C++ 标准模板库 (STL, Standard Template Library)：包含一些常用数据结构与算法的模板的 C++ 软件库。其包含四个组件——算法 (Algorithms)、容器 (Containers)、仿函数 (Functors)、迭代器 (Iterators).

示例：

- 算法： `sort(a.begin(), a.end())`
- 容器： `priority_queue<int> pque`
- 仿函数： `greater<int>()`
- 迭代器： `vector<int>::iterator it = a.begin()`

1 前言

STL 作为一个封装良好，性能合格的 C++ 标准库，在算法竞赛中运用极其常见。灵活且正确使用 STL 可以节省非常多解题时间，这一点不仅是由于可以直接调用，还是因为它封装良好，可以让代码的可读性变高，解题思路更清晰，调试过程 往往 更顺利。

不过 STL 毕竟使用了很多复杂的结构来实现丰富的功能，它的效率往往是比不上自己手搓针对特定题目的数据结构与算法的。因此，STL 的使用相当于使用更长的运行时间换取更高的编程效率。因此，在实际比赛中要权衡 STL 的利弊，不过这一点就得靠经验了。

接下来，我会分享在算法竞赛中常用的 STL 容器和算法，对于函数和迭代器，就不着重展开讲了。

2 常用容器

2.1 内容总览

打勾的是本次将会详细讲解的，加粗的是算法竞赛中有必要学习的。

- 顺序容器
  - [ ] **array**
  - [x] **vector**
  - [ ] **deque**
  - [ ] forward_list
  - [ ] **list**
- 关联容器
  - [x] **set**
  - [x] **map**
  - [ ] **multiset**

  - [ ] **multimap**
- 无序关联容器
  - [ ] **unordered_set**
  - [ ] **unordered_map**
  - [ ] **unordered_multiset**
  - [ ] **unordered_multimap**
- 容器适配器
  - [x] **stack**
  - [x] **queue**
  - [x] **priority_queue**
  - [ ] flat_set
  - [ ] flat_map
  - [ ] flat_multiset
  - [ ] flat_multimap
- 字符串
  - [x] **string** (basic_string<char>)
- 对与元组
  - [x] **pair**
  - [ ] **tuple**

2.2 向量 vector

```
#include <vector>
```

连续的顺序的储存结构（和数组一样的类别），但是有长度可变的特性。

2.2.1 常用方法

构造

```
vector<类型> arr(长度, [初值])
```

时间复杂度：$O(n)$

常用的一维和二维数组构造示例，高维也是一样的（就是会有点长）.

```
vector<int> arr;         // 构造int数组
vector<int> arr(100);    // 构造初始长100的int数组
vector<int> arr(100, 1); // 构造初始长100的int数组，初值为1

vector<vector<int>> mat(100, vector<int> ());       // 构造初始100行，不指定列数的二维数组
vector<vector<int>> mat(100, vector<int> (666, -1)) // 构造初始100行，初始666列的二维数组，初值为-1
```

构造二维数组的奇葩写法，千万别用：

```
vector<int> arr[100];         // 正确，构造初始100行，不指定列数的二维数组，可用于链式前向星存图
vector<int> arr[100](100, 1); // 语法错误！
vector<int> arr(100, 1)[100]; // 语法错误！
vector<int> arr[100] {{100, 1}, 这里省略98个 ,{100, 1}}; // 正确但奇葩，使用列表初始化
```

尾接 & 尾删

- **`.push_back(`**元素**`)`**：在 vector 尾接一个元素，数组长度 $+1$.
- **`.pop_back()`**：删除 vector 尾部的一个元素，数组长度 -1

时间复杂度：均摊 $O(1)$

```
// init: arr = []
arr.push_back(1);
// after: arr = [1]
arr.push_back(2);
// after: arr = [1, 2]
arr.pop_back();
// after: arr = [1]
arr.pop_back();
// after: arr = []
```

中括号运算符

和一般数组一样的作用

时间复杂度：$O(1)$

获取长度

`.size()`

获取当前 vector 的长度

时间复杂度：$O(1)$

```
for (int i = 0; i < arr.size(); i++)
    cout << a[i] << endl;
```

清空

`.clear()`

清空 vector

时间复杂度：$O(n)$

判空

`.empty()`

如果是空返回 `true` 反之返回 `false` .

