字符串与流
String and Stream

现代C++基础
Modern C++ Basics

Jiaming Liang, undergraduate from Peking University

- **String and string view**
- **Unicode support and locale**
- **Print function and Formatter**
- **Stream**
- **Regular expression**

Before that…

- We first review and extend something in character and string literals.
  - First, character is just like `‘a’`, or escaped one `‘\n’`, `‘\\’`, or octal representation `‘\0’`, `‘\123’`, or hexadecimal representation `‘\x12’`.
    - Since C++23, you can also use `‘\o{12}’, ‘\x{12}’`.
  - C Strings: like `“abc\x12\n”`.
    - It’s null-terminated, which means it in fact has 6 characters.
      - This distinguishes it from a pure character array(like `char a[] = { ‘1’, ‘2’ }`).
    - Though it’s in fact `const char[]`, assigning to `auto` variable will decay to `const char*`; you cannot assign it to `char*` without `const_cast` in C++.
    - You can concatenate them, e.g. `“123” “456”` is actually same as `“123456”`.
      - This is convenient if you need a long string - just add newlines!

Before that…

- Raw strings: any character will not be escaped.
  - For example, `“\\\n\"”` means `“\\n"”` in fact, but `R“(\\\n\")”` means exactly six characters.
    - This is really useful for e.g. filesystem path in Windows, which uses backslash `\` (instead of slash `/` like Unix) for delimiter.
  - If you want a `‘\n’`, then just input a real new line, like `R“(This is a new line)”` will get `“This is a \n new line”`.
  - To prevent parsing error(e.g. `“"”` is abnormal), the boundary are determined by `“(` and `)”`.
    - But what if we want a `)”` in the string?
    - Then you can change the boundary by adding some internal characters, e.g. `R“+(I want a )”!)+”` .
- Besides, we will call string that shows all characters in escaped form (e.g. `\n` instead of a new line) an **escaped string**.

String and Stream

String and string view

String and stream

- string
- string view

string

- std::string is just an enhancement to std::vector<char>.
  - The basic implementation is usually same as vector, like reallocation.
  - Contiguous range.
  - Random-access iterators, i.e. (c)begin/end, (r)begin/end();
  - Comparison, swap, std::erase(_if).
  - .empty/size/max_size/capacity;
    - .length() is also provided, just same as .size().
  - .at/operator[]/front/back();
  - .clear/shrink_to_fit/reserve/resize/push_back/pop_back();
    - Particularly, reserve in std::string can be used to shrink memory before; but **since C++20**, it's same as std::vector, i.e. no effect if param <= capacity.
  - .assign/insert/erase/insert_range/assign_range (C++23);
    - .append/.append_range is also provided, which is just like insert(str.end(), …).
      - But it returns reference to the new string.

string

```
string (1)     string& insert (size_t pos, const string& str);
substring (2)  string& insert (size_t pos, const string& str, size_t subpos, size_t sublen);
c-string (3)   string& insert (size_t pos, const char* s);
buffer (4)     string& insert (size_t pos, const char* s, size_t n);
fill (5)       string& insert (size_t pos, size_t n, char c);
               string& insert(size_t pos, StringView str, size_t subpos=0, size_t sublen=npos);
```

- Besides, it also provides unique APIs that's suitable for string:
  - Index is more commonly used for string, so `.assign/insert/erase/append()` also provides index-based version.
    - The first parameter is changed from iterator to index; if the index > size, then `std::out_of_range` will be thrown.
    - For insertion/assign, then you can provide a string/C-string/string view, which will insert/assign the total.
      - For C-string, you can also provide a count to explicitly designate the character number you want to insert.
        - Of course, it may be more efficient (compared with only a C-string) to pass the length if you've known it before.
      - For string/string_view, you can provide another index for the begin copy position, and an optional (default to end) count.
    - For erase, only `(index, count)` is provided.
    - For append, it's just same as insertion, with the first index parameter equal to `str.size()`.

string

- All index-based methods returns `std::string&` instead of iterator.
- More other string-related methods.
  - `operator+/+=/hash`;
  - `.starts_with/ends_with` (C++20); `contains` (C++23);
    - Parameter is char/any string.
  - `.substr(index(, count))`: return a new string same as sub-string.
  - `.replace`: replace part of the string with a new string, return `std::string&`.
    - "part of" is specified by the first two parameters, by an iterator pair `(first, last)` or `(index, count)`.
    - "a new string" is specified by the following parameters, which is basically same as insert (e.g. all strings, C-string with count, character with count, strings with `(index(, count))`).
    - Anyway, see IDE prompts if you're not sure.
    - C++23 also supports `.replace_with_range()`, i.e. `(first, last, range)`

string

- `.data()/.c_str()`: get the underlying pointer (i.e. `const char*`).
  - `.data()` returns `char*` (i.e. no `const`, like vector) for non-const string since C++17.
- Search: `.find/rfind/find_first(_not)_of/find_last(_not)_of()`.
  - They're similar to algorithms in standard library, but return index instead of iterator.
    - If not found, `std::string::npos` (i.e. `static_cast<size_t>(-1)`) will be returned.
  - `.find/rfind` are used to find a character/sub-string, with the following parameter as the start position (by default 0).
    - E.g. `str = "PKU>THU"; str.find("TH");` will get 4, `str.find("TH", 5);` will get `std::string::npos`.
    - E.g. `str = "PKU>THU"; str.find_first_of("TH");` will get 4 (because `str[4]` is `'T'` in `"TH"`) is the first occurrence for one of character in `"TH"`), `str.find_first_of("TH", 5);` will get 5 (i.e. `'H'`).

string

- Finally, all `count` can be substituted by `std::string::npos`, which means "until end of string".

- Okay, APIs may be boring to you, so at least have a rough impression of them, and utilize you IDE to give you hints!

- Now let's talk about some interesting things about string…
- Note1: Remember what we've mention in review lecture?

string

- Note2: `std::string` guarantees the underlying string is null-terminated.
  - You can also have `'\0'` in your string too, since it doesn't judge end like C-style string, but by `.size()`.
- Note3: Usually, `std::string` has **SSO** (small string optimization).
  - That is, if the string is small, it may not allocate memory on heap, but just utilize stack memory.
    - Remember? `new/delete` is a rather expensive operation compared with e.g. integer add. So when string is small, it's not worthwhile to do so.
    - The concrete number is not determined, but in x64 libstdc++/VC is 15, libc++ is 22.
  - Even if it has SSO, the total size is small enough; they're usually smaller than 40 bytes, meaning that copying them is basically as cheap as copying several `std::uint64_t`.

string

- When the string is long enough, the storage will be allocated on heap, and it's basically same as `std::vector<char>`.
- Note4: C++23 introduces another optimization for resizing.
  - Resize only gives fixed character (by default `'\0'`) to fill in, which usually needs following assignments; that's performance loss…
    - Of course, we can use `reserve + insert` sometimes;
    - But there should be a more convenient (and possibly more efficient) API for users to reallocate and write!
  - `.resize_and_overwrite(newSize, Op)` is for that.
    - `Op` should accept `(char* ptr, size_t len)`, and then overwrite it.
      - `len` is equal to `newSize`, and `[ptr, ptr + min(oldSize, newSize))` is same as `[.data(), .data() + min(oldSize, newSize))`.
      - Return `realSize (<= newSize)` so that finally `.size() == realSize`.
      - It shouldn't throw any exception.

string

- Note5: you can also convert a string to/back from a number.
  - `std::stoi/sto(u)l/sto(u)ll(string, std::size_t* end = nullptr, int base = 10);`
    - It will stop at the end of the first parsed number, and try to write the stop index to `*end` (e.g. `"123 456"` will write `*end = 3`; past-end is `size`).
      - If the first digit is not valid (i.e. stop without finding a number), `std::invalid_argument` will be thrown.
    - Base(进制) can be 2-36; for base > 10, alphabet will be used for digit.
      - Base=0 means identify base automatically by prefix (i.e. 0 for octal, 0x for hexadecimal, otherwise decimal).
    - If the result cannot be represented by `int/…`, `std::out_of_range` will be thrown.
  - `std::stof/stod/stold(string, std::size_t* end = nullptr)`
  - `std::to_string()`, accept floating points or integers.
    - Breaking change since C++26: it will be same as `std::format("{}", val)`; we'll cover `format` later. See [P2587] for changing details.
      - Before it will show fixed point (i.e. always 6 digits after dot) for floating points, which may not be able to be converted back by e.g. `stof()`. For example, `std::to_string(1e-7f)` will get `"0.000000"` before C++26, and `"1e-7f"` after that.

String and stream

- string
- string view

string view

- Sometimes, you may only read the string in a function, so you use `const std::string&` as parameter.
  - However, if you pass a C-string, e.g. “PKU”, then it has to create a temporary string. But we only want to read it, so why should we endure this performance loss?
  - `std::string_view` defined in `<string_view>` since C++17 is for that!
    - Before, you may have to define overloads for C-string (that’s why methods of `std::string` has so many overloads…)
    - `std::string` and C-string can convert to `std::string_view` implicitly, and you can also use iterator pair to construct it.
  - It’s like a specialization of `span<const char>`, i.e. it just has a **`const`** `char*` with a length.
    - Thus, use `std::string_view` instead of `const char*` without `count` for `std::string` methods may boost performance.
  - It has almost all non-modified methods of `std::string`.

string view

C++26 is likely to add `operator+` to concatenate `std::string` and `std::string_view`, see P2591.

