C++ STL 常用组件

By 吴羽晗.

简介

C++ 标准模板库（Standard Template Library，STL）包含一系列高效的数据结构与算法。C++ 编译器自带 STL，编程时可以直接使用。

本文只介绍常用 STL 组件的部分功能，完整的 STL 文档见于 CPP Reference。

- **首次学习建议只看示例；**
- **部分功能需要开启 C++11。See how to enable support for C++11.**

`pair`

二元组。可以组合任意两个不同类型（或相同类型）的元素。

头文件

```
#include <utility>
```

成员变量

变量名	内容
`first`	第一个值
`second`	第二个值

示例

```
#include <iostream>
```

```cpp
#include <utility>

int main() {

    std::pair<char, double> p1 = {'x', 2.5};

    std::pair<int, std::pair<char, double>> p2 = {1, p1};
    // 可嵌套

    std::cout << p2.first << ' ';
    std::cout << p2.second.first << ' ';
    std::cout << p2.second.second << ' ';

    return 0;
}
```

```
1 x 2.5
```

stack

栈。只允许在栈顶进行插入和删除操作。

头文件

```cpp
#include <stack>
```

定义与初始化

```cpp
std::stack<int> s;
// s = {}
```

成员函数

函数名	功能	时间复杂度
`top()`	返回栈顶元素	$O(1)$

函数名	功能	时间复杂度
`empty()`	返回是否为空	$O(1)$
`size()`	返回元素个数	$O(1)$
`push(value)`	在栈顶插入 `value`	$O(1)$
`pop()`	删除栈顶元素	$O(1)$

示例

```cpp
#include <iostream>
#include <stack>

int main() {

    std::stack<int> s;

    s.push(1);
    s.push(2);
    s.push(3);
    s.push(4);
    s.push(5);
    // s = {1, 2, 3, 4, 5(top)}

    s.pop();
    // s = {1, 2, 3, 4(top)}

    while (! s.empty()) {
        std::cout << s.top() << ' ';
        s.pop();
    }
    // s = {}

    return 0;
}
```

```
4 3 2 1
```

queue

队列。只允许在队尾插入元素，在队头删除元素。

头文件

```
#include <queue>
```

定义与初始化

```
std::queue<int> q;
// q = {}
```

成员函数

函数名	功能	时间复杂度
`front()`	返回队头元素	$O(1)$
`back()`	返回队尾元素	$O(1)$
`empty()`	返回是否为空	$O(1)$
`size()`	返回元素个数	$O(1)$
`push(value)`	在队尾插入 `value`	$O(1)$
`pop()`	删除队头元素	$O(1)$

示例

```
#include <iostream>
#include <queue>

int main() {

    std::queue<int> q;
```

```
    q.push(1);
    q.push(2);
    q.push(3);
    q.push(4);
    q.push(5);
    // q = {1(front), 2, 3, 4, 5(back)}

    q.pop();
    // q = {2(front), 3, 4, 5(back)}

    while (! q.empty()) {
        std::cout << q.front() << ' ';
        q.pop();
    }
    // q = {}

    return 0;
}
```

```
2 3 4 5
```

priority_queue

优先队列。会自动排序，但其内部元素不可见，只允许访问最大的元素。

头文件

```
#include <queue>
```

定义和初始化

```
std::priority_queue<int> pq;
```

`priority_queue` 默认为大根堆（只允许访问最大的元素）。以下是小根堆的写法：

```
#include <vector> // 包含 std::vector 和 std::greater

```

```
std::priority_queue<int, std::vector<int>, std::greater<int>> pq;
```

成员函数

函数名	功能	时间复杂度
`top()`	返回最大的元素	$O(1)$
`empty()`	返回是否为空	$O(1)$
`size()`	返回元素个数	$O(1)$
`push(value)`	往堆中插入 `value`	$O(\log n)$
`pop()`	删除最大的元素	$O(\log n)$

示例

```cpp
#include <iostream>
#include <queue>

int main() {

    std::priority_queue<int> pq;

    pq.push(3);
    pq.push(1);
    pq.push(2);
    // pq = {1, 2, 3(top)}

    while (! pq.empty()) {
        std::cout << pq.top() << ' ';
        pq.pop();
    }

    return 0;
}
```

```
3 2 1
```

