts reference instead of pointer.

Time zone

- Note 1: `%Z` and `%z` are separately processed so it's possible that the offset is not aligned with the time zone name.
- Note 2: unlike other specifiers, `%Z` is not essentially inversible.
  - That is, usually name output by `%Z` cannot locate a time zone. Two cases:
  1. It's a pseudo name that's not a time zone.
     - E.g. GPS, TAI.
  2. `locate_zone()` uses **name** to locate, but `%Z` outputs **abbreviation**.
     - E.g. our previous example: `2026-03-21 15:15:06 GMT+8`
     - GMT+8 is not a valid time zone name.
     - Only occasionally, abbr. is also a **unique** name (like UTC).
  - Solution: output `.name()` when formatting, or scan the time zone database to let users determine (covered later).

Time zone

- Some final notes…
- Note 1: Methods of `time_zone`:
  - Most of methods in `zoned_time` just call them.
  - And of course, `to_sys` has a `choose` param variant.
  - You can see that `time_zone` also provides `.get_info` for `local_time`…
    - It returns `local_info`, which describes the result of converting `local_time` to `sys_time`:
    - When `.result` is `ambiguous` or `nonexistent`, `first` is the `sys_info` for the previous zone and `second` is for the later one.

Member functions

`name`	obtains the name of this `time_zone` (public member function)
`get_info`	obtain information associated with a `sys_time` or `local_time` (public member function)
`to_sys`	converts a `local_time` in this time zone to a `sys_time` (public member function)
`to_local`	converts a `sys_time` to a `local_time` in this time zone (public member function)

Nonmember functions

`operator==` `operator<=>` (C++20)	compares two `time_zone` objects (function)

Member constants

Name	Value
`constexpr int` **`unique`** [static]	`0` (public static member constant)
`constexpr int` **`nonexistent`** [static]	`1` (public static member constant)
`constexpr int` **`ambiguous`** [static]	`2` (public static member constant)

Member objects

Member object	Type
`result`	`int`
`first`, `second`	`std::chrono::sys_info`

Time zone

- Note 2: to be rigorous, time zone pointers are part of template:

```
template<
    class Duration,
    class TimeZonePtr = const std::chrono::time_zone*
> class zoned_time;
```

  - And all previous claims that return `const stdc::time_zone*` essentially return `TimeZonePtr`.
  - Details are covered in homework for not being very important.
- Note 3: currently accessing time zone information in MS-STL is slightly slow, which may be fixed in the future.

Time zone database

- Finally, we'll introduce the underlying database.
  - C++ standard library regulates to use IANA timezone database, but OS may only adopt it partially.
  - It can even be absent in your OS; for example, an embedded system may not care about zone at all so the tailored OS removes it.
    - Particularly, Windows before Win10 just has no such database.
- This leads to facts that:
  1. When the database doesn't exist, `std::runtime_error` will be thrown in `locate_zone`, etc.
  2. For a long-running program, the database can be obsolete and gives wrong time.
     - For example, new leap second and new time zone may be absent.

Time zone database

- Essentially, the program maintains a time zone list.
  - You can get it by `stdc::get_tzdb_list() -> stdc::tzdb_list`.
  - Every element in the list is a database `stdc::tzdb`.
- Previous `locate_zone()` and `current_zone()` just use `tzdb` at the head of the list (equiv. to `stdc::get_tzdb()`).
  - You can update the head by `stdc::reload_tzdb()`, which will compare the version with `stdc::remote_version()`.
    - If the head version is older, the latest one got from remote will be **inserted** at the front of the list.
  - Then for a long-running program, calling `reload_tzdb` can make `locate_zone` up-to-date without updating the OS.
  - So…why does the standard library maintain a list?

Time zone database

- Reason: there may exist `time_zone*` that refers to entries in the old database in the program.
  - Removing it directly can cause illegal access.
  - But you can do it manually by `tzdb_list` methods:

`front`	access the first element (public member function)
`erase_after`	erases an element after an element (public member function)
`begin` `cbegin`	returns an iterator to the beginning of the list (public member function)
`end` `cend`	returns an iterator past the end of the list (public member function)

- So these statements are equivalent:

```
auto z1 = stdc::current_zone();
auto z2 = stdc::get_tzdb().current_zone();
auto z3 = stdc::get_tzdb_list().front().current_zone();
```

Time zone database

`name()` is the alternative name, and `target()` is the original name.

- Finally members of `tzdb`:

Member objects

Member object	Description
`version`	A `std::string` that contains the version of the database
`zones`	A sorted `std::vector<std::chrono::time_zone>` containing description of time zones
`links`	A sorted `std::vector<std::chrono::time_zone_link>` containing description of alternative names of time zones (links)
`leap_seconds`	A sorted `std::vector<std::chrono::leap_second>` containing description of leap seconds

Member functions

`locate_zone`	locate a time zone with the given name (public member function)
`current_zone`	return the local time zone (public member function)

- `stdc::leap_second` contains information of when leap second happened:

Member functions

`date`	obtains the time of leap second insertion (public member function)

`->sys_seconds`

Nonmember functions

`operator==` **`operator<`** **`operator<=`** **`operator>`** **`operator>=`** **`operator<=>`**	(C++20)	compares two leap_secor (function template)

Helper classes

`std::hash``<std::chrono::leap_second>`	(C++26)	hash s (class te

Nonmember f

`operator==` **`operator<=>`**	(C++20)

`name`	accesses the
`target`	(public member

- Solution: output `.name()` when formatting, or scan the time zone database to let users determine (covered later).

- So we can write a program to check all time zones with some abbr.:

```cpp
std::string abbrev{ "GMT+8" };

auto day = stdc::sys_days{ 2021y / 1 / 1 };
auto& db = stdc::get_tzdb();
std::cout << db.version << " " << stdc::remote_version() << "\n";

// print time and name of all timezones with abbrev:
std::cout << day << " UTC maps to these '" << abbrev << "' entries:\n";
// iterate over all timezone entries:
for (const auto& z : db.zones) {
    if (z.get_info(day).abbrev == abbrev) {
        stdc::zoned_time zt{ &z, day };
        std::cout << "  " << zt << "  " << z.name() << '\n';
    }
}
std::cout << "And these links:\n";
for (const auto& link : db.links) {
    if (link.name() == abbrev) {
        stdc::zoned_time zt{ link.target(), day };
        std::cout << "  " << zt << "  " << link.target() << '\n';
    }
}
```

```
2022g.27 2022g.27
2021-01-01 UTC maps to these 'GMT+8' entries:
  2021-01-01 08:00:00 GMT+8  Asia/Brunei
  2021-01-01 08:00:00 GMT+8  Asia/Choibalsan
  2021-01-01 08:00:00 GMT+8  Asia/Hong_Kong
  2021-01-01 08:00:00 GMT+8  Asia/Irkutsk
  2021-01-01 08:00:00 GMT+8  Asia/Kuala_Lumpur
  2021-01-01 08:00:00 GMT+8  Asia/Kuching
  2021-01-01 08:00:00 GMT+8  Asia/Macau
  2021-01-01 08:00:00 GMT+8  Asia/Makassar
  2021-01-01 08:00:00 GMT+8  Asia/Manila
  2021-01-01 08:00:00 GMT+8  Asia/Shanghai
  2021-01-01 08:00:00 GMT+8  Asia/Singapore
  2021-01-01 08:00:00 GMT+8  Asia/Taipei
  2021-01-01 08:00:00 GMT+8  Asia/Ulaanbaatar
  2021-01-01 08:00:00 GMT+8  Australia/Perth
  2021-01-01 08:00:00 GMT+8  Etc/GMT-8
And these links:
```

For `abbrev = "CST"`:

```
2022g.27 2022g.27
2021-01-01 UTC maps to these 'CST' entries:
And these links:
  2020-12-31 18:00:00 GMT-6  America/Chicago
```

The specific list is system-dependent.