- Random-access iterators, i.e. `(c)begin/end, (r)begin/end()`;
- Comparison, swap, hash.
- `.empty/size/length/max_size`;
  - No capacity since it's a **view**; it only observes the memory.
- `.at/operator[]/front/back()`;
- `.clear/shrink_to_fit/resize/push_back/pop_back()`;
- `.starts_with/ends_with/contains()`;
- `.find/rfind/find_first(_not)_of/find_last(_not)_of()`;
  - `std::string_view::npos`.
- `.substr(index(, count))/remove_prefix(n)/remove_suffix(n)`;
  - They all create `std::string_view` instead of `std::string`, so it's $O(1)$.
- `.data()`; it always return `const char*`, and there is no `.c_str()`;
- It doesn't support `operator+`, since you need to create `std::string` to make sense, but it's costly so you need to explicitly do it.

string view

- Some examples:
  - Transparent operator:

```cpp
class Hasher
{
public:
    using is_transparent = void;
    size_t operator()(std::string_view sv) const {
        return std::hash<std::string_view>{}(sv);
    }
};
```

  - std::sort by substring:

```cpp
std::vector<std::string> vec{ "PKU", "THU", "CMU"};
stdr::sort(vec, [](const std::string& s1, const std::string& s2) {
    return std::string_view{ s1 }.substr(1) < std::string_view{ s2 }.substr(1);
});
std::print(vec);
```

```
C:\WINDOWS\system32\cmd.exe
[THU, PKU, CMU]请按任意键继续. . .
```

    - It’s an optimization compared with `s1.substr()` since it will not create a temporary string, but just a cheap view to sub-part of it!
    - Before, you have to use `std::lexicographical_compare`.

Caveats on string view

- 1. `std::string_view` is not required to be null-terminated.
  - Anyway, it's just a `const char*` with a length!
  - So, it may be not safe to use `.data()` to pass into an API that requires null termination, like C-string APIs.
  - If you really need it, there are two possible ways:
    - Construct `string_view` to make `.back() == '\0'`, e.g. by iterator pairs.
      - But this will make `.size()` different from `strlen`, since it counts termination additionally.
    - Or make sure `str[size] == '\0'`;
      - This should be guaranteed by the user, since you're reading past end, which is "logically" wrong.
  - Output it by stream doesn't require null-termination.

Caveats on string view

But since C++23, you cannot use `nullptr` to construct `std::string_view` directly; you need `{nullptr, 0}` or default ctor.

- 2. The pointer it contains can be `nullptr` (as default ctor does).
  - This usually happens when `.size()` is 0, so check it specially.
- 3. You should be really cautious if you want to use `std::string_view` as return value.
  - It's just like span or a `ref_view`; if the referred object goes out of its lifetime, then it's dangling!
  - It may be hidden sometimes, for example:
    - If you pass a temporary string (e.g. "PKU"), then its lifetime ends once the function ends, so `std::string_view` is dangling.
    - Ranges have similar problem, but it will use `owning_view` for rvalues, so it's slightly safer (but for this example it's still dangerous since it's not rvalue).
  - Another example: `auto s = CreatePerson().GetName();` is dangerous, since the temporary person has been destructed, invalid view to its name.

```cpp
std::string_view func(const std::string& str) {
    return str;
}
```

Caveats on string view

  - Instead, return `std::string` is safer.
  - So, if you really want to return `std::string_view`, document and name the function in an explicit and noticeable way!
- 4. Template parameter that may be related to `std::string_view` should also pay attention to return type.
  - If we pass into two `std::string_view`, then returned thing is still `std::string_view`.
  - Use `auto` instead.

```cpp
template<typename T>
T Add(T a, T b) { return std::string{ a } + std::string{ b }; }
```

- 5. If you will create the string anyway (like in a ctor), pass a `std::string_view` is not a good idea.
  - We'll tell you why in the next lecture after learning move semantics!

User-defined literals

- You may find that it’s troublesome to create a string/string_view from a C-string.
    - e.g. `std::string_view{ “PKU” }`.
    - Why cannot we use something like `1ull` to denote the type?
    - User-defined literals are for that!
    - There are some pre-defined standard literals, e.g. for string/string_view we can use `“PKU”s` and `“PKU”sv`.
        - Besides strings, there are other two kinds of standard literals:
            - Time-related: `1s` for seconds, `1.1ms` for milliseconds, `1d` for 1 day, etc.
            - Complex-related: `1i` for pure imaginary number, `1.2if/2.5id` for explicit types (float/double imaginary).
        - Remember `using namespace std::literals;` in your local scope!

User-defined literals

- You can also define your own literals, e.g. cache simulator I've coded:

```
CacheConfig L1config{ 32_KB, 8, 64_B, 1, 0, CacheConfig::WritePolicy::WriteBack_Allocate };
CacheConfig L2config{ 256_KB, 8, 64_B, 8, 6, CacheConfig::WritePolicy::WriteBack_Allocate };
CacheConfig LLCconfig{ 8_MB, 8, 64_B, 20, 20, CacheConfig::WritePolicy::WriteBack_Allocate };
```

- You need to define literal operators:

```
constexpr unsigned int operator"" _KB(unsigned long long m) { return static_cast<unsigned int>(m) * 1024; }
```

  - It's recommended that literals you define are `_xx`, i.e. begin with a underscore, to prevent possible confliction with standard literals.
  - Here we assume < 4GB, so we use `unsigned int` as return type; `unsigned long long` is better for general case.
  - We'll explain "`constexpr`" in detail in the future; basically it means "compiler will try to calculate at compile time".

User-defined literals

- The parameter type is limited:
  - For integers, only `unsigned long long` is permitted.
    - This is because it's usually the largest integer, which can unify all integers instead of defining lots of overloads.
    - Conversion will not cause performance loss, since they're finished in compilation time.
  - For floating points, only `long double`.
  - For characters, e.g. `char` and Unicode characters, they're all OK.
  - For C-strings, `(const CHAR*, std::size_t)` is needed, where `CHAR` is any character type.
    - Thus, you can utilize length to maximize efficiency!
  - Finally, a `(const char*)` is also provided, as a fallback of integers and floating points.
    - It will treat them as strings.
    - This is rarely used, though.

```cpp
void operator""_print(const char* str)
{
    std::cout << str << '\n';
}
```

```cpp
0x123ABC_print;
```

charconv

- Finally, `stoi/to_string` will create new `std::string`; we may want to provide storage ourselves.
  - E.g. `stoi(std::string{view})` is costly, since we only read the string.
  - Also, they may throw exceptions, which are expensive sometimes.
- You can use `std::from_chars` and `std::to_chars` in `<charconv>`!
  - `std::from_chars(const char* begin, const char* end, val)` will try to save the result into `val` (an integer or a floating point).
    - It returns `std::from_chars_result`, which includes `.ptr` as stopping point and `.ec` as error code.
      - When `ec == std::errc{}`, success; it's also possible that `ec == std::errc::invalid_argument`, or `std::errc::result_out_of_range`.
      - You can use structured binding, e.g. `if(auto [ptr, ec] = xx; ec != std::errc{})`.

Notice that C++26 can drop `ec != std::errc{}`, since `std::from/to_chars_result` can be converted to `bool` directly.

charconv

```cpp
for (std::string_view const str : {"1234", "15 foo", "bar", " 42", "5000000000"})
{
    std::cout << "String: " << std::quoted(str) << ". ";
    int result{};
    auto [ptr, ec] = std::from_chars(str.data(), str.data() + str.size(), result);

    if (ec == std::errc())
        std::cout << "Result: " << result << ", ptr -> " << std::quoted(ptr);
    else if (ec == std::errc::invalid_argument)
    {
        std::cout << "This is not a number.";
        std::cout << " Remaining chars: " << ptr;
    }
    else if (ec == std::errc::result_out_of_range)
    {
        std::cout << "This number is larger than an int.";
        std::cout << " Remaining chars: " << ptr;
    }
    std::cout << "\n";
}
```

找到相关问题

器 命令窗口 错误列表 输出

C:\WINDOWS\system32\cmd.exe

```
String: "1234". Result: 1234, ptr -> ""
String: "15 foo". Result: 15, ptr -> " foo"
String: "bar". This is not a number. Remaining chars: bar
String: " 42". This is not a number. Remaining chars:  42
String: "5000000000". This number is larger than an int. Remaining chars:
请按任意键继续. . .
```

Out of range will still get advanced pointer!

charconv

- `base/std::chars_format` can also be provided as the last parameter.
  - For integers, `base` should be in [2, 36] and by default 10.
    - Notice that only minus sign will be recognized; e.g. Leading whitespaces, "`0x`" are not.
  - For floating point, `std::chars_format::xx` should be provided.
    - `scientific`: like `(-)d.ddde±dd`;
    - `fixed`: like `(-)d.ddd`;
    - `hex`: like `(-)h.hhhp±hh`;(hex floating point is rarely used, so not covered).
    - `general`: `scientific | fixed`, both are OK.
    - Particularly, "`NAN`", "`INF`"(case-insensitive) are all Okay.
    - Results are rounded to nearest.

