
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 your IDE to give you hints!
- Now let's talk about some interesting things about string...
- Note1: Remember what we've mentioned in review lecture?

你见过最烂的代码长什么样子?



天苍苍

+ 关注

25 人赞同了该回答

```
1 #include<iostream>
2 int main()
3 {
4
5     std::string s = "hello world";
6     if (s.find("t")>=0)
7     {
8         std::cout << "found\\n";
9     }
10    else
11    {
12        std::cout << "not found\\n";
13    }
14
15
16 }
```

found

F:\projects\CPP_PROJECTS\CppTest\x64\Debug\CppTest.exe (进程 58780) 已退
要在调试停止时自动关闭控制台, 请启用“工具”->“选项”->“调试”->“调试停止时
按任意键关闭此窗口”

我之前遇到一个问题, 这是代码简化版本, 看大家能理解么

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/stou/stoull(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:

```
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:

- 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`.

```
std::vector<std::string> vec{ "PKU", "THU", "CMU"};
std::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);
```



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.

```
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`.

```
template<typename T>
T Add(T a, T b) { return std::string{ a } + std::string{ b }; }
```
 - Use `auto` instead.
- 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.

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

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

```
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";
}
```



```
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.

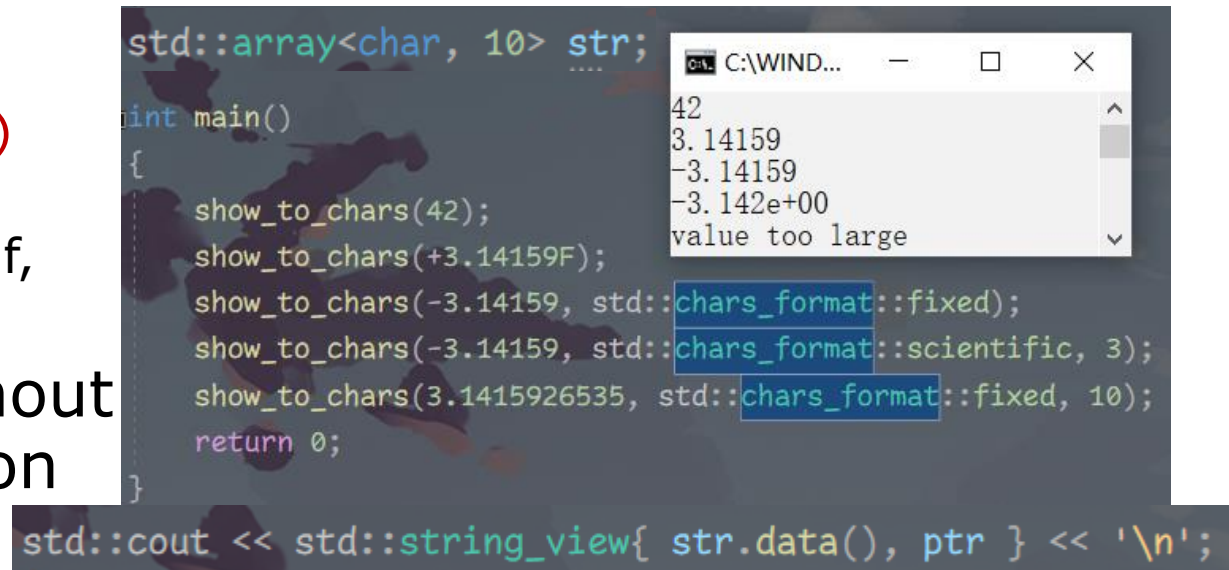
Unfortunately, `libc++` doesn't support `std::from_chars` for floating points yet even in 2024/6 (see [impl status](#)). `libstdc++`/MS-STL all support it.

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.



The screenshot shows a C++ program in a code editor and its output in a console window. The code defines a `std::array<char, 10> str;` and a `main()` function. Inside `main()`, it calls `show_to_chars(42);`, `show_to_chars(+3.14159F);`, `show_to_chars(-3.14159, std::chars_format::fixed);`, `show_to_chars(-3.14159, std::chars_format::scientific, 3);`, and `show_to_chars(3.1415926535, std::chars_format::fixed, 10);`. The console window shows the output: `42`, `3.14159`, `-3.14159`, `-3.142e+00`, and `value too large`. The code at the bottom of the editor is `std::cout << std::string_view{ str.data(), ptr } << '\n';`.

```
std::array<char, 10> str;

int main()
{
    show_to_chars(42);
    show_to_chars(+3.14159F);
    show_to_chars(-3.14159, std::chars_format::fixed);
    show_to_chars(-3.14159, std::chars_format::scientific, 3);
    show_to_chars(3.1415926535, std::chars_format::fixed, 10);
    return 0;
}

std::cout << std::string_view{ str.data(), ptr } << '\n';
```

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 Å).
 - 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!

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

 - 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.

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++



The screenshot shows two GitHub issues from the 'Unicode in the Library' repository. The top issue is 'P2729 Unicode in the Library, Part 2: Normalization' (B3 - add), opened on Jan 16, 2023, by wg21bot, with a link to a 2024 telecon. The bottom issue is 'P2728 R6 Unicode in the Library, Part 1: UTF Transcoding' (B), opened on Jan 16, 2023, by wg21bot, with a 'unicode' label.

- 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.

```
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;
}
```

```
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:

```
C:\WINDOWS\system
std::string s = "北京";
std::u8string s2 = u8"北京";
```

e5 8c 97 e4 ba ac
e5 8c 97 e4 ba ac

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

```
std::string s = "北京";
std::u8string s2 = u8"北京";
std::u16string s3 = u"北京";
std::u32string s4 = U"北京";

DecomposeToByte(s);
DecomposeToByte(s2);
DecomposeToByte(s3);
DecomposeToByte(s4);
```

C:\WINDOWS\system32\cm
b1 b1 be a9
e5 8c 97 e4 ba ac
17 53 ac 4e
17 53 00 00 ac 4e 00 00

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...

⚙	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
000EAA50	00 00 00 00 6E 61 6E 00 6E 61 6E 28 73 6E 61 6E
000EAA60	29 00 00 00 69 6E 69 74 79 00 00 00 61 6E 00 00
000EAA70	69 6E 64 00 73 6E 61 6E 00 00 00 00 6E 66 00 00
000EAA80	69 6E 69 74 79 00 00 00 61 6E 00 00 69 6E 64 00
000EAA90	73 6E 61 6E 00 00 00 00 30 70 2B 30 00 00 00 00
000EAAA0	30 70 2B 30 00 00 00 00 30 65 2B 30 30 00 00 00
000EAAB0	30 65 2B 30 30 00 00 00 CE D2 B7 C5 CA D6 00 00
000EAAC0	E6 88 91 E4 BB BB E8 B5 B0 00 00 00 00 00 00
000EAAD0	47 50 12 6D 31 81 00 00 01 8C C2 61 11 62 1A 59
000EAAE0	48 4E 0D 4E 0D 82 97 5F 00 00 00 00 00 00 00
000EAAF0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000EAB00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000EAB10	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000EAB20	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000EAB30	A0 D4 0F 40 01 00 00 00 16 5E 00 40 01 00 00
000EAB40	52 18 00 40 01 00 00 00 00 00 00 00 00 00 00

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.

我任走 in UTF-8 假洒脱 in UTF-16

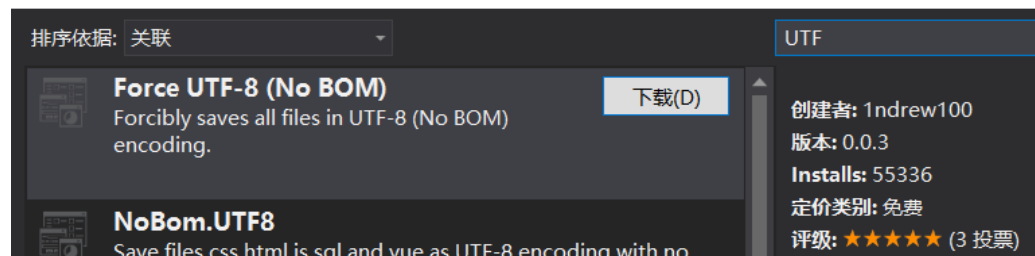
谁懂我多么不舍得 in wide character encoding (UTF-16 on Windows)