`vector`

动态数组。能根据需要自动扩容，也能手动调整容量。

头文件

```
#include <vector>
```

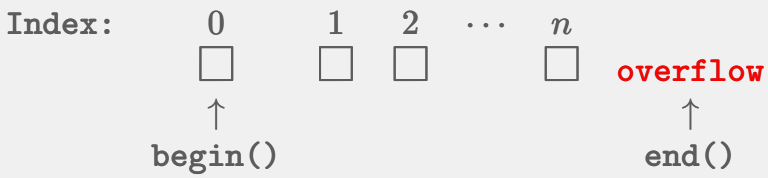
定义和初始化

```
std::vector<float> vec1;
// vec1 = {}

std::vector<int> vec2 = {1, 1, 4, 5, 1, 4};
// vec2 = {1, 1, 4, 5, 1, 4}

std::vector<double> vec3(4, 0.5);
// vec3 = {0.5, 0.5, 0.5, 0.5}
```

成员函数

函数名	功能	时间复杂度
`assign(count, value)`	初始化为 `count` 个 `value`	$O(n)$
`at(pos)`	返回第 `pos` 个元素	$O(1)$
`operator [pos]`	返回第 `pos` 个元素	$O(1)$
`front()`	返回第一个元素	$O(1)$
`back()`	返回最后一个元素	$O(1)$
`begin()`	返回头部迭代器	$O(1)$
`end()`	返回尾部迭代器	$O(1)$
`empty()`	返回是否为空	$O(1)$

函数名	功能	时间复杂度
`size()`	返回元素个数	$O(1)$
`clear()`	清空	$O(n)$
`push_back(value)`	在尾部插入 `value`	$O(1)$
`pop_back()`	删除尾部元素	$O(1)$
`resize(count)`	将容量调整为 `count`	$O(n)$

> STL 中的 `begin()` 迭代器均指向头部元素；`end()` 迭代器均指向尾部元素的后继，而不是尾部元素。
>
> 
>

示例

```cpp
#include <iostream>
#include <vector>

int main() {

    std::vector<int> vec(4, 3);
    // vec = {3, 3, 3, 3}

    vec.push_back(6);
    vec.push_back(9);
    // vec = {3, 3, 3, 3, 6, 9}

    vec[2] = 1;
    // vec = {3, 3, 1, 3, 6, 9}

    vec.pop_back();
    // vec = {3, 3, 1, 3, 6}

    for (int el : vec) {
        std::cout << el << ' ';
    }
```

```
    return 0;
}
```

```
3 3 1 3 6
```

deque

双端队列。相较于 `vector` 增加了在头部的插入、删除操作。

头文件

```
#include <deque>
```

定义和初始化

```
std::deque<float> dq1;
// dq1 = {}

std::deque<int> dq2 = {1, 1, 4, 5, 1, 4};
// dq2 = {1, 1, 4, 5, 1, 4}

std::deque<double> dq3(4, 0.5);
// dq3 = {0.5, 0.5, 0.5, 0.5}
```

成员函数

函数名	功能	时间复杂度
`assign(count, value)`	初始化为 `count` 个 `value`	$O(n)$
`at(pos)`	返回第 `pos` 个元素	$O(1)$
`operator [pos]`	返回第 `pos` 个元素	$O(1)$
`front()`	返回第一个元素	$O(1)$

函数名	功能	时间复杂度
`back()`	返回最后一个元素	$O(1)$
`begin()`	返回头部迭代器	$O(1)$
`end()`	返回尾部迭代器	$O(1)$
`empty()`	返回是否为空	$O(1)$
`size()`	返回元素个数	$O(1)$
`clear()`	清空	$O(n)$
`push_back(value)`	在尾部插入 `value`	$O(1)$
`pop_back()`	删除尾部元素	$O(1)$
`push_front(value)`	在头部插入 `value`	$O(1)$
`pop_front()`	删除头部元素	$O(1)$
`resize(count)`	将容量调整为 `count`	$O(n)$