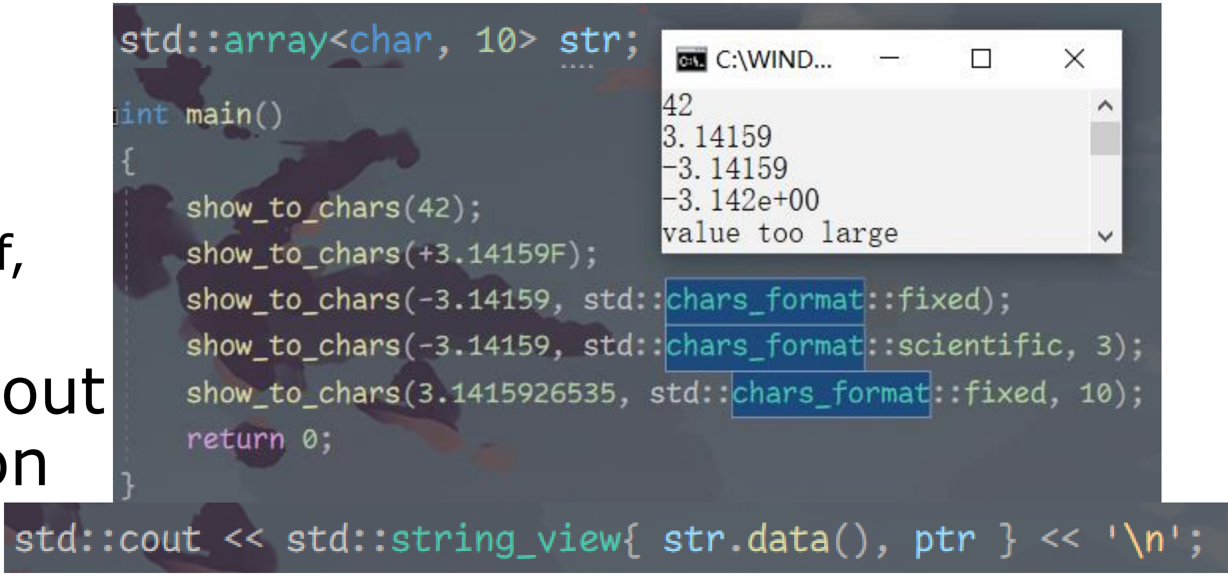
Unfortunately, libc++ doesn't support `std::from_chars` for floating points yet even in 2024/6 (see impl status). libstdc++/MS-STL all support it.

charconv

- `std::to_chars(char* begin, char* end, value)` will try to write `val` (an integer or a floating point) into `[begin, end)`.
  - Notice that null-termination will not be written!
  - It returns `std::to_chars_result`, which also includes `.ptr` as stopping point and `.ec` as error code.
  - When `ec == std::errc{}`, success; it's also possible that `ec == std::errc::value_too_large` (`ptr == end`, `[begin, end)` may be anything).
- `base/std::chars_format` can also be provided as the last parameter.
  - For integers, `base` should be in [2, 36] and by default 10.
  - For floating point, `std::chars_format::xx` should be provided.
    - You can also provide an `int precision`, which specifies number of digits after dot.
    - Particularly, floating points that are integrals or some values that can be precisely represented by it (as you've learnt in ICS, e.g. `2.0f, 2.125f`) with `std::chars_format::general` will truncate instead of aligning.
      - E.g. `(general, 4)` will get `2` or `2.125`, but `(fixed, 4)` will get `2.0000` or `2.1250`.

charconv

- Precision is 10, meaning that 10 digits are required after dot, which is not possible for `std::array<char, 10>`.
  - Changing to 3.14 will still fail!
  - Besides, null termination is not written, so we don’t write `str.data()` directly.
    - Add a null termination at `ptr` yourself, or construct a `string_view` like this.
- Finally, `from_chars/to_chars` without precision limit can do round-trip on the same platform.



We don’t show entire function body since we haven’t learn how to pass into varying parameters.

String and Stream

Unicode support and locale

String and stream

- Unicode support and locale
  - Unicode
  - Unicode support in C++
  - Locale

Credit:
CppCon 2014: "Unicode in C++", James McNellis

Unicode

- Characters that we used before is just `char`, which usually uses ASCII.
  - However, ASCII cannot represent all symbols, like French, Chinese, etc..
  - Many coding standards are provided, e.g. GBK(国标扩展码) for Chinese, EUCKR (Extended Unix Code for Korean) for Korean, etc…
  - But they're usually not cross-platform; for example, GBK file will be a total mess for some terminals (you may encounter that when using Python…).
  - Thus, Unicode is provided as unified character code.
- Unicode also evolves step by step, and we'll cover it roughly.
  - UTF1.0: each character has 16 bits; it's also called UCS-2.
    - Since it needs 2 bytes, byte order should be determined; BOM (Byte-Order Mask) `0xFE 0xFF` thus may be provided if your file may use in another machine with different endianness.

Unicode

- UTF2.0: considering that $2^{16} = 65536$, it's still limited for characters in the whole world.
  - This is usual case for Hieroglyphs(象形文字), e.g. GBK has 21003 Chinese characters.
  - Emoji are also coded in Unicode; even larger!
  - So UTF2.0 is introduced, with different code representation.
    - They all have the same Unicode id (e.g. ☞ is 0x1F449), but have different coding representations in computer.
  - UTF-32: make each character occupy 32 bits; it's direct but useful, since at most 4 billion characters can be used.
    - But it may occupy too much space, e.g. "abc" needs only 4 bytes before, but currently 16 bytes!
    - It also needs BOM.

Unicode

- UTF-8: to solve space waste, UTF-8 uses code with varying length.
  - Different character length has different coding prefix (like Huffman tree that we've learnt) to ensure no ambiguity.
    - This makes it waste some code space, so some characters may need more than 4 bytes.
  - For example, ASCII characters, including null-termination `'\0'`, still occupy 1 byte in UTF-8 (the coding is same too).
  - UTF-8 is the most commonly used character set in modern systems.
- UTF-16: to be compatible with UTF-1.0 (UCS-2), UTF-16 is also introduced.
  - It's similar to UTF-8, but extend by 2 bytes (e.g. when 2 bytes not enough, 4 bytes are used).
  - 16-bits codes are totally same as UCS-2.
  - **UTF-16 is used as default internal encoding for strings in Java, C#, etc.**.

Notice that UTF8 doesn't need BOM, but Windows identifies Unicode by BOM long long before (UCS-2 needs that), and it has been part of Windows and hard to change, so Windows may require you to give UTF8 a BOM.

This only happens when encoding is automatically detected, but Notebook software usually gives you an option to designate the encoding explicitly, and then BOM is not necessary.

Unicode

- Character normalization:
    - Many characters act as “modifier”, e.g. tone in 拼音, top things on ÀÁÂÃÄÅ….
    - Unicode also supports composite characters, i.e. combining the modifier with the character (e.g. we combine ̀ with A to get À).
        - They’re coded into two characters, but only shown as one.
    - However, some of the composite characters may already have their Unicode id, so each symbol may have different representations.
    - If you want to compare two Unicode strings, you may need to **normalize** them to get a unified representation.
- Unicode characters may also have their alias name, e.g. GREEK CAPITAL LETTER OMEGA means Ω. You can check them here.

Unicode

- So to conclude, Unicode has these basic elements:
  - **Byte**, i.e. computer representation.
  - **Code unit**, i.e. (byte count ÷ minimal bytes) used to represent a character (1 for UTF-8, 2 for UTF-16, 4 for UTF-32).
  - **Code point**, i.e. each Unicode character.
  - **Text element**, i.e. what humans really see on the screen.
    - We need normalization to compare two strings in the level of text element.
- For example, omit null-termination, 1Ä🍸 has 3 text elements, 4 code points (since there is a composite character), and:

	Representation	byte count	code unit
UTF-8	(31) (41) (CC 88) (F0 9F 8D B8)	8	8
UTF-16	(0031) (0041) (0308) (D83C DF78)	10	5
UTF-32	(00000031) (00000041) (00000308) (0001F378)	16	4

String and stream

- Unicode support and locale
  - Unicode
  - Unicode support in C++
  - Locale

Unicode support in C++

- Unluckily, Unicode support in C++ is rather weak.
  - There are `char8_t/char16_t/char32_t` for UTF-8/16/32, but it's used as one code unit, instead of one code point.
    - They're at least 8/16/32 bits to hold **one code unit**.
    - You can use `u8/u/U` as prefix separately.
    - But one code point may (and usually) have more than one code unit, so some weird things may happen:
    - There are also types like `std::u8string`, but they are all in code units!
      - This makes traversal really hard, e.g. `for(auto ch : std::u8string{u8"刘"})` will not get "刘", but several code units.
      - `.size(), .find_first_of()`… are also for code unit.

```
char8_t ch1 = u8'A'; // correct.
char8_t ch2 = u8'刘'; // cannot be represented in 1 byte, compile eror.
const char8_t str1[] = u8"刘"; // correct, but sizeof(str1) == 3, 2 for 刘
and 1 for null-termination.
```

Unicode support in C++

- So in fact, e.g. `std::string`, `std::string_view` are just instantiation of template!
  - More specifically, i.e. `std::basic_string<char>` and `std::basic_string_view<char>`.
  - So, e.g. `std::u8string`, `std::u8string_view` are just `std::basic_string<char8_t>` and `std::basic_string_view<char8_t>`.
  - Since `char8_t` itself is code unit, the string represents vector of code units too.
  - It also accepts `Traits = std::char_traits<charT>` as the second template parameter, which regulates operations used by `basic_string` like comparison of characters. So theoretically (but usually not used) you can define you own character type as long as it’s standard-layout and trivial, then give a `YourString` with the `NewTraits<NewCharType>`.
  - You can also provide an allocator, just like STL. We’ll cover it in the future.

Unicode support in C++

- Weak Unicode support is mostly due to its complexity, so why not use the standard library provided by Unicode?
  - C++ core guideline recommends you to use ICU (International Components for Unicode), Boost.Locale, etc. for complete Unicode support.
  - You may also use utf8cpp, which operates UTF-8 by **code point**, since ICU is too large for small project,.

- Besides, input/output Unicode is troublesome in C++.
  - This is improved in C++23 (but still output UTF-8 only), and we'll mention it sooner.

Unicode regular expression is not supported in standard library too; we'll mention it in regex section.

Unicode support in C++

- There is also a wide character `wchar_t` in C/C++.
  - It’s platform dependent, only regulated that “it has the same property with one of integer types”.
    - Practically, it’s UTF-16 on Windows, while UTF-32 on Linux.
  - So, it can be used if your platform is the same.
  - Prefix for wide character/string literals is `L`, e.g. `L"爱情可以慷慨又自私"`.
  - And, you can also use `std::wcout/wcin/wstring(_view)`.
    - Well, there is no `std::u8cout` etc. in C++.