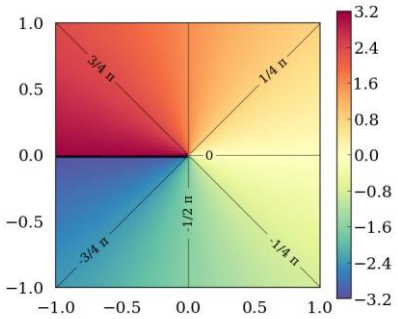
Supplementary and Summary

Math functions

Supplementary

- Math functions
  - Miscellaneous topics
  - Random number

Math functions



- Just list a table (in `<cmath>`) with special notes:

Class	Functions	Notes
Trigonometric 三角函数	`sin, cos, tan, asin, acos, atan, atan2`	`atan2(y,x)` distinguishes y/x and $-y/-x$.
Hyperbolic 双曲函数	`sinh, cosh, tanh, asinh, acosh, atanh`	N/A
Exponential & Logarithm 指数/对数函数	`exp, exp2, expm1, log, log2, log1p, log10, logb`	`expm1` means $e^x - 1$ (exp minus 1), `log1p` means $\ln(x+1)$ (ln 1 plus), `logb` means `floor(log2(x))`.
Power 幂函数	`pow, sqrt, cbrt, hypot`	`cbrt` means $\sqrt[3]{x}$, `hypot(x,y[,z])` means hypotenuse (斜边) $\sqrt{x^2+y^2(+z^2)}$
Rounding 舍入	`ceil, floor, trunc, round, nearbyint, rint`	`trunc(2.1) == 2, trunc(-2.1) == -2`; `round` doesn't consider current rounding mode but `nearbyint` does; `rint == nearbyint` except for raising error when integer is not exactly representable in current float.

Math functions

std::div_t

```
struct div_t { int quot; int rem; };
```

or

```
struct div_t { int rem; int quot; };
```

Not frequently used now since compilers can combine division and modulo automatically to save instructions.

Class	Functions	Notes
Division 除法	`fmod`, `remainder`, `remquo`, `div`	Let `quo = x / y`, `remainder` is `x - round(quo)*y`, `fmod` is `x-trunc(quo)*y`. `remquo(x, y, *quo)` can be used for periodic functions. `div(x, y) -> div_t` is for integer to get both remainder and quotient.
(De)composition 分解与组合	`frexp`, `ldexp`, `scalbn`, `modf`	`frexp(x, *e) -> m`: $x = m * 2^e, m \in [0.5, 1)$; `ldexp(m, e)`: $x = m * 2^e$ (`scalbn` is same, except for n-bit system that $n \neq 2$); `modf(x, *intptr) -> frac`: $x = \text{int} + \text{frac}$.
Classification 分类	`fpclassify`, `isfinite`, `isinf`, `isnan`, `isnormal`	`normal`: 规格化数。 `fpclassify`: return integer for class.
Bit operations 比特操作	`signbit`, `copysign`, `nextafter`, `nexttoward`	`signbit(x) -> bool`; `copysign(x, y)`: `x` copies sign from `y`; `nextafter(from, to)`: return next representable number towards `to`. Minor difference with `nexttoward`.

Math functions

Given $N(\mu, \sigma^2)$, $\Pr[L_a \le X \le L_b] = \int_{L_a}^{L_b} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \mathrm{d}x = \frac{1}{2}\left(\mathrm{erf}\left(\frac{L_b-\mu}{\sqrt{2}\sigma}\right) - \mathrm{erf}\left(\frac{L_a-\mu}{\sqrt{2}\sigma}\right)\right)$, $\Pr[X \le L] = \frac{1}{2}\left(\mathrm{erf}\left(\frac{L-\mu}{\sqrt{2}\sigma}\right) + 1\right)$

Class	Functions	Notes
Comparison 比较	`is(greater/less)(equal)`, `islessgreater`, `isunordered`	N/A
Other-1 其他-1	`erf`, `erfc`, `tgamma`, `lgamma`	$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} \mathrm{d}t$, which is integral of Gaussian Distribution $N(0, 0.5)$ in $[-x, x]$ (i.e. probability of happening in $[-x, x]$). $\mathrm{erfc}(x) = 1 - \mathrm{erf}(x)$. `tgamma` is gamma function $\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} \mathrm{d}t$, `lgamma` is $\ln\lvert\Gamma(x)\rvert$.
Other-2 其他-2	`fmin`, `fmax`, `abs`, `fdim`, `fma`, `nan`	`fdim(x, y) = max(0, x-y)`; `fma(x, y, z) = x * y + z`; `nan(const char* arg)` converts a string to `NaN`; the string is used to determine the specific binary (e.g. `nan("1")` is `0x7ff0000000000001`).

Math functions

- Note 1: these functions often correspond to special instructions, which will be applied by either the standard library or the compiler optimization.
  - For example, `1.0f / sqrtf(x)` will be optimized to `RSQRTSS` in Intel;
  - Sometimes performance loss may even occur…
    1. GNU extension provides `sincos`, which is historically faster for `fsincos` instruction in x87 architecture; but it's slower than separate computation now.
    2. `ldexp(m, e)` (i.e. $x = m * 2^e$) can be slower than `m * (1 << e)`, when SIMD instructions are used.
- Note 2.1: almost all functions have variations with suffix `f` or `l`, meaning for `float` or `long double`.
  - E.g. `sinf` for `float`, `sinl` for `long double`.
  - `scalbn` is slightly special (though it's rarely used):

```
float       sinf( float num );
long double sinl( long double num );
```

```
template< class Integer >
double scalbn( Integer num, int exp );
template< class Integer >
double scalbln( Integer num, long exp );
```

Math functions

- These functions exist for compatibility with `<math.h>` in C.
  - C doesn't have function overloading, so it has to provide different names (no suffix means for `double`).
  - C++ actually adds overloads:

    For extended floating-point numbers in C++23 `<stdfloat>`.

    Defined in header <cmath>

```
float       sin ( float num );
double      sin ( double num );                          (until C++23)
long double sin ( long double num );              (1)
/*floating-point-type*/                                  (since C++23)
            sin ( /*floating-point-type*/ num );         (constexpr since C++26)
```

    Additional overloads (since C++11)

    Defined in header <cmath>

```
template< class Integer >
double      sin ( Integer num );                  (A)    (constexpr since C++26)
```

    - But suffix ones can still be used when you need explicit type control.
- Note 2.2: some functions have variations for with prefix `i`, `l` or `ll`, meaning for returning `int`, `long` or `long long`.
  - For example, `logb` returns `floor(log2(x))`, `ilogb` returns `int(log2(x))`.
  - Since it differs on return type, overloading cannot help here.

