香港電腦奧林匹克競賽
Hong Kong Olympiad in Informatics

Advanced C++ STL

David Wai {wjx}
2024-03-02

C++ STL

C++ **S**tandard **T**emplate **L**ibrary

A **part** of C++ Standard Library

Contains four components:

- Algorithms
- Containers
- Functions
- Iterators

Why use C++ STL

- STL contains many algorithms and data structures that are useful in competitive programming
- Can write shorter code
  - Less implementation time (especially if your typing speed is slow)
- No need to care about the implementation detail of the algorithm / data structure
  - Less debugging time
  - Have more time to focus on other parts of a problem
- Learn about how to write your code in a more standard way
  - You don't need to know the implementation detail in most of the time
  - But if you want to learn more, you can still look at it and learn some coding conventions from it

C++ Standard Library

Functions and classes in C++ Standard Library are declared within the `std` namespace and they are defined in different headers

- If you are using a GCC C++ compiler, you can use the following line to include most C++ Standard Library headers:
  - `#include <bits/stdc++.h>`
- To avoid typing `std::` prefix for each standard function / class, you can use the following line to move everything in `std` namespace to your program:
  - `using namespace std;`

Other than C++ STL, We will also talk about some features in C++ Standard Library that are useful in competitive programming in this session

C++ Standard

Starting from 2011, 3 years a standard (C++11, C++14, ...)

- C++20 has been published, and work is now underway on C++23
- In most cases, new standards are backward compatible
  - There are some special cases like `gets` and `random_shuffle`
- g++ compilation flag: `-std=c++11`, `-std=c++14`, ...

Different contests and online judges may support different standards

- HKOI Online Judge, codeforces supports C++20
- IOI, APIO supports C++17
- NOI supports C++14
- Be careful about the supported version as some functions may be in new standard, check the compilation flags if you are not sure about it

Template

There are many types of templates in C++

- Class template
- Function template
- Alias template (since C++11)
- Variable template (since C++14)

Compiler will generate a copy for each used type

Template

```cpp
template <typename T> // class template
struct Point { T x, y; };
template <typename T> // function template
T sum(T a, T b) { return a + b; }
template <typename T> // alias template
using P = Point<T>;
template <typename T> // variable template
T pi = T(acos(-1));
int main() {
    Point<int> a = {1, 2};
    P<double> b = {0.1, 0.2};
    cout << sum(a.x, a.y) << ' ' << sum(b.x, b.y) << '\n';
    cout << pi<int> << ' ' << pi<double> << '\n';
    return 0;
}
```

Output

```
3 0.3
3 3.14159
```

Template

Usually, you need to specify the the template parameters

- But sometimes you don't need to specify them for functions and classes if you provide template arguments
  - `int x = max<int>(1, 2)` works
  - `int x = max(1, 2)` also works
  - `vector<int> v(10, 0)` works
  - `vector v(10, 0)` also works (since C++17)

A lot of C++ Standard Library functions and classes are templates

In most cases, you don't have to write your own templates, you just need to know how to use templates

Auto (since C++11)

To get rid of long type names, you may use `auto` to declare variables

An initial value must be assigned when declaring a variable, and the value is used to deduce the type of the variable

Examples:

- `auto a = 5678ll` (deduced type: `long long`)
- `auto b = 3.0` (deduced type: `double`)
- `auto c = 'a'` (deduced type: `char`)
- `auto f = true` (deduced type: `bool`)
- `auto s = "123"` (deduced type: `const char*`)
- `auto s = "123"s` (deduced type: `string`)

Pair

Defined in header `<utility>`

A struct template to store two objects

To declare a pair storing two int types:

- `pair<int, int> p`
- `pair p(0, 0)` (since C++17)

To access the first / second element:

- `p.first` / `p.second`
- `get<0>(p)` / `get<1>(p)` (since C++11)

Tuple (since C++11)

Defined in header `<tuple>`

A generalization of pair (2 -> *N*, *N* should be a fixed number in compile time)

To declare a tuple storing two int types and one double type:

- `tuple<int, int, double> p`
- `tuple p(0, 0, 0.0)` (since C++17)

To access the N^{th} element:

- `get<N>(p)` (*N* should be a fixed number in compile time)

Pair and tuple

```
pair<int, int> p = {1, 2};
pair p2(3, 4);
tuple<int, int, double> t;
tuple t2(1, 2, 3);
cout << p.first << ' ' << get<1>(p) << '\n';
cout << get<0>(t2) << ' ' << get<1>(t2) << ' ' << get<2>(t2) <<
'\n';
cout << (p > p2) << ' ' << (t < t2) << '\n';
```

Output

```
1 2
1 2 3
0 1
```

Comparison (<, <=, ==, ...) works with lexicographical order if all types are comparable

Accessing members of pair and tuple

What if you don't want to write `p.first` / `p.second` / `get<N>(p)` every time?

