

c++与stl

回顾一下c++

struct与class

相比于c语言，c++新增了class这个东西。它和struct有什么区别呢？其实区别不大。struct也允许成员函数，运算符重载，继承，public/private/protected等作用域。

struct和class主要不同包括

- struct成员默认public，class成员默认private
- 继承时struct默认为public继承，class默认为private继承

在考场上，我们不用编写具有复杂面向对象特性的代码，因此不用考虑封装，继承，多态等问题；考场上的代码属于“一次性”代码，我们也不用担心全部public带来坏处。因此如果需要，我们会使用struct



c语言的struct和c++的struct有很大不同，c++的struct进行了很多面向对象的扩展

引用

有时候写c++可以逃避指针。如果我们要写一个函数，交换两个数的值，那么用c语言的方法是这样

C++

```
1 void swap(int *a, int *b){
2     int tmp = *a;
3     *a = *b;
4     *b = tmp;
5 }
6 int main(){
7     int a = 1, b = 2;
8     swap(&a, &b);
9 }
```

我们把定义函数时小括号里写的参数称为形参，调用函数时传入的参数为实参。函数实际调用时，相当于实参赋值给了形参。如果直接传入值而不是指针，那么只会交换两个形参，也就是只交换了实参的复制品，实参本身没有动。所以要传入指针，通过指针访问实际要交换的两个数，进行交换

c++的引用相当于指针的一层封装。如果我们这样定义一个变量b

C++

```
1 int a;  
2 int &b = a;
```

b就成为了a的一个别名，它们实际相当于一个变量。如果我改变b，a也会变化，反之亦然。因此交换函数在c++里可以这样写

C++

```
1 void swap(int &a, int &b){  
2     int tmp = a;  
3     a = b;  
4     b = tmp;  
5 }  
6 int main(){  
7     int a = 1, b = 2;  
8     swap(a, b);  
9 }
```

形参和实参就相当于一个东西了。

运算符重载

c++的基本类型可以通过各种运算符，比如小于号进行比较，也可以通过各种运算符，比如加号进行计算。但自定义的结构体就不行

C++

```
1 struct Point{  
2     int x, y;  
3     Point(int X = 0, int Y = 0): x(X), y(Y){}  
4 };  
5  
6 int main(){  
7     Point a(1, 2), b(3, 4);  
8     Point c = a + b; // 编译错误, no match for operator +  
9 }
```

这一点不能怪编译器，因为编译器也不知道使用者具体要干什么。当然可以写一个函数来代替运算符

C++

```
1 struct Point{
2     int x, y;
3     Point(int X = 0, int Y = 0): x(X), y(Y){}
4
5     Point add(const Point &p) const {
6         return Point(x + p.x, y + p.y);
7     }
8 };
9
10 int main(){
11     Point a(1, 2), b(3, 4);
12     Point c = a.add(b);
13 }
```

这样可以解决问题，但不太方便。如果把函数名换成 `operator +` 就可以重载加法运算了

C++

```
1 struct Point{
2     int x, y;
3     Point(int X = 0, int Y = 0): x(X), y(Y){}
4
5     Point operator + (const Point &p) const {
6         return Point(x + p.x, y + p.y);
7     }
8 };
9
10 int main(){
11     Point a(1, 2), b(3, 4);
12     Point c = a + b;
13 }
```

template

写一份代码，可以给多种类型使用

比如一个整数取最小值的函数

C++

```
1 int min_int(int a, int b){
2     if(a < b) return a;
3     else return b;
4 }
```

如果我们需要一个浮点数取最小值的函数

C++

```
1 double min_double(double a, double b){
2     if(a < b) return a;
3     else return b;
4 }
```

这两份代码几乎一模一样，只是参数类型和返回值类型不同，我们就要把代码写两遍。用template可以简化写法

C++

```
1 template <class T>
2 T min(T a, T b){
3     if(a < b) return a;
4     else return b;
5 }
```

这就是一个模板函数。我们使用了一个模板参数T，T可以代表任何类型。这样我们在使用时，编译器会自动根据传入的参数，判断T是什么类型，然后根据模板自动帮着写出一个实际的确定参数类型的函数，也就是把T换成实际的类型

