# Advanced C++ STL

David Wai {wjx}

2021-04-03

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

# About C++ STL

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

It is a part of C++ Standard Library

It has four components:

- Algorithms (sort, binary_search, etc.)
- Containers (vector, set, map, etc.)
- Functions
- Iterators

# Why C++ STL?

Write shorter code (less bugs)

No need to implement some algorithms and data structures by yourself

Learn more about C++

Instead of how it works, we will focus more on how to use it

# C++ Standard Library

Features of the C++ Standard Library are declared within the std namespace and they are defined in different headers

You can use the following two lines to simplify your code:

```
#include<bits/stdc++.h> // not a standard header file
using namespace std;
```

Some features in the C++ Standard library will also be included in this session

# C++11/14/17

Some features in this session requires C++11 or newer standards (C++14, C++17)

In most cases, new standards are backward compatible

g++ flag: -std=c++11 (C++11), -std=c++14 (C++14), -std=c++17 (C++17)

C++17 is supported in HKOI Online Judge, IOI, APIO

C++11 is supported in NOI

# Template

There are two types of template: class template and function template

They can be applied to different types

Compiler will generate a function for each used type

```cpp
template<class T> // class template
struct myStruct {
    T a, b;
};

template<class T> // function template
T sum(T a, T b) {
    return a + b;
}

int main() {
    myStruct<int> a = {1, 2};
    cout << sum(a.a, a.b) << endl; // 3
    myStruct<int> a = {0.1, 0.2};
    cout << sum(a.a, a.b) << endl; // 0.3
    return 0;
}
```

# std::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**

```
int main() {
    int a[5] = {3, 2, 4, 5, 1};
    sort(a, a + 5);
    for (int i = 0; i < 5; i++) cout << a[i] << ' '; // 1 2 3 4 5
    cout << endl;
    return 0;
}
```

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

# std::sort

If you want to sort the elements in descending order, there are two ways:

- Use std::greater (defined in header <functional>)
- Define your own comparison function

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

int main() {
    int a[5] = {3, 2, 4, 5, 1};
    sort(a, a + 5, greater<int>());
    sort(a, a + 5, cmp); // same as the line above
    for (int i = 0; i < 5; i++) cout << a[i] << ' '; // 5 4 3 2 1
    cout << endl;
    return 0;
}
```

# std::sort

In C++11, you can also write like this by using lambda expressions:

```
sort(a, a + 5, [](int a, int b) { return a > b; });
```

For self-defined types, you can either define `operator<` or use comparison function

```
struct myStruct {
    int a, b;
    bool operator< (const myStruct &rhs) const {
        return a < rhs.a;
    }
};

bool cmp(const myStruct &a, const myStruct &b) {
    return a.a < b.a;
}
```

# Binary Search Functions

Defined in header <algorithm>

Uses binary search algorithm (see Searching and Sorting for details)

std::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

std::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 Functions

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

Note that *first* and *last* should be **random access iterators** and the range should be sorted

Time Complexity: O(log *n*)

# Binary Search Functions

```cpp
int main() {
    int a[5] = {1, 2, 2, 4, 5};
    cout << lower_bound(a, a + 5, 2) - a << ' ' << *lower_bound(a, a + 5, 2) << endl; // 1 2
    cout << lower_bound(a, a + 5, 3) - a << ' ' << *lower_bound(a, a + 5, 3) << endl; // 3 4
    cout << upper_bound(a, a + 5, 2) - a << ' ' << *upper_bound(a, a + 5, 2) << endl; // 3 4
    cout << upper_bound(a, a + 5, 3) - a << ' ' << *upper_bound(a, a + 5, 3) << endl; // 3 4
    cout << binary_search(a, a + 5, 2) << endl; // 1
    cout << binary_search(a, a + 5, 3) << endl; // 0
    return 0;
}
```

# std::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.