Two solutions:

- Tie (since C++11)
- Structured binding declaration (since C++17)

Tie

```
pair p(1, 2);
int x, y;
tie(x, y) = p;
cout << x << ' ' << y << '\n';
x = 2;
cout << x << ' ' << y << '\n';
cout << p.first << ' ' << p.second << '\n';
p = tie(x, y);
cout << p.first << ' ' << p.second << '\n';
```

Output

```
1 2
2 2
1 2
2 2
```

Changing the values of the tied variables won't change the member values in pair, so you need to tie them again and assign it to the pair

Structured binding declaration

Binds the specified names to members of a type

Supports 3 types: arrays, tuple-like types, self-defined structures

To get copies (pass by value):

- `auto [x, y] = p`

To get references (pass by reference):

- `auto& [x, y, z] = t`

Structured binding declaration

```cpp
array<int, 2> a = {1, 2};
auto [b, c] = a;
b = 2;
cout << b << ' ' << c << '\n';
cout << a[0] << ' ' << a[1] << '\n';
tuple t(1, 2, 3);
auto& [x, y, z] = t;
x = 5;
cout << x << ' ' << y << ' ' << z << '\n';
cout << get<0>(t) << ' ' << get<1>(t) << ' ' << get<2>(t) << '\n';
```

Output

```
2 2
1 2
5 2 3
5 2 3
```

Vector

Defined in header `<vector>`

A dynamic sized array (with *O(1)* random access)

Similar to pair, comparison (<, <=, ==, ...) works with lexicographical order

To declare an empty int vector:

- `vector<int> a`

To declare a long long vector of size 100:

- `vector<long long> a(100)`
- `vector a(100, 0ll)` (since C++17)

Vector

To declare a 2D int vector of size *n* * *m* with all elements initialized to -1:

- `vector<vector<int>> a(n, vector<int>(m, -1))`
- `vector a(n, vector(m, -1))` (since C++17)

To add an element *x* to the end of the vector:

- `a.push_back(x)`
- `a.emplace_back(x)` (since C++11)

To remove the last element of the vector:

- `a.pop_back()`

Vector

To get the size of the vector

- `a.size()` (be careful that the type of the return value is size_t, which is an unsigned integer type)

To access the first element of the vector:

- `a.front()`

To access the last element of the vector:

- `a.back()`

To access the i^{th} element (0 based) of the vector:

- `a[i]`
- `a.at(i)` (slower, but have bound checking)

Iterator

A generalization of pointer

Similar to pointers, can use `*it` to dereference

The iterator in vector is a random access iterator

- Type of the iterator of `vector<int>`: `vector<int>::iterator`

To find the distance of two iterators:

- `distance(it1, it2)`
  - *O(n)* in general (*n* is the distance between `it1` and `it2`)
  - *O(1)* if they are random access iterators (vector iterator supports)

Iterator

To increment `it` by *n* elements:

- `advance(it, n)`
  - *O(n)* in general
  - *O(1)* if it is a random access iterator

To get the next iterator (since C++11):

- `next(it)`

To get the previous iterator (since C++11):

- `prev(it)`

Vector iterators

To get the iterator pointing to the first element of the vector:

- `a.begin()`

To get the iterator pointing to the element **after** the last element of the vector:

- `a.end()`

There are also reverse iterators, which can be used to iterate through the vector in reverse order

- `a.rbegin()`: the iterator pointing to the **last** element of the vector
- `a.rend()`: the iterator pointing to the element **before** the **first** element of the vector

Vector

To iterate through the whole vector:

- Use an index:
  - `for (int i = 0; i < a.size(); i++)`
  - Access the value using `a[i]`
- Use a iterator:
  - `for (auto it = a.begin(); it != a.end(); it++)`
  - Access the value using `*it`
- Use range-based for loop (since C++11):
  - `for (auto v : a)` (pass by value)
  - `for (auto& v : a)` (pass by reference)
  - Access the value using v

Vector

```cpp
vector<int> a;
for (int i = 0; i < 10; i++) a.emplace_back(10 - i);
for (auto x : a) cout << x << ' ';
cout << '\n';
cout << a.back() << '\n';
a.pop_back();
cout << a.size() << '\n';
for (int i = 0; i < a.size(); i++) cout << a[i] << ' ';
cout << '\n';
for (auto& x : a) x = 5;
for (auto it = a.begin(); it != a.end(); it++) cout << *it << ' ';
cout << '\n';
```

Output

```
10 9 8 7 6 5 4 3 2 1
1
9
10 9 8 7 6 5 4 3 2
5 5 5 5 5 5 5 5 5
```

Vector efficiency

The size of a vector is dynamic, how to ensure it does not take too much time compared to an array?