C++

```
1 min(1, 2); // int 类型, 返回1
2 min(2.4, 3.5); // double 类型, 返回2.4
```

另外，还有模板类，定义方法和模板函数类似，但使用时我们需要指定模板参数，因为编译器的自动类型推导不足以推断出所有信息。比如

C++

```
1 vector<int> v;
2 // 定义一个vector
```

类名称后面会加一个尖括号，里面是模板参数。编译器会把模板参数中的内容替换进原文，自动得到一个完整定义的类。



实际进行程序设计竞赛过程中，我们不需要手动编写模板，只要做到熟练运用c++自带的模板即可

STL即Stand Template library，标准模板库，接下来我们将介绍c++的标准库，尤其是STL的知识

<bits/stdc++.h>

这是一个特殊的头文件，包含了c++标准库的所有头文件。

这个头文件在考场上使用非常方便

- 不需要记忆每个类或函数在哪个头文件了，只需要include这个头文件即可

但如果平时写一个不用于考试/竞赛的代码，最好不要使用它

- 不是c++标准，和平台相关。比如visual studio（不是visual studio code）的编译器就不支持它

另外一个缺点是引入了所有头文件，会让编译变慢。但考虑到考试环境和评测环境都是Ubuntu，使用g++编译，考场上使用它还是很方便的



windows下的mingw也支持这个头文件，毕竟mingw就是minimum gnu for windows的简写

pair与tuple

头文件：<utility>

pair（掌握）

当你需要一个结构体，里面有两个成员变量，可以这样写

C++

```
1 struct s{
2     int x;
3     double y;
4 };
```

通常来说这样没什么问题，但如果我们需要定义多个类似这样的结构体，并且每种结构体成员的类型还不一样，那或许就需要这样写

C++

```
1  struct s1{
2      int x;
3      double y;
4  };
5  struct s2{
6      string z;
7      char w;
8  };
9  struct s3{
10     double a;
11     double b;
12 };
```

写起来有点麻烦。并且有些时候，这些结构体只是临时用来把数据放在一起，结构体本身没什么特别明确的含义，结构体和成员的名字也不重要。为了免去定义这种小结构体的繁琐，c++里可以使用 **pair**

pair是一个类模板，有两个模板参数，分别代表两个成员的类型。pair的两个成员变量名字分别叫first和second，我们可以通过这两个名字来访问pair的内容。我们可以像这样定义一个pair类型的变量

C++

```
1  pair<int, double> p;
```

这个操作基本等价于

C++

```
1  // 先定义结构体s
2  struct s{
3      int first;
4      double second;
5  };
6
7  // 定义一个s类型的变量p
8  s p;
```

也可以定义一个数组

C++

```
1 pair<int, double> p[10];
2
3 // 输入p[2]的两个成员
4 cin >> p[2].first >> p[2].second;
```

定义时，我们也可以使用带两个参数的构造函数来直接指定first和second的值

当然，pair不止于此

- pair可以直接用 `=` 进行整体赋值，像一个普通结构体一样
- pair可以直接用小于号/大于号/等于号/不等号等符号来比较大小，当然前提是这两个pair类型相同，并且两个成员的类型都可以独自比较。pair会先比较first的大小，如果first不同，则直接返回first的比较结果；如果first相同，则返回second比较的结果

例如：

C++

```
1 pair<int, string> p1(2, "aab"), p2(3, "aaa"), p3(2, "aaa");
2 p1 < p2; // true 因为 2 < 3
3 p1 < p3; // false 因为 2 == 2 并且 "aab" > "aaa"
```

另外，可以使用 `make_pair` 函数构造一个pair并返回

C++

```
1 pair<int, int> a;
2 a = make_pair(3, 5);
```

在c++11后，可以通过大括号来直接构造，上面代码等价于

C++

```
1 pair<int, int> a;
2 a = {3, 5};
```

tuple（了解）

另外一个问题是，如果你需要不只有2个元素，比如三个或四个，应该怎么做

一个解决方法是pair嵌套，比如

C++

```
1 pair<int, pair<double, string> > p;  
2 cin >> p.first >> p.second.first >> p.second.second;
```

但这样写起来太繁琐，也不好看。c++提供了tuple，来实现这种功能。

我们可以像这样定义tuple

C++

```
1 tuple<int, double, string> p; // 三个成员的tuple  
2 tuple<int, int, int, int> q; // 四个成员的tuple  
3 // tuple 允许更多成员，不再举例
```

怎么访问成员变量呢？不可能像pair一样通过first和second来访问。c++提供了 `get<T>` 来访问。

如果我们需要访问p的第2个成员（从0开始算），那么这样写

C++

```
1 get<2>(p); // 访问p的第二个成员
```

这其实是一个函数模板，模板参数是访问的第几个成员，函数参数是要访问的tuple变量名。需要注意的是，由于是模板，模板参数那个数字**必须是常量**，这样编译器才能在编译期确定应该访问哪个成员。



模板实例化是编译阶段完成的，因此模板参数，也就是尖括号中的内容，必须是类型名或常量等可以在编译器确定的内容

tuple也可以用带参数的构造函数，在定义变量时直接指定各个成员的值

像pair一样，tuple也可以通过各种运算符比较大小，比较规则也和pair类似：从第0个元素开始比较，如果相等，就比较下一个元素，否则直接返回当前元素的比较结果