Unicode support in C++

- Here we assume all character types can be converted to `int` without precision loss.

```cpp
template<typename T>
void DecomposeToByte(const T& str)
{
    for (int i = 0; i < str.size(); i++)
    {
        auto ch = static_cast<unsigned int>(str[i]);
        unsigned int mask = (1 << CHAR_BIT) - 1;
        for (int j = 0; j < sizeof(str[0]); j++)
        {
            std::cout << std::format("{:02x} ", (mask & ch));
            ch >>= 8;
        }
        std::cout << "=>";
    }
    std::cout << "\n";
    return;
}
```

```cpp
std::wstring s1 = L"\U0001F449";
std::u8string s2 = u8"\U0001F449";
std::u16string s3 = u"\U0001F449";
std::u32string s4 = U"\U0001F449";
DecomposeToByte(s1);
DecomposeToByte(s2);
DecomposeToByte(s3);
DecomposeToByte(s4);
```

```
C:\WINDOWS\system32
3d d8 =>49 dc =>
f0 =>9f =>91 =>89 =>
3d d8 =>49 dc =>
49 f4 01 00 =>
```

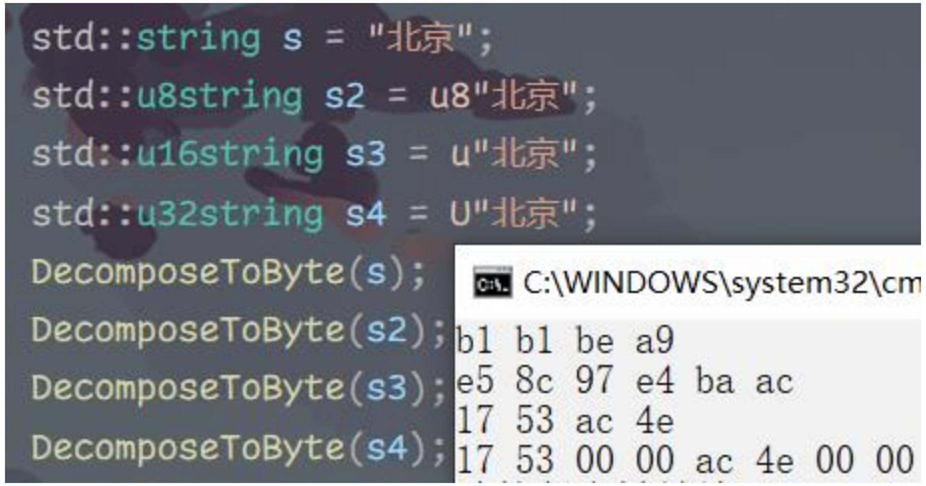
\Uxxxxxxxx(8 digits) or \uxxxx(4 digits) means Unicode id, which is not affected by computer representation. \U0001F449 is 👉.

Since C++23, you can also use `\u{1F449}`, whose digits can be of any length. Symbols that have alias name can be specified as `\N{…}` (e.g. `“\N{GREEK CAPITAL LETTER OMEGA}”`).

Unicode support in C++

- Finally, you may write non-ascii characters in C-string before, but still get correct input/output.
  - So why? Shouldn't they be Unicode?
  - In fact, C-string or `std::string` is more like **a byte array** instead of an ASCII string.
  - For example, most Chinese computers use GBK character set by default:
  - But if we specify /utf-8 in VS, then you will get:

    - You need to choose UTF8 as file format to get rid of warnings.



GBK coding table can be found here.

For string literals without any prefix, the encoding is determined by execution charset.

我放手 in GBK

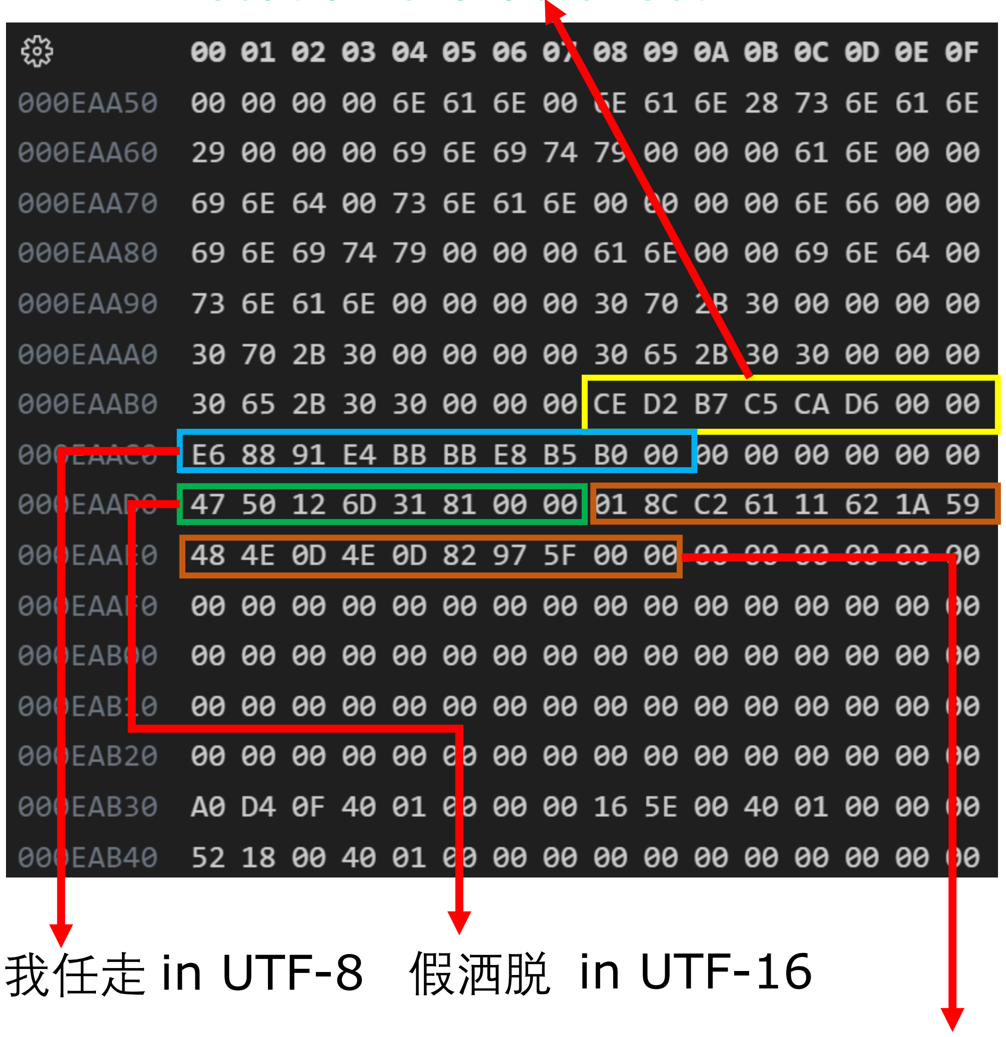
Execution character set

Input character set

All characters are of same encoding.

```
int main()
{
  std::cout << "我放手";
  const char8_t str[] = u8"我任走";
  const char16_t str2[] = u"假洒脱";
  const wchar_t str3[] = L"谁懂我"
                         L"多么不舍得";
}
```

Compiler work…



我任走 in UTF-8　假洒脱 in UTF-16

谁懂我多么不舍得 in wide character encoding (UTF-16 on Windows)

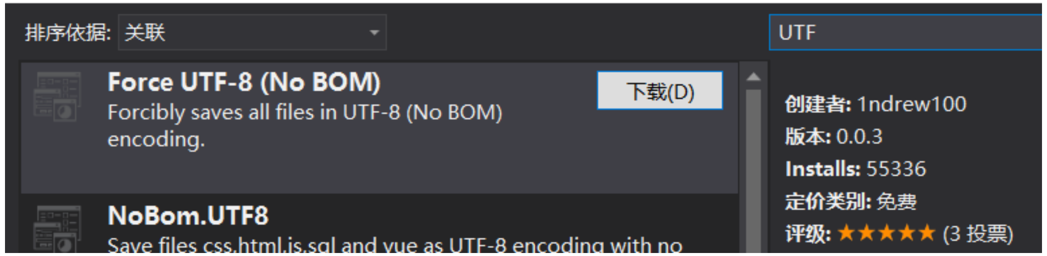
Imagine another programmer who wants to read your code. If your source code is GBK, but he/she opens it as UTF-8, then "锟斤拷"!

Compiler is just the "another programmer"; input charset is to let it know how to read.

