C++ STL 学习笔记

for 循环的四种用法

```
int nArray[] = {0, 1, 2, 3, 4, 5};
vector<int> vec(nArray, nArray + 6);
```

1. **用下标**

```
for (int i = 0; i < vec.size(); ++i)
        cout << vec[i] << " ";
```

2. **用迭代器**

```
for (auto it = vec.begin(); it != vec.end(); ++it)
        cout << (*it) << " ";

for (vector<int>::iterator it = vec.begin(); it != vec.end(); ++it)
        cout << (*it) << " ";
```

3. **STL函数**

```
// [](int item) { cout << item << " "; } 为lambda表达式，也就是说for_each的第三个参数为一个函数
for_each(vec.begin(), vec.end(), [](int item) { cout << item << " "; });
```

   - lambda 表达式
     - 基本语法

```
//直接调用
[] { cout << "hello lambda1" << endl; }();

//传递给对象
auto l = [] { cout << "hello lambda2" << endl;
};
l();
```

- Lambda 可以返回某物。但不需要指明返回类型

```
int a = [] { return 42; }();
```

- 方括号内，可以指明一个*capture*用来处理外部作用域内未被传递为实参的数据

```
int x = 1, y = 42;
auto q = [x, &y] {
    y++;
    cout << "x: " << x << endl;
    cout << "y: " << y << endl;
};
x = y = 77;
q();//输出结果 x：1 y：78
////////////////////////////////////
int x = 1, y = 42;
x = y = 77;
auto q = [x, &y] {
    y++;
    cout << "x: " << x << endl;
    cout << "y: " << y << endl;
};
q();//输出结果 x：77 y：78
```

4. **新增特性**

```
for (int item : vec)
        cout << item << " ";
```

常用函数(#include <Algorithom>)

1. **sort(start, end, cmp)**
   - 第一个参数是要排序的数组的起始地址。
   - 第二个参数是结束的地址。
   - 第三个参数是排序的方法（可以不写，默认从小到大）
     - stl内置了几个cmp，一个是greater<int>()，一个是less/()

```
sort(a, a + 5, greater<int>());
sort(a, a + 5, less<int>());
```

     - 也可以自己定义cmp

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

2. **lower_bound(start, end, val, cmp)**
   - 第四个参数是比较方法，可以省略
   - **注意：使用lower_bound()必须提前排序。**
   - 在 **[start, end)** 区域内查找不小于(>=) value 的元素

```
int a[6] = {1, 3, 5, 7, 9, 11};
int* i = lower_bound(a, a + 6, 10);
cout << "数值为: " << (*i);   // 11
cout << "下表为: " << i - a;  // 5
```

   - 在 **[start, end)** 区域内查找第一个不符合 cmp 规则的元素

```
int a[6] = {1, 3, 5, 7, 9, 11};
int* i = lower_bound(a, a + 6, 6, [](int x, int y) { return x <= y; });
cout << "数值为: " << (*i);   // 7
cout << "下表为: " << i - a;  // 3
```

   - 返回值：如果找到返回找到元素的地址否则返回end的地址。(这里注意有可能越界)

```
int a[6] = {1, 3, 5, 7, 9, 11};
int* i = lower_bound(a, a + 6, 12);
cout << "数值为：" << (*i); // 鏁板€间负锛?
```

3. **upper_bound(start, end, val, cmp)**
   - 与lower_bound()同理，可以理解为；upper_bound()是>，而lower_bound是>=

```
int a[6] = {1, 3, 5, 7, 9, 11};
int* i = upper_bound(a, a + 6, 7);
cout << "数值为：" << (*i) << endl;  // 9
cout << "下表为：" << i - a;          // 4
```

4. **next_permutation(start, end, cmp)**
   - 第四个参数是比较方法，可以省略
   - 求一个排序的后面排列的函数

```
int a[3] = {1, 3, 2};
do {
    for (int x : a) {
        cout << x << " ";
    }
    cout << endl;
} while (next_permutation(a, a + 3));
/* 1 3 2
2 1 3
2 3 1
3 1 2
3 2 1*（没有1 2 3）/
```

   - cmp指定排序方法

```
bool Compare(int x, int y) {
    if (x > y)
        return true;
    else
        return false;
}

int a[3] = {1, 3, 2};
```

```
do {
    for (int x : a) {
        cout << x << " ";
    }
    cout << endl;
} while (next_permutation(a, a + 3,Compare));
/* 1 3 2
1 2 3*/
```