常用算法函数

在algorithm库中，有若干c++自带的函数，用来实现某些算法。

sort(掌握)

sort是用来排序的函数。时间复杂度为稳定的 $O(n\log n)$ 。但不保证稳定排序



稳定排序版本为stable_sort，用法和sort相同

基础用法

基础用法很简单，两个参数，分别传入待排序区间的首和尾，要求左闭右开，那么就会把区间内的数按照升序排好

C++

```
1 int a[9] = {1,9,2,8,3,7,4,6,5};
2 sort(a, a + 9);
3 // a: {1,2,3,4,5,6,7,8,9}
```

上面例子我们注意到，sort传入的两个参数分别是a和a+9。a很容易理解，这是数组名，也是数组的首地址。a+9代表数组结束的位置。也就是从a+9开始，包括a+9自己，后面都不是要排序的区间了。于是，sort函数会将a到a+9但不包含a+9的这个区间内的数进行排序。

降序排序

sort可以传入第3个参数，这个参数是一个函数，而这个函数代表用于排序的比较方式

这个函数要求，传入两个参数，分别是待排序类型的两个变量，返回一个bool值：当第一个变量应该排在第二个变量前面时，返回true，否则返回false

如果我们要降序排一个数组，那么这个函数可以这样写

C++

```
1 bool cmp(int a, int b){
2     if(a > b) return true;
3     else return false;
4 }
```

当然，考虑到a > b本身就是个布尔值，我们可以简化一下

C++

```
1 bool cmp(int a, int b){
2     return a > b;
3 }
```

接下来，在排序中，第三个参数写这个函数名即可

C++

```
1 int a[9] = {1,9,2,8,3,7,4,6,5};
2 sort(a, a + 9, cmp);
3 // a: {9,8,7,6,5,4,3,2,1}
```

或许我们会觉得这样写还是太繁琐，因为我们还要额外写个函数。其实，这个函数c++也提供了一个模板。我们可以使用greater<T>() 来得到同样效果

greater<T>是一个类模板，它重载了()小括号运算符，该运算符传入两个参数，均为T类型的常引用。如果第一个参数比第二个参数大，返回true，否则返回false。这个小括号运算符和我们刚才写的cmp功能几乎一样，于是我们可以直接这样写来完成降序排序

C++

```
1  sort(a, a + 9, greater<int>());
```

我们把cmp换成了greater<int>(), 代表我们这里传入的是greater<int>的小括号运算

结构体排序

如果你定义了一个结构体，它默认是不能进行比较的，所以如果你要排序的话，就需要指定一种比较方式。可以像排降序的方法一样，传入一个函数作为参数，这个函数描述比较方式。还有一种比较简便的方法，给结构体重载小于号，这样sort函数会默认调用小于号来比较，实现排序

我们现在有一个结构体s，我们希望按照a为第一关键字进行升序排序，b为第二关键字进行降序排序，c为第三关键字进行升序排序

C++

```
1  struct s{
2      int a, b, c;
3      bool operator < (const s &x) const {
4          //先按照a比较
5          if(a < x.a) return true;
6          if(a > x.a) return false;
7
8          //若a相同再按照b比较
9          if(b > x.b) return true;
10         if(b < x.b) return false;
11
12         //若ab均相同按照c比较
13         return c < x.c;
14     }
15 };
16
17 s arr[10];
18 /*
19     进行赋值
20 */
21 sort(arr, arr + 10); // 自动调用小于运算符，按照指定规则排序
```

二分相关（讲二分时会详细展开）

lower_bound

用法：传入3个参数，分别代表待查找区间的开始位置，结束位置（左闭右开区间），和待查找的值。返回一个位置，这个位置为第一个**大于等于**给定值的位置，如果不存在（所有数都小于给定值），则返回结束位置。

要求传入的区间必须为不降序（从前往后越来越大，允许相邻的相同数字）排列，否则不能返回正确结果

upper_bound

用法和lower_bound相同，但查找的是第一个**大于**给定值的位置

使用lower_bound和upper_bound的过程中，我们经常给返回值减去区间首地址，得到位置对应的下标

举例：

C++

```
1  int a[10] = {1,2,3,4,5,5,5,6,7,8}
2  lower_bound(a, a + 10, 5) - a; // 4
3  upper_bound(a, a + 10, 5) - a; // 6
```

reverse（了解）

反转指定区间，传入首地址和结束地址（左闭右开）即可。

其实你即使记不住这个函数也完全可以自己写一个循环来实现

min_element和max_element（了解）

传入区间首地址和结束地址（左闭右开），返回最大值/最小值的位置

同样是个记不住也不影响自己写的函数

unique(掌握)

去重函数，会把原先序列的相邻相同元素去掉。传入区间首地址和结束地址（左闭右开），返回去重后，新的数组结束地址

C++

```
1  int a[10] = {1,1,2,3,3,5,3,3,4,1};
2  unique(a, a + 10); //返回a + 7, a到a + 7区间内变成1,2,3,5,3,4,1
```

unique函数用于离散化非常方便，后续课程会详细讲解

next_permutation(了解)