Math functions

- A thorough example:
  - But most of functions don't have all these permutations of prefix and suffix…

Rounding to floating-point types

```
float       round ( float num );
double      round ( double num );
long double round ( long double num );                                  (1)
constexpr /* floating-point-type */
            round ( /* floating-point-type */ num );
float       roundf( float num );                                        (2)
long double roundl( long double num );                                  (3)
```

(1): (since C++11) (until C++23); (since C++23)
(2): (since C++11) (constexpr since C++23)
(3): (since C++11) (constexpr since C++23)

Rounding to `long`

```
long lround ( float num );
long lround ( double num );
long lround ( long double num );                                        (4)
constexpr long lround( /* floating-point-type */ num );
long lroundf( float num );                                              (5)
long lroundl( long double num );                                        (6)
```

(4): (since C++11) (until C++23); (since C++23)
(5): (since C++11) (constexpr since C++23)
(6): (since C++11) (constexpr since C++23)

Rounding to `long long`

```
long long llround ( float num );
long long llround ( double num );
long long llround ( long double num );                                  (7)
constexpr long long llround( /* floating-point-type */ num );
long long llroundf( float num );                                        (8)
long long llroundl( long double num );                                  (9)
```

(7): (since C++11) (until C++23); (since C++23)
(8): (since C++11) (constexpr since C++23)
(9): (since C++11) (constexpr since C++23)

Additional overloads

Defined in header `<cmath>`

```
template< class Integer >
double round( Integer num );                                            (A)
template< class Integer >
long lround( Integer num );                                             (B)
template< class Integer >
long long llround( Integer num );                                       (C)
```

(A): (since C++11) (constexpr since C++23)
(B): (since C++11) (constexpr since C++23)
(C): (since C++11) (constexpr since C++23)

Math functions

- Note 2.3: it's recommended to fully qualify function names (i.e. `std::xxx`) to prevent ambiguity with C functions.
  - For example: what does `abs(-3.1416)` return?
  - Oops, it depends on calling C++ function or C function.
    - C doesn't have `abs(float)` and introduces `fabs`!
  - So `abs(-3.1416)` can be `3` or `3.1416`.
  - Just use `std::abs`!

abs(float) C++ only
abs(int) **fabs**
labs **fabsf**
llabs **fabsl**

- Note 3: there also exist some constants, but `std::numeric_limits` in `<limits>` should be used in C++ instead.

HUGE_VALF (C++11) **HUGE_VAL** **HUGE_VALL** (C++11)	indicates the overflow value for `float`, `double` and `long double` respectively (macro constant)
INFINITY (C++11)	evaluates to positive infinity or the value guaranteed to overflow a `float` (macro constant)
NAN (C++11)	evaluates to a quiet NaN of type `float` (macro constant)

Math functions

```
#define MATH_ERRNO        1                          (since C++11)
#define MATH_ERREXCEPT    2                          (since C++11)
#define math_errhandling  /*implementation defined*/  (since C++11)
```

The macro constant `math_errhandling` expands to an expression of type `int` that is either equal to `MATH_ERRNO`, or equal to `MATH_ERREXCEPT`, or equal to their bitwise OR (`MATH_ERRNO | MATH_ERREXCEPT`).

- Note 4: error handling of math functions is by either floating-point exception or errno (specific way determined by macro).

The following floating-point error conditions are recognized:

Condition	Explanation	errno	Floating-point exception	Example
Domain error	The argument is outside the range in which the operation is mathematically defined (the description of each function lists the required domain errors)	`EDOM`	`FE_INVALID`	`std::acos(2)`
Pole error	The mathematical result of the function is exactly infinite or undefined	`ERANGE`	`FE_DIVBYZERO`	`std::log(0.0)`, `1.0 / 0.0`
Range error due to overflow	The mathematical result is finite, but becomes infinite after rounding, or becomes the largest representable finite value after rounding down	`ERANGE`	`FE_OVERFLOW`	`std::pow(DBL_MAX, 2)`
Range error due to underflow	The result is non-zero, but becomes zero after rounding, or becomes subnormal with a loss of precision	`ERANGE` or unchanged (implementation-defined)	`FE_UNDERFLOW` or nothing (implementation-defined)	`DBL_TRUE_MIN / 2`
Inexact result	The result has to be rounded to fit in the destination type	Unchanged	`FE_INEXACT` or nothing (unspecified)	`std::sqrt(2)`, `1.0 / 10.0`

floating-point exception can be tested by `fetestexcept` (details not covered here).

Math functions

- Note 5: most of these functions become `constexpr` since C++23.

- Note 6: most of these functions add explicit SIMD overloads since C++26.
  - Since C++26 introduces `<simd>`; but we don't cover it here.

SIMD overload (since C++26)

Defined in header `<simd>`

```
template< /*math-floating-point*/ V >
constexpr /*deduced-simd-t*/<V>                    (S)    (since C++26)
          exp ( const V& v_num );
```

- Note 7: some special notes on numeric precision…
  - Before that, we first introduce **u**nit in the **l**ast **p**lace (ULP).

ULP

- ULP means spacing around current floating points.
  - i.e. `ULP(x) = nextafter(x, +inf) - x` for `x > 0`;
    `ULP(x) = x - nextafter(x, -inf)` for `x < 0`;
  - Under IEEE 754, that means:
    - Assuming floating-point has `p` bits for mantissa, and exponent of `x` is `e`;
    - For normal numbers, ULP is 2^{e-p};
    - For denormal numbers, ULP is 2^{e-p+1}.
  - Say we want to compute ULP for `float x = 1.0f`.
    - 32-bit `float` has 23 bits for mantissa; exponent of `x` is `0` (since `1.0` is 1.0×2^0).
    - So ULP(1.0f) is 2^{-23}.
- Using ULP, we can describe computation error for finite precision of floating points.

Catch2 provides `WithinULP(x, d)` for checking error to be within $d \cdot \mathrm{ULP}(x)$.

```
REQUIRE_THAT( nextafter( 1.f, 2.f ), WithinULP( 1.f, 1 ) );
```

ULP

- IEEE-754 guarantees some errors to be within 0.5 ULP:
  - Rounding: given any value x, converting it to floating-point `x` will choose the closest representable value, i.e. error in `0.5 * ULP(x)`.
  - Plus: $|(a \oplus b) - (a + b)| \in 0.5\ \mathrm{ULP}(a + b)$.
    - That is, given computation result of `float a + float b` (denoted as $a \oplus b$), and the real value $a + b$ (also called infinite-precision value), their space is no more than $0.5\ \mathrm{ULP}(a + b)$.
    - Of course, it assumes a, b are already exactly representable.
  - Similarly, minus, multiplication and division are in $0.5\ \mathrm{ULP}(a + b)$ too.
- Most importantly, IEEE-754 requires `sqrt` and `fma` to be in $0.5\ \mathrm{ULP}$ too.
  - So when you have precision requirement, use `fma` instead of $a \otimes b \oplus c$.

FYI, you can check some extreme error care in graphics.

Math functions

- Note 8: some functions add computation for complex number in `<complex>`.
  - `std::complex<T>` is generally like a wrapper for `T real, imag`; details not covered here.

Exponential functions

exp(std::complex)	complex base *e* exponential (function template)
log(std::complex)	complex natural logarithm with the branch cuts along the negative real axis (function template)
log10(std::complex)	complex common logarithm with the branch cuts along the negative real axis (function template)

Power functions

pow(std::complex)	complex power, one or both arguments may be a complex number (function template)
sqrt(std::complex)	complex square root in the range of the right half-plane (function template)

Trigonometric functions

sin(std::complex)	computes sine of a complex number (*sin(z)*) (function template)
cos(std::complex)	computes cosine of a complex number (*cos(z)*) (function template)
tan(std::complex)	computes tangent of a complex number (*tan(z)*) (function template)
asin(std::complex) (C++11)	computes arc sine of a complex number (*arcsin(z)*) (function template)
acos(std::complex) (C++11)	computes arc cosine of a complex number (*arccos(z)*) (function template)
atan(std::complex) (C++11)	computes arc tangent of a complex number (*arctan(z)*) (function template)

Hyperbolic functions

sinh(std::complex)	computes hyperbolic sine of a complex number (*sinh(z)*) (function template)
cosh(std::complex)	computes hyperbolic cosine of a complex number (*cosh(z)*) (function template)
tanh(std::complex)	computes hyperbolic tangent of a complex number (*tanh(z)*) (function template)
asinh(std::complex) (C++11)	computes area hyperbolic sine of a complex number (*arsinh(z)*) (function template)
acosh(std::complex) (C++11)	computes area hyperbolic cosine of a complex number (*arcosh(z)*) (function template)
atanh(std::complex) (C++11)	computes area hyperbolic tangent of a complex number (*artanh(z)*) (function template)

Methods for `std::complex`, quite easy.