示例

```cpp
#include <iostream>
#include <deque>

int main() {

    std::deque<int> dq(2, 3);
    // dq = {3, 3}

    dq.push_back(6);
    dq.push_back(9);
    // dq = {3, 3, 6, 9}

    dq.push_front(0);
    dq.push_front(-1);
    // dq = {-1, 0, 3, 3, 6, 9}

    dq[2] = 1;
```

```
    // dq = {-1, 0, 1, 3, 6, 9}

    dq.pop_back();
    // dq = {-1, 0, 1, 3, 6}

    dq.pop_front();
    // dq = {0, 1, 3, 6}

    for (int el : dq) {
        std::cout << el << ' ';
    }

    return 0;
}
```

```
0 1 3 6
```

`list`

链表。支持在任意位置插入和删除元素，但不支持随机访问。

> 随机访问：对于任意给定的 i，可以直接访问到第 i 个元素。

头文件

```
#include <list>
```

定义与初始化

```
std::list<int> l1;
// l1 = {}

std::list<int> l2 = {1, 2, 3, 4, 5};
// l2 = {1, 2, 3, 4, 5}
```

成员函数

函数名	功能	时间复杂度
`assign(count, value)`	初始化为 `count` 个 `value`	$O(n)$
`front()`	返回第一个元素	$O(1)$
`back()`	返回最后一个元素	$O(1)$
`begin()`	返回头部迭代器	$O(1)$
`end()`	返回尾部迭代器	$O(1)$
`empty()`	返回是否为空	$O(1)$
`size()`	返回元素个数	$O(1)$
`clear()`	清空	$O(n)$
`insert(pos, value)`	在迭代器 `pos` 处插入 `value`	$O(1)$
`erase(pos)`	删除迭代器 `pos` 处的元素	$O(1)$
`push_back(value)`	在尾部插入 `value`	$O(1)$
`pop_back()`	删除尾部元素	$O(1)$
`push_front(value)`	在头部插入 `value`	$O(1)$
`pop_front()`	删除头部元素	$O(1)$
`resize(count)`	将容量调整为 `count`	$O(n)$

> `list` 不支持随机访问，因此对于 `list` 的两个迭代器 `p1` 和 `p2`，不能用 `p1 - p2` 计算它们的距离，必须使用 `std::distance(p1, p2)`，其时间复杂度是 $O(n)$。

示例

```cpp
#include <iostream>
#include <list>
```

```cpp
int main() {

    std::list<int> l = {1, 2, 3, 4, 5};

    auto pos = l.begin();
    // 获取头部迭代器

    advance(pos, 2);
    // 使迭代器前进两步

    l.insert(pos, -1);
    // l = {1, 2, -1, 3, 4, 5}

    for (int el : l) {
        std::cout << el << ' ';
    }

    return 0;
}
```

```
1 2 -1 3 4 5
```

set

集合。插入其中的每种元素都只保留其一，并自动升序排序。不支持随机访问。

> STL 提供 `multiset` （多重集），相同的元素允许存在多个。其余功能与 `set` 一致。

> STL 提供 `unordered_set`，功能与 `set` 一致，但时间复杂度有所区别。

头文件

```cpp
#include <set>
```

定义与初始化

```cpp
std::set<int> s1;
```

```cpp
// s1 = {}

std::set<char> s2 = {'a', 'p', 'p', 'l', 'e'};
// s2 = {'a', 'e', 'l', 'p'}
```

成员函数

函数名	功能	时间复杂度	时间复杂度（`unordered`）
`begin()`	返回头部迭代器	$O(1)$	$O(1)$
`end()`	返回尾部迭代器	$O(1)$	$O(1)$
`empty()`	返回是否为空	$O(1)$	$O(1)$
`size()`	返回元素个数	$O(1)$	$O(1)$
`clear()`	清空	$O(n)$	$O(n)$
`insert(value)`	插入一个 `value`	$O(\log n)$	平均 $O(1)$，最坏 $O(n)$
`erase(value)`	删除所有 `value`	$O(\log n)$	平均 $O(1)$，最坏 $O(n)$
`erase(iter)`	删除迭代器 `iter` 指向的单个元素	$O(\log n)$	平均 $O(1)$，最坏 $O(n)$
`count(value)`	返回 `value` 的个数	$O(\log n)$	平均 $O(1)$，最坏 $O(n)$
`find(value)`	返回一个 `value` 的迭代器	$O(\log n)$	平均 $O(1)$，最坏 $O(n)$