时间复杂度：$O(1)$

改变长度

`.resize(`新长度`, [`默认值`])`

修改 vector 的长度

- 如果是缩短，则删除多余的值
- 如果是扩大，且指定了默认值，则新元素均为默认值**（旧元素不变）**

时间复杂度：$O(n)$

2.2.2 适用情形

一般情况 `vector` 可以替换掉普通数组，除非该题卡常。

有些情况普通数组没法解决：$n \times m$ 的矩阵，$1 \leq n, m \leq 10^6$ 且 $n \times m \leq 10^6$

- 如果用普通数组 `int mat[1000010][1000010]` ，浪费内存，会导致 MLE。
- 如果使用 `vector<vector<int>> mat(n + 10, vector<int> (m + 10))` ，完美解决该问题。

另外， `vector` 的数据储存在堆空间中，不会爆栈。

2.2.3 注意事项

提前指定长度

如果长度已经确定，那么应当直接在构造函数指定长度，而不是一个一个 `.push_back()`. 因为 `vector` 额外内存耗尽后的重分配是有时间开销的，直接指定长度就不会出现重分配了。

```
// 优化前: 522ms
vector<int> a;
for (int i = 0; i < 1e8; i++)
    a.push_back(i);
// 优化后: 259ms
vector<int> a(1e8);
for (int i = 0; i < a.size(); i++)
    a[i] = i;
```

当心 size_t 溢出

vector 获取长度的方法 `.size()` 返回值类型为 `size_t`，通常 OJ 平台使用的是 32 位编译器（有些平台例如 cf 可选 64 位），那么该类型范围为 $[0, 2^{32})$.

```
vector<int> a(65536);
long long a = a.size() * a.size(); // 直接溢出变成0了
```

2.3 栈 stack

```
#include <stack>
```

通过二次封装双端队列 (deque) 容器，实现先进后出的栈数据结构。

2.3.1 常用方法

作用	用法	示例
构造	`stack<类型> stk`	`stack<int> stk;`
进栈	`.push(元素)`	`stk.push(1);`
出栈	`.pop()`	`stk.pop();`
取栈顶	`.top()`	`int a = stk.top();`
查看大小 / 清空 / 判空	略	略

2.3.2 适用情形

如果不卡常的话，就可以直接用它而不需要手写栈了。

另外，vector 也可以当栈用，vector 的 `.back()` 取尾部元素，就相当于取栈顶，`.push_back()` 相当于进栈，`.pop_back()` 相当于出栈。

2.3.3 注意事项

不可访问内部元素！**下面都是错误用法**

```
for (int i = 0; i < stk.size(); i++)
    cout << stk[i] << endl;
for (auto ele : stk)
    cout << stk << endl;
```

2.4 队列 queue

```
#include <queue>
```

通过二次封装双端队列 (deque) 容器，实现先进先出的队列数据结构。

2.4.1 常用方法

作用	用法	示例
构造	`queue<类型> que`	`queue<int> que;`
进队	`.push(元素)`	`que.push(1);`
出队	`.pop()`	`que.pop();`
取队首	`.front()`	`int a = que.front();`
取队尾	`.back()`	`int a = que.back();`
查看大小 / 清空 / 判空	略	略

2.4.2 适用情形

如果不卡常的话，就可以直接用它而不需要手写队列了。

2.4.3 注意事项

不可访问内部元素！**下面都是错误用法**

```
for (int i = 0; i < que.size(); i++)
    cout << que[i] << endl;
for (auto ele : que)
    cout << ele << endl;
```