Unicode support in C++

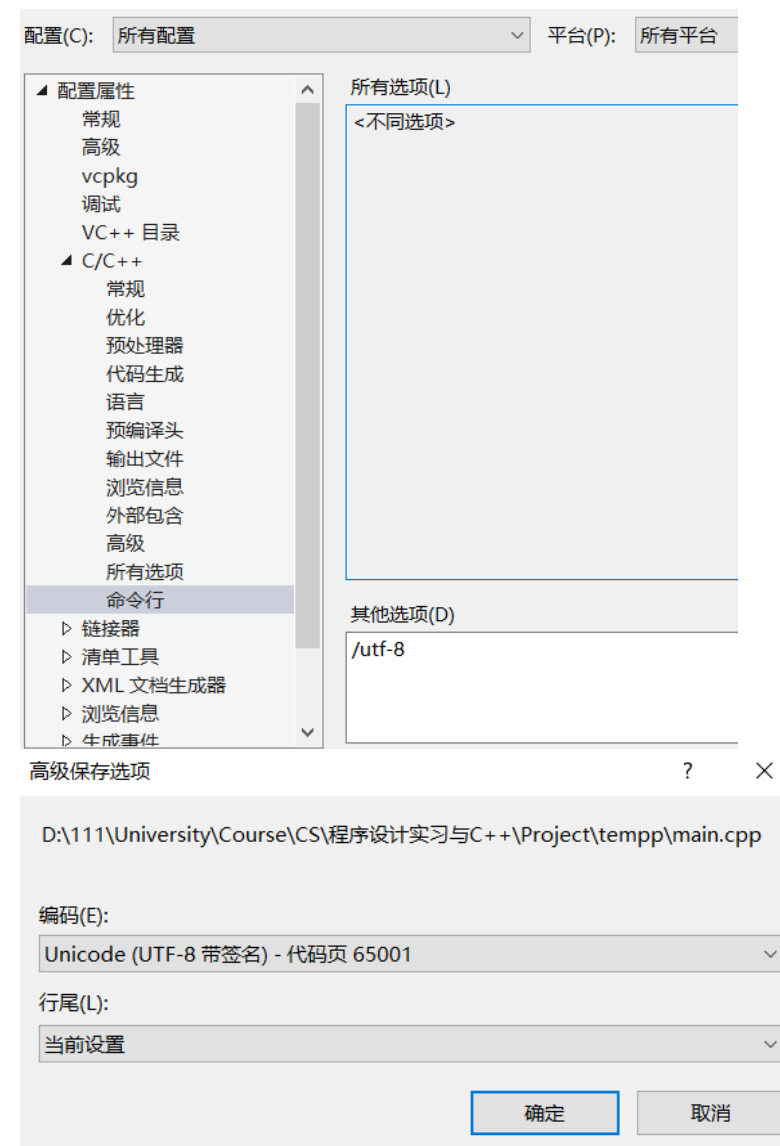
/utf-8 == /source-charset:utf-8
+ /execution-charset:utf8

- 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).

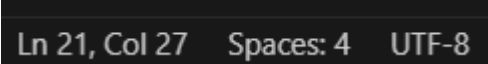
- One way is 扩展 -> 搜索UTF -> 下载:



- 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: Ln 21, Col 27 Spaces: 4 UTF-8
- 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`, `num_punct`.
 - `num_get` will affect how `std::cin` works, `num_put` -> `std::cout`, `num_punct` 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/codecv`t, how characters are classified and converted, e.g. is upper.
 - BTW, `<codecv`t> is deprecated in C++17 and removed in C++26, but in fact standard `codecv`t is defined in `<locale>`, so it doesn't affect anything.
 - Before `codecv`t 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*

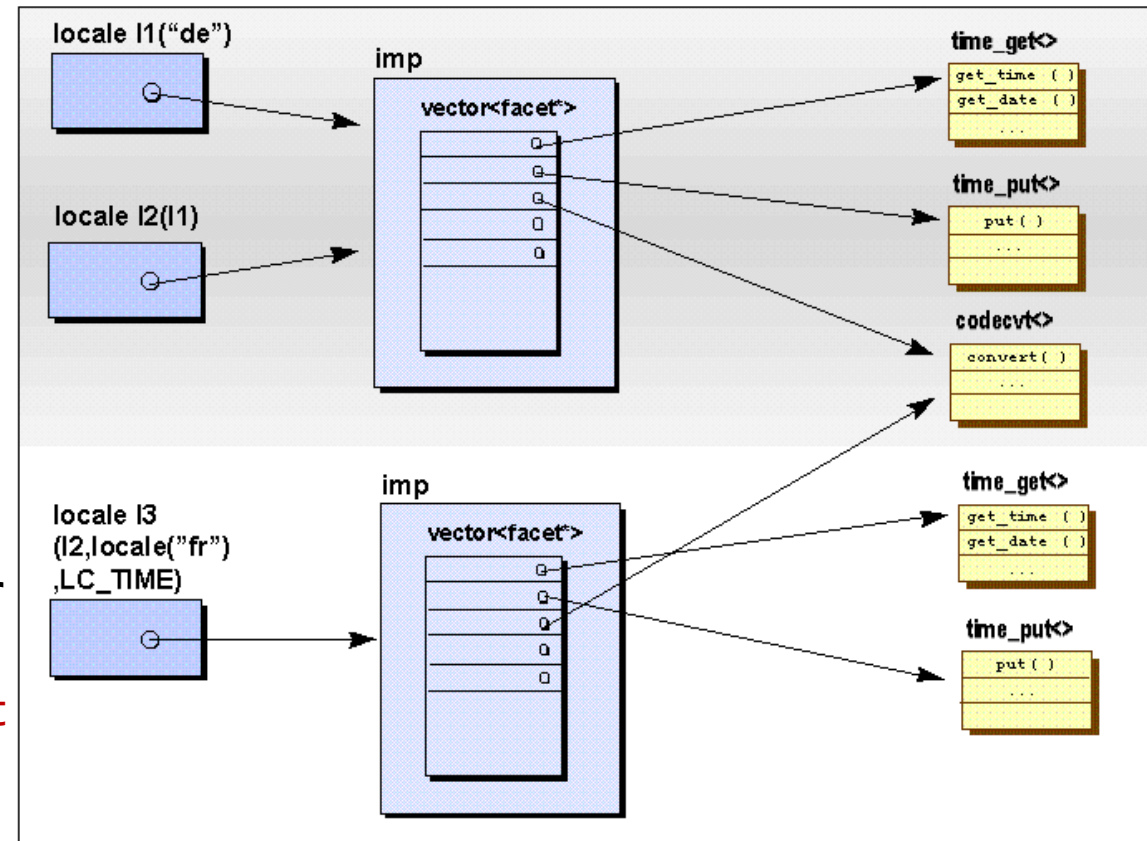
`<cctype>` have similar functions, which uses the global locale (by default `classic()`), but you can use `std::locale::global(Loc)` to set another one, so it's possibly not safe.

- `<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()`.

[illegible]

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`.



```
struct french_bool : std::num_punct<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 num_punct: "
               << 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 `num_punct` 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*

- 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.

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

```
class Umlaut : public std::locale::facet { //1
public:
    static std::locale::id id; //2 New id address, new facet group.
    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
};

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';
    }
}
```

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: `/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
 - Release, x64, msvc:

```
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\cmd

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

```
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.

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

```
#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
```

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 `map`).
 - `n`: strip the surrounding wrapper, e.g. `[xx, yy]` will become `xx, yy` (so that you can customize your parentheses).
 - `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

```
auto test = u8"好吧";  
std::println("{}", reinterpret_cast<const char*>(test));  
std::cout << reinterpret_cast<const char*>(test) << '\n';
```

Micro
好吧
濂藉怡

- 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 = 0xFFFFFFFF };
```

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.

```
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.

```

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.

```
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);
    };
}
```

```
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 = 0xFFFFFFFF** so that it's legal.
 - Remember? Range of scoped enumeration is the $(1 \ll (\text{MSB}(\text{MaxEnum}) + 1)) - 1$, e.g. if we only use {R, G, B}, then the max allowed value is $(1 \ll (\text{MSB}(\text{B}) + 1)) - 1 = (1 \ll (\text{MSB}(2) + 1)) - 1 = 4 - 1 = 3$; otherwise UB.
- So:

```
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:

```
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 });
```

C:\ 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:

```
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);
    }
};

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

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

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:

```
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*.

```
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.

String and Stream

Stream

String and stream

- Stream
 - Stream overview
 - Output stream
 - Input stream
 - Bidirectional stream and stream linking
 - Standard streams

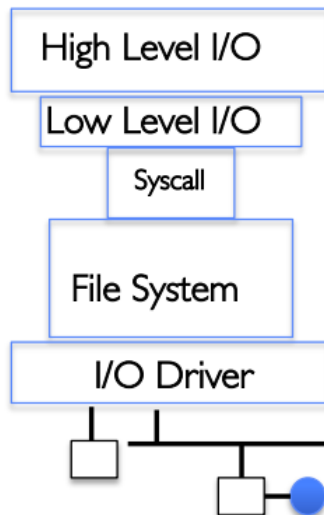
Stream

- Stream is a high-level abstraction of readable/writeable buffer; No matter it's in fact console, file or just memory, users usually don't need to care about the details, but just see it as a continuous buffer to read/write.
- It's a successful OOP design in C++98, but unluckily console ones (i.e. `std::(w)cout/(w)cin`) aren't favored by many because of their horrible efficiency.
 - However, it's still worthwhile to learn other streams (e.g. there shouldn't be great performance gap between file stream and `FILE*`).
 - There is also an `<iomanip>` to manipulate format of output result, but it's recommended to just use format specifiers of `std::format/print` because it's more efficient and user-friendly, so we may only cover `iomanip` roughly and it's also just optional if you can use C++20.

Stream

- Before we go on, let's talk about IO system in OS.
 - You'll learn it thoroughly in your OS course, and here we just give a brief view.
 - Roughly speaking, IO system can be divided into the following parts:

Application / Service



Streams

File Descriptors

open(), read(), write(), close(), ...

Open File Descriptions

Files/Directories/Indexes

Commands and Data Transfers

Disks, Flash, Controllers, DMA

What you've learnt and used in ICS
(file descriptors, Linux file APIs).

Stream

- We know that system call is relatively expensive, so if syscall is used every time when users need something in file, then it would be very slow.
 - Yeap, it's similar to disk storage; you need a cache in memory!
 - That is, we prepare a buffer and read a lot of things from the file into the buffer even if users haven't needed it, and then just adjust pointer to buffer when reading.
 - Once the pointer reaches its end, we need to load new content from file again.
 - This makes an illusion to users to read continuously from the file.
 - So space locality is also important in IO, which can reduce system call.
 - Similarly, when users write to the stream, it doesn't really go down to the disk directly, but is saved in the buffer; once the write buffer is full, then everything is **flushed** into the disk.

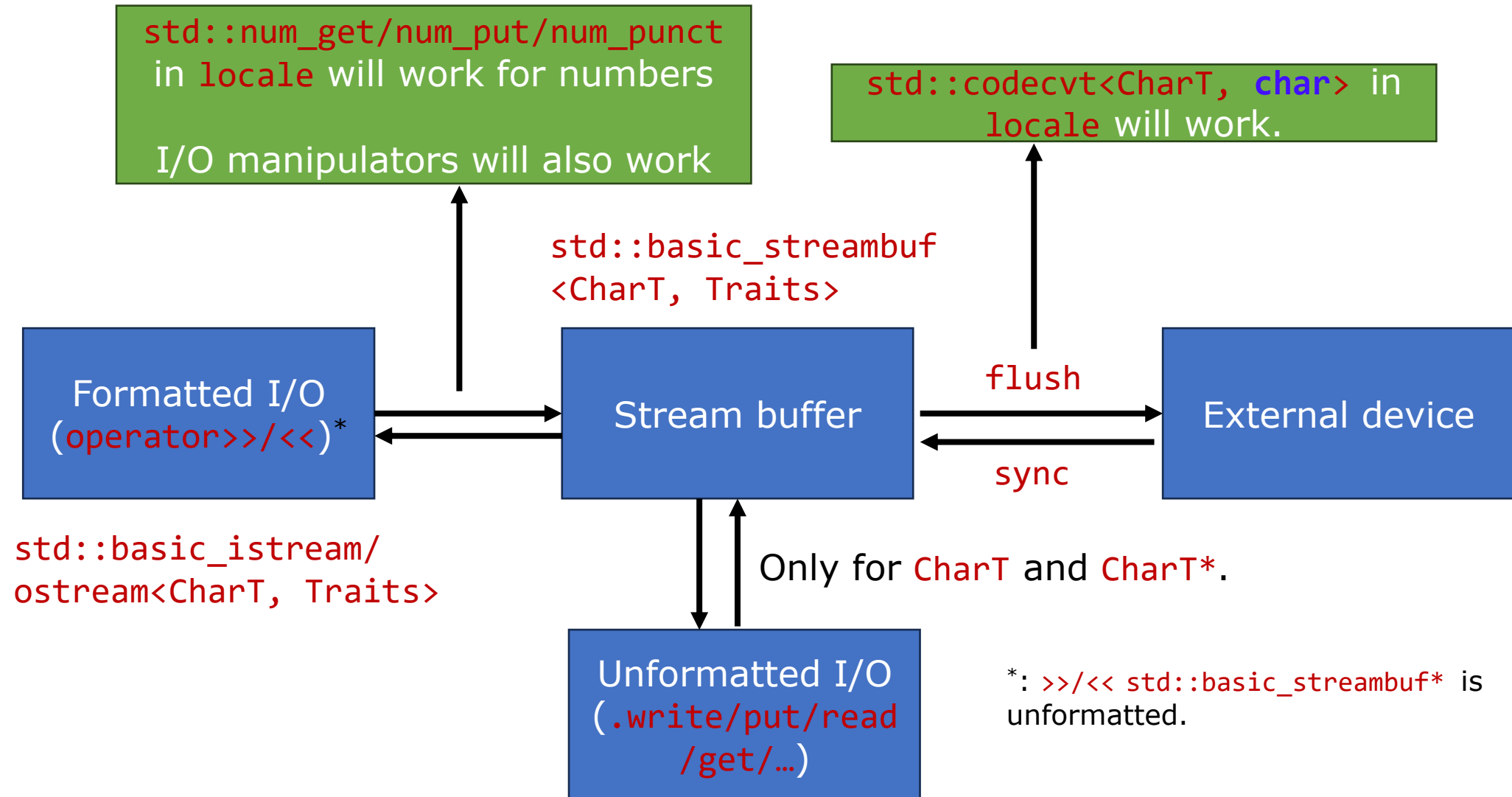
"Flush to disk" may be not rigorous because it depends on OS; That is, flush in our application level may still be cached in the OS buffer, which needs OS-dependent API to go to disk thoroughly.

std::endl v.s. '\n'

- After knowing this, an important and basic problem can be solved: what's the difference between `std::endl` and `'\n'`?
 - Your first C++ program is likely to be `std::cout << "Hello, world!" << std::endl`.
- `std::endl` is in fact a special IO manipulator, meaning that "output a newline, and **flush the stream**".
 - That is, if you output a `'\n'`, file/console may not get it immediately until proper time; `std::endl` will force it to be there.
 - So, it will be dramatically slow if you call `std::endl` over and over again.
 - If you don't want flush that frequently, it's recommended to use `'\n'`, and `<< std::flush` when you really need.
 - When the stream is destructed, it will be flushed automatically too.

Notice that there is also a `std::cerr`, which is similar to `stderr` (instead of `stdout/stdin` that you've learnt in ICS) and has no buffer because it wants to warn users error immediately; if you want buffered one, you can use `std::clog`.

Design overview



String and stream

- Stream
 - Stream overview
 - Output stream
 - Input stream
 - Bidirectional stream and stream linking
 - Standard streams

Output stream

benchmark name	samples		iterations		estimated	
	mean	std dev	low	mean	high	mean
			low	std dev	high	std dev
write		100		37		1.776 ms
	489.514 ns		441.838 ns		607.432 ns	
	355.161 ns		180.138 ns		719.308 ns	
<<		100		35		1.8235 ms
	655.943 ns		539.229 ns		952.8 ns	
	886.931 ns		426.673 ns		1.77422 us	

- There are two kinds of streams: output ones and input ones.
- For output stream, it's inherited from `std::basic_ostream`.
 - It's in fact a template, i.e. `<Char, CharTraits = std::char_traits<Char>>`.
 - Formatted output: just use `<<` (sometimes named "insertion operator"), like what you've done in `std::cout`.
 - Unformatted output: not affected by io manipulator.
 - Only for raw character (& its array) output; return reference to `*this`.
 - `.put(ch)` and `.write(chArr, len)`.
 - They're similar to `putc()`, `putchar()`, `puts()` in C.
 - For example, `.write(fmtStr.data(), fmtStr.size())` is same as `<< fmtStr` with no manipulator.
 - Sometimes, you may want to jump to other positions (as if you move the pointer to some place of the whole buffer).
 - You can use `.tellp()` and `.seekp()` to get or set the "**put**" position.

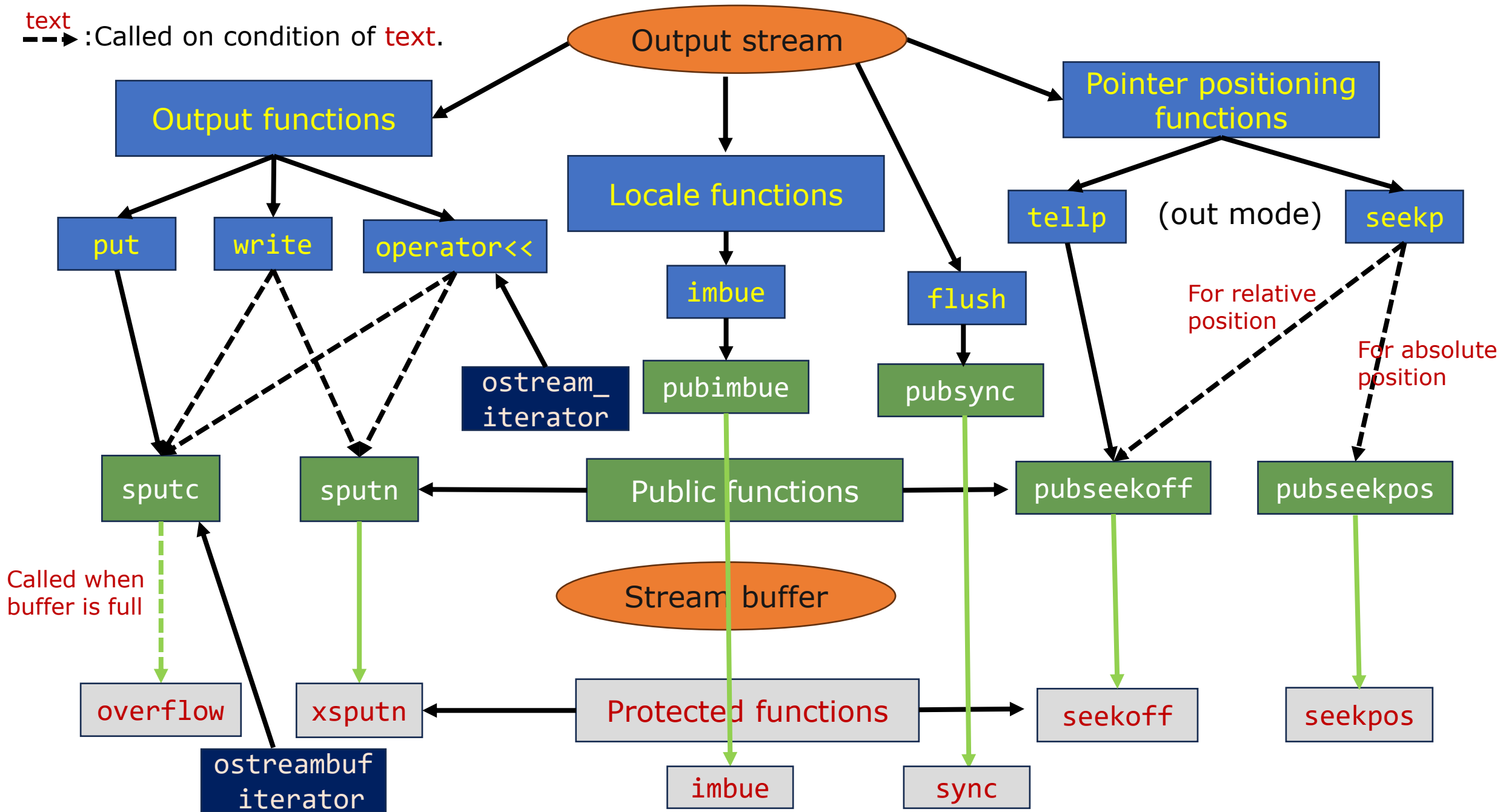
Output stream

Note that `std::cout` etc. don't have meaningful `.seekp/tellp()`, and the behavior is platform-dependent.