> 精心设计的输入数据会使 `unordered_set` 始终保持最坏时间复杂度。

> `set` 不支持随机访问，因此对于 `set` 的两个迭代器 `p1` 和 `p2`，不能用 `p1 - p2` 计算它们的距离，必须使用 `std::distance(p1, p2)`，其时间复杂度是 $O(n)$。

示例

```cpp
#include <iostream>
#include <set>
```

```cpp
int main() {

    std::set<int> s;

    s.insert(1);
    s.insert(2);
    s.insert(3);
    // s = {1, 2, 3}

    s.insert(2);
    // s = {1, 2, 3}

    s.erase(2);
    // s = {1, 3}

    for (int value : s) {
        std::cout << value << ' ';
    }

    return 0;
}
```

```
1 3
```

map

映射。相当于 `[]` 内可填任意键值（可以是 `int`，`double` 等任意类型的对象）的数组。

> STL 提供 `unordered_map`，功能与 `map` 一致，但时间复杂度有所区别。

头文件

```cpp
#include <map>
```

定义与初始化

```cpp
std::map<int, int> m1;
```

```cpp
// m1 = {}

std::map<char, int> m2 = { {'a', 3}, {'b', 2}, {'c', 1} };
// m2 = { {'a', 3}, {'b', 2}, {'c', 1} }
```

> `map<T1, T2>` 相当于 `set<pair<T1, T2>>` 。实际上， `map` 内部存储的就是 `pair` 。

成员函数

函数名	功能	时间复杂度	时间复杂度（`unordered`）
`at(key)`	返回 `key` 映射的元素	$O(\log n)$	平均 $O(1)$，最坏 $O(n)$
`operator [key]`	返回 `key` 映射的元素（如没有则创建）	$O(\log n)$	平均 $O(1)$，最坏 $O(n)$
`begin()`	返回头部迭代器	$O(1)$	$O(1)$
`end()`	返回尾部迭代器	$O(1)$	$O(1)$
`empty()`	返回是否为空	$O(1)$	$O(1)$
`size()`	返回元素个数	$O(1)$	$O(1)$
`clear()`	清空	$O(n)$	$O(n)$
`count(key)`	返回 `key` 映射的元素个数	$O(\log n)$	平均 $O(1)$，最坏 $O(n)$
`erase(key)`	删除从 `key` 出发的映射	$O(\log n)$	平均 $O(1)$，最坏 $O(n)$

> 精心设计的输入数据会使 `unordered_map` 始终保持最坏时间复杂度。

样例

```cpp
#include <iostream>
#include <map>

int main() {
```

```
    std::map<char, int> m;

    m['b'] = 1;
    m['r'] = 2;
    m['o'] = 3;
    // m = { {'b', 1}, {'r', 2}, {'o', 3} }

    m.erase('o');
    // m = { {'b', 1}, {'r', 2} }

    for (auto el : m) {
        std::cout << el.first << ' ' << el.second << std::endl;
    }

    return 0;
}
```

```
b 1
r 2
```

string

字符串。相当于 `vector<char>` 的独立优化版本。

头文件

```
#include <string>
```

定义与初始化

```
std::string s1;
// s1 = ""

std::string s2 = "banana";
// s2 = "banana"

std::string s3(5, 'a');
// s3 = "aaaaa"
```