当你需要生成一个序列的全排列时，相比于写递归搜索，这个方法要简单很多

传入区间首地址和结束地址（左闭右开），函数会给区间内元素重排为下一个排列的顺序。如果已经枚举完成没有下一个排列，那么返回false，否则返回true

需要注意的是，下一个排列是最小的字典序大于当前排列的排列。如果你想枚举所有的排列，你需要从字典序最小的排列开始，也就是全升序的排列开始。

通常用法如下：

C++

```
1 p[5] = {1,2,3,4,5};
2 do{
3     //在这里根据排列进行操作
4 }while(next_permutation(p, p + 5));
```

可以理解并尝试运行以下完整代码

C++

```
1 #include<bits/stdc++.h>
2 using namespace std;
3 const int n = 4;
4 int a[n];
5 int main(){
6     for(int i = 0; i < n; i++){
7         a[i] = i;
8     }
9     do{
10         for(int i = 0; i < n; i++){
11             cout << a[i] << ' ';
12         }
13         cout << '\n';
14     }while(next_permutation(a, a + n));
15     return 0;
16 }
```

顺序容器

vector（掌握）

头文件<vector>

vector是数组实现的线性表。你可能会想，vector有什么用，实际上在算法竞赛的大部分场合，确实可以用普通的数组代替。

先说它像一个普通数组的方面

定义

定义vector，需要指定元素类型，可以指定长度，默认为0，也可以设置数组的初始值。和普通数组不同的是，长度可以是变量，不一定是常量

C++

```
1 vector<int> a; // 定义一个vector, 类型int, 长度为0
2 vector<int> b(10); // 定义一个vector, 类型int, 长度为10
3 vector<int> c(10, 3); // 定义一个vector, 类型int, 长度为10, 并且初始值均为3
```

访问元素

可以通过方括号来访问元素，这一点和普通的数组一模一样。下标可以从0取到尺寸减一，当心别越界了

C++

```
1 vector a(10);
2 a[2] = 5;
3 // a[10] = 3; 错误, 越界, 最大可以访问到a[9]
4 a.back() = 4; // 将a中最后一个数改成4
```

改变长度

和定长数组不同的是，vector可以在末尾增加或删除元素，操作均为O(1)的时间复杂度

C++

```
1 vector<int> a(10);
2 a.push_back(3); // a的末尾增加一个元素3, 尺寸增大了1
3 a.pop_back(); // 把a末尾的元素删除。如果a本身长度为0会产生错误
```

也可以直接指定目标长度来实现长度变化，如果长度增加，新增的部分可以设置为指定值。时间复杂度为O(n)，其中n为增加的长度（如果长度减少则复杂度为O(1)）

C++

```
1 vector<int> a = {1,2,3,4,5,6,7,8,9}; // 使用初始化列表构造
2 a.resize(11); // a = {1,2,3,4,5,6,7,8,9,0,0};
3 a.resize(13, 5); // a = {1,2,3,4,5,6,7,8,9,0,0,5,5};
4 a.resize(7); // a = {1,2,3,4,5,6,7}
```

考虑到vector可以改变长度，c++提供了两个函数得到vector现在的尺寸和是否为空的情况

C++

```
1  int x = a.size(); // 返回一个整数，为a的长度
2  bool y = a.empty(); // 返回一个bool，若a长度为0则返回true，否则返回false
```



上面代码中的把x定义为int实际上不严谨，用size_t更加严谨，而现在的平台下size_t通常为unsigned long long的宏定义。但算法竞赛中，容器尺寸用int表示已经足够，使用有符号数也可以防止减去一个数后小于0溢出。所以还是用int吧

迭代器

我们先回顾一下什么是迭代器

迭代器，其实类似一个指针，只不过指针指向的是一个物理地址，而迭代器，是一个虚拟的结构，指向的数据结构中的一个位置。可以通过迭代器访问它所指的内容，也可以根据迭代器访问数据结构中相邻的内容，这一点类似指针的++，--，[]操作

vector的迭代器可以指向vector中的任意元素，除此之外还有个特殊的迭代器：end，它指向最后一个元素的下一个位置。

我们可能经常遇到需要给vector内的内容进行排序或者去重之类操作的情况。sort之类的函数需要传入区间的起始位置和结束位置，vector对应的成员函数是begin()和end()。这两个函数均会返回一个**迭代器**，即vector<T>::iterator类型，其中T为vector存储元素的类型。

vector的迭代器和指针几乎完全一样

a是一个vector。假如我们想给a中的数进行排序，可以这样

C++

```
1  sort(a.begin(), a.end());
```

前两个参数分别填写begin和end，第三个参数和在其他地方用sort相同，填一个用于比较的函数，就可以给整个vector排序。可以更自由地写，假如想给a中除了第0个元素，其他的元素进行排序