```
int main() {
    int a[5] = {1, 2, 2, 4, 2};
    int n = unique(a, a + 5) - a;
    for (int i = 0; i < n; i++) cout << a[i] << ' '; // 1 2 4 2
    cout << endl;
    return 0;
}
```

# Discretization

By using the above algorithms, we can do discretization very easily (see Optimization for details)

```
int n, num = 0;
cin >> n;
for (int i = 1; i <= n; i++) cin >> a[i];
for (int i = 1; i <= n; i++) b[i] = a[i];
sort(b + 1, b + 1 + n);
num = unique(b + 1, b + 1 + n) - b - 1;
for (int i = 1; i <= n; i++) a[i] = lower_bound(b + 1, b + 1 + num, a[i]) - b;
```

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

# std::reverse

Defined in header <algorithm>

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

```
int main() {
    int a[5] = {1, 2, 3, 4, 5};
    reverse(a, a + 5);
    for (int i = 0; i < 5; i++) cout << a[i] << ' '; // 5 4 3 2 1
    cout << endl;
    return 0;
}
```

# std::pair and std::tuple

Defined in headers <utility> and <tuple> respectively

std::pair is a struct template to store two objects

std::tuple is a generalization of std::pair (2 -> n)

To declare a pair: `pair<int, long long> a;`

To declare a tuple: `tuple<int, long long, char> b;`

# std::pair and std::tuple

Data members of pair can be accessed by `a.first` and `a.second`

Data members of tuple can be accessed by `get<0>(b)`, `get<1>(b)`, `get<2>(b)`, etc. (n in `get<n>(b)` should be known in compile time)

In C++17, you can use a feature called structured binding declaration to assign the data members to some variables:

```
auto [x, y] = a;
auto [u, v, w] = b;
```

Then you can access the elements by using the variables

# std::pair and std::tuple

```cpp
pair<int, int> a = {1, 2};
tuple<int, char, int> b = {3, 'a', 5};
cout << a.first << ' ' << a.second << endl; // 1 2
cout << get<0>(b) << ' ' << get<1>(b) << ' ' << get<2>(b) << endl; // 3 a 5
auto [x, y] = a;
cout << x << ' ' << y << endl; // 1 2
auto [u, v, w] = b;
cout << u << ' ' << v << ' ' << w << endl; // 3 a 5
```

# std::pair and std::tuple

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

```
int a[5] = {2, 3, 4, 1, 5};
pair<int, int> b[5];
for (int i = 0; i < 5; i++) b[i] = {a[i], i};
sort(b, b + 5);
for (int i = 0; i < 5; i++) cout << b[i].second << ' ' << b[i].first << "  "; // 3 1  0 2  1 3  2 4  4 5
cout << endl;
```

# std::vector

Defined in header <vector>

Similar to an array (O(1) random access), but with dynamic size

Comparison (<, <=, ==, ...) works with lexicographical order

To declare an empty int vector:

```
vector<int> a;
```

# std::vector

To declare a long long vector of size 100:

```
vector<long long> a(100);
```

To declare a char vector of size 10 with element initialized to ‘a’:

```
vector<char> a(10, ‘a’);
```

To declare a 2D int vector of size *n* * *m*:

```
vector<vector<int>> a(n, vector<int>(m));
```

# std::vector

To add an element to the end:

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

To remove the last element:

```
a.pop_back();
```

To clear a vector:

```
a.clear();
```

# std::vector

```cpp
vector<int> a;
for (int i = 0; i < 10; i++) a.emplace_back(10 - i);
for (auto x : a) cout << x << ' '; // 10 9 8 7 6 5 4 3 2 1
cout << endl;
a.pop_back();
for (int i = 0; i < a.size(); i++) cout << a[i] << ' '; // 10 9 8 7 6 5 4 3 2
cout << endl;
sort(a.begin(), a.end());
a.pop_back();
for (auto it = a.begin(); it != a.end(); it++) cout << *it << ' '; // 2 3 4 5 6 7 8 9
cout << endl;
```

# std::vector

Difference between `push_back` and `emplace_back`:

- `emplace_back` is faster especially for a large struct
- Also, their implementation are slightly different