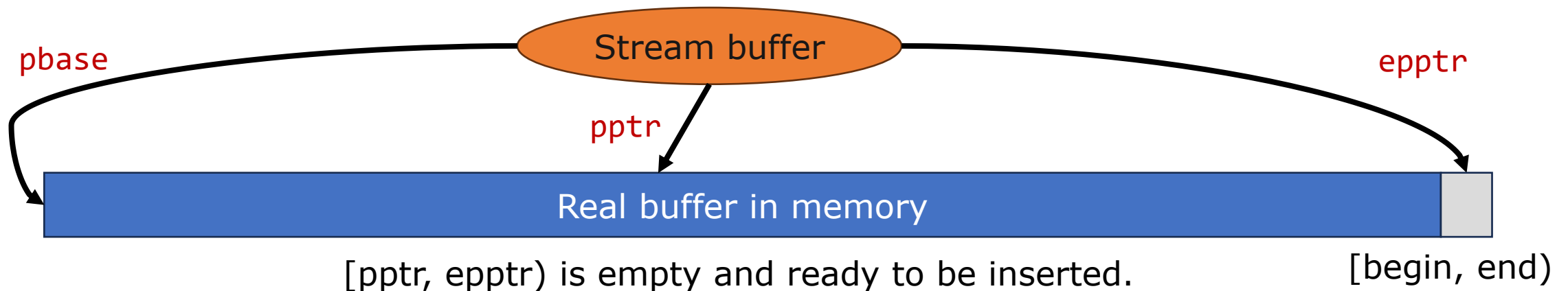
- `.tellp()` returns & `.seekp()` accepts `std::basic_ostream<xx>::pos_type`, which can be seen as a signed integer (in fact a class `std::streampos`).
- Besides, you can use relative position to seek, i.e. by `.seekp(off_type off, std::ios_base::seekdir dir)`.
 - `std::ios_base::seekdir` has three constant global values (not enumerations!).
 - `beg`, `end` and `cur`.
 - `off_type` is usually a signed integer type, so you may advance or withdraw the pointer.
- For example, `.seekp(-1, std::ios_base::end)` will place the pointer to the last character of the stream, `.seekp(0, std::ios_base::beg)` will rewind the stream, and `.seekp(2, std::ios_base::cur)` will advance the pointer by 2.

--> :Depend on implementation to use the one or the another or both.

text
--> :Called on condition of text.



- `std::basic_streambuf` also handles buffering automatically, so derived class doesn't need to care "when buffer is full".
 - Usually the derived class has a default memory buffer, while users can use `.pubsetbuf()` to set it by themselves.
 - This calls `.setbuf()` of the derived class, so it depends on the implementation of the derived class to either use it or not.
 - The buffer is abstracted as three components: begin, curr, end.
 - For output, it's the protected methods `pbase/pptr/epptr`.
 - For example, `sputc` will write into `*pptr` directly and advance `pptr`.
 - Only when `pptr == epptr` (i.e. the buffer is full so put is impossible) will it call virtual method `overflow` to let the derived class to give some extra space.
 - E.g. for file, flush all things to the disk and rewind `pptr` to `pbase` by `setp`.
 - If you want to adjust `pptr` manually, you can use `.pbump(int off)`.



Output stream

- The stream buffer can be got by `.rdbuf()`, which returns `std::basic_streambuf*`.
 - It's a **base class** that has a `protected` ctor, which uses **polymorphism** to access the actual buffer (e.g. `std::basic_filebuf` for file stream).
 - This facilities your possible customization of the buffer.
 - As we mentioned, for output buffer of stream, there are several things that can be used or customized (these functions are `protected`):
 - `.pbase()/.pptr()/.epptr()`: return `begin/cur/end` pointer.
 - `.setp(pBeg, pEnd)`: set the `begin/cur/end` pointer to `pBeg/pBeg/pEnd`.
 - `.pbump(int off)`: `cur += off`; `cur` can go out of `[begin, end)`.
 - And virtual methods that can be overridden in derived class...
 - Notice that unless specified explicitly, all virtual methods implemented in base class have no effect or only return failure (like `eof`), which requires the derived class (like `basic_ifstream`) to implement meaningful things.

Output stream

- `.overflow(int_type ch = Traits::eof())`: if `ch` is `eof`, then do nothing; otherwise put the `ch` to the proper position.
 - Return `eof` if fail, or any other value if succeed.
 - This will only be invoked when `cur` reaches/goes beyond `end`; you may need to reposition your pointers by `.setp` after flushing.
- `.xspn(const Char* arr, std::streamsize cnt)`: output a string to the proper position.
 - The default implementation for base class is a loop to call `.putc` (and `.overflow` if necessary).
 - If your implementation can output a bunch of things more efficiently, then just implement `.xspn`; for example, if the string is very long, you may only need to output a big part of the string without caching it in the buffer to flush over and over again.
 - Return the number of characters that are written successfully.

In the most general way, you may use `Traits::eq_int_type(ch, Traits::eof())` to judge `eof`; but `ch != Traits::eof()` is enough for most cases.

Stream

- For both input and output buffer, some `protected` positioning can be overridden.
 - `.seekoff(off_type off, seekdir, std::ios_base::openmode mode)`: seek position by relative offset; the buffer should adjust its pointers to proper position; return the absolute position of type `pos_type`, or `pos_type(-1)` if fail.
 - Input/output buffer is specified by `mode`; if it's `std::ios_base::in`, then adjust input buffer; if it's `std::ios_base::out`, then adjust output buffer.
 - We'll talk about stream with both input & output buffer later, then it's also legal to use `in | out` (also default parameter).
 - It's also usually lazy, i.e. only reload for the next read if exceeds buffer.
 - This is exposed by public method `.pubseekoff()`.
 - Notice `.tellg()` also uses this method, i.e. `return .pubseekoff(0, std::ios_base::cur)`.

Stream

- `.seekpos(pos_type pos, openmode mode)`: seek position by absolute offset; the buffer should adjust its pointers to proper position; return the absolute position of type `pos_type`, or `pos_type(-1)` if fail.
 - This is exposed by public method `.pubseekpos()`.
- `.sync()`: write buffer back for output stream and empty the buffer for input stream. Exposed by `.pubsync()`.
 - It depends on implementation to reload immediately for input buffer or not.
- `.setbuf(Char* s, std::streamsize n)`: allow to set the underlying buffer manually; you can use it to set a larger buffer. Return `this`. Exposed by `.pubsetbuf()`.

Output stream