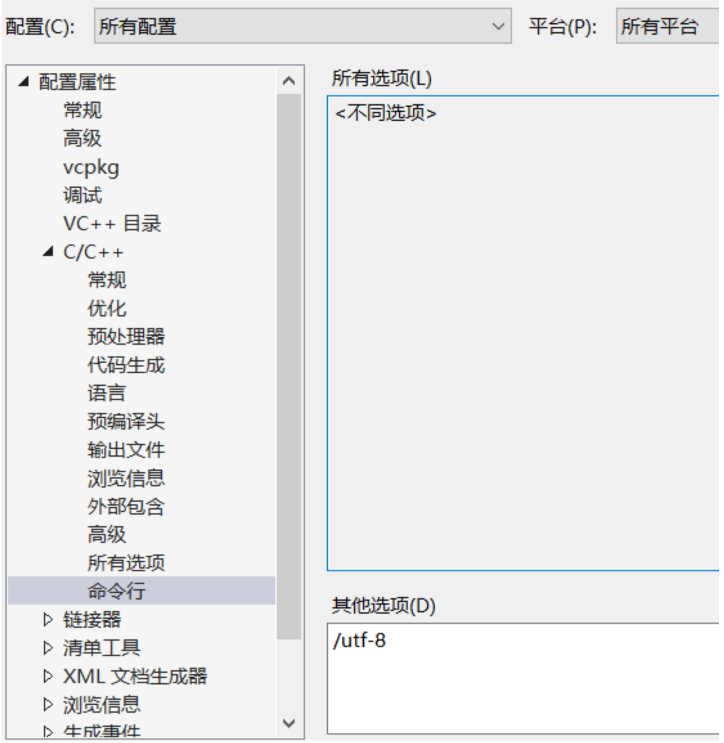
Unicode support in C++

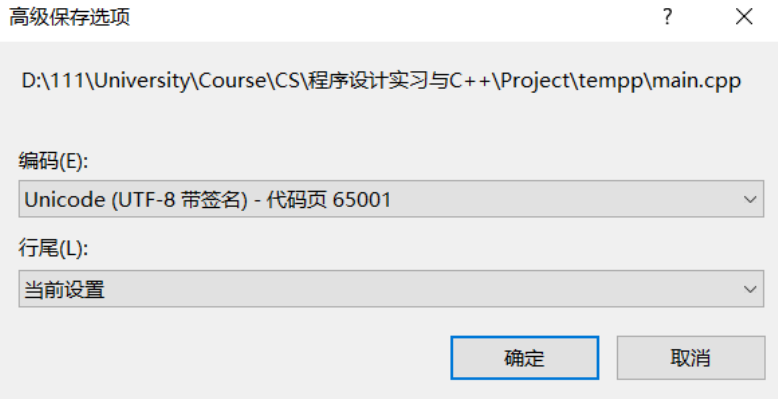
/utf-8 == /source-charset:utf-8 + /execution-charset:utf8

- To specify UTF8 as string character set, you need to go to 项目 -> 属性, then add /utf-8 like this:
- Besides, you need to save your file as UTF8 (otherwise warnings are prompted if you use non-ascii characters, since GBK cannot identify UTF8).
  - One way is 扩展 -> 搜索UTF -> 下载:



  - The other way is to use VS settings, see here for more details.





Unicode support in C++

- For gcc/clang, you can use `–finput-charset=utf-8 –fexec-charset=utf-8` to specify it.
  - Clang only supports UTF8 as execution charset.
- VSCode can easily select encoding at right bottom: 

- However, for Windows console output, you may need `SetConsoleOutputCP(CP_UTF8)` in `<Windows.h>`; console ~~doesn't support UTF-8 input yet(no matter what settings)~~ supports UTF-8 input for terminal v1.18 since Windows 11 in 2023/5.

- Final word: if you really want localized things, like operating Unicode strings in some complex ways, it's recommended to use `UnicodeString` in ICU or `Qstring` in QT.
  - We'll briefly introduce locale in C++, but it's also recommended to use other localization libraries.

In xmake, you can use `set_encodings("utf-8")`.

String and stream

- Unicode support and locale
  - Unicode
  - Unicode support in C++
  - Locale

Locale*

- Preface: A great resource to understand locale and facet is [Apache's doc] (Chapter 24 - 26). If you want to learn more, don't hesitate to read it!
- Locale is used for localization.
  - Each `std::locale` contains information for a set of culture-dependent features by `std::facet` objects.
    - That is, you can think `std::locale` as a collection of pointers to `std::facet`, and each `std::facet` contains one feature of some culture.
  - Standard facets are divided into six categories:
    - Collate: i.e. how characters are compared, transformed or hashed.
      - They don't affect default comparison/hash behavior, you still need to explicitly pass corresponding functors.
    - Numeric: `num_get, num_put, numpunct`.
      - `num_get` will affect how `std::cin` works, `num_put` -> `std::cout`, `numpunct` only affect punctuation.

Locale*

    - For example, in German, `1.234,56` means `1234.56`; if you use Germany locale, then inputting `1.234,56` will correctly get it instead of `1.234`.
  - Time: `time_get/time_put;`
  - Monetary: `money_get/money_put/moneypunct`;
  - Message: transform some error message to another language.
  - Ctype: `ctype/codecvt`, how characters are classified and converted, e.g. is upper.
    - BTW, <codecvt> is deprecated in C++17 and removed in C++26, but in fact standard `codecvt` is defined in <locale>, so it doesn't affect anything.
    - Before `codecvt` also has Unicode conversion, but it's deprecated since C++26. See [LWG3767]() and now we need to wait standard Unicode library to do so.
- They will affect how input, output, character identification and regular expression works.
  - For example, to output Chinese characters for `wstring`, you may need `.imbue`:

```
std::wcerr.imbue(std::locale("chs"));
```

Create a locale by name (the name depends on the OS). The native locale can be get by `std::locale(“”)`.

Locale*

<cctype> have similar functions, which uses the global locale (by default `classic()`), but you can use `std::locale::global(Loc)` to set another one, so it's possibly not safe.

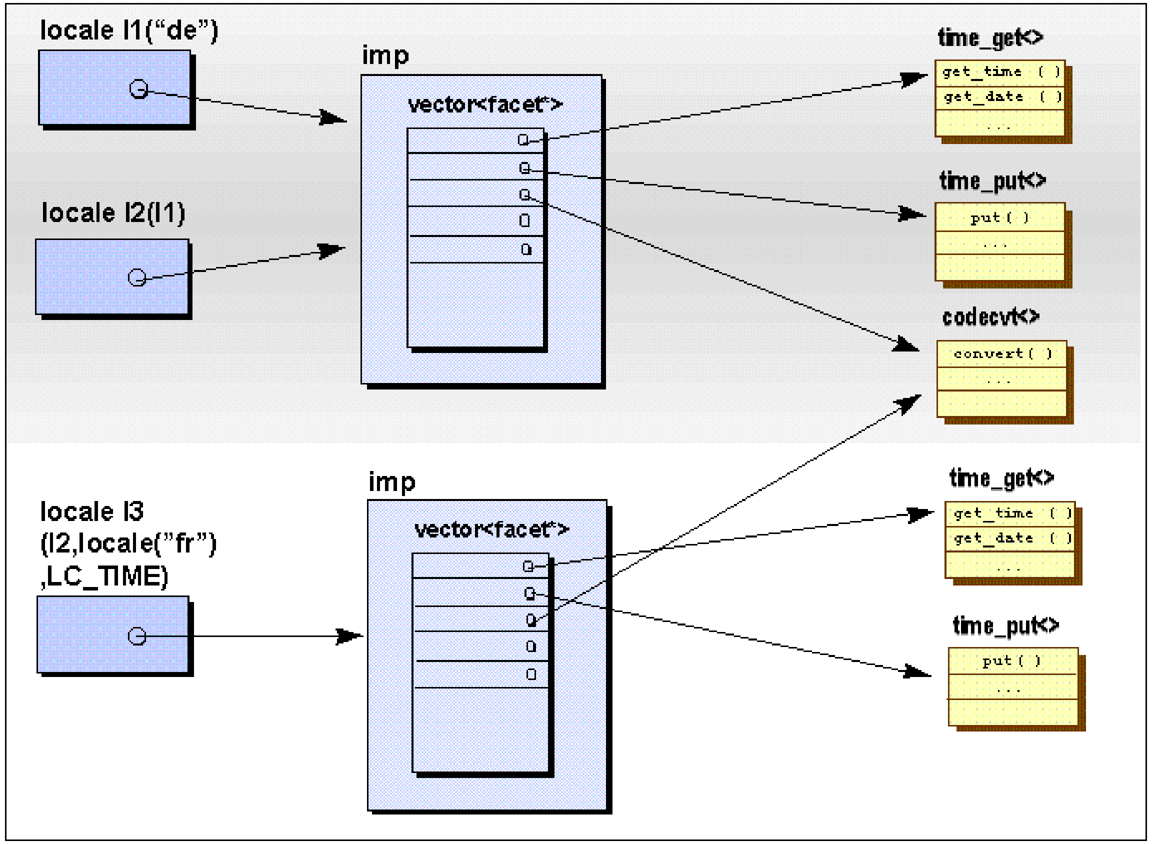
- <locale> also defines character classification functions.
  - It accepts locale as the second parameter (who then uses ctype facet to classify), so that you may e.g. use French-version `std::toupper`.
  - ASCII one is `std::locale::classic()`.

isspace(std::locale)	checks if a character is classified as whitespace by a locale (function template)
isblank(std::locale) (C++11)	checks if a character is classified as a blank character by a locale (function template)
iscntrl(std::locale)	checks if a character is classified as a control character by a locale (function template)
isupper(std::locale)	checks if a character is classified as uppercase by a locale (function template)
islower(std::locale)	checks if a character is classified as lowercase by a locale (function template)
isalpha(std::locale)	checks if a character is classified as alphabetic by a locale (function template)
isdigit(std::locale)	checks if a character is classified as a digit by a locale (function template)
ispunct(std::locale)	checks if a character is classified as punctuation by a locale (function template)
isxdigit(std::locale)	checks if a character is classified as a hexadecimal digit by a locale (function template)
isalnum(std::locale)	checks if a character is classified as alphanumeric by a locale (function template)
isprint(std::locale)	checks if a character is classified as printable by a locale (function template)
isgraph(std::locale)	checks if a character is classified as graphical by a locale (function template)