```
vector<pair<int, int>> a;
a.push_back(make_pair(1, 2));
a.emplace_back(1, 2); // same as the line above
```

# std::vector

Application example: Find dfs order of a tree

```cpp
vector<vector<int>> e; // store all the edges here
vector<int> dfs_order;

void dfs(int u, int fa) {
    dfs_order.emplace_back(u);
    for (auto v : e[u])
        if (v != fa) dfs(v, u);
}
```

# std::vector

How does vector work with dynamic size and random access iterators?

The main idea is reallocation

When the ”array” is not large enough, a larger ”array” will be ”created” (usually with double size)

Everything in the old ”array” will then be moved to the new ”array”

”Array” size <= $2n$

Number of moves <= $1 + 2 + 4 + \ldots + 2^{\lceil \log n \rceil} < 4n$

# std::vector

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

Output:

```
size: 0 capacity: 0
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
```

Overall time complexity for pushing $n$ elements: $O(n)$

# std::vector

How to improve the time performance of vector?

- Use reserve so the elements do not need to move many times
- Use resize if you know the final size, then use it like an normal array

```
vector<int> a;
a.reserve(100);
cout << a.size() << ' ' << a.capacity() << endl; // 0 100
a.resize(200);
cout << a.size() << ' ' << a.capacity() << endl; // 200 200
```

# std::deque

Defined in header <deque>

A double-ended queue with random access (see Data Structures (I) for details)

To push an element to the front: `push_front` or `emplace_front`
To push an element to the end: `push_back` or `emplace_back`
To pop an element at the front: `pop_front`
To pop an element at the end: `pop_back`

# std::deque

To access the first element: `front`

To access the last element: `back`

```
deque<int> q{4, 5, 6};
q.emplace_front(3); // 3 4 5 6
q.emplace_back(7);  // 3 4 5 6 7
cout << q[2] << endl; // 5
q.pop_back();       // 3 4 5 6
cout << q.back() << endl; // 6
q.pop_front();      // 4 5 6
cout << q.front() << endl; // 4
```

# std::deque

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

The answer is the memory usage:

std::deque: $n + c_1$

std::vector: $2n + c_2$

$c_1 > c_2$

The internal storage of std::deque are not contiguous

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

# std::deque

std::vector

```
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
```

std::deque

```
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
```

# std::stack and std::queue

Defined in headers <stack> and <queue> respectively

Container adapters for LIFO and FIFO data structures respectively (See Data Structures (I) for details)

No random access

No iterators

# std::stack and std::queue

`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

```
queue<int> q;
q.emplace(1);
q.emplace(2);
cout << q.front() << endl; // 1
q.pop();
cout << q.front() << endl; // 2
```

`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

```
stack<int> st;
st.emplace(1);
st.emplace(2);
cout << st.top() << endl; // 2
st.pop();
cout << st.top() << endl; // 1
```

# std::list

Defined in header <list>

Implemented as a doubly-linked list (see Data Structures (I) for details)

No random access

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

# std::list

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

To sort the list: `l.sort();` (You can't use `sort(l.begin(), l.end());`)

To reverse the list: `l.reverse();` or `reverse(l.begin(), l.end());`

`l.insert(it, x)`: Inserts *x* before the element pointed by *it* and returns an iterator pointing to the element inserted

`l.erase(it)`: Removes the element pointed by *it* and returns an iterator pointing to the next element after *it*

# std::list