```
template<typename Char = char, typename CharTraits = std::char_traits<Char>>
class NullDevBuf : public std::basic_streambuf<Char, CharTraits>
{
    using Super_ = std::basic_streambuf<Char, CharTraits>;
protected:
    // Normally we need to use setp to adjust pointers, but null device drops
    // everything so that no buffer is needed, thus only return 0.
    Super_::int_type overflow(Super_::int_type ch) override { return 0; } // non-eof val.
    std::streamsize xsputn(const Char* arr, std::streamsize cnt) override { return cnt; }
};

int main()
{
    std::cout << "Here is it.\n";
    NullDevBuf buf;
    std::cout.set_rdbuf(&buf);
    std::cout << "Not printed.\n";
    return 0;
}
```

C:\WINDOWS\system32\cmd.exe

Here is it.
请按任意键继续. . .

We'll show you a more useful but complex example in the homework.

String and stream

- Stream
 - Stream overview
 - Output stream
 - Input stream
 - Stream status
 - Bidirectional stream and stream linking
 - Standard streams

Input stream

Notice: since C++20, you can only `>> char[N]` instead of `>> char*`, since the latter may overflow.

- For input stream, it's inherited from `std::basic_istream`.
 - Formatted input: just use `>>` (sometimes named "extraction operator"), like what you've done in `std::cin`.
 - Unformatted input: not affected by io manipulator.
 - Only for raw character (& its array) input.
 - `.get()`, `.get(ch)` and `.read(chArr, len)`.
 - They're similar to `getc()`, `getchar()`, `fgets()` in C.
 - `.get()` will return `int_type`, while `.get(Char& c)` will save the result into `c`.
 - Why `.get()` doesn't return `Char`?
 - It's possible to have no character anymore (e.g. end of file)!
 - In that case, return `Trait::eof()` and it may be not a valid `Char` value.
 - A typical way in C is return `EOF` (a negative integer value), and that's why `getc()` return `int` instead of `char`.
 - For `.get(Char& ch)`, it just remains the character unchanged for this case.
 - For `.read(chArr, len)`, remaining characters can be less than `len`.

Input stream

- Sometimes, you may want to jump to other positions (as if you move the pointer to some place of the whole buffer).
 - You can use `.tellg()` and `.seekg()` to get or set the “**get**” position.
 - `.tellg()` returns & `.seekg()` accepts `std::basic_ostream<xx>::pos_type`; you can also use relative position to seek, i.e. by `.seekg(off_type off, std::ios_base::seekdir dir)`.
- Beyond these, input stream provides more utility functions.
 - Some more unformatted input functions:
 - `>> streambuf` (only this `>>` is unformatted).
 - `.get(Char* buffer, std::streamsize count, delim = Char('\n'))`: input to the buffer, until `delim` or character numbers are `count`.
 - `.get(streambuf buf, delim = Char('\n'))`: input to the output stream buffer, until `delim`.

Input stream

- `.getline(Char* buffer, std::streamsize count, delim = Char('\n'))`: similar to `.get`, except that it will eat the `delim` while `.get()` will keep it in the stream (so that the next time you can still get the `delim`).
 - If you want to `getline` to a `std::string`, you shouldn't use the member function, but `std::getline(stream, std::string, delim = '\n')`.
- `.readsome(Char* buffer, std::streamsize count)`: extract at most `count` characters from **buffer in memory** (i.e. no need to reload from e.g. file).
 - `std::cin.readsome()` may or may not fill in the buffer.
- `.ignore(count = 1, delim = Traits::eof())`: eat at most `count` characters / until reaching `delim`.
- `.peek()`: get the next character without really changing the status of the stream (i.e. next `.get()` will get the peeked result). Return `eof` if stream reaches its end.
 - Note that `cin.peek()` blocks instead of returning `eof` if no next character is available.

Input stream

- `.unget()`: try to put back the last extracted character; `.peek()` is basically equivalent to `.get() + .unget()`.
- `.putback(ch)`: try to put an arbitrary character back to the stream.
- `.gcount()`: get number of characters got from the last **unformatted** input functions.
 - `.read(chArr, count)` may get characters less than `count`, and you can use `.gcount` to get the actual length!

The following member functions of `basic_istream` change the value of subsequent `gcount()` calls:

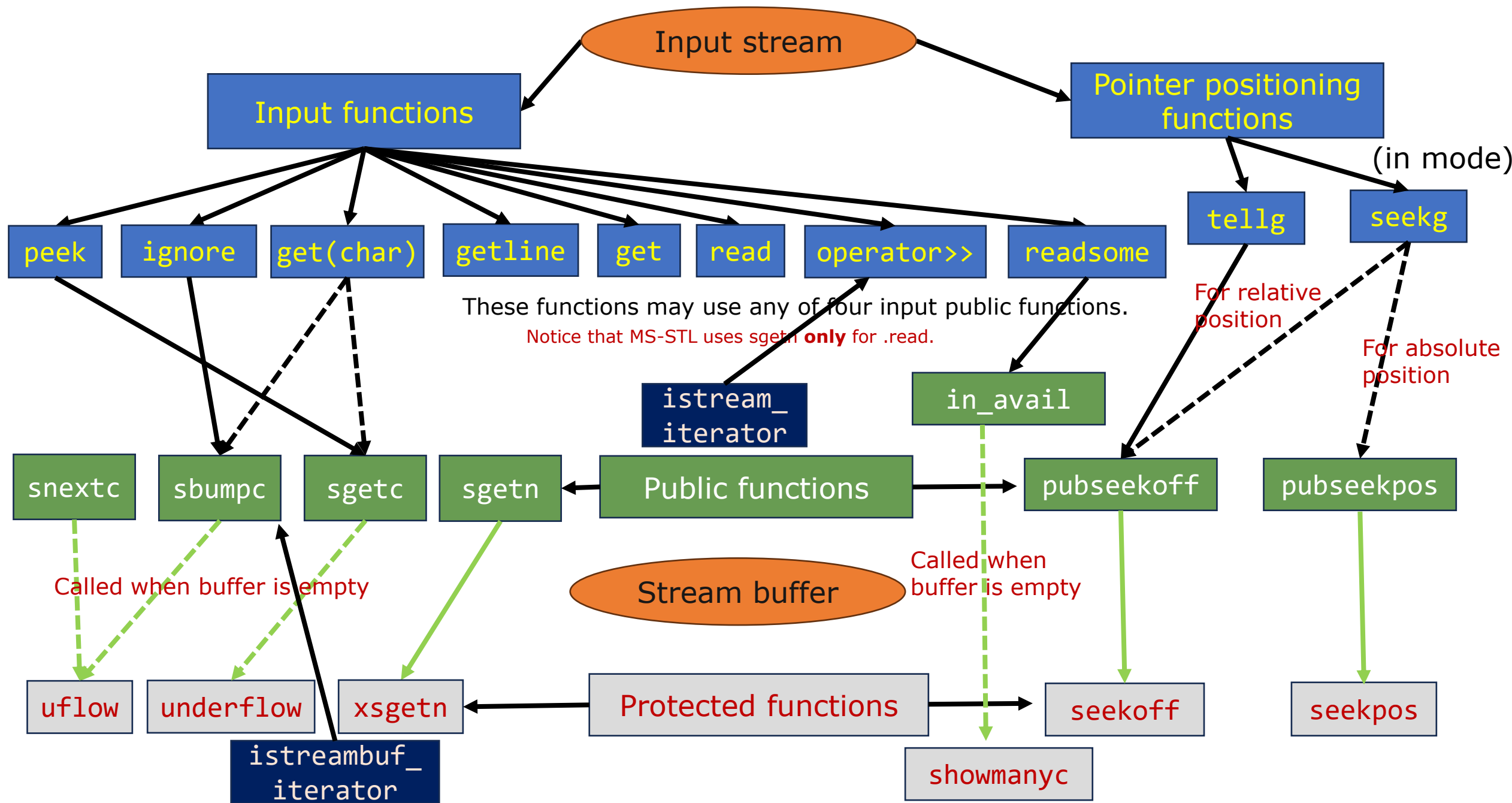
- `move constructor`
- `swap()`
- `get()`
- `getline()`
- `ignore()`
- `read()`
- `readsome()`
- `operator>>(basic_streambuf*)`

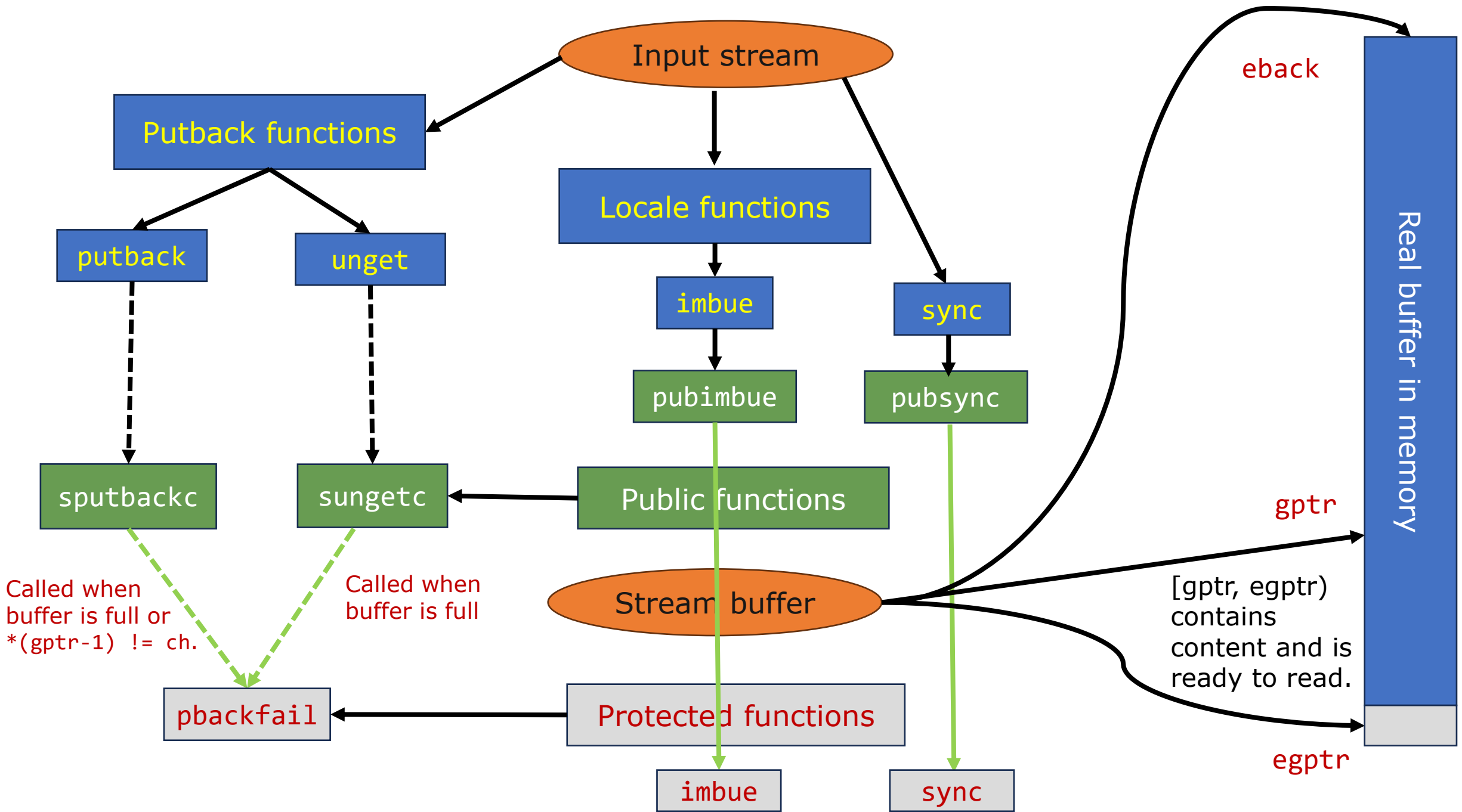
The following functions set `gcount()` to zero:

- `constructor`
- `putback()`
- `unget()`
- `peek()`

Input stream

- And a synchronization function:
 - `.sync()`: synchronize the buffer with the underlying sequence; for example, if the file is written, input stream can use this method to reload.