Member functions

(constructor)	constructs a complex number (public member function)
operator=	assigns the contents (public member function)
real	accesses the real part of the complex number (public member function)
imag	accesses the imaginary part of the complex number (public member function)
operator+= operator-= operator*= operator/=	compound assignment of two complex numbers or a complex and a scalar (public member function)

```
std::complex<double> z4 = 1.0 + 2i, z5 = 1.0 - 2i; // conj
std::cout << "(1 + 2i) * (1 - 2i) = " << z4 * z5 << '\n';

const auto zz = {0.0 + 1i, 2.0 + 3i, 4.0 + 5i};
```

You can also use user-defined literals.

operator""if
operator""i (C++14)
operator""il

Non-member functions

operator+ operator-	applies unary operators to complex numbers (function template)
operator+ operator- operator* operator/	performs complex number arithmetic on two complex values or a complex and a scalar (function template)
operator== operator!= (removed in C++20)	compares two complex numbers or a complex and a scalar (function template)
operator<< operator>>	serializes and deserializes a complex number (function template)
get (std::complex) (C++26)	obtains a reference to real or imaginary part from a **std::complex** (function template)
real	returns the real part (function template)
imag	returns the imaginary part (function template)
abs (std::complex)	returns the magnitude of a complex number (function template)
arg	returns the phase angle (function template)
norm	returns the squared magnitude (function template)
conj	returns the complex conjugate (function template)
proj (C++11)	returns the projection onto the Riemann sphere (function template)
polar	constructs a complex number from magnitude and phase angle (function template)

Math functions

- Finally, C++17 adds many special math functions.
  - Details not covered here; just use it when your task needs.

assoc_laguerre (C++17) **assoc_laguerref** (C++17) **assoc_laguerrel** (C++17)	associated Laguerre polynomials (function)
assoc_legendre (C++17) **assoc_legendref** (C++17) **assoc_legendrel** (C++17)	associated Legendre polynomials (function)
beta (C++17) **betaf** (C++17) **betal** (C++17)	beta function (function)
comp_ellint_1 (C++17) **comp_ellint_1f** (C++17) **comp_ellint_1l** (C++17)	(complete) elliptic integral of the first kind (function)
comp_ellint_2 (C++17) **comp_ellint_2f** (C++17) **comp_ellint_2l** (C++17)	(complete) elliptic integral of the second kind (function)
comp_ellint_3 (C++17) **comp_ellint_3f** (C++17) **comp_ellint_3l** (C++17)	(complete) elliptic integral of the third kind (function)
cyl_bessel_i (C++17) **cyl_bessel_if** (C++17) **cyl_bessel_il** (C++17)	regular modified cylindrical Bessel functions (function)
cyl_bessel_j (C++17) **cyl_bessel_jf** (C++17) **cyl_bessel_jl** (C++17)	cylindrical Bessel functions (of the first kind) (function)
cyl_bessel_k (C++17) **cyl_bessel_kf** (C++17) **cyl_bessel_kl** (C++17)	irregular modified cylindrical Bessel functions (function)
cyl_neumann (C++17) **cyl_neumannf** (C++17) **cyl_neumannl** (C++17)	cylindrical Neumann functions (function)
ellint_1 (C++17) **ellint_1f** (C++17) **ellint_1l** (C++17)	(incomplete) elliptic integral of the first kind (function)
ellint_2 (C++17) **ellint_2f** (C++17) **ellint_2l** (C++17)	(incomplete) elliptic integral of the second kind (function)
ellint_3 (C++17) **ellint_3f** (C++17) **ellint_3l** (C++17)	(incomplete) elliptic integral of the third kind (function)
expint (C++17) **expintf** (C++17) **expintl** (C++17)	exponential integral (function)
hermite (C++17) **hermitef** (C++17) **hermitel** (C++17)	Hermite polynomials (function)
legendre (C++17) **legendref** (C++17) **legendrel** (C++17)	Legendre polynomials (function)
laguerre (C++17) **laguerref** (C++17) **laguerrel** (C++17)	Laguerre polynomials (function)
riemann_zeta (C++17) **riemann_zetaf** (C++17) **riemann_zetal** (C++17)	Riemann zeta function (function)
sph_bessel (C++17) **sph_besself** (C++17) **sph_bessell** (C++17)	spherical Bessel functions (of the first kind) (function)
sph_legendre (C++17) **sph_legendref** (C++17) **sph_legendrel** (C++17)	spherical associated Legendre functions (function)
sph_neumann (C++17) **sph_neumannf** (C++17) **sph_neumannl** (C++17)	spherical Neumann functions (function)

For example, graphics usually uses Spherical Harmonics, which can be calculated by `sph_legendre`.

Numbers

- C++20 adds many math constants in `<numbers>`.
  - namespace is `std::numbers`.
  - When `_v` is removed, it means `double` constant (e.g. `std::numbers::e`).
    - If you want `float` constant, you need to use e.g. `std::numbers::e_v<float>`.

e_v	the mathematical constant e (variable template)
log2e_v	$log_2 e$ (variable template)
log10e_v	$log_{10} e$ (variable template)
pi_v	the mathematical constant π (variable template)
inv_pi_v	$\frac{1}{\pi}$ (variable template)
inv_sqrtpi_v	$\frac{1}{\sqrt{\pi}}$ (variable template)
ln2_v	$ln\ 2$ (variable template)
ln10_v	$ln\ 10$ (variable template)
sqrt2_v	$\sqrt{2}$ (variable template)
sqrt3_v	$\sqrt{3}$ (variable template)
inv_sqrt3_v	$\frac{1}{\sqrt{3}}$ (variable template)
egamma_v	the Euler–Mascheroni constant γ (variable template)
phi_v	the golden ratio Φ ($\frac{1+\sqrt{5}}{2}$) (variable template)

Side note

- Also a side note: C++26 adds some integer arithmetic utilities.
  - Saturation: in `<numeric>`; `saturating_xxx<T>(x, y) -> T` will *saturate* computation of x and y, so when the result is out-of-bound, it will be **clamped** to representable value.
    - `xxx` can be `add, sub, mul, div`.
    - And `saturating_cast<T>(x) -> T`; by comparison, `(T)x` will modulo it. E.g. `(int8_t)-696` is `72`, while `saturating_cast<int8_t>(-696)` is `-128`.
    - Example: `(std::saturating_sub<int>(INT_MIN + 4, 5) == INT_MIN) // saturated`
  - Checked: in `<stdckdint.h>` (which is first accepted in C23); when the computation result of x and y is out-of-bound, `ckd_xxx(*result, x, y) -> bool` will not set `*result`, and will return `false`.
    - `xxx` can be `add, sub, mul`.

Supplementary

- Math functions
  - Miscellaneous topics
  - Random number

Random number

- C just provides a very simple function `rand` to get a pseudo-random integer.
  - And you can only manipulate it by setting seed using `srand`.
- But it has many drawbacks…
  1. Not thread-safe, so have to add lock to protect;
  2. Many math formula use special distributions; providing mere random integers isn't enough.
  3. The implemented algorithm, for backward compatibility, is very naïve and likely to generate low-quality random numbers.
     - That is, random number may encounter loop quickly.
- To overcome these problems, C++11 introduces `<random>`.