The main idea is reallocation

When the vector is full, a larger memory space will be allocated (usually with double size)

Everything in the old space will then be moved to the new space

Vector efficiency

```
vector<int> a;
for (int i = 0; i < 10; i++) {
   a.emplace_back(i);
   cout << "size: " << a.size() << " capacity: " << a.capacity() << '\n';
}
```

The allocated size of a vector can be queried by using `a.capacity()`

Output

```
size: 1 capacity: 1
size: 2 capacity: 2
size: 3 capacity: 4
size: 4 capacity: 4
size: 5 capacity: 8
size: 6 capacity: 8
size: 7 capacity: 8
size: 8 capacity: 8
size: 9 capacity: 16
size: 10 capacity: 16
```

Vector efficiency

Assume that you call `push_back()` n times.

The total cost comprises of:

- Cost of adding an element
  - 1 operation per `push_back`
  - Total n operations for n `push_back`
- Cost of moving elements when vector is full
  - $1, 2, 4, 8, ..., 2^k$ (where $2^k < n$)
  - The sum of above = $2^{k+1} - 1 < 2n$

Total cost for n `push_back` = $n + (<2n) < 3n$, and therefore is $O(n)$

We can say that `push_back` is amortized $O(1)$

Vector efficiency

If you know the expected maximum size of the vector

- use `reserve()` to preallocate the memory space so reallocate is not needed

If you just want to use it like a fix-sized array

- Use constructor when declaring the vector
- Use `resize()` to set the size of the vector

Push_back and emplace_back

`emplace_back()` is faster especially for a large struct

- `push_back()` copies the object to the vector
- `emplace_back()` constructs the object inside the vector

Their implementation are slightly different

For example, pushing `pair(1, 2)` to a vector:

- `a.push_back(make_pair(1, 2))`
- `a.push_back({1, 2})` (since C++11)
- `a.push_back(pair(1, 2))` (since C++17)
- `a.emplace_back(1, 2)`

Sort

Defined in header `<algorithm>`

Sorts the elements in the range [*first*, *last*) in non-descending order, where *first* and *last* are **random access iterators**

To sort an array of size *n*:

- `sort(a, a + n)`

To sort the whole vector:

- `sort(a.begin(), a.end())`

Time Complexity: *O(n log n)* in **worst case**

Sort

To sort a int vector in reverse order:

- `sort(a.begin(), a.end(), greater<int>())`
- `sort(a.begin(), a.end(), greater<>())` (since C++14)
- `sort(a.begin(), a.end(), greater{})` (since C++14)
- `sort(a.rbegin(), a.rend())`
- `sort(a.begin(), a.end())` followed by `reverse(a.begin(), a.end())`

To sort a int vector by a self-defined comparison function `cmp`:

- `sort(a.begin(), a.end(), cmp)`

Reverse

Defined in header `<algorithm>`

Reverses the order of the elements in the range [*first*, *last*)

Reverse

```cpp
vector a = {1, 4, 2, 3, 5};
reverse(a.begin(), a.end());
for (auto x : a) cout << x << ' ';
cout << '\n';
```

Output

```
5 3 2 4 1
```

Comparison function

The comparison function should take two parameters *a* and *b* and return a boolean value indicating whether $a < b$

How to write it?

Two main ways:

- A normal function
- A lambda expression (since C++11)

Comparison function - normal function

```cpp
bool cmp(int a, int b) { return a > b; }

vector<int> a = {1, 4, 2, 3, 5};
sort(a.begin(), a.end(), cmp);
for (auto x : a) cout << x << ' ';
cout << '\n';
```

Output

```
5 4 3 2 1
```

Lambda expression

An unnamed function object

Can be declared inside a function, or even inside a lambda expression (nested lambdas)

Other than behave like a normal function (takes parameters), it can also capture variables

Syntax: `[ captures ] ( parameters ) { body }`

Capture list (see cppreference for more details):

- =: pass by value
- &: pass by reference

Lambda expression

```cpp
auto sum = [](int a, int b) { return a + b; };
cout << sum(1, 2) << '\n';
auto generic_sum = [](auto a, auto b) { return a + b; };
cout << generic_sum(1.2, 3.4) << '\n';
```

Output

```
3
4.6
```

We can store it to a variable and use it like a normal function
We can also define generic lambdas (since C++14), which work like templates
Be careful when you want to capture a large array by value, it will copy the whole contents of the array, which takes a lot of time
In most cases, you can just write something like `auto f = [&](...) { ... };` to capture all variables by reference

Recursive lambda

Lambdas are unnamed function objects, so there is no name for us to call itself recursively

Is it possible to recursively call it?