C++

```
1  sort(a.begin() + 1, a.end());
```

除此之外，begin和end还可以用来遍历



一个非常重要的事情是，vector改变长度后，原有的迭代器可能失效

遍历

最简单的方法，通过下标访问

C++

```
1 for(int i = 0; i < a.size(); i++){
2     // a[i] 为访问的元素
3 }
```

或者使用迭代器，从begin到end就是所有元素

C++

```
1 for(vector<int>::iterator i = a.begin(); i != a.end(); i++){
2     // *i 即为我们要访问的元素
3 }
```

上面一种写法太繁琐，如果使用c++ 11的新特性auto就会简单的多

C++

```
1 for(auto i = a.begin(); i != a.end(); i++){
2     // *i 即为我们要访问的元素
3     // i 的类型依然为vector<int>::iterator, 编译器可以根据它的初值自动推导出类型
4 }
```

c++ 11 同时引入了range for写法。如果不需要在遍历时得到下标的话，这是最简便的写法。

C++

```
1 for(int &i: a){ // int 可以替换为 auto
2     // i 本身就是访问元素的引用
3 }
```



Range for底层实际上也调用了迭代器，从begin访问到end

插入删除（了解）

vector支持在某个位置插入一个元素或删除一个元素。这个操作在算法竞赛中用得不多，其中一个原因是最坏复杂度为 $O(n)$ ，其中 n 为vector长度，这个复杂度通常来说还是比较高的

插入一个元素，需要传入插入位置和插入内容。其中插入位置参数是一个迭代器，代表以这个迭代器为分界线，将数组分成两部分：它前面为一部分，它以及它后面为另一部分；然后新元素将插入到这两部分之间



通过insert插入元素同样可能导致原有迭代器失效

C++

```
1 vector<int> a = {1,3,5,7,9}
2 a.insert(a.begin() + 2, 0); // a = {1,3,0,5,7,9}
```

删除一个元素，需要传入删除位置的迭代器，然后它后面的元素会自动前移补上空缺，vector的尺寸也会相应减少

C++

```
1 vector<int> a = {1,3,5,7,9}
2 a.erase(a.begin() + 2); // a = {1,3,7,9}
```

也可以删除一个区间内所有数

C++

```
1 vector<int> a = {1,2,3,4,5,6,7,8,9}
2 a.erase(a.begin() + 1, a.begin() + 4); // a = {1,5,6,7,8,9}
```

进阶操作（了解）

emplace与emplace_back

假如vector内存储的是一个结构体，你需要向它末尾插入一个新的值

C++

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

使用push_back可以解决问题，但效率略微不大好，因为传入的参数是个结构体，就代表传参时需要先构造一个临时变量，把这个结构体构造出来，再传入。使用emplace_back可以消除这个临时变量，而传入的参数不是构造好的结构体，而是结构体的构造函数参数

C++

```
1 vector<pair<int, int> > a;
2 a.emplace_back(1, 2); // 构造函数的参数为(1, 2)
```

insert函数也有同样的问题，对应的方法是emplace函数

C++

```
1 vector<pair<int, int> > a(5);
2 a.emplace(a.begin() + 3, 1, 2); // 在begin()+3的位置插入结构体，其中构造函数的参数为(1, 2)
```



对于其他容器，如set，map，queue，stack等，插入操作通常也有emplace版本，感兴趣的可以自己搜索资料查看用法

二维变长数组

如果vector每个元素存储的类型也为vector，那么这就是个二维数组，并且每一维都可以改变长度

C++

```
1 vector<vector<int> > a(n, vector<int>(m, 0)); // a是一个n*m初值为0的二维数组
```

好处与缺点

相比于普通的数组，好处是显而易见的

- 可以不定长。定义时长度可以为变量。普通数组定义时长度必须是常量（不考虑g++编译器支持VLA）；普通数组在函数内部定义时，长度不能太大，否则会栈溢出，但vector不会；如果使用new的方式通过指针构建数组，需要手动delete，但vector不用
- 可以变长。vector可以自由改变长度，如果只在末尾增加或删除一个元素，时间复杂度还很优秀，为常数级的

缺点也存在

- 效率不如普通数组高。所以在竞赛中尽量使用普通数组

list（了解）

list是c++提供的双向链表。不过由于竞赛中很少遇到需要链表的题目，而且遇到需要链表的题目，通常自己手动模拟一个链表效率更高也更便捷，于是list在竞赛中使用很少