 - Notice again that it's implementation-defined whether the buffer is reloaded immediately when calling `.sync()` or later in the next read.
 - Return 0 if success, -1 if fail.





Input stream

- List them all:
 - `.eback()/.gptr()/.egptr()`: return `begin/cur/end` pointer.
 - `.setg(gBeg, gCurr, gEnd)`: set the `begin/cur/end` pointer to `gBeg/gCurr/gEnd`.
 - `.gbump(int off)`: `cur += off`; `cur` can go out of `[begin, end)`.
 - And many methods that can be overridden in derived class:
 - `.underflow()`: return the next character and adjust pointers to the proper position. Return `eof` if fail.
 - This will only be invoked when `cur` reaches `end`; you may need to reposition your pointers by `.setg` after reloading new contents.
 - `.uflow()`: `.underflow()` and then advance `cur` by 1.
 - The default implementation is to call `.underflow()` and increments `gptr`.
 - `.xsgetn(Char* arr, std::streamsize cnt)`: input a string to `arr` and return the number of characters that are written successfully.
 - Default: a loop to call `.sbumpc` (and `.underflow` if necessary).

Input stream

- `.pbackfail(int_type ch = Traits::eof())`: this method is called if `eback == gptr` or `gptr == nullptr` or `*(gptr-1) != ch` (i.e. fail to go back); buffer should put the character back (e.g. for file buffer, it may need to reload buffer by going backwards). Return `eof` if fail.
- `.showmanyc()`: abbr. for **stream-how-many-character**; return estimated remaining number of characters of the underlying sequence (or 0 if not sure; default of base class).
 - `.in_avail()`: return `(end - cur)` of type `std::streamsize`, meaning the content that can be got immediately without reloading.
 - `.showmanyc` is used by `.in_avail()` when the buffer is empty (i.e. `gptr == egptr`); by default 0, which means unknown.
- One case that I can think it's useful:
 - If we implement a `TCP_streambuf` ourselves, and we want to know how many characters can be got from the system buffer immediately.
 - Then you can call system API to know it, and returned by `.showmanyc!`

Stream status

- We've found that stream buffer will report `eof` to denote failure; but it seems that stream itself doesn't expose it!
 - So how can we know this failure?
 - Stream uses stream status to denote that!
- There are three bits for that: `eofbit`, `failbit` and `badbit`.
 - `std::ios_base::eofbit` is commonly used to denote end of stream.
 - `std::ios_base::failbit`: commonly used when parsing error, e.g. if you `cin >> SomeFloat` but input a character `'a'`; or failing to open/close file.
 - `std::ios_base::badbit`: some irrecoverable error happens.
 - You can use `.rdstate()` to get the state flags, and test it by e.g. `s & eofbit`, `.clear(states = goodbit)` to set flags, `.setstate(states)` to add flags (equivalent to `.clear(rdstate() | states)`).
 - Particularly, 0 or `std::ios_base::goodbit` means no error.

For details of when these bits are set when stream buffer failure happens, see [here](#) if you need it/are interested.

Stream status

- However, it's usually not convenient to operate on bits, so methods are also provided directly:
 - `.good()/operator bool`: return whether the stream is good.
 - Combined with the fact that all input/output methods return the stream itself, so that you can use `while(std::cin >> xx)!`
 - `operator!` is also overloaded.
 - `.eof()`: return whether eofbit is set.
 - `.fail()`: return whether **failbit** or **badbit** is set.
 - `.bad()`: return whether badbit is set.
- Besides, if you want to throw `std::ios_base::failure` (derived from `std::system_error/runtime_error`) when error states occur, you can use `.exceptions(states)`.
 - `.exceptions()` can get the exception states; default is none.

Stream status

- When the stream is not good, any input/output will do nothing; so don't forget to `.clear()` if you want to continue to use the stream normally!
 - Some operations (i.e. `putback/unget/seekg()`) may reset the eofbit as side effects too.
- Final word: all streams are move-only, move-constructible and swappable.
 - We will cover move semantics in the next lecture, but now it's just enough to know it's not copyable.

String and stream

- Stream
 - Stream overview
 - Output stream
 - Input stream
 - Bidirectional stream and stream linking
 - Standard streams

Bidirectional stream

- Streams can also both input and output by `std::basic_iostream`.
 - It has a single buffer in the `streambuf`, with two sets of `begin/cur/end` pointers.
 - It has all methods from `istream` and `ostream`, e.g. `.tellg/seekg/tellp/seekp()`.
 - Whether they share the same position is determined by the derived class. For example, **file stream share while other streams don't**.