Random number

Implementations usually just use `/dev/urandom` or OS random library.
But you can use other strings in ctor, e.g. `/dev/random`, to specify explicitly.

- In physics, there are many “true” random process.
  - For example, whether a radioactive particle (放射性粒子) decays is completely random; we can only depict its overall behavior statistically by half-life (半衰期).
  - In computers, we may use noise of reverse-biased diodes (反向偏置二极管).
- C++ provides such a “true” random number generator `std::random_device`.
  - Typically, it gets random number from hardware device.
  - However, when such true number generator is absent, it can also be implemented as software-based pseudo-random number generator (PRNG).
  - You can check it by `.entropy() -> double`;
    - PRNG implementation “will” return 0 while others return positive value.

Random number

- But typically, performance of `std::random_device` will degrade sharply when called frequently (when it has to "flush the noise").
  - Thus, it's usually used to generate a seed for PRNG.
- C++ provides a bunch of PRNGs with different algorithms:

linear_congruential_engine (C++11)	implements linear congruential algorithm (class template)
mersenne_twister_engine (C++11)	implements Mersenne twister algorithm (class template)
subtract_with_carry_engine (C++11)	implements a subtract-with-carry (lagged Fibonacci) algorithm (class template)
philox_engine (C++26)	a counter-based parallelizable generator (class template)

  - We'll briefly introduce these engines later…

Random number

- These PRNGs generate random integers in uniform distribution;
  - But in scientific problems, we usually need random numbers that obey some **distribution**.
  - C++ also provides a lot of distributions that accept PRNGs, and transform the generated number to obey distribution.
    - We won't cover them in detail and check manual when you need.

Uniform distributions	
`uniform_int_distribution` (C++11)	produces integer values evenly distributed across a range (class template)
`uniform_real_distribution` (C++11)	produces real values evenly distributed across a range (class template)
Bernoulli distributions	
`bernoulli_distribution` (C++11)	produces `bool` values on a Bernoulli distribution (class)
`binomial_distribution` (C++11)	produces integer values on a binomial distribution (class template)
`negative_binomial_distribution` (C++11)	produces integer values on a negative binomial distribution (class template)
`geometric_distribution` (C++11)	produces integer values on a geometric distribution (class template)
Poisson distributions	
`poisson_distribution` (C++11)	produces integer values on a Poisson distribution (class template)
`exponential_distribution` (C++11)	produces real values on an exponential distribution (class template)
`gamma_distribution` (C++11)	produces real values on a gamma distribution (class template)
`weibull_distribution` (C++11)	produces real values on a Weibull distribution (class template)
`extreme_value_distribution` (C++11)	produces real values on an extreme value distribution (class template)
Normal distributions	
`normal_distribution` (C++11)	produces real values on a standard normal (Gaussian) distribution (class template)
`lognormal_distribution` (C++11)	produces real values on a lognormal distribution (class template)
`chi_squared_distribution` (C++11)	produces real values on a chi-squared distribution (class template)
`cauchy_distribution` (C++11)	produces real values on a Cauchy distribution (class template)
`fisher_f_distribution` (C++11)	produces real values on a Fisher's F-distribution (class template)
`student_t_distribution` (C++11)	produces real values on a Student's t-distribution (class template)
Sampling distributions	
`discrete_distribution` (C++11)	produces integer values on a discrete distribution (class template)
`piecewise_constant_distribution` (C++11)	produces real values distributed on constant subintervals (class template)
`piecewise_linear_distribution` (C++11)	produces real values distributed on defined subintervals (class template)

Random number

- So a commonly used pattern to generate random number is:

```
// Add thread_local if called by multiple threads concurrently.
static std::random_device device;
static std::default_random_engine engine{ device() };
// Generate U[-1, 1];
std::uniform_real_distribution<float> distribution{ -1.0f, 1.0f };
// Sample from the distribution using engine.
for (int i = 0; i < 10; i++)
    std::cout << distribution(engine) << " ";
```

1. Generate a device;
2. Get a seed to initialize PRNG;
3. Create a distribution;
4. Sample the distribution using PRNG.

- Now let’s cover a little bit of details about these utilities…
  - But we **won’t cover detailed algorithms**; if you have special requirements on quality of random number, check them yourself.
    - It’s really, really an expert-level work.

Random number

- For PRNG, they have common interface below:

Construction and Seeding

(constructor)	constructs the engine (public member function)
seed	sets the current state of the engine (public member function)

Generation

operator()	advances the engine's state and returns the generated value (public member function)
discard	advances the engine's state by a specified amount (public member function)

Characteristics

min [static]	gets the smallest possible value in the output range (public static member function)
max [static]	gets the largest possible value in the output range (public static member function)

Ctor: should be copyable; PRNG copies share the same internal state.

`op()`: generate a random integer in `[min, max]`, and advance to the next state.
`.discard(unsigned long long n)`: discard the next `n` random integers. Though naïvely it can be implemented by looping `op()` by `n` times, some engines may have better algorithm.

Non-member functions

operator== (C++11) **operator!=** (C++11)(removed in C++20)	compares the internal states of two pseudo-random number engines (function)
operator<< **operator>>** (C++11)	performs stream input and output on pseudo-random number engine (function template)

PRNG can be equality-compared and input/output (I/O format is impl-defined).

Random number

- Ctor and `.seed()` accept a seed to (re)set its state.
  - There are several variants:
  1. Accept a single integer, as we show before.
  2. Accept nothing, as if accept a default integer for `1.`.
  - But is it enough?

```
std::random_device device;
std::default_random_engine engine{ device() };
```

- NO.
  - Reason: an integer of M bits has only 2^M states; but some engines have much more state bits (e.g. for `std::mt19937`, it has 624 32-bit integers).
  - Therefore, a variant for accepting a seed range is needed. So:
  3. Accept a *`SeedSequence`*, which is capable of generating many integers.
     - This overload is a template, and the standard library already provides a class `std::seed_seq` that fulfills *`SeedSequence`*.

Random number

- For example:

```
// Generate a normal distribution around that mean
std::seed_seq seed2{r(), r(), r(), r(), r(), r(), r(), r()};
std::mt19937 e2(seed2);
std::normal_distribution<> normal_dist(mean, 2);
```

  - We give it 8 random 32-bit integers for seeding; though it's still far less than 624, it should be better than only a single integer.

- Therefore, *SeedSequence* is a mapping that:
  - Accepts M random bits;
  - Output N random bits that eliminate bias (as much as possible), which will be used to initialize PRNG status.

Random number

- `std::seed_seq` has:
  1. Ctor: initialized from a range of 32-bit integers;
  2. `.generate(begin, end)`: fill `[begin, end)` with random integers that eliminate bias.
  3. Get status of `v`.

Data members

Member	Description
`std::vector<result_type>` *v*	the underlying seed sequence (exposition-only member object*)

Member functions

(constructor)	constructs and seeds the `std::seed_seq` object (public member function)
operator= [deleted]	`std::seed_seq` is not assignable (public member function)
generate	calculates the bias-eliminated, evenly distributed 32-bit values (public member function)
size	obtains the number of stored 32-bit values (public member function)
param	copies all stored 32-bit values (public member function)

- Note: it doesn't guarantee a bijection when $M = N$ (i.e. the input status space is same as the output space). For instance, given two 32-bit integers, you cannot get every value of 64-bit integers.