成员函数

函数名	功能	时间复杂度
`assign(count, value)`	初始化为 `count` 个 `value`	$O(n)$
`at(pos)`	返回第 `pos` 个字符	$O(1)$
`operator[pos]`	返回第 `pos` 个字符	$O(1)$
`front()`	返回第一个字符	$O(1)$
`back()`	返回最后一个字符	$O(1)$
`c_str()`	返回 c 风格字符串	$O(n)$
`begin()`	返回头部迭代器	$O(1)$
`end()`	返回尾部迭代器	$O(1)$
`empty()`	返回是否为空	$O(1)$
`size()`	返回字符串长度	$O(1)$
`clear()`	清空	$O(n)$
`push_back(ch)`	在尾部插入 `ch` 字符	$O(1)$
`pop_back()`	删除尾部字符	$O(1)$
`append(str)`	在尾部插入 `str` 字符串	$O(n)$
`operator += str`	在尾部插入 `str` 字符串	$O(n)$
`resize(count)`	将字符串长度调整为 `count`	$O(n)$
`substr(pos, count)`	截取第 `pos` 个字符开始、长度为 `count` 的子串	$O(n)$
`substr(pos)`	截取第 `pos` 个字符开始到末尾的子串	$O(n)$

非成员函数

函数名	功能	时间复杂度
`operator str1 + str2`	拼接字符串	$O(n)$
`operator str1 == str2`	等于	$O(n)$
`operator str1 < str2`	小于（按字典序比较）	$O(n)$
`operator str1 > str2`	大于（按字典序比较）	$O(n)$
`operator str1 <= str2`	小于等于（按字典序比较）	$O(n)$
`operator str1 >= str2`	大于等于（按字典序比较）	$O(n)$
`stoi(str)`	字符串转 `int`	$O(n)$
`stoll(str)`	字符串转 `long long`	$O(n)$
`stoull(str)`	字符串转 `unsigned long long`	$O(n)$
`stof(str)`	字符串转 `float`	$O(n)$
`stod(str)`	字符串转 `double`	$O(n)$
`stold(str)`	字符串转 `long double`	$O(n)$
`to_string(value)`	数字转字符串（支持 `int`，`double` 等）	$O(n)$

- `cin` 可以直接输入 `string`；
- `cout` 可以直接输出 `string`；
- `scanf` 不可以输入 `string`；
- `printf` 可以输出 `string` 的 c 风格形式，如 `printf("%s", str.c_str());`。

示例 1

```cpp
#include <iostream>
#include <string>

int main() {

    std::string str1 = "hello";
```

```
    std::string str2 = "world";

    std::string str3 = str1 + "," + str2;

    std::cout << str3 << std::endl;
    std::cout << str3.substr(2, 7) << std::endl;

    return 0;
}
```

```
hello,world
llo,wor
```

示例 2

```
#include <iostream>
#include <string>

int main() {

    int a = std::stoi("25");
    long long b = std::stoll("8");

    std::string c = std::to_string(a * b);

    std::cout << c << std::endl;

    return 0;
}
```

```
200
```

`sort`

给一个序列排序（默认为升序排序）。

支持数组，`vector`，`deque`，`string` 等支持随机访问的容器。

时间复杂度为 $O(n \log n)$。

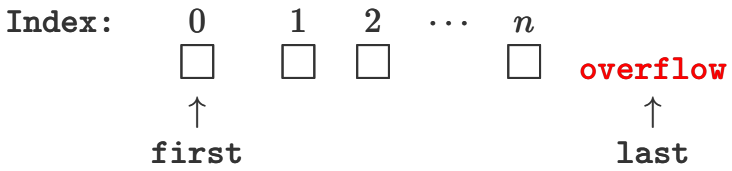
头文件

```
#include <algorithm>
```

函数签名

```
void sort(Iterator first, Iterator last);
void sort(Iterator first, Iterator last, Compare cmp);
```

- `first`：头部元素的迭代器（或指针）；
- `last`：尾部元素的后继的迭代器（或指针）；
- `cmp`（非必须）：自定义比较函数，用于控制排序的升降。



示例 1

```
#include <iostream>
#include <algorithm>

int main() {

    int a[] = {1, 1, 4, 5, 1, 4};

    std::sort(a, a + 6);

    for (int el : a) {
        std::cout << el << ' ';
    }

    return 0;
}
```