Two workarounds:

- Declare a variable by using function first
- Pass itself as a parameter

Recursive lambda - use function

```
function<int(int)> fib;
fib = [&](int n) {
   if (n <= 1) return n;
   return fib(n - 1) + fib(n - 2);
};
cout << fib(15) << '\n';
```

Output

```
610
```

Recursive lambda - pass itself

```
auto fib = [](auto&& self, int n) {
    if (n <= 1) return n;
    return self(self, n - 1) + self(self, n - 2);
};
cout << fib(fib, 15) << '\n';
```

Output

```
610
```

Comparison function - lambda expression

```cpp
vector<int> a = {1, 4, 2, 3, 5};
sort(a.begin(), a.end(), [](int a, int b) { return a > b; });
for (auto x : a) cout << x << ' ';
cout << '\n';
```

Output

```
5 4 3 2 1
```

Application - tree traversal

Please read the problem 01038 - Preorder Tree Traversal

Application - tree traversal

```cpp
vector<vector<int>> edge; // adjacency list to store the connected
nodes for each node
vector<int> ans;

void dfs(int u, int par) {
   ans.emplace_back(u);
   for (auto v : edge[u])
       if (v != par) dfs(v, u);
}
```

Output

Binary search functions

Defined in header `<algorithm>`

Uses binary search algorithm

`lower_bound()`: Returns an iterator pointing to the first element in the range [*first*, *last*) that is **not less** than *value*, or *last* if no such element is found

`upper_bound()`: Returns an iterator pointing to the first element in the range [*first*, *last*) that is **greater** than *value*, or *last* if no such element is found

`binary_search()`: Checks if an element is equivalent to *value* appears within the range [*first*, *last*)

Binary search functions

The range must be sorted in ascending order

- If you want to perform binary search in a range sorted in descending order, you need to write the comparison function, or you can reverse the range first

To achieve *O(log n)* time complexity, *first* and *last* should be **random access iterators**

Binary search functions

```
vector a = {1, 2, 2, 4, 5};
cout << lower_bound(a.begin(), a.end(), 2) - a.begin() << '\n';
cout << lower_bound(a.begin(), a.end(), 3) - a.begin() << '\n';
cout << *lower_bound(a.begin(), a.end(), 2) << '\n';
cout << *lower_bound(a.begin(), a.end(), 3) << '\n';
cout << upper_bound(a.begin(), a.end(), 2) - a.begin() << '\n';
cout << upper_bound(a.begin(), a.end(), 3) - a.begin() << '\n';
cout << *upper_bound(a.begin(), a.end(), 2) << '\n';
cout << *upper_bound(a.begin(), a.end(), 3) << '\n';
cout << binary_search(a.begin(), a.end(), 2) << '\n';
cout << binary_search(a.begin(), a.end(), 3) << '\n';
```

Output

```
1
3
2
4
3
3
4
4
1
0
```

Unique

Defined in header `<algorithm>`

Eliminates all except the first element from every consecutive group of equivalent elements from the range [*first*, *last*) and returns a past-the-end iterator for the new logical end of the range

Usually followed by calling `erase()` which clears the unused range at the end

Unique

```cpp
vector a = {1, 2, 2, 4, 2};
auto it = unique(a.begin(), a.end());
cout << it - a.begin() << '\n';
cout << a.size() << '\n';
for (auto x : a) cout << x << ' ';
cout << '\n';
a.erase(it, a.end());
cout << a.size() << '\n';
for (auto x : a) cout << x << ' ';
cout << '\n';
```

Output

```
4
5
1 2 4 2 2
4
1 2 4 2
```

Discretization

By using the above algorithms, we can do discretization very easily:

- Make a copy of the original array
- Sort it, then all elements of the same values will be in continuous ranges
- Call `unique()` to get an array containing distinct elements
- Use `lower_bound()` to find the rank of each element in the original array (which is the index of the new array + 1)