5. **prev_permutation(start, end ,cmp)**
    - 和 next_permutation 函数一样，只是求一个排序的前面排列的函数
6. **unique(start, end)**
    - 使用该函数前，一定要先对序列进行排序
    - unique()将不重复的元素放到容器的前面，返回值是去重之后的尾地址。

```
int a[4] = {1, 2, 3, 3};
int k = unique(a, a + 4)-a;
for (int i = 0; i < k;i++){
    cout << a[i];
}// 123
```

常用容器

1. **string**

   常用方法：
    - size()、length()——返回字符串长度
    - empty()——判断字符串是否为空
    - clear()——清空字符串
    - substr(start, size)——返回子串
        - start 是要获取的子串的起始地址，size是要获取的长度

```
string a = "abcd";
string b = a.substr(1, 2);
cout << b;
//bc
```

- c_str()——返回字符串所在字符数组的起始地址

```
string a = "abcd";
const char* b = a.c_str();
cout << b;
```

2. **queue<T>（先进先出）**

   常用方法：

   - size()——返回队列大小
   - empty()——判断队列是否为空
   - push()——放入元素
   - **front()——返回队头元素**
   - back()——返回队尾元素
   - pop()——弹出队头元素，没有返回值

3. **priority_queue<T>（优先队列）**

   定义：priority_queue<Type, Container, Functional>

   - Type是存储数据的类型
   - Container是存储数据的容器的类型（默认为vctor）
   - Functional是比较的方法（要用仿函数的形式实现，默认为less（升序排列））
     - 仿函数不是函数，它是一个类
     - 仿函数重载了()运算符，使得它的对你可以像函数那样子调用

```
class Cmp {
    public:
     bool operator()(int x, int y) { return x > y; }
};

Cmp cmp;
cout << cmp(1, 2);//0
```

常用方法：

- 自定义排序

  priority_queue要自定义排序有两种方法

  - 一是直接将 < 运算符重载

```
struct node {
    //pair将一对值(T1和T2)组合成一个值
    //两个值可以分别用pair的两个公有函数first和second访问。
    pair<int, int> a;
    //注意这里的const一定要带上，巨坑！！
    bool operator<(node b) const {
        if (a.first == b.a.first) {
            return a.second < b.a.second;
        } else {
            return a.first < b.a.second;
        }
    }
};

priority_queue<node> q;
for (int i = 0; i < 5; i++) {
    pair<int, int> x = {i + 1, i - 1};
    node x1 = {x};
    q.push(x1);
}
while (!q.empty()) {
    cout << q.top().a.first << ' ' << q.top().a.second << endl;
    q.pop();
}
/*5 3
3 1
2 0
4 2
1 -1*/
```

  - 二是重写仿函数

```
struct Cmp {
```

```
    bool operator()(pair<int, int> a, pair<int, int> b) {
        if (a.first == b.first)
            return a.second > b.second;
        else
            return a.first > b.first;
    }
};

priority_queue<pair<int, int>> q;
for (int i = 0; i < 5; i++) {
    q.push(pair<int, int>{i + 1, i - 1});
}
while (!q.empty()) {
    cout << q.top().first << ' ' << q.top().second << endl;
    q.pop();
}
/*5 3
4 2
3 1
2 0
1 -1*/
```

- push()——插入元素
- top()——返回堆顶元素
- **pop()——弹出堆顶元素（注意区别于queue的front()）**

4. **stack（先进后出）**

   常用方法：

   - size()——返回长度
   - empty()——判空
   - push()——添加元素
   - pop()——弹出栈顶元素，无返回值
   - top()——返回栈顶元素

5. **map**

   - 使用[]进行插入

```
map<char, int> a;
a['a'] = 1;
```

- insert(pos, value)

```
map<char, int> a;
a.insert({'a', 1});
```

- at(key)——返回key的value，如果没有的话报错

```
map<char, int> a;
a.insert({'a', 1});
cout << a.at('a');//1
//cout << a['a']; 如果没有的话不会报错，会返回0
```

- eraze(key)——删除指定键值对，删除成功返回1，否则0

```
map<char, int> a;
a.insert({'a', 1});
cout << a.erase('a') << endl;
cout << a['a'] << endl;//1 0
```

6. **unordered_map**
   - map是有序的，但是unordered_map是无序的
   - 内置哈希，效率很高。
7. **vector**

   支持数组形式直接访问

   常用函数：
   - size()
   - empty()
   - clear()
   - front()
   - back()
   - push_back()
   - pop_back()
   - begin()
   - end()