如果想高效率使用list，必须要掌握迭代器的用法，否则链表将几乎毫无优势

声明

C++

```
1 list<int> l1;
```

改变长度操作

C++

```
1  l1.push_back(1); // 末尾增加一个元素
2  l1.push_front(1); // 最前面插入一个元素
3  l1.pop_front(); // 删除第一个元素
4  l1.pop_back(); // 删除最后一个元素
5  // 均为  $O(1)$  时间复杂度
6
7  list<int>::iterator it = l1.begin();
8  it++; // 迭代器移动
9  l1.insert(it, 1); // 在it之前插入一个新元素
10 it = l1.erase(it); // 删除it所在位置的元素，返回被删除元素的下一个元素的迭代器。原先迭代器由于指向位置被删除而失效
11 // 均为  $O(1)$  时间复杂度
12
13 l1.clear(); // 删除所有元素，时间复杂度为  $O(n)$ 
```

list的迭代器不支持随机访问，不能给它加减一个常数，只能通过++或--来移动迭代器

所以，一些函数如sort，就不能直接使用，但list自带一些成员函数

C++

```
1  l1.sort(); // 对l1的内容进行升序排列
2  l1.unique(); // 对l1的内容进行相邻去重
```

deque（了解）

头文件：<queue>

deque是双端队列，但实际上比队列功能要多。deque支持vector的几乎一切操作，同时还支持在 $O(1)$ 的复杂度内，在最前面插入或删除一个元素。

与vector不同的是，插入元素不会让deque的迭代器失效，不过副作用就是，deque底层存储的数据并不是完全连续的内存。deque底层结构同样会让deque的效率比不上vector和普通的数组

定义，访问元素等和vector相同，主要介绍push_front和pop_front，这个操作和list的又基本相同

C++

```
1  deque<int> a(10);
2  a.push_front(5); // 在a的开头插入一个5
3  a.pop_front(); // 删除a开头的数
```

string（掌握）

头文件：<string>



注意和<string.h>与<cstring>区分开。<string.h>与<cstring>里面都是对字符数组进行的操作

String 是字符串。在c语言中，字符串是字符数组，而不是像int，double是一个基础类型，使用起来处处受限。c++提供了string类，可以让程序编写更加简单

String可以用cin，cout输入输出，可以用加号进行拼接

C++

```
1  string s1 = "123";
2  string s2;
3  cin >> s2;
4  cout << s1 << '\n'; // 可以使用 cin 和 cout 输入和输出string
5
6  cout << s1.size() << '\n'; // 长度为3
7
8  string s3 = "abc";
9
10 s1 += s3; // s1 = "123abc"
11 s1 += 'x' // s1 = "123abcx"
12 cout << s1 << '\n';
13
14 s2 = s1 + s3; // s2 = 123abcxabc
15
16 s2 = s1.substr(2, 4); // 截取子串, 从第二个开始截取长度为4的子串 s2 = "3abc"
```



上面代码中我使用了'\n'输出换行，而不是endl，因为endl会刷新缓冲区，导致速度变慢



通常不建议使用scanf输入string，因为string不是c语言的基本类型，它和字符串数组完全不同

string也可以比较，等于号可以判断两个string是否相等，小于号和大于号可以判断string的字典序大小



什么是字典序？先比较第一个字母，如果相同，就比较下一个...直到找到两个不同的字母，这是哪个字母小，哪个字符串就小。如果两个字符串前面都相同，其中一个字符串先结束了（即一个字符串是另一个字符串前缀），那么短的字符串字典序更小

C++

```
1 string s = "aab";
2 string t = "aba";
3
4 cout << (s < t) << '\n'; // s < t 为true, 输出 1
5 if(s == t) cout << "s is equal to t\n";
6 else cout << "s is not equal to t\n"; // 不相等
```

关联容器

set（掌握）

头文件：<set>

set是集合。它可以用来装一些元素，在比较优秀的时间内插入新元素，删除元素，或者判断元素是不是在集合中，同时始终保持每个元素在集合中不会重复

插入删除

C++

```
1 set<int> s; // 定义一个set, 里面装整数。一开始是空集
2
3 s.insert(3); // s中插入3
4 s.insert(3); // s中又插入3, 但没有实际效果, 因为本来就有了
5
6 s.erase(3); // 删除s中的3。现在s又是一个空集了
7 s.erase(4); // 删除s中的4。但没有实际效果, 因为s中本来就没有4
8 // insert 和 erase 的复杂度均为O(logn)
```

在set中，元素实际上是逻辑上按顺序从小到大排列的，放在一个红黑树中

迭代器

set的迭代器是双向迭代器，可以通过自增自减来移动，不能加一个数或减一个数来移动。set的迭代器可以指向set中任何一个元素，也可以指向set最大元素的下一个位置，即end()。



迭代器自增或自减的时间复杂度为均摊 $O(1)$ ，如果你用它遍历整个set，不用担心带来额外的复杂度。但单次最坏可能达到 $O(\log n)$

查找

可以使用find函数判断一个数在集合中的什么位置。返回一个set的迭代器，若这个数不存在则返回end()。复杂度为 $O(\log n)$

C++

```
1 set<int> s = {1,2,3,4,5,7,8};
2 auto it1 = s.find(4), it2 = s.find(6); // 这里的auto可以改成set<int>::iterator
3 if(it1 == s.end()) cout << "4 is not found\n";
4 if(it2 == s.end()) cout << "6 is not found\n";
```