Time complexity: $O(n \log n)$

Discretization

```
vector a = {748934, 23, 3232, 1, 328490, 2342, 123, 1, 2342, 123,
3232, 1};
vector b = a;
sort(b.begin(), b.end());
b.erase(unique(b.begin(), b.end()), b.end());
for (auto& x : a) x = lower_bound(b.begin(), b.end(), x) - b.begin()
+ 1;
for (auto x : a) cout << x << ' ';
cout << '\n';
```

Output

```
7 2 5 1 6 4 3 1 4 3 5
1
```

Application - discretization

Please read the problem S152 - Apple Garden

String

Defined in header `<string>`

C++ strings are very easy to use

Similar to vector

- You can also use ranged-based for loop to iterate over a string
- You can get iterators from string to perform operations

You can concatenate strings together using the + operator

You can compare strings directly using comparison operators

- You can also use `compare()`, which returns 0 when the strings are equal, negative number when the left string is smaller, and positive otherwise

Iterate over a string

```
string s = "abcdef";
for (int i = 0; i < s.length(); i++) cout << s[i] << '\n';
for (char& c : s) c -= 32;
cout << s << '\n';
```

Output

```
a
b
c
d
e
f
ABCDEF
```

Using string::iterator

```
string s = "abcdef";
for (auto it = s.begin(); it != s.end(); ++it) cout << *it << '\n';
reverse(s.begin(), s.end());
cout << s << '\n';
```

Output

```
a
b
c
d
e
f
fedcba
```

Concatenate strings

```
string s = "ab";
string t = "d";
s += 'c';  // append a character
t += "ef"; // append a string
cout << s.length() << '\n';
cout << s + t << '\n';
```

Output

```
3
abcdef
```

Compare strings

```
cout << ("abc"s == "abc"s) << '\n';
cout << ("abc"s < "def"s) << '\n';
cout << ("abcd"s > "abc"s) << '\n';
cout << "abc"s.compare("abx") << '\n';
cout << "xyz"s.compare("xyz") << '\n';
cout << "def"s.compare("a") << '\n';
```

Output

```
1
1
1
-21
0
3
```

Read one line

```
string s;
getline(cin, s);
cout << s << '\n';
cin >> s;
cout << s << '\n';
```

Output

Find string

```cpp
string s = "This is a string";
int n = s.find("is");
cout << n << '\n';
n = s.find("is", 5);
cout << n << '\n';
n = s.find('q');
cout << n << '\n';
cout << string::npos << '\n';
```

Output

```
2
5
-1
18446744073709551615
```

Modify string

```cpp
string s = "abc";
s.insert(1, "abc", 2);
cout << s << '\n';
s.erase(3);
cout << s << '\n';
s.replace(1, 1, "123");
cout << s << '\n';
```

Output

```
aabbc
aab
a123b
```

Get substring

```
string s = "https://judge.hkoi.org";
cout << s.substr(14, 4) << '\n';
cout << s.substr(8) << '\n';
```

Output

```
hkoi
judge.hkoi.org
```

String conversion (since C++11)

```
string s = "123";
int a = stoi(s);
cout << to_string(a + 1) << '\n';
s += ".456";
double b = stod(s);
cout << to_string(b + 0.123) << '\n';
```

Output

```
124
123.579000
```

Application - string processing

Please read the problem 01000 - Append Insert Replace

Deque

Defined in header `<deque>`
A double-ended queue with random access

To push an element to the front: `q.push_front(x)` / `q.emplace_front(x)`
To push an element to the end: `q.push_back(x)` / `q.emplace_back(x)`
To pop an element at the front: `q.pop_front()`
To pop an element at the end: `q.pop_back()`
To access the first element: `q.front()`
To access the last element: `q.back()`
To access the i^{th} element (0 based): `q[i]` or `q.at(i)`

Deque

```
deque q{4, 5, 6};
q.emplace_front(3);
q.emplace_back(7);
cout << q[2] << '\n';
q.pop_back();
cout << q.back() << '\n';
q.pop_front();
cout << q.front() << '\n';
for (auto x : q) cout << x << ' ';
cout << '\n';
```

Output

```
5
6
4
4 5 6
```

Deque

It seems that deque is stronger than vector, why we use vector?