2.5 优先队列 priority_queue

```
#include <queue>
```

提供常数时间的最大元素查找，对数时间的插入与提取，底层原理是二叉堆。

2.5.1 常用方法

构造

priority_queue<类型，容器，比较器> pque

- 类型：要储存的数据类型
- 容器：储存数据的底层容器，默认为 vector<类型>，竞赛中保持默认即可
- 比较器：比较大小使用的比较器，默认为 less<类型>，可自定义

```
priority_queue<int> pque1;                              // 储存int的大顶堆
priority_queue<int, vector<int>, greater<int>> pque2; // 储存int的小顶堆
```

> 对于需要自定义比较器的情况，涉及一些初学时容易看迷糊的语法（重载小括号运算符 / lambda 表达式），在此就不展开讲了。如果想要了解，可以查阅 cppreference 中的代码示例。

其他

作用	用法	示例
进堆	.push(元素)	que.push(1);
出堆	.pop()	que.pop();
取堆顶	.top()	int a = que.top();
查看大小 / 判空	略	略

进出队复杂度 $O(\log n)$，取堆顶 $O(1)$.

2.5.2 适用情形

持续维护元素的有序性：每次向队列插入大小不定的元素，或者每次从队列里取出大小最小/最大的元素，元素数量 n，插入操作数量 k.

- 每次插入后进行快速排序：$k \cdot n \log n$
- 使用优先队列维护：$k \cdot \log n$

2.5.3 注意事项

仅堆顶可读

只可访问堆顶，其他元素都无法读取到。**下面是错误用法：**

```
cout << pque[1] << endl;
```

所有元素不可写

堆中所有元素是不可修改的。**下面是错误用法：**

```
pque[1] = 2;
pque.top() = 1;
```

如果你恰好要修改的是堆顶元素，那么是可以完成的：

```
int tp = pque.top();
pque.pop();
pque.push(tp + 1);
```

2.6 集合 set

```
#include <set>
```

提供对数时间的插入、删除、查找的集合数据结构。底层原理是红黑树。

集合三要素	解释	set	multiset	unordered_set
确定性	一个元素要么在集合中，要么不在	✓	✓	✓

集合三要素	解释	set	multiset	unordered_set
互异性	一个元素仅可以在集合中出现一次	✓	✗（任意次）	✓
无序性	集合中的元素是没有顺序的	✗（从小到大）	✗（从小到大）	✓

2.6.1 常用方法

构造

`set<类型, 比较器> st`

- 类型：要储存的数据类型
- 比较器：比较大小使用的比较器，默认为 `less<类型>`，可自定义

```
set<int> st1;                // 储存int的集合（从小到大）
set<int, greater<int>> st2; // 储存int的集合（从大到小）
```

> 对于需要自定义比较器的情况，涉及一些初学时容易看迷糊的语法（重载小括号运算符 / lambda 表达式），在此就不展开讲了。

遍历

可使用迭代器进行遍历：

```
for (set<int>::iterator it = st.begin(); it != st.end(); ++it)
    cout << *it << endl;
```

基于范围的循环（C++ 11）：

```
for (auto &ele : st)
    cout << ele << endl;
```

其他

作用	用法	示例
插入元素	`.insert(元素)`	`st.insert(1);`
删除元素	`.erase(元素)`	`st.erase(2);`

作用	用法	示例
查找元素	`.find(元素)`	`auto it = st.find(1);`
判断元素是否存在	`.count(元素)`	`st.count(3);`
查看大小 / 清空 / 判空	略	略

增删查时间复杂度均为 $O(\log n)$

2.6.2 适用情形

- 元素去重：$[1,1,3,2,4,4] \to [1,2,3,4]$
- 维护顺序：$[1,5,3,7,9] \to [1,3,5,7,9]$
- 元素是否出现过：元素大小 $[-10^{18},10^{18}]$，元素数量 10^6，vis 数组无法实现，通过 set 可以完成。