Random number

```
template<
    class UIntType,
    UIntType a,
    UIntType c,
    UIntType m
> class linear_congruential_engine;
```

congruential: 同余

- The basic properties of PRNGs are:
  1. `std::linear_congruential_engine`: a.k.a. LCG, widely used in C `rand()` implementation; $x_{i+1} = (a \cdot x_i + c) \bmod m$, x_0 is seed.
     - Occupies least space (only one integer x_i), moderately fast, worst random quality.
  2. `std::mersenne_twister_engine`: Mersenne Twister algorithm.
     - Occupies most space (624 integers for `mt19937`), very fast, high random quality (cycle is $2^M - 1$ where M is Mersenne prime; in `mt19937` $M = 19937$).

```
template<
    class UIntType, std::size_t w, std::size_t n, std::size_t m, std::size_t r,
    UIntType a, std::size_t u, UIntType d, std::size_t s,
    UIntType b, std::size_t t, UIntType c, std::size_t l, UIntType f
> class mersenne_twister_engine;
```

Space is `n` integers of type `UIntType`.

  3. `std::subtract_with_carry_engine`: subtract with carry algorithm.
     - Occupies moderate space, moderately fast, moderate random quality.

```
template<
    class UIntType,
    std::size_t w, std::size_t s, std::size_t r
> class subtract_with_carry_engine;
```

Space is `r` integers of type `UIntType`, plus a flag (either 0 or 1).

Random number

- And some PRNG algorithms may need transformation on previous PRNGs, so there are three adaptors:
  1. `std::discard_block_engine`: for every `P` random numbers, keep only the first `R` random numbers.

```
template<
    class Engine,
    std::size_t P, std::size_t R
> class discard_block_engine;
```

  2. `std::independent_bits_engine`: generate a random number whose last `W` numeric bits are independent.
     - So it can create a larger range of random number, e.g. using a 32-bit engine to generate a 64-bit number.
     - The result is stored in `UIntType`, whose bits should be no less than `W`.

```
template<
    class Engine,
    std::size_t W,
    class UIntType
> class independent_bits_engine;
```

  3. `std::shuffle_order_engine`: shuffle the random number generated from the engine.
     - It keeps a table of size `K`, fill it will `K` random numbers from engine; then select from the table randomly as the next random number, and replace that item in the table with a new random number.

```
template<
    class Engine,
    std::size_t K
> class shuffle_order_engine;
```

It’s an expert-level task to select good parameters for them, so C++ provides lots of predefined aliases:

If you don’t have special requirements, just use `std::default_random_engine` (usually `std::mt19937` in implementations).

Type	Definition
`minstd_rand0` (C++11)	`std::linear_congruential_engine<std::uint_fast32_t, 16807, 0, 2147483647>` Discovered in 1969 by Lewis, Goodman and Miller, adopted as "Minimal standard" in 1988 by Park and Miller
`minstd_rand` (C++11)	`std::linear_congruential_engine<std::uint_fast32_t, 48271, 0, 2147483647>` Newer "Minimum standard", recommended by Park, Miller, and Stockmeyer in 1993
`mt19937` (C++11)	`std::mersenne_twister_engine<std::uint_fast32_t, 32, 624, 397, 31, 0x9908b0df, 11, 0xffffffff, 7, 0x9d2c5680, 15, 0xefc60000, 18, 1812433253>` 32-bit Mersenne Twister by Matsumoto and Nishimura, 1998
`mt19937_64` (C++11)	`std::mersenne_twister_engine<std::uint_fast64_t, 64, 312, 156, 31, 0xb5026f5aa96619e9, 29, 0x5555555555555555, 17, 0x71d67fffeda60000, 37, 0xfff7eee000000000, 43, 6364136223846793005>` 64-bit Mersenne Twister by Matsumoto and Nishimura, 2000
`ranlux24_base` (C++11)	`std::subtract_with_carry_engine<std::uint_fast32_t, 24, 10, 24>`
`ranlux48_base` (C++11)	`std::subtract_with_carry_engine<std::uint_fast64_t, 48, 5, 12>`
`ranlux24` (C++11)	`std::discard_block_engine<std::ranlux24_base, 223, 23>` 24-bit RANLUX generator by Martin Lüscher and Fred James, 1994
`ranlux48` (C++11)	`std::discard_block_engine<std::ranlux48_base, 389, 11>` 48-bit RANLUX generator by Martin Lüscher and Fred James, 1994
`knuth_b` (C++11)	`std::shuffle_order_engine<std::minstd_rand0, 256>`
`default_random_engine` (C++11)	an implementation-defined *RandomNumberEngine* type

Random number

- Previous PRNGs are all iterative, i.e. $x_{i+1} = f(x_i)$ to get next random number.
  - However, it significantly restricts usage of random number in parallel.
  - To overcome that, one way is to use e.g. `thread_local` PRNGs…
- But, such approach still has many drawbacks:
  1. Initializing PRNG is often costly, but every thread has to do so;
  2. Weak reproducibility when you add more threads.
     - A new thread will introduce a completely different sequence and leads to different numeric results.
  3. Multiple good seeds need to be provided.
- Instead, what if we can provide a PRNG with $x_i = f(i)$?
  - Providing a task id for every thread is a very common pattern in multi-threading tasks, so such PRNG makes the random number easy to get.

Random number

```
template<
    class UIntType, std::size_t w, std::size_t n, std::size_t r,
    UIntType... consts
>
class philox_engine;
```

- C++26 adds a counter-based PRNG `std::philox_engine`.
  - This engine is also widely used in Pytorch on different devices.
  - Its cycle is $n * 2^{n*\text{sizeof}(\text{UIntType})}$.
- `philox_engine` has state as:
  1. An integer counter; to represent a long integer, an array of size `std::array<UIntType, n>` is used.
  2. An integer seed with half bit length, i.e. an array of size `n / 2`.
  3. Buffer `b` for generated result with same bit length, i.e. an array of size `n`.
  4. An index `i` to split the buffer and get next random number properly.
- Its `operator() -> UIntType` returns `b[i]` and increases `i`.
  - When `i` reaches `n`, the philox algorithm runs to generate next `b`, and the counter is increased by 1.

Random numbe

Defined in header <random>

Type	Definition
philox4x32 (C++26)	std::philox_engine<std::uint_fast32_t, 32, 4, 10, 0xCD9E8D57, 0x9E3779B9, 0xD2511F53, 0xBB67AE85>
philox4x64 (C++26)	std::philox_engine<std::uint_fast64_t, 64, 4, 10, 0xCA5A826395121157, 0x9E3779B97F4A7C15, 0xD2E7470EE14C6C93, 0xBB67AE8584CAA73B>

- For example:

```
uint32_t global_seed = 999;
std::philox4x32 eng(global_seed);
std::normal_distribution rnd;
auto n1 = rnd(eng), n2 = rnd(eng);
```

1. Ctor: Need 4 / 2 = 2 integers of 32 bits as seed; we provide one and the other will be set as 0.
2. Counter: initialized as all 0.
3. `operator()`: first generate four 32-bit integers, and consume two here.
   - If we consume 5 integers, then generating the 5^{th} integer needs to (1) increase the counter (2) generate the next four 32-bit integers.

- Now it can provide $4 \cdot 2^{4\times32}$ random numbers.
  - So how does it facilitate parallel tasks?
  - C++ provides `.set_counter()` to offset the counter, so you can divide the numbers into any chunk!

Random number

For algorithm details, check this blog or the original paper.

- For example, assuming our loop below is parallelized:

Every sub-task occupies the sequence headed by `{atom_id, time_step}`, so it can freely consume $4 \cdot 2^{2 \times 32}$ random numbers.

```
uint32_t global_seed = 999;
for(uint32_t time_step = 0; time_step < time_steps_num; ++time_step) {
    for(uint32_t atom_id = 0; atom_id < atoms_num; ++atom_id) {
        std::philox4x32 eng(global_seed);
        eng.set_counter({atom_id, time_step, 0, 0});
        std::normal_distribution rnd;
        auto n1 = rnd(eng), n2 = rnd(eng);
        // ...
    }
}
```

- Though other PRNGs can `.discard` to achieve similar effects, they consume $O(\log n)$ or $O(n)$ where n is jumping distance; `philox_engine` does so in $O(1)$.