```
list<int> l{10, 20};
l.emplace_back(30); // 10 20 30
l.emplace_front(0); // 0 10 20 30
cout << l.front() << endl; // 0
cout << l.back() << endl; // 30
auto it = l.begin();     // >0 10 20 30
it++;                    // 0 >10 20 30
it = l.erase(it);        // 0 >20 30
it++;                    // 0 20 >30
it = l.emplace(it, 25); // 0 20 >25 30
l.insert(l.end(), 40);  // 0 20 >25 30 40
cout << *it << endl; // 25
```

# std::priority_queue

Defined in header <queue>

Implementation of a max heap (See Data Structures (II) for details)

Use std::vector as the default container

# std::priority_queue

`push(x)` or `emplace(x)`: Inserts *x*

`top()`: Returns the largest element

`pop()`: Removes the largest element

```
priority_queue<int> pq;
pq.emplace(1);
pq.emplace(3);
pq.emplace(2);
cout << pq.top() << endl; // 3
pq.pop();
cout << pq.top() << endl; // 2
```

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

# std::priority_queue

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

- Use std::greater (defined in header <functional>)
- Define `operator()`

For self-defined types, use `operator<`

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

int main() {
    priority_queue<int, vector<int>, greater<int>> pq;
    priority_queue<int, vector<int>, cmp> pq; // same as the line above
    pq.emplace(1);
    pq.emplace(3);
    pq.emplace(2);
    cout << pq.top() << endl; // 1
    pq.pop();
    cout << pq.top() << endl; // 2
    return 0;
}
```

# std::set and std::multiset

Defined in header <set>

Associative containers

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

std::multiset contains a sorted set of keys

Usually implemented as a red-black tree

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

# std::set and std::multiset

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

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

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

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

# std::set and std::multiset

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*))

# std::set and std::multiset

std::set

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

std::multiset

```cpp
multiset<int> s{1, 2, 2, 3, 4};
s.emplace(5);
cout << s.size() << endl; // 6
s.erase(2);
auto it = s.lower_bound(2);
if (it != s.end()) cout << *it << endl; // 3
else cout << "None" << endl;
it = s.emplace(5);
cout << *it << endl; // 5
s.erase(it);
it = s.find(3);
if (it != s.end()) cout << *it << endl; // 3
else cout << "None" << endl;
for (auto x : s) cout << x << ' '; // 1 3 4 5
cout << endl;
```

# std::map and std::multimap

Defined in header <map>

Associative containers

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

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

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

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

# std::map and std::multimap

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

Use std::map to store frequency of a key instead of std::multiset with `count`

# std::unordered_set and std::unordered_map

Defined in headers <unordered_set> and <unordered_map> respectively (since C++11)

Similar to std::set and std::map, but use hash table to implement (see Data Structures (II) for details)

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

Expected time complexity: O(1) for each operation

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

# std::unordered_set and std::unordered_map

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

To define a hash: Use `operator()`

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

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

# std::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)

# std::bitset

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

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

To set a bit to 0: `s.reset(x);` or `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 count number of bits set to 1: `s.count();`

```
bitset<10> s(100);
cout << s << endl; // 0001100100
cout << s.count() << endl; // 3
s.set(0);
cout << s << endl; // 0001100101
s.reset(3);
cout << s << endl; // 0001100101
s.flip(1);
cout << s << endl; // 0001100111
s[5] = 0;
cout << s << endl; // 0001000111
s.reset();
cout << s << endl; // 0000000000
```

# More in C++ Standard Library

std::array (Defined in header <array>, since C++11)

std::string (Defined in header <string>)

std::stable_sort (Defined in header <algorithm>)

std::next_permutation (Defined in header <algorithm>)

std::max_element and std::min_element (Defined in header <algorithm>)

std::accumulate (Defined in header <numeric>)

std::partial_sum (Defined in header <numeric>)

std::gcd and std::lcm (Defined in header <numeric>, since C++17)

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

# Non-standard Library

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

# Reference

https://en.cppreference.com

https://en.wikipedia.org/wiki/Standard_Template_Library

https://en.wikipedia.org/wiki/C%2B%2B_Standard_Library

https://assets.hkoi.org/training2017/cppstl.pdf

https://assets.hkoi.org/training2019/adv-cpp.pdf