2.6.3 注意事项

不存在下标索引

set 虽说可遍历，但仅可使用迭代器进行遍历，它不存在下标这一概念，无法通过下标访问到数据。**下面是错误用法：**

```
cout << st[0] << endl;
```

元素只读

set 的迭代器取到的元素是只读的（因为是 const 迭代器），不可修改其值。如果要改，需要先 erase 再 insert. **下面是错误用法：**

```
cout << *st.begin() << endl; // 正确。可读。
*st.begin() = 1;             // 错误！不可写！
```

不可用迭代器计算下标

set 的迭代器不能像 vector 一样相减得到下标。**下面是错误用法：**

```
auto it = st.find(2);       // 正确，返回2所在位置的迭代器。
int idx = it - st.begin(); // 错误！不可相减得到下标。
```

2.7 映射 map

```
#include <map>
```

提供对数时间的有序键值对结构。底层原理是红黑树。

映射：

$$\begin{aligned} 1 &\to 2 \\ 2 &\to 2 \\ 3 &\to 1 \\ 4 &\to 5 \\ &\vdots \end{aligned}$$

性质	解释	map	multimap	unordered_map
互异性	一个键仅可以在映射中出现一次	✓	✗（任意次）	✓
无序性	键是没有顺序的	✗（从小到大）	✗（从小到大）	✓

2.7.1 常用方法

构造

`map<键类型, 值类型, 比较器> mp`

- 键类型：要储存键的数据类型
- 值类型：要储存值的数据类型
- 比较器：键比较大小使用的比较器，默认为 `less<类型>`，可自定义

```
map<int, int> mp1;                 // int->int 的映射（键从小到大）
map<int, int, greater<int>> st2; // int->int 的映射（键从大到小）
```

> 对于需要自定义比较器的情况，涉及一些初学时容易看迷糊的语法（重载小括号运算符 / lambda 表达式），在此就不展开讲了。

遍历

可使用迭代器进行遍历：

```cpp
for (map<int, int>::iterator it = mp.begin(); it != mp.end(); ++it)
    cout << it->first << ' ' << it->second << endl;
```

基于范围的循环（C++ 11）：

```cpp
for (auto &pr : mp)
    cout << pr.first << ' ' << pr.second << endl;
```

结构化绑定 + 基于范围的循环（C++17）：

```cpp
for (auto &[key, val] : mp)
    cout << key << ' ' << val << endl;
```

其他

作用	用法	示例
增 / 改 / 查元素	中括号	`mp[1] = 2;`
查元素（返回迭代器）	`.find(元素)`	`auto it = mp.find(1);`
删除元素	`.erase(元素)`	`mp.erase(2);`
判断元素是否存在	`.count(元素)`	`mp.count(3);`
查看大小 / 清空 / 判空	略	略

增删改查时间复杂度均为 $O(\log n)$

2.7.2 适用情形

需要维护映射的场景可以使用：输入若干字符串，统计每种字符串的出现次数。（`map<string, int> mp`）

2.7.3 注意事项

中括号访问时默认值

如果使用中括号访问 map 时对应的键不存在，那么会新增这个键，并且值为默认值，因此中括号会影响键的存在性。

```
map<char, int> mp;
cout << mp.count('a') << endl; // 0
mp['a'];                       // 即使什么都没做，此时mp['a']=0已经插入了
cout << mp.count('a') << endl; // 1
cout << mp['a'] << endl;       // 0
```

不可用迭代器计算下标

map 的迭代器不能像 vector 一样相减得到下标。**下面是错误用法：**

```
auto it = mp.find('a');      // 正确，返回2所在位置的迭代器。
int idx = it - mp.begin();   // 错误！不可相减得到下标。
```

