Advanced C++ STL

Joseph Cheung {Mariouz}

2022-04-02

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

References

- The slide is mainly adapted from Advanced C++ STL slides (2021) by **David Wai**

Contents

C++ STL

Standard **T**emplate **L**ibrary

They are also C++ standard library like <iostream>, <cmath>, etc.

It has four components:

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

Advantage

- Less code (less bug)
- Spend less time
- Focus on solving task instead of annoying implementation

C++ Standard Library

No one want to write extra code, you may use this to simplify your code:

```
#include<bits/stdc++.h>
using namespace std;
```

C++ Standard (C++11/14/17/20)

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), -std=c++20 (c++20)

C++20 is supported in HKOI Judge

C++17 is supported in IOI, APIO

C++11 is supported in NOI

Template

2 types of Template:

Class Template and **Function Template**

Can be applied to different types

Compiler will generate a function for each used type

```
template<class T>
struct Vector{ //class template
    T i, j;
};

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

int main(){
    Vector<int> v = {1,2};
    cout << sum(v.i, v.j) << endl; // 3
    Vector<double> u = {0.1, 0.2};
    cout << sum(u.i, u.j) << endl; // 0.3
}
```

std::sort()

Define in <algorithm>

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

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

Time Complexity: **O(NlogN)**

std::sort()

If you want to sort in descending order,

you may use **std::greater** (define in <functional>)

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

std::sort()

Or you may use self-define comparison function:

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

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

std::sort()

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

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

For self-define types, like vector, you can define **operator<**:

```
template<class T>
struct Vector{
    T i, j;
    bool operator < (const Vector v) const {
        if(v.i == i) return v.j < j;
        return v.i < i;
    }
};
```

Binary Search Functions

Define in <algorithm>

Use Binary Search Algorithm

Array that are going to be search should be sorted

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

```
int main(){
    int a[5] = {1, 2, 3, 5, 6};
    cout << lower_bound(a, a + 5, 2) - a << ' ' << *lower_bound(a, a + 5, 2) << '\n'; // 1 2
    cout << lower_bound(a, a + 5, 4) - a << ' ' << *lower_bound(a, a + 5, 4) << '\n'; // 3 5
}
```

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

```
int main(){
    int a[5] = {1, 2, 3, 5, 6};
    cout << upper_bound(a, a + 5, 2) - a << ' ' << *upper_bound(a, a + 5, 2) << '\n'; // 2 3
    cout << upper_bound(a, a + 5, 4) - a << ' ' << *upper_bound(a, a + 5, 4) << '\n'; // 3 5
}
```

std::binary_search()

Returns a boolean value, checks if an element equivalent to *value* appears within the range [*first*, *last*)

```
int main(){
    int a[5] = {1, 2, 3, 5, 6};
    cout << binary_search(a, a + 5, 3) << '\n'; // 1
    cout << binary_search(a, a + 5, 4) << '\n'; // 0
}
```

Binary Search Functions

Since they are using Binary Search Algorithm,

Time complexity: **O(logN)**, where **N** is the number of elements.

std::unique()

Define in <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[10] = {1, 1, 3, 4, 5, 5, 5, 7, 7, 6};
    int n = unique(a, a + 10) - a;
    for(int i=0;i<n;i++) cout << a[i] << ' '; // 1 3 4 5 7 6
}
```

Discretization

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

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

Time complexity: **O(nlogn)**

std::reverse()

Define in <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
}
```

std::max_element and std::min_element

Define in <algorithm>

Return an iterator pointing to the max. or the min. element from range *[first,last)*

```
int a[5] = {4, 5, 3, 1, 2};
cout << max_element(a, a + 5) - a << ' ' << *max_element(a, a + 5) << endl; // 1 5
cout << min_element(a, a + 5) - a << ' ' << *min_element(a, a + 5) << endl; // 3 1
```

std::partial _sum

Define in <numeric>

Calculate the partial sum from a range *[first,last)*

Useful for calculate partial sum. (see Optimization for details)

```
int a[5] = {1, 2, 3, 4, 5};
int b[5];
partial_sum(a, a + 5, b);
for(int i=0;i<5;i++) cout << b[i] << ' '; // 1 3 6 10 15
```

std::partial _sum

You can also define a function to compute “partial sum” for specific task.

```
int func(int a, int b) {
    return a * b;
}

int main(){
    int a[5] = {1, 2, 3, 4, 5};
    int b[5];
    partial_sum(a, a + 5, b, func);
    for(int i=0;i<5;i++) cout << b[i] << ' '; // 1 2 6 24 120
}
```

std::gcd and std::lcm

Define in <numeric> since C++17

Return the gcd or lcm of the **absolute** value of 2 integers

```
cout << gcd(8, 4) << endl; // 4
cout << lcm(8, 4) << endl; // 8
```

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)

Since 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

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

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

std::vector

Define in <vector>, not the vector mentioned in Computational Geometry

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

```
int main(){
    vector<int> v;
    for(int i=5;i>=0;i--) v.push_back(i);
    for(auto x:v) cout << x << ' '; // 5 4 3 2 1 0
    cout << endl;
    for(int i=0;i<6;i++) cout << v[i] << ' '; // 5 4 3 2 1 0
    cout << endl;
    sort(v.begin(), v.end());
    for(vector<int>::iterator it = v.begin();it != v.end();it++) cout << *it << ' '; // 0 1 2 3 4 5
}
```

std::vector

Difference between `push_back` and `emplace_back`:

- `emplace_back` is faster especially for a large struct
- Also, their implementation are slightly different
- This 2 lines are doing the same thing

```
vector<pair<int, int> > v;
v.push_back({1, 2});
v.emplace_back(1, 2);
```

std::vector

Some useful applications: Finding the pre-order of a tree

```
vector<int> e[1005]; // store edges here, you may also use 2D vector

void dfs(int a, int p) {
    cout << a << ' ';
    for(auto x:e[a]) {
        if(x != p) dfs(x, a);
    }
}
```

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

```
vector<int> v;
for(int i=0;i<10;i++) {
    v.push_back(i);
    cout << "Size: " << v.size() << " Capacity: " << v.capacity() << endl;
}
```

```
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
```

Overall time complexity of pusing **n** elements: O(**n**)

std::deque

Define in <deque>

A **d**ouble-**e**nded **que**ue 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{1, 2, 3};
q.push_front(4); // 4 1 2 3
q.push_back(0);  // 4 1 2 3 0
cout << q[0] << endl; // 4
q.pop_back();    // 4 1 2 3
cout << q.back() << endl; // 3
q.pop_front();  // 1 2 3
cout << q.front() << endl; // 1
```

std::deque

Why using **std::vector**? Seems **std::deque** is more useful

That because the memory of **std::deque** is not guarantee contiguous

i.e.

```
deque<int> q{1, 2, 3};
int *q_ptr = &q[0]; // 1
q_ptr++;
cout << *q_ptr << endl; // not nessary 2

vector<int> v{1, 2, 3};
int *v_ptr = &v[0]; // 1
v_ptr++;
cout << *v_ptr; // must be 2
```

std::stack

Define in <stack>

Container adapters for LIFO data structures(See Data Structures (I) for details), the STL for stack basically

No random access

No iterators

std::stack

push => push an element into the stack

pop => pop an element from the stack

top => the toppest element of the stack

```
stack<int> s;
s.push(1);
s.push(2);
cout << s.top() << endl; // 2
s.pop();
cout << s.top() << endl; // 1
s.pop();
cout << s.top() << endl; // RE
```

std::queue

Define in <queue>

Container adapters for FIFO data structures(See Data Structures (I) for details), the STL for queue

No random access

No iterators

std::queue

push => enqueue an element into the back of the queue

pop => dequeue an element from the front of the queue

`front` => the first element of the queue

```
queue<int> q;
q.push(1); // [1]
q.push(3); // [1, 3]
cout << q.front() << endl; // 1
q.pop(); // [3]
cout << q.front() << endl; // 3
q.pop(); // []
cout << q.front() << endl; // RE
```

std::list

Define in <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::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> q;
q.push(2);
q.push(3);
cout << q.top() << endl; // 3
q.pop();
cout << q.size() << endl; // 1
```

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<`, just like **std::sort**

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

int main(){
    priority_queue<int, vector<int> , cmp> q;
    q.push(2);
    q.push(3);
    cout << q.top() << endl; // 2
    q.pop();
    cout << q.top() << endl; // 3
}
```

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, **repeated** keys will also be stored

Usually implemented as a red-black tree

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

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

set:

```
set<int> s{1, 2, 3, 4, 5};
s.insert(5);
cout << s.size() << endl; // 5
set<int>::iterator it = s.lower_bound(3);
if(it != s.end()) cout << *it << endl; // 3
else cout << "None\n";
s.erase(it);
cout << s.size() << endl; // 4
it = s.upper_bound(5);
if(it != s.end()) cout << *it << endl; //None
else cout << "None\n";
```

multiset:

```
multiset<int> s{1, 2, 3, 4, 5};
s.insert(5);
cout << s.size() << endl; // 6
set<int>::iterator it = s.lower_bound(5);
if(it != s.end()) cout << *it << endl; // 5
else cout << "None\n";
s.erase(it);
cout << s.size() << endl; // 5
it = s.upper_bound(4);
if(it != s.end()) cout << *it << endl; //5
else cout << "None\n";
```

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(log *n*) for each operation

std::map and std::multimap

```
map<int, char> mp;
mp[2] = 'a';
mp[1] = 'b';
cout << mp[2] << endl; // a
auto it = mp.find(1);
cout << it->first << ' ' << it->second << end; // 1 b
```

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

std::unordered_set and std::unordered_map

Defined in <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() (pair<int, int> a) const {
        return a.first ^ a.second;
    }
};

int main(){
    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(101); //101 in binary = 1100101
cout << s << endl; //0001100101
s.reset(2);
s[1] = 1;
cout << s << endl; //0001100011
cout << s.count() << endl; //4
s = ~s;    // same as s.flip()
cout << s << endl; //1110011100
```

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::accumulate (Defined in header <numeric>)

Explore cppreference for more

Non-standard Library

Possibly not existing in some C++ compilers

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

For example: __buildin_popcount, __gcd

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

Practice Problem

https://judge.hkoi.org/task/M2002

https://judge.hkoi.org/task/N1511

https://judge.hkoi.org/task/M1904

https://judge.hkoi.org/task/M1122

References

https://assets.hkoi.org/training2021/adv-cpp.pdf

https://en.cppreference.com

https://www.geeksforgeeks.org/cpp-stl-tutorial/