```
void Test(std::iostream& s)
{
    int a;
    s >> a;
    std::cout << s.tellg() << " " << s.tellp() << "\n";
    s << (a + 1);
    std::cout << s.tellg() << " " << s.tellp() << "\n";
    return;
}
```

```
std::stringstream s1{ "123 456" };
std::fstream s2{ "test.txt" };
Test(s1);
Test(s2);
return 0;
```

3	0
3	3
3	3
6	6

Stream linking

- Beyond that, you can also link an `istream` and `ostream`, so that when the `istream` reads something, `ostream` will call `flush()` automatically.
 - By `.tie(basic_ostream*)`, which returns the pointer to the original linked `basic_ostream`.
 - Since both `flush` and `sync` only calls `pubsync()`, it also indirectly synchronizes input buffer.
- Note1: use `.tie()` to get the original link directly, and `.tie(nullptr)` to decouple original link.
- Note2: You can also tie an output stream on another output stream, but this seems to be rarely used.

Stream linking

- The difference between bidirectional stream and stream linking is that the latter are two individual streams, which use two states, two locales, etc..
- The most typical example is `std::cin` and `std::cout`; they are tied automatically (i.e. input will flush the output).
 - They also have a `.sync_with_stdio()`, which controls whether `printf/scanf` can be used when using `std::cin/std::cout`.
 - So in some ACM solutions, you may see `std::cin.sync_with_stdio(false)`, `std::cout.sync_with_stdio(false)`, and `std::cin.tie(nullptr)`.
 - But it's usually not used in real projects, since it's hard to guarantee no external library uses `printf`.
 - And `std::print` in C++23 is fast enough!

String and stream

- Stream
 - Stream overview
 - Output stream
 - Input stream
 - Bidirectional stream and stream linking
 - Standard streams

Standard streams

- There are three pre-defined standard streams with their buffers, inherited from `std::basic_istream/ostream/iostream/streambuf`.
 - File stream: `std::basic_ifstream/ofstream/fstream`, with `std::basic_filebuf` as buffer. They're defined in `<fstream>`.
 - String stream: `std::basic_istringstream/ostringstream/stringstream`, with `std::basic_stringbuf` as buffer. They're defined in `<sstream>`.
 - There is a `<strstream>`, but deprecated in C++98, removed in C++26 and replaced by span stream!
 - Span stream (C++23): `std::basic_ispanstream/ospanspanstream/spanstream`, with `std::basic_spanbuf` as buffer. They're defined in `<spanstream>`.
- And a stream used with other streams:
 - Synchronized **output** stream: C++20, `std::basic_osyncstream`, with `std::basic_syncbuf` as buffer. They're defined in `<syncstream>`.

File stream

```
std::ifstream fin{ "in.txt" };  
float a; int b;  
fin >> a >> b;
```

- As its name, the underlying byte sequence of file stream is file.
 - Thus, to construct a new file stream, you need a path and **open mode**.
 - Path can be a `std::string` and `const char*`; you can also use `std::filesystem::path` (which is a canonical way to represent a path) or its **native** string view. We'll cover them in the future.
 - Open mode is widely used in many languages and platform APIs, e.g. C, Python, Go, etc., with slight difference on name or functionality.
 - We've seen `in / out`, which is in fact the default mode for `ifstream/ofstream`; they specify only read / can write the file; `out` will also create the file if it doesn't exist, and **truncate** (i.e. delete all contents) if the file exists.
 - However, if you use `in | out` (as the default mode for `fstream`), truncation will not happen automatically; you need to add `| trunc` explicitly.
 - Also, if you want to fail rather than create a new file when it doesn't exist, you can use `noreplace` (also called "exclusive mode") since C++23.

File stream

Opens in text (translated) mode. In this mode, carriage return-line feed (CR-LF) combinations are translated into single line feeds (LF) on input, and LF characters are translated into CR-LF combinations on output. Also, CTRL+Z is interpreted as an end-of-file character on input. In files opened for reading or reading and writing, `fopen` checks for CTRL+Z at the end of the file and removes it, if possible. It's removed because using the `fseek` and `ftell` functions to move within a file ending with CTRL+Z may cause `fseek` to behave improperly near the end of the file.

- There are also two specifiers to affect initial position of sequence pointer.
 - `ate`: after opening the file, seek the pointer to the end of file immediately.
 - This is often used with `out`, which makes it **not truncate** and may append some contents at the end first.
 - `app`: seek the pointer to the end of file every time before writing happens.
 - The difference with `ate` is that `ate` will only seek to end once, while `app` will always append at the end (even `.seekg()` doesn't affect it).
- Besides, files can be opened in text mode (default) or binary mode (`| binary`).
 - For example, in Windows, newline will output `"\r\n"` (CRLF) though you only write a `'\n'`.
 - If you open in binary mode, then only `'\n'` will be written.
 - Also, some unprintable characters will cause error in text mode.

File stream

- **ATTENTION**: This is the **only** difference between binary mode and text mode!
 - In binary mode, you can also use `operator<<` (e.g. `fout << 1`). Parameters are formatted to string and the characters are then written!
 - Since numbers are all printable, then it's same as text mode.
 - If you want to write "real binary data", you need to use:

```
int a = 10;  
fout.write((const char*)&a, sizeof(int));
```

- `a` is then a character array, e.g. 0x00, 00, 00, 0a (depend on endianness).
- So what if we use code above in text mode?
 - 0x00 is unprintable, so it may be wrong in some systems.
 - 0x0a is newline, so on Windows it will be converted to `\r\n`!
- And that's why we need binary mode.

See Windows [doc1](#) and [doc2](#).

File stream

- There are many files that should be read as pure byte sequence, like some compressed file, video/image, etc.
- Similarly, you can use `.read()` to read bytes directly.
 - It's theoretically illegal to `.read(reinterpret_cast<char*>(&obj), sizeof(obj))` and then use `obj` because of lifetime concern, but practically it's usually Okay.
 - It's definitely OK to create the byte array with proper alignment, read in the buffer, and then `start_lifetime_as<Object>`.
 - It's definitely Okay to `.write(reinterpret_cast<const char*>(&obj), sizeof(obj))` for objects, since it's legal to **read** as byte sequence.
 - Remember? only "trivially copyable" objects can be safely copied from a bunch of bytes got from object of the same type.

The file is opened as if by calling `std::fopen` with the second argument (file access mode) determined by the result of `mode & ~std::ios_base::ate` as follows, `open()` fails if the result is not some combination of flags shown in the table:

mode & ~std::ios_base::ate						std::fopen access mode	Action if file already exists	Action if file does not exist
binary	in	out	trunc	app	noreplace (since C++23)			
-	+	-	-	-	-	"r"	Read from start	Failure to open
+	+	-	-	-	-	"rb"		
-	+	+	-	-	-	"r+"		Error
+	+	+	-	-	-	"r+b"		
-	-	+	-	-	-	"w"	Destroy contents	Create new
-	-	+	+	-	-			
+	-	+	-	-	-	"wb"		
+	-	+	+	-	-			
-	+	+	+	-	-	"w+"		
+	+	+	+	-	-	"w+b"		
-	-	+	-	-	+	"wx"	Failure to open	Create new
-	-	+	+	-	+			
+	-	+	-	-	+	"wbx"		
+	-	+	+	-	+			
-	+	+	+	-	+	"w+x"		
+	+	+	+	-	+	"w+bx"		
-	-	+	-	+	-	"a"	Write to end	Create new
-	-	-	-	+	-			
+	-	+	-	+	-	"ab"		
+	-	-	-	+	-			
-	+	+	-	+	-	"a+"		
-	+	-	-	+	-			
+	+	+	-	+	-	"a+b"		
+	+	-	-	+	-			

Summary on
open mode.

If the open operation succeeds and `(openmode & std::ios_base::ate) != 0` (the ate bit is set), repositions the file position to the end of file, as if by calling `std::fseek(file, 0, SEEK_END)`, where `file` is the pointer returned by calling `std::fopen`. If the repositioning fails, calls `close()` and returns a null pointer to indicate failure.

File stream

- If the file fails to open, like file doesn't exist with `in` mode, or you don't have permission to open the file in some mode (e.g. read-only to you), then failbit will be set.
 - You can use `.is_open()` to check whether the file stream associates with a file explicitly.
 - You can also use `.close()` to close a file, but dtor will do it automatically.
 - Unless you want to open a new file (by `.open(...)`) on the same stream object, you don't need to call `.close()` explicitly.
 - Since C++26, you can also call `.native_handle()` to get the native handle of the file stream.

Codecvt

- Finally we need to mention codecvt in stream...
 - The conversion happens when goes to external device, i.e. the encoding of the memory buffer haven't changed at all.
 - Codecvt in file stream is always `std::codecvt<CharT, char>`, i.e. convert to `char` when writing.
 - Notice that MS-STL will cancel `codecvt` when `.pubsetbuf(CharT*, size)`, but `libc++/libstdc++` won't.
 - The default locale is same as `std::locale::global()`, which is C locale by default (i.e. wide characters -> `char` always fails).
 - Binary mode for file stream will **still go through codecvt**.

```
const wchar_t str[] = L"爱情可以慷慨又自私";
std::wofstream fout{ "test-locale.txt" };
fout.imbue(std::locale("chs"));
fout << str;
```

Codecvt

- For example, if you want to use `std::wfstream` to output `L"你好"` in its native encoding (e.g. UTF-16 on Windows).
 - It will by default fail since C locale cannot do so.
 - It still fails when switching to binary mode, since binary mode still go through codecvt.
 - [It works on MS-STL](#) if `pubsetbuf()` + binary mode, which disables codecvt and can write "unprintable characters" (for `char`) normally.
 - But not work on `libstdc++/libc++`.
 - And don't forget to write BOM!
 - To make it cross-platform, we need to write our own codecvt + binary mode to make it success.
 - We'll leave it in our homework; check it!

String stream

- The underlying sequence of the string stream is just memory.
 - You can use any string type to construct it, with an optional open mode (yes, you may just treat it as file in memory with no flush/sync).
 - The string stream will reallocate when overflow.
 - You can use `.str()` to **copy** it out, or replace with a new string by `.str(newStr)`.
 - Since C++20, you can also `.view()` to get the `string_view` to it.
 - But notice that future reallocation may make the view invalid!
 - For example:

Remember? `tellx()` returns `std::streampos`, which can be **seen** as an unsigned integer but in fact a class. It should subtract another stream size to get `std::streamoff`, which is a real signed integer.