2.8 字符串 string

```
#include <string>
```

顾名思义，就是储存字符串的。

2.8.1 常用方法

构造

构造函数： `string(长度, 初值)`

```
string s1;           // 构造字符串，为空
string s2 = "awa!";  // 构造字符串，并赋值awa!
string s3(10, '6');  // 构造字符串，通过构造函数构造为6666666666
```

输入输出

C++

```
string s;
cin >> s;
cout << s;
```

C

```
string s;
char buf[100];
scanf("%s", &buf);
s = buf;
printf("%s", s.c_str());
```

其他

作用	用法	示例
修改、查询指定下标字符	`[]`	`s[1] = 'a';`
是否相同	`==`	`if (s1 == s2) ...`
字符串连接	`+`	`string s = s1 + s2;`
尾接字符串	`+=`	`s += "awa";`
取子串	`.substr(起始下标, 子串长度)`	`string sub = s.substr(2, 10);`
查找字符串	`.find(字符串, 起始下标)`	`int pos = s.find("awa");`

数值与字符串互转（C++11）

源	目的	函数
int / long long / float / double / long double	string	to_string()
string	int	stoi()
string	long long	stoll()
string	float	stof()
string	double	stod()
string	long double	stold()

2.8.2 适用情形

非常好用！~~建议直接把字符数组扔了，赶快投入 string 的怀抱。~~

2.8.3 注意事项

尾接字符串一定要用 `+=`

string 的 += 运算符，将会在原字符串原地尾接字符串。而 + 了再 = 赋值，会先生成一个临时变量，在复制给 string.

通常字符串长度可以很长，如果使用 + 字符串很容易就 TLE 了。

```
// 优化前：15139ms
string s;
for (int i = 0; i < 5e5; i++)
    s = s + "a";

// 优化后：< 1ms (计时器显示0)
string s;
for (int i = 0; i < 5e5; i++)
    s += "a";
```

`.substr()` 方法的奇葩参数

一定要注意，C++ string 的取子串的第一个参数是**子串起点下标**，第二个参数是**子串长度**。

第二个参数不是子串终点！不是子串终点！要与 java 等其他语言区分开来。

`.find()` 方法的复杂度

该方法实现为暴力实现，时间复杂度为 $O(n^2)$.

~~不要幻想 STL 内置了个 $O(n)$ 的 KMP 算法~~

2.9 二元组 pair

```
#include <utility>
```

顾名思义，就是储存二元组的。

2.9.1 常用方法

构造

`pair<第一个值类型，第二个值类型> pr`

- 第一个值类型：要储存的第一个值的数据类型

- 第二个值类型：要储存的第二个值的数据类型

```
pair<int, int> p1;
pair<int, long long> p2;
pair<char, int> p3;
// ...
```

赋值

老式

```
pair<int, char> pr = make_pair(1, 'a');
```

列表构造 C++11

```
pair<int, char> pr = {1, 'a'};
```

取值

直接取值

- 取第一个值： `.first`
- 取第二个值： `.second`

```
pair<int, char> pr = {1, 'a'};
int awa = pr.first;
char bwb = pr.second;
```

结构化绑定 C++17

```
pair<int, char> pr = {1, 'a'};
auto &[awa, bwb] = pr;
```

判同

直接用 `==` 运算符

```
pair<int, int> p1 = {1, 2};
pair<int, int> p2 = {1, 3};
if (p1 == p2) { ... } // false
```

2.9.2 适用场景

所有需要二元组的场景均可使用，效率和自己定义结构体差不多。

2.9.3 注意事项

无

3 迭代器简介

3.1 迭代器是什么?

不搞抽象，直接举例。

对于一个 vector，我们可以用下标遍历：

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

我们同时也可以用迭代器来遍历：

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

- `a.begin()` 是一个迭代器，指向的是第一个元素
- `a.end()` 是一个迭代器，指向的是最后一个元素**再后面一位**
- 上述迭代器具有自增运算符，自增则迭代器向下一个元素移动
- 迭代器与指针相似，如果对它使用解引用运算符，即 `*it` ，就能取到对应值了

3.2 为何需要迭代器?