The answer is the memory usage:

- deque: $n + c_1$
- vector: $2n + c_2$
- $c_1 > c_2$

The internal storage of deque are not contiguous

Be careful of the memory used by many deques of small size

Deque

```
n=1     Maximum resident set size (kbytes): 3912
n=2     Maximum resident set size (kbytes): 3944
n=3     Maximum resident set size (kbytes): 4040
n=4     Maximum resident set size (kbytes): 3948
n=5     Maximum resident set size (kbytes): 4136
n=123   Maximum resident set size (kbytes): 8824
n=124   Maximum resident set size (kbytes): 8760
n=125   Maximum resident set size (kbytes): 8776
n=126   Maximum resident set size (kbytes): 8744
n=127   Maximum resident set size (kbytes): 8888
n=128   Maximum resident set size (kbytes): 8792
n=129   Maximum resident set size (kbytes): 13888
n=130   Maximum resident set size (kbytes): 13884
n=131   Maximum resident set size (kbytes): 13696
n=132   Maximum resident set size (kbytes): 13844
n=133   Maximum resident set size (kbytes): 13752
n=255   Maximum resident set size (kbytes): 13888
n=256   Maximum resident set size (kbytes): 13772
n=257   Maximum resident set size (kbytes): 23560
```

```
n=1     Maximum resident set size (kbytes): 10088
n=2     Maximum resident set size (kbytes): 10232
n=3     Maximum resident set size (kbytes): 10112
n=4     Maximum resident set size (kbytes): 10092
n=5     Maximum resident set size (kbytes): 10216
n=123   Maximum resident set size (kbytes): 10212
n=124   Maximum resident set size (kbytes): 10080
n=125   Maximum resident set size (kbytes): 10092
n=126   Maximum resident set size (kbytes): 10128
n=127   Maximum resident set size (kbytes): 10120
n=128   Maximum resident set size (kbytes): 15256
n=129   Maximum resident set size (kbytes): 15236
n=130   Maximum resident set size (kbytes): 15388
n=131   Maximum resident set size (kbytes): 15180
n=132   Maximum resident set size (kbytes): 15236
n=133   Maximum resident set size (kbytes): 15228
n=255   Maximum resident set size (kbytes): 15332
n=256   Maximum resident set size (kbytes): 20488
n=257   Maximum resident set size (kbytes): 20336
```

Stack

Defined in header `<stack>`

A container adaptor for a LIFO (last-in, first-out) data structure

Default container: deque

No random access and iterators

`push_back()` in deque → `push()` in stack

`emplace_back()` in deque → `emplace()` in stack

`pop_back()` in deque → `pop()` in stack

`back()` in deque → `top()` in stack

Stack

```cpp
stack<int> st;
st.emplace(1);
st.emplace(2);
cout << st.top() << '\n';
st.pop();
cout << st.top() << '\n';
```

Output

```
2
1
```

Queue

Defined in header `<queue>`

A container adaptor for a FIFO (first-in, first-out) data structure

Default container: deque

No random access and iterators

`push_back()` in deque → `push()` in queue

`emplace_back()` in deque → `emplace()` in queue

`pop_front()` in deque → `pop()` in queue

`front()` in deque → `front()` in queue

Queue

```cpp
queue<int> q;
q.emplace(1);
q.emplace(2);
cout << q.front() << '\n';
q.pop();
cout << q.front() << '\n';
```

Output

```
1
2
```

List

Defined in header `<list>`

Implemented as a doubly-linked list

No random access

Supports `push_front()`, `push_back()`, `pop_front()`, `pop_back()`, etc.

To declare an empty int list: `list<int> l`

To sort the list:

- `l.sort()` (time complexity: $O(n \log n)$)
- `sort(l.begin(), l.end())` (time complexity: $O(n^2)$)

List

To insert *x* before the element pointed by *it*:

- `l.insert(it, x)`
- Returns an iterator pointing to the element inserted

To remove the element pointed by *it*:

- `l.erase(it)`
- Returns an iterator pointing to the next element after *it*

List

```cpp
list l{10, 20};
l.emplace_back(30);
l.emplace_front(0);
cout << l.front() << ' ' << l.back() << '\n';
auto it = l.begin();
it++;
it = l.erase(it);
for (auto x : l) cout << x << ' ';
cout << '\n';
it++;
it = l.emplace(it, 25);
for (auto x : l) cout << x << ' ';
cout << '\n';
l.emplace(l.end(), 40);
for (auto x : l) cout << x << ' ';
cout << '\n';
cout << *it << '\n';
```