```
std::string s0{ "1.234 567 " };
std::stringstream s{ s0 };
float a; int b;
s >> a >> b;
std::cout << std::format("a = {}, b = {}\n", a, b);
s << b;
std::cout << std::format("get pos = {}, put pos = {}\n",
    s.tellg() - std::streampos{ 0 }, s.tellp() - std::streampos{ 0 });
std::cout << std::format("original string = {},\nunderlying string = {}", s0, s.str());
```

Span stream

- String stream will always create a new underlying string; but what if we want to provide the buffer ourselves?
 - Changes are applied on the buffer directly, without worrying about whether the output exceeds the buffer (just truncates it rather than reallocate).
 - `badbit` will be set in this case.
 - But then you need to ensure that the lifetime of buffer \geq scope for spanstream to operate on it.
 - Ctor: use a `span<CharT>` + mode.
 - `rdbuf()->pubsetbuf(CharT*, len)` (or move ctor/assignment) can reset it.
 - Get view: `.span()`, get `span<CharT>` (instead of `basic_string_view<CharT>!`)
 - Notice: if the mode contains `out`, then it returns `[pbase, pptr)`, i.e. the part that's already written.
 - Otherwise the whole buffer.

String stream and span stream don't have "external device";
`flush & sync` do nothing and `codecvt` will never work.

Span stream

```
1  #include <spanstream>
2  #include <array>
3  #include <print>
4
5  int main()
6  {
7      std::array<char, 10> buffer;
8      std::spanstream s{ buffer }; // span可以直接从range构造
9
10     s << 1 << " test";
11
12     auto currSpan = s.span();
13     std::println("{} {} {}", currSpan.size(),
14                 std::string_view{currSpan.data(), currSpan.size()},
15                 std::string_view{ buffer.data(), currSpan.size() });
16
17     int a = 0;
18     s >> a;
19     std::println("{} get pos: {}, put pos: {}", a,
20                 s.tellg() - std::streampos{}, s.tellp() - std::streampos{});
21
22     s << " 9999999999999999";
23     auto newSpan = s.span();
24     std::println("{} {} {}", newSpan.size(),
25                 std::string_view{ newSpan.data(), newSpan.size() },
26                 std::string_view{ buffer.data(), newSpan.size() });
27
28     std::println("Bad or not: {}", s.bad());
29
30     return 0;
31 }
```

Microsoft Visual Studio 调试

```
6 1 test 1 test
1, get pos: 1, put pos: 6
10 1 test 999 1 test 999
Bad or not: true

D:\Work\CS\C++\Test\Proje
按任意键关闭此窗口...
```

Synchronized stream

- If you've coded some multi-threading program that uses `std::cout`, you may find output is messed up...
 - E.g. `std::cout << "a=" << a << "c=" << c`; in two threads may have any permutation (e.g. `"a=a=1c=c=22"`).
 - Before, you may have to create `std::ostringstream`, and `std::cout << .str()` so that it will output the string as a whole.
 - `std::format` will help to eliminate that problem too.
 - However, this only holds water for standard output because it has an inner global lock.
 - Other streams, like file stream you create, will still cause data race because two threads may operate on file pointer simultaneously. You need to add a lock manually!
 - `std::osyncstream` is to solve that!

Synchronized stream

- You need to create two `std::osyncstream` and each thread has its own.
 - You may think `osyncstream` prepares an additional buffer; when you use `std::flush_emit` explicitly, the buffer is output to the attached stream (`fout` here) without data race.
 - This is done by sharing mutex on streams with the same buffer address; only OS APIs can do so currently.
 - If you want / don't want to emit every `std::flush`, you can use `<< std::emit_on_flush / std::noemit_on_flush`.
 - The stream will be emitted too when destructed.
 - You can also use `.get_wrapped()` to get the buffer of the attached stream, too.

```
void Work(std::osyncstream& s) {  
    s << "When I go with the force, "  
      << "the force is with me." << std::flush_emit;  
}  
  
int main()  
{  
    std::ofstream fout{ "out.txt" };  
    std::osyncstream s1{ fout }, s2{ fout };  
    Work(s1); // we assume they're dispatched  
    Work(s2); // to two different threads.  
  
    return 0;  
}
```

String and stream

- Stream
 - Stream overview
 - Output stream
 - Input stream
 - Bidirectional stream and stream linking
 - Standard streams
- IO manipulator*

IO manipulator*

- As we've said, you can now use `std::format` for formatting, so most of IO manipulators are not that useful (especially for output) and we'll only give a brief view.
 - The vexing fact of IO format manipulators is that many of them will keep the manipulation state so that any following output will be affected.
 - For example, `std::cout << std::hex << 16; std::cout << 32;` will also make `32` hexadecimal.
 - If you want to restore the status, you either remember what status you've changed and adjust it back (e.g. `std::cout << std::dec << 32`), or `auto flags = stream.flags()` before status changing, and `stream.flags(flags)` at the end to restore all settings.
- First, they're defined in `<ios>` & `<iomanip>`.

IO manipulator*

Defined in header `<ios>`

boolalpha noboolalpha	switches between textual and numeric representation of booleans (function)
showbase noshowbase	controls whether prefix is used to indicate numeric base (function)
showpoint noshownpoint	controls whether decimal point is always included in floating-point representation (function)
showpos noshownpos	controls whether the + sign used with non-negative numbers (function)
skipws noskipws	controls whether leading whitespace is skipped on input (function)
uppercase nouppercase	controls whether uppercase characters are used with some output formats (function)
unitbuf nounitbuf	controls whether output is flushed after each operation (function)
internal left right	sets the placement of fill characters (function)
dec hex oct	changes the base used for integer I/O (function)
fixed scientific hexfloat (C++11) defaultfloat (C++11)	changes formatting used for floating-point I/O (function)

adjustfield left | right | internal

basefield dec | oct | hex

floatfield scientific | fixed
and other two.

IO manipulator*

- For those three that have multiple options, their default values are `left/dec/fixed`.
 - You can also use e.g. `<< std::resetiosflags(std::ios::basefield/adjustfield/floatfield)` to restore the default one.
 - There is also a `std::setiosflags; << std::setiosflags(std::ios::fixed)` is basically same as `<< std::fixed`.
- For fill-align-width-type, you can use:

<code>setbase</code>	changes the base used for integer I/O (function)
<code>setfill</code>	changes the fill character (function template)
<code>setprecision</code>	changes floating-point precision (function)
<code>setw</code>	changes the width of the next input/output field (function)

`setbase` only accepts 8/10/16.

`setw` has no after effects!

IO manipulator*

- And some locale-related manipulators:

<code>get_money</code> (C++11)	parses a monetary value (function template)
<code>put_money</code> (C++11)	formats and outputs a monetary value (function template)
<code>get_time</code> (C++11)	parses a date/time value of specified format (function template)
<code>put_time</code> (C++11)	formats and outputs a date/time value according to the specified format (function template)

- And a final one `std::quoted(str, delim='"', escape='\\')`.
 - It will add `delim` on both sides, and add `escape` for characters in the `str` that equals to `delim`.
 - This can be basically substituted by `?` type in C++23.

Default delimiter case:

```
read in      [std::quoted() quotes this string and embedded "quotes" too]
stored as    ["std::quoted() quotes this string and embedded \"quotes\" too"]
written out  [std::quoted() quotes this string and embedded "quotes" too]
```

```
void default_delimiter()
{
    const std::string in = "std::quoted() quotes this string and embedded \"quotes\" too";
    std::stringstream ss;
    ss << std::quoted(in);
    std::string out;
    ss >> std::quoted(out);

    std::cout << "Default delimiter case:\n"
               << "read in      [" << in << "]\n"
               << "stored as    [" << ss.str() << "]\n"
               << "written out [" << out << "]\n\n";
}
```

IO manipulator*

- And finally non-formatting manipulators:

Defined in header `<istream>`

<code>ws</code>	consumes whitespace (function template)
-----------------	--

Defined in header `<ostream>`

<code>ends</code>	outputs <code>'\0'</code> (function template)
-------------------	--

<code>flush</code>	flushes the output stream (function template)
--------------------	--

<code>endl</code>	outputs <code>'\n'</code> and flushes the output stream (function template)
-------------------	--

<code>emit_on_flush</code> <code>noemit_on_flush</code> (C++20)	controls whether a stream's <code>basic_syncbuf</code> emits on flush (function template)
--	--

<code>flush_emit</code> (C++20)	flushes a stream and emits the content if it is using a <code>basic_syncbuf</code> (function template)
---------------------------------	---