很多数据结构并不是线性的（例如红黑树），对于非线性数据结构，下标是无意义的。无法使用下标来遍历整个数据结构。

迭代器的作用就是定义某个数据结构的遍历方式，通过迭代器的增减，代表遍历到的位置，通过迭代器便能成功遍历非线性结构了。

例如，set 的实现是红黑树，我们是没法用下标来访问元素的。但是通过迭代器，我们就能遍历 set 中的元素了：

```
for (set<int>::iterator it = st.begin(); it != st.end(); ++it)
    cout << *it << endl;
```

3.3 迭代器用法

对于 vector 容器，它的迭代器功能比较完整，以它举例：

- `.begin()`：头迭代器
- `.end()`：尾迭代器
- `.rbegin()`：反向头迭代器
- `.rend()`：反向尾迭代器
- 迭代器 `+` 整型：将迭代器向后移动
- 迭代器 `-` 整型：将迭代器向前移动
- 迭代器 `++`：将迭代器向后移动 1 位
- 迭代器 `--`：将迭代器向前移动 1 位
- 迭代器 `-` 迭代器：两个迭代器的距离
- `prev(it)`：返回 it 的前一个迭代器
- `next(it)`：返回 it 的后一个迭代器

对于其他容器，由于其结构特性，上面的功能不一定都有（例如 set 的迭代器是不能相减求距离的）

3.4 常见问题

`.end()` 和 `.rend()` 指向的位置是无意义的值

对于一个长度为 10 的数组：`for (int i = 0; i < 10; i++)`，第 10 位是不可访问的

对于一个长度为 10 的容器：`for (auto it = a.begin(); it != a.end(); ++it)`，.end 是不可访问的

不同容器的迭代器功能可能不一样

迭代器细化的话有正向、反向、双向，每个容器的迭代器支持的运算符也可能不同，因此不同容器的迭代器细节很有可能是不一样的。

删除操作时需要警惕

为什么 3 没删掉?

```cpp
vector<int> a{1, 2, 3, 4};
for (auto it = a.begin(); it != a.end(); ++it)
    if (*it == 2 || *it == 3)
        a.erase(it);
// a = [1, 3, 4]
```

为啥 RE 了?

```cpp
vector<int> a{1, 2, 3, 4};
for (auto it = a.begin(); it != a.end(); ++it)
    if (*it == 4)
        a.erase(it);
```

建议：如无必要，别用迭代器操作容器。（遍历与访问没关系）

4 常用算法

4.1 内容总览

打勾的是本次将会详细讲解的，其他的是算法竞赛中建议学习的，不在下表列出的在比赛中基本用不到。

（很多函数的功能很简单，自己都能快速写出来，但是使用函数可以让代码可读性变得更高，这在比赛中是至关紧要的）

- 算法库 Algorithm
  - [ ] `count()`
  - [ ] `find()`
  - [ ] `fill()`
  - [x] `swap()`
  - [x] `reverse()`
  - [ ] `shuffle()` C++11
  - [x] `unique()`
  - [x] `sort()`
  - [x] `lower_bound()` / `upper_bound()`
  - [x] `max()` / `min()`
  - [ ] `max_element()` / `min_element()`
  - [ ] `prev_permutation()` / `next_permutation()`
- 数学函数 cmath

- [x] abs()
- [x] exp()
- [x] log() / log10() / log2()
- [x] pow()
- [x] sqrt()
- [ ] sin() / cos() / tan()
- [ ] asin() / acos() / atan()
- [ ] sinh() / cosh() / tanh()
- [ ] asinh() / acosh() / atanh() C++11
- [x] ceil() / floor()
- [x] round() C++11