ASCII values			characters	iscntrl iswcntrl	isprint iswprint	isspace iswspace	isblank iswblank	isgraph iswgraph	ispunct iswpunct	isalnum iswalnum	isalpha iswalpha	isupper iswupper	islower iswlower	isdigit iswdigit	isxdigit iswxdigit
decimal	hexadecimal	octal													
0–8	\x0–\x8	\0–\10	control codes (NUL, etc.)	≠0	0	0	0	0	0	0	0	0	0	0	0
9	\x9	\11	tab (\t)	≠0	0	≠0	≠0	0	0	0	0	0	0	0	0
10–13	\xA–\xD	\12–\15	whitespaces (\n, \v, \f, \r)	≠0	0	≠0	0	0	0	0	0	0	0	0	0
14–31	\xE–\x1F	\16–\37	control codes	≠0	0	0	0	0	0	0	0	0	0	0	0
32	\x20	\40	space	0	≠0	≠0	≠0	0	0	0	0	0	0	0	0
33–47	\x21–\x2F	\41–\57	!"#$%&'()*+,-./	0	≠0	0	0	≠0	≠0	0	0	0	0	0	0
48–57	\x30–\x39	\60–\71	0123456789	0	≠0	0	0	≠0	0	≠0	0	0	0	≠0	≠0
58–64	\x3A–\x40	\72–\100	:;<=>?@	0	≠0	0	0	≠0	≠0	0	0	0	0	0	0
65–70	\x41–\x46	\101–\106	ABCDEF	0	≠0	0	0	≠0	0	≠0	≠0	≠0	0	0	≠0
71–90	\x47–\x5A	\107–\132	GHIJKLMNOP QRSTUVWXYZ	0	≠0	0	0	≠0	0	≠0	≠0	≠0	0	0	0
91–96	\x5B–\x60	\133–\140	[\]^_`	0	≠0	0	0	≠0	≠0	0	0	0	0	0	0
97–102	\x61–\x66	\141–\146	abcdef	0	≠0	0	0	≠0	0	≠0	≠0	0	≠0	0	≠0
103–122	\x67–\x7A	\147–\172	ghijklmnop qrstuvwxyz	0	≠0	0	0	≠0	0	≠0	≠0	0	≠0	0	0
123–126	\x7B–\x7E	\172–\176	{|}~	0	≠0	0	0	≠0	≠0	0	0	0	0	0	0
127	\x7F	\177	backspace character (DEL)	≠0	0	0	0	0	0	0	0	0	0	0	0

Locale*

- Locale is cheap to copy; it uses reference count when copied and points to same set of facets.
- Facets use the address of a static member id to group; e.g. if you want a new facet, you need to inherit from std::locale::facet and have static std::locale::id id.




```cpp
struct french_bool : std::numpunct<char>
{
    string_type do_truename() const override { return "vrai"; }
    string_type do_falsename() const override { return "faux"; }
};

int main()
{
    std::cout << "default locale: "
              << std::boolalpha << true << ", " << false << '\n';
    std::cout.imbue(std::locale(std::cout.getloc(), new french_bool));
    std::cout << "locale with modified numpunct: "
              << std::boolalpha << true << ", " << false << '\n';
}
```

Inherit and override the parent method.

Substitute a facet of the original locale. It belongs to the same group as numpunct because they have the same id.

Facet will be deleted when it's never referenced, so new is correct. It also uses reference count.

Locale*

```
class Umlaut : public std::locale::facet {                          //1
  public:
    static std::locale::id id;                                      //2
    Umlaut(std::size_t refs=0): std::locale::facet(refs) {}         //3
    bool is_umlaut(char c) const {return do_isumlaut(c);}           //4
  protected:
    virtual bool do_isumlaut(char) const;                           //5
};
```

New id address, new facet group.

- Create a new group of facet:
  - Great extensibility.

    `std::ctype<char>` is regulated by the standard, so use it without check.

    When the facet group doesn't exist, `std::bad_cast` will be thrown.

```
std::locale loc(std::locale(""),    // native locale
           new Umlaut);              // the new facet     //1
char c,d;
while (std::cin >> c){
 d = std::use_facet<std::ctype<char> >(loc).tolower(c); //2
 if (std::has_facet<Umlaut>(loc)) {                      //3
   if (std::use_facet<Umlaut>(loc).is_umlaut(d))         //4
       std::cout << c << "belongs to the German alphabet!" << '\n';
 }
}
```

- We'll have a locale exercise in homework adopted from Chapter26 of Apache's doc. It's **optional** and just for your interest.

String and Stream

Print function and formatter

Format

- You may be annoying when you write a series of `<<` for stream.
  - E.g. `std::cout << "a=" << a << ",b=" << b << ",c=" << c << '\n'`;
  - But `printf` is just like `printf("a=%d,b=%d,c=%d\n", a, b, c);`
    - Sometimes, you may also hear of that `printf` is faster than `std::cout` dramatically…
    - However, `printf` is not type-safe, i.e. type cannot be identified to match `%d`.
    - Also, `printf` cannot be customized, but stream can overload `operator<<`.
  - So, with the great power of C++, why cannot we create a type-safe, customizable, convenient and fast function?
  - That's what `std::format` for!
    - Defined in `<format>`.
    - You can use `std::cout << std::format("a={},b={},c={}\n",a,b,c)`;
      - C++23 introduces `std::print`, which also utilizes format.

Format

- `std::format` is even faster than `sprintf()`!
  - For msvc, this requires e.g. /utf-8 because:
  - Release, x64, msvc:

/utf-8 . If you don't use the /utf-8 option then performance can be significantly degraded because we need to retrieve your system locale to correctly parse the format string. While we're

```
int main()
{
    measure("sprintf()     ", checkSPrintf);
    measure("ostringstream", checkOStringStream);
    measure("to_string()   ", checkToString);
    measure("to_chars()    ", checkToChars);
    measure("format()      ", checkFormat);
    measure("format_to()   ", checkFormatTo);
    return 0;
```

```
C:\WINDOWS\system32\cm
sprintf()     : 0.4757ms
 42 7.700000
ostringstream: 2.0376ms
 42 7.7
to_string()   : 0.9862ms
 42 7.700000
to_chars()    : 0.1477ms
 42 7.7
format()      : 0.3478ms
 42 7.7
format_to()   : 0.2317ms
 42 7.7
```

Format

- So, let's first tell you how to use these APIs.
  - The most basic ones are like `std::format("a={},b={},c={}\n",a,b,c)`.
    - To print `{` or `}`, you need to use `{{` or `}}`.
  - You can also specify order explicitly, e.g. like `std::format("c={1},a={0},c={1}\n",a,c)`.
- Beyond these, you can write format specifiers after `:`, e.g. `c={:xx}` or `a={0:xx}` (`0` is order).

Format

- The format is like `fill – align – sign - # - 0 – width - .precision – L – type`, and all parts are optional.
  - `fill`, `align`, `width` are used to fill into additional characters to keep a fixed length.
    - `width` means the total character numbers (including element itself).
    - `fill` means the character to fill in (by default whitespace).
    - `align` can be left(`<`), right(`>`) or middle(`^`) (by default left, except for **integers and floating points** to be **right**).
    - For example: `std::format(“{:0^8}”, 1)` will get `00010000`, `std::format(“{:08}”, 1)` will get `00000001`.
    - Note1: if element itself is longer than `width` (e.g. here we pass 10000000), then these specifiers will do nothing.
    - Note2: if only a number is given, then it’s seen as `width` instead of `fill`.
    - Note3: for [most of East Asian characters]and many emojis, width is seen as 2 instead of 1.

- Any code point whose Unicode property East_Asian_Width has value Fullwidth (F) or Wide (W)
- U+4DC0 - U+4DFF (Yijing Hexagram Symbols)
- U+1F300 – U+1F5FF (Miscellaneous Symbols and Pictographs)
- U+1F900 – U+1F9FF (Supplemental Symbols and Pictographs)

震惊，易经六十四卦出现在了C++标准中（

Format

- sign: can be - (default, i.e. only show sign when the number is negative, including -0.0), + (always show sign), and whitespace (only show negative sign, but non-negative number will fill into a whitespace).
  - For example, `std::format("{:0^+8}", 1)` will get `000+1000`, `std::format("{:+8}", 1)` will get `      +1`, `std::format("{: }", 1)` will get ` 1`
- type: how to show the element in many forms.
  - Integer: `b/B/d/o/x/X`; by default d(ecimal). Prefix is not output.
    - Particularly, `bool` has an additional `s` (default), which will get `true/false`;
    - `char/wchar_t` has an additional `c` (default), which will get the character; and an additional `?` since C++23, which will output raw character (e.g. `'\n'` instead of a new line).
  - Floating point: `e/f/g/a/E/F/G/A`; by default g(eneral).
    - Same as `std::chars_format::scientific/fixed/general/hex`, while default parameter to precision of `e/f/g/E/F/G` is 6.

Remember special case for precision in `general`?

Format

- String (including C-string, std::string, etc.): `s`; by default s.
  - C++23 adds a `?`, which outputs escaped string (surrounding “” will be output too).
- pointer: `p`; by default p. Same as `reinterpret_cast<std::uintptr_t>(ptr)`.
  - It will additionally output a `0x`.
  - C++26 adds a `P`, which will force hexadecimal letters to be uppercase.
- For example, `std::format(“{:m^+8x}”, 32)` will get `mm+20mmm`; `std::format(“{:e}”, 32.F)` will get `3.200000e+01`.

- `#`: alternative form.
  - For integers with type `x/X/o/b/B`, prefix `0x/0X/0/0b/0B` will be added.
  - For floating points, dot will always be shown (e.g. `42.f` by default is `“42”`, but will get `“42.”` with `#`).
    - Particularly, for explicit `#g/#G`, all zeros will be shown, e.g. `42.0000`.
  - For example, `std::format(“{:m^#8x}”, 32)` will get `mm0x20mm`.