也可以用lower_bound查找最小的大于等于给定值的位置（若不存在返回end），upper_bound查找最小的大于给定值的位置（若不存在返回end）。复杂度为 $O(\log n)$

C++

```
1 set<int> s = {1,2,3,4,5,7,8};
2 auto it1 = lower_bound(s.begin(), s.end(), 5), it2 = upper_bound(s.begin(), s.end(), 5);
3 cout << *it1 << ' ' << *it2 << '\n'; // 5 7
```

可以使用size()函数得到set内元素个数，empty()得到set是否为空集

C++

```
1 set<int> s = {1,2,3,4,5,7,8};
2 cout << s.size() << '\n'; // 7
3 if(s.empty()) cout << "s is empty\n";
```

可以通过clear函数删除set内所有元素

C++

```
1 set<int> s = {1,2,3,4,5,7,8};
2 s.clear();
3 if(s.empty()) cout << "s is empty\n";
```

另外，使用count可以得到set中某个数的数量。由于set中元素不重复，所以返回值要么是0，要么是1。这一操作用于判断一个元素是否在set中非常方便

C++

```
1 set<int> s = {1,2,3};
2 cout << s.count(1) << '\n'; // 1
3 cout << s.count(4) << '\n'; // 0
```

遍历


可以使用迭代器，从begin循环到end，也可以用c++11的range for。遍历顺序为从小到大依次访问

C++

```
1  set<int> s = {1,2,3,4,5,6,7,8,9};
2  for(auto i = s.begin(); i != s.end(); i++){
3      cout << *i << ' ';
4  }
5  cout << '\n';
6
7  for(const int &i: s){ // const int 可以改成 auto
8      cout << i << ' ';
9  }
```

set里存储结构体

由于set里的元素逻辑上是按照从小到大排列的，存入set的类型必须要提供一种比较方式。c++要求存入set的类型必须有小于运算符

 为什么只用小于运算符就可以了？因为其他都没有必要，可以用小于替代。

a > b 可以用 b < a 替代

a <= b 可以用 !(b < a) 替代

a >= b 可以用 !(a < b) 替代

a == b 可以用 !(b < a) && !(a < b) 替代

a != b 可以用 a < b || b < a 替代

自定义的结构体需要重载小于号运算符

假如我们要想set中存储一个结构体point，我们可以像下面一样

C++

```
1  struct Point{
2      int x, y;
3      bool operator < (const Point &p) const{
4          if(x != p.x) return x < p.x;
5          else return y < p.y;
6      }
7  }
8
9  set<Point> pointSet; // 这样就可以定义结构体了
```

map(掌握)

头文件：<map>

map是映射。map维护了一个字典，即若干键到值的映射

在讨论map实际功能之前，先讨论一下普通的数组。假如我们像这样定义了一个数组：

C++

```
1 double a[10];
```

我们可以通过一个下标，访问数组的值，也就是说，数组提供一个下标到数组内容的映射。但数组的限制比较大，因为：

- 数组下标从0开始，一直连续到数组长度减一。而不一定数组的每个位置都有用
- 数组下标只能是整数，而不能是字符串或者其他内容

map就相当于一个更高级的数组。我们把对应数组下标的称为键，数组内容的称为值

定义

C++

```
1 map<string, int> mp; // 定义一个map，键是string类型，
```



map中的键值对是按照键的大小，从小到大排列的。因此需要键的类型有比较操作，如果是自定义结构体则需要重载小于号。这一点和set相同

增删查改

很多时候我们不会特意插入一组键值对，而是当一个键被访问时，这个键值对就自动插入了

C++

```
1 map<string, int> mp;  
2 mp["sdu"] = 10;  
3 mp["acm"] = 34;
```

通过方括号可以访问某个键对应的值。如果访问时不存在，则自动插入，值为默认值

C++

```
1 // 接上面  
2 cout << mp["sdu"] << '\n' // 10  
3 cout << mp["csp"] << '\n' // 0  
4 // 此时 mp 中有 acm->34, csp->0, sdu->10三组键值对
```

可以用过erase删除一个键值对。如果本身不存在则无影响

C++

```
1 mp.erase("csp");
```

可以通过count获取一个键在map中出现的次数。显然返回值只能是0或者1，用法和set相似，不再举例

需要注意的是以上增删查改的方法，时间复杂度均为 $O(\log n)$

可以使用size，empty等 $O(1)$ 的方法获取尺寸或是否为空，也可以通过clear清除所有元素，这一点和set相同，不再举例

迭代器

map的迭代器和set类似，可以自增或自减来移动，但map的迭代器指向一个pair<const K, V>，其中K和V分别是键和值的类型

可以通过find操作查找某个键所在键值对的位置，返回一个迭代器，若不存在返回end。用法和set类似

也可以通过lower_bound和upper_bound查找，同样和set类似。当然，这样查找，即使查找不到，也不会自动插入新的键值对

C++

```
1 map<int, int> mp = {{1,4}, {2,5}, {3,4}}; // 可以通过初始化列表构造
2 map<int, int>::iterator it = mp.find(3); // 类型可以用auto简写
3 if(it != mp.end()) cout << it->second << '\n'; // 输出查找的值
```