- 数值算法 numeric
  - [ ] iota() C++11
  - [ ] accumulate()
  - [x] gcd() C++17
  - [x] lcm() C++17
- 伪随机数生成 random
  - [ ] mt19937
  - [ ] random_device()

4.2 swap()

交换两个变量的值

用法示例

```
template< class T >
void swap( T& a, T& b );
```

```
int a = 0, b = 1;
swap(a, b);
// now a = 1, b = 0

int arr[10] {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
swap(arr[4], arr[6]);
// now arr = {0, 1, 2, 3, 6, 5, 4, 7, 8, 9}
```

注意事项

这个 swap 参数是引用的，不需要像 C 语言一样取地址。

4.3 sort()

使用快速排序给一个可迭代对象排序

用法示例

```
template< class RandomIt, class Compare >
void sort( RandomIt first, RandomIt last, Compare comp );
```

默认排序从小到大

```
vector<int> arr{1, 9, 1, 9, 8, 1, 0};
sort(arr.begin(), arr.end());
// arr = [0, 1, 1, 1, 8, 9, 9]
```

如果要从大到小，则需要传比较器进去。

```
vector<int> arr{1, 9, 1, 9, 8, 1, 0};
sort(arr.begin(), arr.end(), greater<int>());
// arr = [9, 9, 8, 1, 1, 1, 0]
```

如果需要完成特殊比较，则需要手写比较器。

比较器函数返回值是 bool 类型，传参是需要比较的两个元素。记我们定义的该比较操作为 $\star$：

- 若 $a \star b$，则比较器函数应当返回 `true`
- 若 $a \not\star b$，则比较器函数应当返回 `false`

注意：如果 $a = b$，比较器函数必须返回 `false`

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

int main()
{
    vector<pair<int, int>> arr{{1, 9}, {2, 9}, {8, 1}, {0, 0}};
        sort(arr.begin(), arr.end(), cmp);
    // arr = [(0, 0), (8, 1), (2, 9), (1, 9)]
}
```

4.4 lower_bound() / upper_bound()

在**已升序排序**的元素中，应用二分查找检索指定元素，返回对应元素迭代器位置。**找不到则返回尾迭代器。**

- `lower_bound()`：寻找 $\geq x$ 的第一个元素的位置
- `upper_bound()`：寻找 $> x$ 的第一个元素的位置

怎么找 $\leq x$ / $< x$ 的第一个元素呢?

- $> x$ 的第一个元素的前一个元素（如果有）便是 $\leq x$ 的第一个元素
- $\geq x$ 的第一个元素的前一个元素（如果有）便是 $< x$ 的第一个元素

返回的是迭代器，如何转成下标索引呢？减去头迭代器即可。

用法示例

```
template< class ForwardIt, class T >
ForwardIt lower_bound( ForwardIt first, ForwardIt last, const T& value );
```

```
vector<int> arr{0, 1, 1, 1, 8, 9, 9};
vector<int>::iterator it = lower_bound(arr.begin(), arr.end(), 7);
int idx = it - arr.begin();
// idx = 4
```

我们通常写成一行：

```
vector<int> arr{0, 1, 1, 1, 8, 9, 9};
idx = lower_bound(arr.begin(), arr.end(), 7) - arr.begin(); // 4
idx = lower_bound(arr.begin(), arr.end(), 8) - arr.begin(); // 4
idx = upper_bound(arr.begin(), arr.end(), 7) - arr.begin(); // 4
idx = upper_bound(arr.begin(), arr.end(), 8) - arr.begin(); // 5
```

4.5 reverse()

反转一个可迭代对象的元素顺序

用法示例

```
template< class BidirIt >
void reverse( BidirIt first, BidirIt last );
```

```
vector<int> arr(10);
iota(arr.begin(), arr.end(), 1);
// 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
reverse(arr.begin(), arr.end());
// 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
```

4.6 max() / min()

返回最大值 / 最小值的**数值**

用法示例