Output

```
0 30
0 20 30
0 20 25 30
0 20 25 30 40
25
```

Priority queue

Defined in header `<queue>`

Implementation of a **max** heap

Default container: vector

To insert an element: `q.push(x)` / `q.emplace(x)`

To get the largest element: `q.top()`

To remove the largest element: `q.pop()`

Time complexity: *O(1)* for `top()` and *O(log n)* for `push()` / `pop()`

Priority queue

```cpp
priority_queue<int> q;
q.emplace(1);
q.emplace(3);
q.emplace(2);
cout << q.top() << '\n';
q.pop();
cout << q.top() << '\n';
```

Output

```
3
2
```

Min heap

If you want a min heap, there are 3 ways:

- Use greater
  - `priority_queue<int, vector<int>, greater<int>> q`
  - You can use alias template so you can use it in the future easily:
    - `template<typename T> using min_heap = priority_queue<T, vector<T>, greater<T>>`
    - `min_heap<int> q`
- Define a struct that contains `operator()`
- Define a lambda expression, and use `decltype()` to get its type

Min heap - self-defined struct

```cpp
struct cmp {
   bool operator()(int a, int b) { return a > b; }
};

priority_queue<int, vector<int>, cmp> q;
q.emplace(1);
q.emplace(3);
q.emplace(2);
cout << q.top() << '\n';
q.pop();
cout << q.top() << '\n';
```

Output

```
1
2
```

Min heap - self-defined lambda expression

```
auto cmp = [](int a, int b) { return a > b; };

priority_queue<int, vector<int>, decltype(cmp)> q(cmp);
q.emplace(1);
q.emplace(3);
q.emplace(2);
cout << q.top() << '\n';
q.pop();
cout << q.top() << '\n';
```

Output

```
1
2
```

Set and multiset

Defined in header `<set>`

Associative containers

set contains a sorted set of **unique** keys

multiset contains a sorted set of keys

Usually implemented as a red-black tree

Time complexity: *O(log n)* for each operation

Set and multiset

To declare an empty int set: `set<int> s`

To insert an element *x*: `s.insert(x)` / `s.emplace(x)`

To remove **elements** that are equal to *x*: `s.erase(x)`

To remove the element at *it*: `s.erase(it)`

To find *x*: `s.find(x)`

To get the lower bound of *x*: `s.lower_bound(x)`
(`lower_bound(s.begin(), s.end(), x)` compiles but is *O(n)*)

To get the upper bound of *x*: `s.upper_bound(x)`
(`upper_bound(s.begin(), s.end(), x)` compiles but is *O(n)*)

Set

```cpp
set s{1, 2, 2, 3, 4};
s.emplace(5);
cout << s.size() << '\n';
s.erase(2);
auto it = s.lower_bound(2);
if (it != s.end()) cout << *it << '\n';
else cout << "None\n";
it = s.emplace(5).first;
cout << *it << '\n';
s.erase(it);
it = s.find(3);
if (it != s.end()) cout << *it << '\n';
else cout << "None\n";
for (auto x : s) cout << x << ' ';
cout << '\n';
```

Output

```
5
3
5
3
1 3 4
```

Multiset

```
multiset s{1, 2, 2, 3, 4};
s.emplace(5);
cout << s.size() << '\n';
s.erase(2);
auto it = s.lower_bound(2);
if (it != s.end()) cout << *it << '\n';
else cout << "None\n";
it = s.emplace(5);
cout << *it << '\n';
s.erase(it);
it = s.find(3);
if (it != s.end()) cout << *it << '\n';
else cout << "None\n";
for (auto x : s) cout << x << ' ';
cout << '\n';
```

Output

```
6
3
5
3
1 3 4 5
```

Map and multimap

Defined in header `<map>`

Associative containers

map contains **key-value pairs** with **unique** keys

multimap contains a sorted list of **key-value pairs**

The value can be accessed by `operator[]` in map

Time complexity: *O(log n)* for each operation

Map

```cpp
map<int, int> mp;
auto it = mp.emplace(1, 2).first;
cout << it->first << ' ' << it->second << '\n';
mp[2] = 0;
mp[2]++;
it = mp.find(2);
if (it != mp.end()) cout << it->first << ' ' << it->second << '\n';
else cout << "None\n";
for (auto [key, val] : mp) cout << key << ' ' << val << '\n';
```

Output

```
1 2
2 1
1 2
2 1
```

In practice, we use map to store the frequency of a key instead of using multiset with `count()`

Unordered set and unordered map (since C++11)

Defined in headers `<unordered_set>` and `<unordered_map>` respectively

Similar to set and map, but use hash table to implement

`operator<` is no longer required, but a hash function is required (built-in hash for int, long long, ...)