Final Notes

1. About reproducibility:
   - PRNG algorithms are strictly regulated, so as long as the seed is the same, **PRNG generates the same sequence in all platforms**.
   - However, distribution algorithms are not regulated, so even if the PRNG is the same, **distributions can generate different sequences in different platforms**.
   - If you want to ensure the same random values everywhere, `<random>` is not enough.
2. Cryptographical security (密码安全) is not concerned in `<random>`.
   - Cryptographical security means that, even if you know the PRNG algorithm, and you know some part of the sequence, you're still unable to know the rest of the sequence, or guess PRNG seed in polynomial complexity.
   - You should use e.g. OpenSSL for such random sequence.

Final Notes

```
template< class RealType, std::size_t Bits, class Generator >
RealType generate_canonical( Generator& g );
```

3. Some uncovered utilities:
   - `generate_canonical()`: given a PRNG, generate a random floating-point number in $[0, 1)$ with at most `Bits` (but no more than `sizeof(RealType)`) randomness.
     - Implementations use PRNG to generate random bits.
     - For example, program below generates `double` with 10-bit randomness (i.e. at most 2^{10} possible values).

```
std::random_device rd;
std::mt19937 gen(rd());
for (int n = 0; n < 10; ++n)
    std::cout << std::generate_canonical<double, 10>(gen) << ' ';
```

   - And some random-related algorithms in `<algorithm>`:
     1. `shuffle`: shuffle the random-access range with a PRNG.
        - URBG means uniform random bits generator, which is equiv. to PRNG in C++ since they generate integers uniformly.

```
template< class RandomIt, class URBG >
void shuffle( RandomIt first, RandomIt last, URBG&& g );
```

Random number

Details of random algorithms are also not strictly regulated, so not reproducible in different platforms.

2. `sample`: since C++17, select `n` elements from `[first, last)` randomly to `out`, each element has equal probability of appearance.
   - Every element will appear at most once in the output range.

```
template< class PopulationIt, class SampleIt, class Distance, class URBG >
SampleIterator sample( PopulationIt first, PopulationIt last,
                       SampleIt out, Distance n, URBG&& g );
```

- The above two also have `std::ranges` version since C++20.

3. `generate_random(range, PRNG[, distribution])`: since C++26, `std::ranges` version only.
   - `range` can also be two params `begin, end`;
   - Effects: fill the range with PRNG or distribution.

```
std::ranges::generate_random(intArray, g);
```

```
std::array<float, arrayLength> fltArray;
std::mt19937 g(777);
std::uniform_real_distribution d(1.f, 2.f);

std::ranges::generate_random(fltArray, g, d);
```

Question: why not just `std::generate`?

It is equivalent to:

```
for(auto& el : intArray)
    el = g();
```

It is equivalent to:

```
for(auto& el : fltArray)
    el = d(e);
```

Random number

The standard encourages standard library implementations to add their `.generate_random` for PRNGs and distributions if optimizations are possible.

- Reason: `stdr::generate_random` allows for customization.
  - For example, some PRNGs or distributions can be vectorized, but the plain fallback loop will suppress it.
  - These PRNGs or distributions can provide its member function `.generate_random()`, and `stdr::generate_random` will call them.

1) Calls `g.generate_random(std::forward<R>(r))` if this expression is well-formed.

   Otherwise, let `I` be `std::invoke_result_t<G&>`. If R models `sized_range`, fills `r` with `ranges::size(r)` values of `I` by performing an unspecified number of invocations of the form `g()` or `g.generate_random(s)`, if such an expression is well-formed for a value `N` and an object `s` of type `std::span<I, N>`.

   Otherwise, performs the fallback operation.

3) Calls `d.generate_random(std::forward<R>(r), g)` if this expression is well-formed.

   Otherwise, let `I` be `std::invoke_result_t<D&, G&>`. If R models `sized_range`, fills `r` with `ranges::size(r)` values of type `I` by performing an unspecified number of invocations of the form `std::invoke(d, g)` or `d.generate_random(s, g)`, if such an expression is well-formed for a value `N` and an object `s` of type `std::span<I, N>`.

   Otherwise, performs the fallback operation.

Supplementary and Summary

Summary and future prospect

Course Outline

- Back to our outline in the first lecture…

1. Introduction
2. Basic review & extension
3. Containers
4. Ranges and algorithms
   - We've talked about how algorithms and containers are implemented; and a bit of C++20 ranges.
5. Lifetime (& Type Safety)
   - Lifetime and storage duration, strict aliasing rules, slicing problems, C++-style type conversion, `variant` and `any`.

Course Outline

6. Programming in multiple files
   - Basic principles like preprocessor, Translation Unit, ODR, namespace and linkage (and `inline`), so we can explain why header and source files work.
   - A bit of xmake, how to make a library.
   - And a bit of C++20 modules.
7. Error Handling
   - Starting from C error code, we gradually introduced `optional` & `expected`, exceptions, assertions and debug helpers (e.g. `stacktrace`).
     - Note that C++ also wraps error code, and it's discussed in homework of this lecture (i.e. Lecture 16).
   - We emphasized exception safety and talked about copy-and-swap idiom.
   - And a bit of Catch2 to do unit test.

Course Outline

8. String and Stream
   - String literals and raw strings, `string` and `string_view`, and `<charconv>`.
   - How Unicode works, and how it's supported in C++.
   - Format and print functions, extension to range, how to specialize your own formatter.
   - A bit of how stream works.
   - And finally regex, but `<regex>` is not discussed.

9. ~ 11. Move Semantics for 2.5 lectures.
   - In Lecture 9, we introduced why we need move semantics, how to write move ctor and assignment, Rule of Five / Zero, moved-from states and some simple algorithms for move semantics.

Course Outline

- In Lecture 10, we discussed value category (plus `decltype`), reference qualifier and deducing this, copy elision (RVO, NRVO, implicit move) and some analysis on performance of different types.
- In Lecture 11, we talked about universal reference and perfect forwarding.

11. ~ 12. Templates for 1.5 lectures.

- In Lecture 11, we taught basics of templates, including `constexpr` (`consteval`, `constinit`), specializations and overload resolutions, some tricky details (`this->`, `typename`, `template`, nested specialization) and finally C++20 concept.
- In Lecture 12, we dived into harder parts such as:
  - NTTP and template template parameter, type deduction (and CTAD), friend template and lazy instantiation, SFINAE.
  - Variadic templates and folder expression.
  - Important techniques: CRTP and type erasure.

Course Outline

13. ~ 14. Multithreading for 1.5 lectures.
    - In Lecture 13, we unveiled utilities from low level to high level, from `thread` (& `jthread` with stop token handling), synchronizations (semaphores, mutex & locks, condition variable, latch & barrier) to future-promise model, `packaged_task` and `async`.
    - In Lecture 14, we discussed memory order and atomic variables in detail.
14. Coroutines for 0.5 lectures.
    - We gave a brief introduction to coroutines, including how to write them in C++, symmetric transfer and `std::generator`.
15. Memory management
    - We covered object layout, operator new & delete, smart pointers and allocators.