```
int mx = max(1, 2); // 2
int mn = min(1, 2); // 1
```

在 C++11 之后，可以使用列表构造语法传入一个列表，这样就能一次性给多个元素找最大值而不用套娃了：

```cpp
// Before C++11
int mx = max(max(1, 2), max(3, 4)); // 4
int mn = min(min(1, 2), min(3, 4)); // 1

// After C++11
int mx = max({1, 2, 3, 4}); // 4
int mn = min({1, 2, 3, 4}); // 1
```

4.7 unique()

消除数组的重复**相邻**元素，数组长度不变，但是有效数据缩短，返回的是有效数据位置的结尾迭代器。

例如：$[1,1,4,5,1,4] \to [1,4,5,1,4,\underline{?}]$，下划线位置为返回的迭代器指向。

```cpp
template< class ForwardIt >
ForwardIt unique( ForwardIt first, ForwardIt last );
```

用法示例

单独使用 unique 并不能达成去重效果，因为它只消除**相邻**的重复元素。但是如果序列有序，那么它就能去重了。

但是它去重后，序列尾部会产生一些无效数据：$[1,1,2,4,4,4,5] \to [1,2,4,5,\underline{?},?,?]$，为了删掉这些无效数据，我们需要结合 erase.

最终，给 vector 去重的写法便是：

```cpp
vector<int> arr{1, 2, 1, 4, 5, 4, 4};
sort(arr.begin(), arr.end());
arr.erase(unique(arr.begin(), arr.end()), arr.end());
```

4.8 数学函数

所有函数参数均支持 `int` / `long long` / `float` / `double` / `long double`

公式	示例
$f(x) = \|x\|$	`abs(-1.0)`
$f(x) = e^x$	`exp(2)`

公式	示例
$f(x) = \ln x$	`log(3)`
$f(x, y) = x^y$	`pow(2, 3)`
$f(x) = \sqrt{x}$	`sqrt(2)`
$f(x) = \lceil x \rceil$	`ceil(2.1)`
$f(x) = \lfloor x \rfloor$	`floor(2.1)`
$f(x) = \langle x \rangle$	`rount(2.1)`

注意事项

由于浮点误差，有些的数学函数的行为可能与预期不符，导致 WA。如果你的操作数都是整型，那么用下面的写法会更稳妥。

> 原文地址：https://codeforces.com/blog/entry/107717

- $\lfloor \frac{a}{b} \rfloor$
  - 别用：`floor(1.0 * a / b)`
  - 要用：`a / b`
- $\lceil \frac{a}{b} \rceil$
  - 别用：`ceil(1.0 * a / b)`
  - 要用：`(a + b - 1) / b` （$\lceil \frac{a}{b} \rceil = \lfloor \frac{a+b-1}{b} \rfloor$）
- $\lfloor \sqrt{a} \rfloor$
  - 别用：`(int) sqrt(a)`
  - 要用：二分查找 https://io.zouht.com/7.html
- a^b
  - 别用：`pow(a, b)`
  - 要用：快速幂 https://io.zouht.com/18.html
- $\lfloor \log_2 a \rfloor$
  - 别用：`log2(a)`
  - 要用：`__lg` （不规范，但是这是竞赛） / `bit_width` （C++20 可用）

4.9 `gcd()` / `lcm()`

（C++17）返回最大公因数 / 最小公倍数

```
int x = gcd(8, 12); // 4
int y = lcm(8, 12); // 24
```

如果不是 C++17，但是是 GNU 编译器（g++），那么可以用内置函数 `__gcd()`.

当然， `gcd` / `lcm` 函数也挺好写，直接写也行（欧几里得算法）：

```
int gcd(int a, int b)
{
    if (!b)
        return a;
    return gcd(b, a % b);
}

int lcm(int a, int b)
{
    return a / gcd(a, b) * b;
}
```