To define a hash function: similar to defining a comparison function object, but returns integers instead of a boolean value

Expected time complexity: *O(1)* for each operation

Worst case time complexity: *O(n)* for each operation

You can use `reserve()` to save time if you know the final size

Hash function - self-defined class

```
struct Hash {
   int operator()(pair<int, int> x) const { return x.first ^
x.second; }
};

unordered_map<pair<int, int>, int, Hash> mp;
```

Output

Bitset

Defined in header `<bitset>`

Represents a **fixed-size** sequence of *n* bits

Supports bitwise operations (&, ^, |, ...)

Can use `operator[]` to access values (like a boolean array)

To declare a bitset: `bitset<n> s;` (*n* must be known in compile time)

Bitset

To set a bit to 1: `s.set(x)` / `s[x] = 1`

To set a bit to 0: `s.reset(x)` / `s[x] = 0`

To flip a bit: `s.flip(x)`

To set all bits to 1: `s.set()`

To set all bits to 0: `s.reset()`

To flip all bits: `s.flip()`

To count number of bits set to 1: `s.count()`

Bitset

```
bitset<10> s(100);
cout << s << '\n';
cout << s.count() << '\n';
s.set(0);
cout << s << '\n';
s.reset(3);
cout << s << '\n';
s.flip(1);
cout << s << '\n';
s[5] = 0;
cout << s << '\n';
s.reset();
cout << s << '\n';
```

Output

```
0001100100
3
0001100101
0001100101
0001100111
0001000111
0000000000
```

Application - bitset

Please read the problem M2002 - Corona and Movies

Mathematical constants (since C++20)

There are a lot of mathematical constants defined in header `<numbers>`, using the `std::numbers` namespace

- Be careful if you want to `using namespace std::numbers`, as there are a lot of names that are commonly used
- A better way is to explicitly state which constant you want to use by `using`, for example: `using numbers::pi;`

More in C++ Standard Library

`<algorithm>`:

- count: count the number of matching elements in a range
- find: find the position of the first matching elements in a range
- fill: fill a value to all elements in a range
- rotate: perform cyclic shift in a range
- is_sorted (since C++11): check is a range is sorted
- partial_sort: partially sort a range
- stable_sort: perform stable sort in a range
- nth_element: find the n^{th} smallest element in a range
- merge and inplace_merge: merge two sorted ranges

More in C++ Standard Library

`<algorithm>`:

- includes, set_difference, set_intersection and set_union: perform set operations in to sorted ranges
- max and min: get max / min of two elements / among any number of elements (by using initializer_list, since C++11)
- max_element and min_element: get max / min among a range
- clamp (since C++17): clamp a value between a pair of boundary values
- is_permutation: check if a range is a permutation of another range
- next_permutation and prev_permutation: find the next / previous permutation of a range

More in C++ Standard Library

`<numeric>`:

- iota: fill a range with continuous increasing numbers
- accumulate: calculate the sum of a range
- inner_product: calculate the sum of the product of two ranges
- adjacent_difference: calculate the differences between adjacent elements in a range
- partial_sum: calculate the partial sum of a range
- gcd and lcm (since C++17): calculate the gcd (greatest common divisor) / lcm (least common multiple) of two values

More in C++ Standard Library

`<numeric>`:

- midpoint (since C++20): calculate the midpoint between two numbers

`<cmath>`:

- lerp (since C++20): calculate the linear interpolation / extrapolation between two numbers

`<complex>`:

- complex: a type for handling complex numbers

Explore cppreference for more

Non-standard Library

Possibly not existing in some C++ compilers

Usable in g++ (which is used in HKOI Online Judge)

Lack of good (and official) documentation

https://gcc.gnu.org/onlinedocs/libstdc++/ext/pb_ds/

https://github.com/kth-competitive-programming/kactl/blob/master/content/data-structures/OrderStatisticTree.h

https://codeforces.com/blog/entry/10355

https://www.luogu.org/blog/Chanis/gnu-pbds

https://www.mina.moe/archives/2481

Conclusion

- C++ standard libraries are useful
- You can save a lot of time during the contest by using them appropriately
- Every problem can be your practice problem
- Try to develop your own coding style base on them
- cppreference is always your good friend

Reference

cppreference.com
Standard Template Library - Wikipedia
C++ Standard Library - Wikipedia
https://assets.hkoi.org/training2017/cppstl.pdf
https://assets.hkoi.org/training2019/adv-cpp.pdf
https://assets.hkoi.org/training2019/cpp.pdf
https://assets.hkoi.org/training2021/cpp.pdf