Course Outline

16. Final
    - Finally, we provided some supplementary on filesystem, chrono and math functions, and gave a summary to the whole course.
- Wow, that’s a long journey…
    - When I mentioned every part, you can definitely recall lots of knowledge you learnt!
    - We’ve covered most of important topics, with some deliberately neglected like ADL.
    - The final video length is 46h + 5.5h (this supplementary chapter), which is actually too long (I expect 40h) because I want to cover every detail.
        - That’s my fault… If I renew this course, many things will be discarded.
- And as we said, our course only covers until C++23…

Future C++

- Though C++ provides powerful utilities, it's still evolving and lots of proposals are submitted every month.
- Particularly, C++26 will bring you these important features:
  - We've mentioned many, like _ for placeholder name, rcu & hazard pointer, variadic friends, `copyable_function` & `function_ref`, `constexpr` placement new, `indirect` & `polymorphic`, `inplace_vector` (mentioned in homework), etc. Besides:
  - Pack indexing:

```
template <typename... T>
constexpr auto first_plus_last(T... values) ->  T...[0] {
    return T...[0](values...[0] + values...[sizeof...(values)-1]);
}
int main() {
    //first_plus_last(); // ill formed
    static_assert(first_plus_last(1, 2, 10) == 11);
}
```

Reflection

- Example: universal formatter to print members.

```
struct B { int m0 = 0; };
struct X { int m1 = 1; };
struct Y { int m2 = 2; };
class Z : public X, private Y { int m3 = 3; int m4 = 4; };

template <> struct std::formatter<B> : universal_formatter { };
template <> struct std::formatter<X> : universal_formatter { };
template <> struct std::formatter<Y> : universal_formatter { };
template <> struct std::formatter<Z> : universal_formatter { };

int main() {
    std::println("{}", Z());
      // Z{X{B{.m0=0}, .m1 = 1}, Y{{.m0=0}, .m2 = 2}, .m3 = 3, .m4 = 4}
}
```

```
struct universal_formatter {
  constexpr auto parse(auto& ctx) { return ctx.begin(); }

  template <typename T>
  auto format(T const& t, auto& ctx) const {
    auto out = std::format_to(ctx.out(), "{}{{", has_identifier(^^T) ? identifier_of(^^T)
                                                                     : "(unnamed-type)");

    auto delim = [first=true]() mutable {
      if (!first) {
        *out++ = ',';
        *out++ = ' ';
      }
      first = false;
    };

    constexpr auto ctx = std::meta::access_context::unchecked();

    template for (constexpr auto base : define_static_array(bases_of(^^T, ctx))) {
      delim();
      out = std::format_to(out, "{}", (typename [: type_of(base) :] const&)(t));
    }

    template for (constexpr auto mem :
                  define_static_array(nonstatic_data_members_of(^^T, ctx))) {
      delim();
      std::string_view mem_label = has_identifier(mem) ? identifier_of(mem)
                                                       : "(unnamed-member)";
      out = std::format_to(out, ".{}={}", mem_label, t.[:mem:]);
    }

    *out++ = '}';
    return out;
  }
};
```

`template for` is called *expansion statement*, which is also introduced in C++26..

Contract

Evaluation semantic	Is a checking semantic	Is a terminating semantic
ignore		
observe	Yes	
enforce	Yes	Yes
quick-enforce	Yes	Yes

- The finally decided semantics include:

```
int f(const int x)
  pre (x != 1)                    // a precondition assertion
  post(r : r == x && r != 2)      // a postcondition assertion; r names the result object of f
{
  contract_assert (x != 3);       // an assertion statement
  return x;
}

void g()
{
  f(0);   // no contract violation
  f(1);   // violates precondition assertion of f
  f(2);   // violates postcondition assertion of f
  f(3);   // violates assertion statement within f
  f(4);   // no contract violation
}
```

Execution

- A general and uniform framework for concurrency.
  - In `<execution>`, namespace `std::execution`.

```
// Declare a pool of 3 worker threads:
exec::static_thread_pool pool(3);

// Get a handle to the thread pool:
auto sched = pool.get_scheduler();

// Describe some work:
// Creates 3 sender pipelines that are executed concurrently by passing to `when_all`
// Each sender is scheduled on `sched` using `on` and starts with `just(n)` that creates a
// Sender that just forwards `n` to the next sender.
// After `just(n)`, we chain `then(fun)` which invokes `fun` using the value provided from `just()`
// Note: No work actually happens here. Everything is lazy and `work` is just an object that statica
// represents the work to later be executed
auto fun = [](int i) { return i*i; };
auto work = stdexec::when_all(
    stdexec::starts_on(sched, stdexec::just(0) | stdexec::then(fun)),
    stdexec::starts_on(sched, stdexec::just(1) | stdexec::then(fun)),
    stdexec::starts_on(sched, stdexec::just(2) | stdexec::then(fun))
);

// Launch the work and wait for the result
auto [i, j, k] = stdexec::sync_wait(std::move(work)).value();

// Prints "0 1 4":
std::printf("%d %d %d\n", i, j, k);
```

And many others…

- Standard library hardening: given special compiler arguments (e.g. `-fharden`), the standard library implementation should always check conditions!
  - For example, `vector::operator[]`:

```
constexpr reference operator[](size_type idx) const;
```

1 *Preconditions*: `idx < size()` is `true`.

2 *Returns*: `*(data() + idx)`.

3 *Throws*: Nothing.

    - It's UB when index is out-of-bound; but hardening requires the implementation to check the precondition.
  - Really nice feature for safety!
    - And note: language UB and library UB are different; plenty of language UBs are not detectable but most of library UBs are detectable.

And many others…

- Task type for coroutine:

```
stdexec::task<int> Print()
{
    std::cout << "Hello, world!\n";
    co_return co_await stdexec::just(0);
}
```

  - We've implemented a tiny `Task` in Coroutine, and the standard library just evolves it further and migrates it into std::execution.

  1. `stdexec::just(0)` is a sender that can be `co_await`ed.
  2. Its `.await_resume` returns its parameter `0`.
  3. `co_return 0` is processed by `stdexec::task::promise_type::return_value`.

- Parallel Range algorithm;
  - We've said that parallel algorithms don't have `std::ranges` version;
  - Since C++26, they are added for random-access range.

And many others…

- SIMD
- submdspan
- User-generated static_assert messages.

```
constexpr string_view str = "Hello";
static_assert(false, str);
```

- Erroneous behavior for uninitialized variables.
  - Erroneous behavior is new behavior that is well-defined (so compilers cannot do wrong assumptions and make aggressive optimizations like UB), but is essentially incorrect.
    - This restricts surprise in "undefined" results.
  - If you really want it uninitialized, use attributes:

```
int g(bool b) {
  unsigned char c;
  unsigned char d = c;    // no erroneous behavior, but d has an erroneous value

  assert(c == d);         // holds, both integral promotions have erroneous behavior

  int e = d;              // erroneous behavior
  return b ? d : 0;       // erroneous behavior if b is true
}
```

```
void g() {
  int x [[indeterminate]], y;
  f(y);       // erroneous behavior [basic.indet]
  f(x);       // undefined behavior
}
```

现代C++基础
Modern C++ Basics

That's ALL!

Jiaming Liang, ~~undergraduate~~ from Peking University

Postgraduate from PKU since 2024.9 :-)

Next lecture?

- Well, our lectures on “Modern C++ Basics” have ended…
  - I really appreciate audiences who persist in learning my course and reach here!
- Though this series “goes beyond its lifetime”, all knowledge has been passed to your data member and will continue their adventure with you.

- You’ll definitely learn more and more in C++…
  - Just believe in yourself and forge ahead!
- And it’s my honor to become part of this lifelong journey.

- This channel will also continue to produce more contents on interesting and brand-new topics about C++, when I have enough spare time.
- Really thank you, and see you in the next video ☺.