Format

`std::format("{:+06}", 120)` -> +00120
`std::format("{:0>+6}", 120)` -> 00+120

- `.precision`: valid for floating point and string.
  - For floating point, same as `to_chars`, i.e. digits after dot.
  - For strings, it's the maximum characters to output.
  - For example, `std::format("{:.4e}", 32.F)` will get `3.2000e+01`.
- `0`: fill into 0 for only integers and floating points after sign and prefix (different from `fill`!), when `align` isn't specified.
- `L`: apply locale information on integers and floating points by the current locale (you can also provide `std::locale` as the first parameter to specify it).

- `fill – align – sign - # - 0 – width - .precision – L – type`.
  - If you've known `printf` formatting thoroughly, or formatting on some other languages, it's easy for you to understand formatting in C++.
- When the format string is invalid, compilation will fail; if the memory is not enough, `std::bad_alloc` will be thrown.

More flexibly, `width` and `precision` can be determined in runtime, i.e. you can use `std::format("{:{}.{}e}", 32.F, 3, 10)` (order is determined by the position of left brace) or `std::format("{0:{2}.{1}e}", 32.F, 10, 3)` are both equivalent to `std::format("{:10.3e}", 32.F)`.

Format

- With the format string, we can easily use `std::format(“xx”, yy)` (or `std::format(locale, “xx”, yy)`, or wstring version).
  - It generates a `std::string`, but we may already prepare a buffer so that dynamic allocation can be prevented (just like `to_string` with `to_chars`).
  - Then you can use `std::format_to(OutIt, …)` where `…` are arguments of `std::format`, which will output the string to the `OutIt`.
    - You must ensure that the destination range is large enough for the output; the size needed can be got by `std::formatted_size(…)`.
    - Of course, you can also use a `std::back_inserter`. Format functions are even optimized for it, which will not insert one by one but insert the whole directly.
  - If you cannot ensure buffer size, you can use `std::format_to_n(OutIt, n, …)`, so that if the size exceeds `n`, the string will be truncated.
  - Notice: these two functions don’t write null-termination!

It can only be passed directly; we'll tell you how to do that in Move Semantics!

Format

```cpp
auto runtime_fmt = std::runtime_format("{}");

auto s0 = std::format(runtime_fmt, 1); // error
auto s1 = std::format(std::move(runtime_fmt), 1); // still error
auto s2 = std::format(std::runtime_format("{}"), 1); // ok
```

- `std::format` only accepts a format string that can be determined in compilation time; what if we want a user-customized format string?
  - Since C++26, you can use `std::runtime_format`.
    - The format string may be invalid since it cannot be determined in compilation time.
      - So `std::format_error` may be thrown.

```cpp
#include <format>
#include <print>
#include <string>
#include <string_view>

int main()
{
    std::print("Hello {}!\n", "world");

    std::string fmt;
    for (int i{}; i != 3; ++i)
    {
        fmt += "{} "; // constructs the formatting string
        std::print("{} : ", fmt);
        std::println(std::runtime_format(fmt), "alpha", 'Z', 3.14, "unused");
    }
}
```

Output:

```
Hello world!
{}  : alpha
{} {}  : alpha Z
{} {} {}  : alpha Z 3.14
```

Notice that VS2019 should be updated to latest version, otherwise `std::format` may throw `std::format_error` though check is done when compiling.

Format

- Before C++26, you can use `std::vformat(_to)`…
  - Similarly for `std::vprint(_unicode/nonunicode)` ~ `std::print`.
  - The arguments slightly change, i.e. it should use `std::make_format_args`.
  - E.g. `std::string fmt{ “{}, {}\n” }; std::vformat(fmt, std::make_format_args(a, b))`.
- **But it’s dangerous!**
  - Reason: `std::make_format_args` stores reference.
    - `auto args = std::make_format_args(std::string{“Dangling”}); std::vformat(fmt, args)` => The temporary string is already destructed.
  - So rvalues will also be rejected since C++26, see P2095R2.
- Anyway, `vformat` should never be called by us since C++26. It’s utilized by the implementer of `<format>` only to “reduce code bloat”.
  - We’ll tell you the technique in Template!

Format

- C++23 reinforces functionality of `std::format`, i.e. it can format a range.
  - Yeap, it saves the burden of `for(const auto& ele : vec) std::cout << ele << ‘ ’;`.
  - By default, ranges are output as [...] (i.e. *sequence*).
    - E.g. `std::vector v{1,2,3}` will get `[1, 2, 3]`, `std::vector<std::vector<int>> v{{1,2},{3,4}}` will get `[[1, 2], [3, 4]]`.
  - Some ranges are specialized:
    - For container adaptors (i.e. stack, queue, priority_queue), they don’t provide iterators so it’s hard to print them before; now they also support `std::format`.
    - For `std::vector<bool>`, the proxy type doesn’t support output before.
    - For `std::pair/std::tuple`, you will get `(xx, yy, …)`.
    - For associative containers, you will get `{xx:yy, aa:bb,…}` or `{xx, aa}` (depending on *map*/*set*).

Format

  - Particularly, if the element of range is char/string, then escaped character/string will be output (i.e. as if it's `{:?}`).
  - You can use `std::format_kind<R>` to check how a range is formatted; it will get `std::range_format::xx`, where `xx` is `disabled/map/set/sequence/string/debug_string` (i.e. escaped string).
  - If you're adept at Python, you may be very happy that they're same!
- Of course, you can add format specifier for elements and ranges.
  - For ranges, the type specifiers have only 5 options:
    - `s/?s`: only valid for range of string, output string/escaped string.
    - `m`: only valid for range of pair/tuple of size 2, output as `{k1: v1, k2: v2, …}` (like element of **m**ap).
    - `n`: strip the surrounding wrapper, e.g. [xx, yy] will become xx, yy (so that you can customize your parathesis).
    - `nm`: combine `n` with `m`, i.e. output as `k1: v1, k2: v2, …`.
  - So, you can use e.g. `std::format("{:n}", v)`.

Format

- If you want to specify format of elements, then you can use multiple colons, i.e. `std::format(“{:n:x}”, v)`.
  - If elements are still ranges, just continue to specify it; anyway, each colon specifies elements at one level.
- For example, for a `vector v{vector{'a'}, vector{'b', 'c'}};` and you want to output it in a flattened way with decimal numbers, you can use `“{:n:n:d}”`.
  - Normally, you’ll get `[[‘a’], [‘b’, ‘c’]]`;
  - The first `n` means to strip `[]` of `v`, i.e. get `[‘a’], [‘b’, ‘c’]`.
  - The second `n` means to strip all `[]` of elements of `v`, i.e. get `‘a’, ‘b’, ‘c’`.
  - Final `d` means to convert characters to decimal values, i.e. get `97, 98, 99`.
  - Notice that for string elements, `“{::}”` is different from `“{}”`, since the former will disable escaped string output.
- Finally, range formatter also support `fill, align, width` specifiers, so thoroughly you can use `fill – align – width – type`.
  - E.g. `“{:*^14n:n:d}”` will get `**97, 98, 99**`.

Print function

- By format, C++23 introduces print functions in `<print>`.
  - `std::print("{}", v)`, or `std::println("{}", v)` to print an additional `'\n'`.
    - C++26 adds `std::println()` (i.e. no parameter), which will only output a new line; Big3 all see it as DR23.
  - It uses C output destination (i.e. `stdout` instead of `std::cout`), but faster than `printf`.
    - So it may have synchronization problem when you both use `std::cout` and `std::print` when you decouple them (we'll tell you how to do that in stream).
  - It can also specify a stream or `FILE*` as the first parameter, so that you can output to a file.
    - You can surely output to `std::cout`, which is still faster than use `std::cout <<` directly.

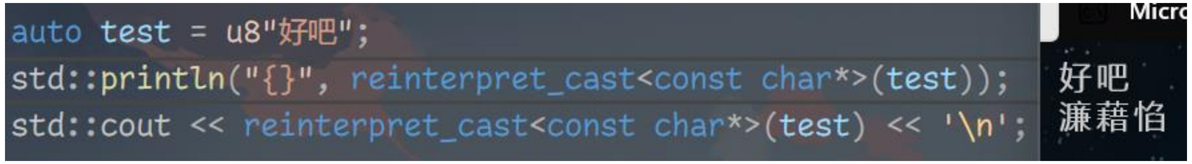
Mac Clang -O3:

Benchmark	Time	CPU	Iterations
printf	87.0 ns	86.9 ns	7834009
ostream	255 ns	255 ns	2746434
print	78.4 ns	78.3 ns	9095989
print_cout	89.4 ns	89.4 ns	7702973
print_cout_sync	91.5 ns	91.4 ns	7903889

VS2019, debug

Benchmark	Time	CPU	Iterations
printf	835 ns	816 ns	746667
ostream	2410 ns	2400 ns	280000
print	580 ns	572 ns	1120000
print_cout	623 ns	614 ns	1120000
print_cout_sync	615 ns	614 ns	1120000

Print function



- Beyond that, `std::print/println` supports UTF-8 output.
  - But bit of ironically, it accepts `char[]` instead of `char8_t[]`.
    - `std::print("\U0001F449")` √
    - `std::print(u8"\U0001F449")` × (compile error).
    - `std::print("{}", u8"\U0001F449")` × (compile error);
      - `char8_t[]` doesn't support format.
- They may throw `std::system_error` when output stream goes wrong, or other exceptions of `std::(v)format`.
  - Particularly, if the Unicode string is invalid, then UB (but it's encouraged to be diagnosed by library implementation by e.g. `assert`).

C++26 accepts two proposals (likely to be DR23) to make `std::print` even faster (20% ~ 200%). See [my analysis article]!