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

示例 2

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

int main() {

    std::vector<int> vec = {1, 1, 4, 5, 1, 4};

    std::sort(vec.begin(), vec.end());

    for (int el : vec) {
        std::cout << el << ' ';
    }

    return 0;
}
```

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

示例 3

```cpp
#include <iostream>
#include <algorithm>

int main() {

    int a[] = {1, 1, 4, 5, 1, 4};

    std::sort(a, a + 6, std::greater<int>());

    for (int el : a) {
        std::cout << el << ' ';
    }

    return 0;
}
```

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

示例 4

```cpp
#include <iostream>
#include <algorithm>

bool cmp(int l, int r) {
    return l > r;
}

int main() {

    int a[] = {1, 1, 4, 5, 1, 4};

    std::sort(a, a + 6, cmp);

    for (int el : a) {
        std::cout << el << ' ';
    }

    return 0;
}
```

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

`reverse`

反转一个序列。

支持数组，`vector`，`deque`，`list`，`string` 等支持顺序访问的容器。

时间复杂度为 $O(n)$。

头文件

```cpp
#include <algorithm>
```

函数签名

```
void reverse(Iterator first, Iterator last);
```

- `first`：头部元素的迭代器（或指针）；
- `last`：尾部元素的后继的迭代器（或指针）。

示例 1

```
#include <iostream>
#include <algorithm>

int main() {

    int a[] = {1, 1, 4, 5, 1, 4};

    std::reverse(a, a + 6);

    for (int el : a) {
        std::cout << el << ' ';
    }

    return 0;
}
```

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

示例 2

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {

    std::vector<int> vec = {1, 1, 4, 5, 1, 4};

    std::reverse(vec.begin(), vec.end());

    for (int el : vec) {
```

```
        std::cout << el << ' ';
    }

    return 0;
}
```

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

示例 3

```
#include <iostream>
#include <string>
#include <algorithm>

int main() {

    std::string s = "!dlroW ,olleH";

    std::reverse(s.begin(), s.end());

    std::cout << s;

    return 0;
}
```

```
Hello, World!
```

`unique`

移除一个序列中 **连续重复** 的元素。`unique` 不会直接删除元素，而是将重复的元素移动到序列末尾，并返回指向新序列末尾的迭代器。

支持数组，`vector`，`deque`，`list`，`string` 等支持顺序访问的容器。

时间复杂度为 $O(n)$。

函数签名

```
Iterator unique(Iterator first, Iterator last);
```

- `first`：头部元素的迭代器（或指针）；
- `last`：尾部元素的后继的迭代器（或指针）。

示例 1

```
#include <iostream>
#include <algorithm>

int main() {

    int a[] = {1, 1, 2, 2, 1, 1, 1};

    std::unique(a, a + 7);

    for (int el : a) {
        std::cout << el << ' ';
    }

    return 0;
}
```

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

示例 2

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {

    std::vector<int> vec = {1, 1, 2, 2, 1, 1, 1};

    auto pos = std::unique(vec.begin(), vec.end());

    vec.erase(pos, vec.end());
```

```
    for (int el : vec) {
        std::cout << el << ' ';
    }

    return 0;
}
```

```
1 2 1
```

lower_bound

在 **有序** 序列中二分查找「第一个大于等于给定值」的元素，并返回它的迭代器。

支持数组，`vector`，`deque`，`string` 等支持随机访问的容器。

时间复杂度为 $O(\log n)$。

> 在使用 `lower_bound` 之前，序列必须是升序的。

> STL 提供 `upper_bound`，用于查找「第一个大于给定值」的元素，用法与 `lower_bound` 相同。

函数签名

```
Iterator lower_bound(Iterator first, Iterator last, T value);
```

- `first`：头部元素的迭代器（或指针）；
- `last`：尾部元素的后继的迭代器（或指针）；
- `value`：用户的给定值。

示例

```
#include <iostream>
#include <algorithm>
#include <vector>
```

```cpp
int main() {

    std::vector<int> vec = {1, 2, 4, 4, 5, 6, 7};

    auto it = std::lower_bound(vec.begin(), vec.end(), 4);
    // 查找第一个 ≥ 4 的元素

    std::cout << std::distance(vec.begin(), it);
    // 输出它对应的下标

    return 0;
}
```

```
2
```