User-defined format

- There is one more thing… streams can be overloaded so it's more convenient than `printf`; so how can we customize format for our own types?
- In essence, resolving format has two phases:
  - **Parsing**: the formatter should parse things in the `{}`, and record necessary states.
  - **Formatting**: according to the recorded states of parsing, the string is inserted from back.
- So similarly, customized format needs these two methods.
- Let's give an example by output scoped enumeration name!

```
enum class Color { Red = 0xFF0000, Green = 0x00FF00, Blue = 0x0000FF, White = 0xFFFFFF };
```

User-defined format

- You need to specialize `std::formatter<T>`. `template<> struct std::formatter<Color>`
  - Just remember it now, and we'll talk about template specialization in the future.
  - Then just define a `constexpr auto parse(const std::format_parse_context&)`.
    - You can think the parameter as a `std::string_view` with strictly constrained methods; iterators can be got by `.begin()/.end()`.
      - The type of iterators is actually `std::format_parse_context::const_iterator`.
      - For `{:…}`, it refers to `…}` (the right brace is still there!).
      - For `{:}/{}`, it refers to either(for libstdc++) `}` or(for MS STL) an empty string view (you may think it as a view to `nullptr` whose size is 0).
    - The return type of `parse` is also `const_iterator`, which specifies the position to continue the following parse.
      - To be exact, you should usually return `context.end() – 1`; if the received string view is empty, then just return `context.end()`.
        - Otherwise compile error (or for `vformat()` exception `std::format_error` is thrown); you can utilize this property to check whether users input redundant specifiers.

`pc.begin()` points to the beginning of the *format-spec* ([format.string]) of the replacement field being formatted in the format string. If *format-spec* is empty then either `pc.begin() == pc.end()` or `*pc.begin() == '}'`.

User-defined format

- We assume that we only accept specifiers ‘x’ or ‘s’, and the former will show the color hexadecimal id (i.e. #rgb, e.g. #FF0000 for pure red), the latter (default one) will show the string (e.g. “Red”).
- Aughhh, too long, just show me the code!
  - When parsing, we first store the specified state.

```cpp
template<>
struct std::formatter<Color>
{
    char type = 's';
    constexpr auto parse(const std::format_parse_context& context)
    {
        auto it = context.begin();
        if (it == context.end() || *it == '}')
            return it;
        type = *(it++);

        if (type != 'x' && type != 's')
            throw std::format_error{ "Unrecognized color format." };
        return it;
    }
```

Default is ‘s’.

Double check for cross-platform ability.

it should be context.end() - 1, otherwise compile error.

```cpp
template<>
struct std::formatter<Color>
{
    char type = 's';
    constexpr auto parse(const std::format_parse_context& context)
    {
        auto it = context.begin();
        if (it == context.end() || *it == '}')
            return it;
        type = *(it++);

        if (type != 'x' && type != 's')
            throw std::format_error{ "Unrecognized color format." };
        return it;
    }
```

- Notice that `constexpr` has many restrictions to be called in compilation time, e.g. you cannot print `[begin, end)` to see the content of the context.
  - You can throw an exception in some condition (here `type != ‘x’ && type != ‘s’`), and if the condition is satisfied, compile error (since exception is runtime thing!).
- To conclude:
  - `{}, {:}, {:x}, {:s}` are all OK.
  - Compile error (or throw exception for `std::vformat`):
    - `{:c}, {:11}`, since we check it in `type != ‘x’ && type != ‘s’`.
    - `{:x11}`, since returned `it != context.end() - 1`.

User-defined format

- After parsing, we need to utilize saved state to fill in string.
  - This is done at runtime, so no `constexpr` needed; but the saved state shouldn't be changed, so `const` should be added.
  - That is, you need `auto format(const T&/T, auto& context) const { … }`, where `T` is the object type (`Color` here).
    - You can use `.out()` to get the output iterator, where you need to append new contents; the function should return new iterator.

You can use **`std::format_context& context`** as parameter to help Intellisense to work, and make it `auto&` when all other things are done.

```cpp
auto format(Color color, auto& context) const
{
    auto formatByType = [outIt = context.out(), type = type]
        (std::string_view stringInfo, std::string_view numberInfo)
    {
        return type == 's' ? std::format_to(outIt, "{}", stringInfo)
            : std::format_to(outIt, "{}", numberInfo);
    };
```

```cpp
switch (color)
{
    using enum Color;
case Red:
    return formatByType("Red", "#FF0000");
case Green:
    return formatByType("Green", "#00FF00");
case Blue:
    return formatByType("Blue", "#0000FF");
case White:
    return formatByType("White", "#FFFFFF");
```

User-defined format

- Additionally, we need a `default` branch, since users may use e.g. `Color{0x1000}` to explicitly create an enumeration!
  - Notice that we have a `White = 0xFFFFFF` so that it's legal.
    - Remember? Range of scoped enumeration is the `(1 << (MSB(MaxEnum) + 1))- 1`, e.g. if we only use `{R, G, B}`, then the max allowed value is `(1 << (MSB(B)+1)) - 1 = (1 << (MSB(2) + 1)) - 1 = 4 – 1 = 3`; otherwise UB.
  - So:

```cpp
default:
{
    auto outIt = context.out();
    if (type == 's')
        outIt = std::format_to(outIt, "Unknown color:");
    return std::format_to(
        outIt, "#{:0>6x}",
        static_cast<std::underlying_type_t<Color>>(color));
    // Or std::to_underlying since C++23.
}
```

User-defined format

- The result is like:

```cpp
std::cout << std::format("{}\n", Color::Red);
std::cout << std::format("{}\n", Color{ 0x0100 });
std::cout << std::format("{:s}\n", Color::Green);
// 0xFF == Color::Blue.
std::cout << std::format("{:s}\n", Color{ 0xFF });
std::cout << std::format("{:x}\n", Color::Blue);
std::cout << std::format("{:x}\n", Color{ 0x1000 });
```

```
选择 C:\WINDOWS\system
Red
Unknown color:#000100
Green
Blue
#0000FF
#001000
请按任意键继续. . .
```

- Note that `context` has some other member functions (**optional, rarely used**):
  - `.advance_to(newIt)`, so that the next time `.out()` is called, `newIt` is returned.
    - This is used by standard library after you return a new iterator.
    - `std::format_parse_context` has a `.advance_to` similarly.
  - `.locale()`.
  - `.arg(size_t id)`, return the `id`th argument; it's in fact `std::basic_format_arg` and can be seen as having a `std::variant<…>`. See here if you're interested.

User-defined format

- Note1: now you may understand why it can be faster than `printf`!
  - It tries to parse when compiling, and only fill string in runtime.
  - C and C functions cannot utilize compile-time evaluation.
- Note2: you can save the context string in the member in `parse`, and then print it in `format` to debug, which saves the trouble that you cannot output in `parse`.
  - For example:

```cpp
std::string state; std::ranges::copy(it, context.end(), std::back_inserter(state));
std::cout << state << '\n';
```

- Note3: the general parameter is in fact `std::basic_format_context<T>/std::basic_format_parse_context<T>`, which can also support e.g. `std::wstring`; if you want that, `auto&` and `const auto&` as parameter types are also OK.

User-defined format

- Note4: it's also acceptable to inherit or own another `std::formatter<T>` as member, so that you can utilize its parsing or formatting. For example:

By inheritance:

```
template<>
struct std::formatter<Always42> : std::formatter<int>
{
  auto format(const Always42& obj, std::format_context& ctx) {
    // delegate formatting of the value to the standard formatter:
    return std::formatter<int>::format(obj.getValue(), ctx);
  }
};
```

`.parse()` is same as `int`.

```
constexpr auto parse(std::format_parse_context& ctx) {
  return f.parse(ctx);
}
```

```
std::formatter<int> f;
```

By member:

```
// delegate formatting of the int value to the standard int formatter:
auto format(const Always42& obj, std::format_context& ctx) const {
  return f.format(obj.getValue(), ctx);
}
```

Credit: *C++20 - The Complete Guide* by Nicolai Josuttis.

User-defined format

- Note5: C++23 also adds `std::range_formatter<T>`, which is used to be inherited by `std::formatter<Container<T>>` for customizing range of `T` to output.
  - Yes, you are usually not supposed to specialize `std::range_formatter<C<T>>`, but `std::formatter<C<T>>` with parent class `std::range_formatter<T>`.
  - `std::range_formatter` has `.set_brackets(left, right)` `/.set_separators(sep)`, and `.underlying()` to get the `std::formatter` of the element, and `.parse()/.format()` (so that usually you don’t need to specify new `parse/.format` after inheriting from it).
    - Particularly, `pair` and `tuple` are not ranges, so their `std::formatter` are specialized with these new methods added, and has no `.underlying`.

User-defined format

You will know why you can write template parameter like this in *Templates*!

- For example:

```cpp
template <std::formattable<char> Key, class Compare, class Allocator>
struct std::formatter<std::set<Key, Compare, Allocator>>
    : std::range_formatter<Key>
{
    constexpr formatter() {
        this->set_brackets("{", "}");
    }
};
```

Inheriting from **template base class** will make `this->` obligated.
We'll also tell you why in *Templates*.

```cpp
template <std::formattable<char> Key, std::formattable<char> T,
            class Compare, class Allocator>
struct std::formatter<std::map<Key, T, Compare, Allocator>>
    : range_formatter<std::pair<Key const, T>>
{
    constexpr formatter() {
        this->set_brackets("{", "}");
        this->underlying().set_brackets({}, {});
        this->underlying().set_separator(": ");
    }
};
```

Possible future of format

- As you can see, format and print functions are super powerful and convenient.
  - However, there are still several minor things to improve.
  - 1. You cannot specify different formats for elements of pair/tuple.
  - 2. Vocabulary types format?

  - Besides, we only introduce output methods, but where are input methods?
    - `std::scan` has been proposed here in 2023, which is said to be faster and more convenient too. Hopefully it can be accepted in C++26.

Notice that C++ has different format for time and thread id; we'll mention them when taking about them.