遍历

可以使用迭代器

C++

```
1 map<string, int> mp = {{ "abc", 2}, {"bcd", 1}, {"aab", 4}, {"aca", 7}};
2 for(auto i = mp.begin(); i != mp.end(); i++){
3     cout << "key = " << i->first << ", value = " << i->second << '\n';
4 }
```

也可以使用range for，更简单

C++

```
1 // 接上面
2 for(auto &i: mp){
3     cout << "key = " << i.first << ", value = " << i.second << '\n';
4 }
```

拓展（了解）

c++11后有了unordered_map和unordered_set，与map和set几乎使用方法相同，只不过，map或set使用红黑树作为底层实现，unordered版本使用哈希表为底层实现。因此有时候，unordered版本时间复杂度更优。如果要存储结构体，需要重载==方法和hash方法，而不用重载小于号。缺点是unordered版本中存储的元素不能按照顺序从小到大排列

如果想在set中存储多个相同的元素，或者map中存储多个相同键的元组，可以使用multiset和multimap。具体使用方法可以自行查找资料学习

容器适配器

容器适配器本身不是完整的容器，而是由底层容器经过封装得来的

stack

头文件：<stack>

stack是栈，一种先进后出的数据结构。c++的支持在栈顶插入一个元素，访问栈顶的元素，以及删除栈顶元素等操作，也支持得到栈的尺寸，以及栈是否为空

C++

```
1 stack<int> s; // 创建一个栈，用于存放int，一开始为空
2
3 s.push(1);
4 s.push(2);
5 s.push(3);
6 // 向栈中插入三个数，分别是1, 2, 3
7
8 cout << s.size() << '\n'; // 3, 栈的尺寸
9
10 s.top() = 4; // 修改栈顶的值
11
12 while(!s.empty()){ // empty 用于判断 栈是否为空
13     cout << s.top() << ' '; // top 用于访问栈顶元素
14     s.pop(); // pop 用于删除栈顶元素
15 } // 结果为 4 2 1
```

栈的底层实际上是个deque，push对应deque的push_back操作，pop对应deque的pop_back()操作，而top()对应deque的back操作



和其他容器适配器一样，栈的底层容器可以更换，只要底层容器能提供用到的那些操作。栈不提供clear函数，所以底层容器也无需提供clear方法

queue

头文件：<queue>

queue是队列，一种先进先出的数据结构。c++的支持在队列尾部插入一个元素，访问队首的元素，以及删除队首元素等操作，也支持得到队列的尺寸，以及队列是否为空

C++

```
1  queue<int> q; // 创建一个栈，用于存放int，一开始为空
2
3  q.push(1);
4  q.push(2);
5  q.push(3);
6  // 向队列中插入三个数，分别是1, 2, 3
7
8  cout << q.size() << '\n'; // 3, 队列的尺寸
9
10 q.front() = 4; // 修改队列首的值
11
12 while(!q.empty()){ // empty 用于判断队列是否为空
13     cout << q.front() << ' '; // top 用于访问队首元素
14     q.pop(); // pop 用于删除队首元素
15 } // 结果为 4 2 3
```

队列的底层实际上是个deque，push对应deque的push_back操作，pop对应deque的pop_front()操作，而front()对应deque的front操作

priority_queue

头文件：<queue>

priority_queue是优先队列，内部是用堆实现的。默认为大根堆



优先队列和队列关系不大

C++

```
1  priority_queue<int> q; // 创建一个优先队列，用于存放int，一开始为空
2
3  q.push(2);
4  q.push(3);
5  q.push(1);
6  // 向优先队列中插入三个数，分别是2, 3, 1
7
8  cout << q.size() << '\n'; // 3, 队列的尺寸
9
10 while(!q.empty()){ // empty 用于判断队列是否为空
11     cout << q.top() << ' '; // top 用于访问队列顶元素
12     q.pop(); // pop 用于删除队列顶元素
13 } // 结果为 3 2 1
```

小根堆

优先队列默认为大根堆，如果要使用小根堆，那么需要修改模板参数

C++

```
1  priority_queue<int, vector<int>, greater<int> > q; //定义一个存放 int 的小根堆
```

模板的第一个参数是优先队列中存放元素类型，第二个参数是优先队列的底层容器，通常我们使用vector即可，第三个参数是一个类，这个类重载小括号运算符实现比较。greater<T>这一类模板在前文讲sort时有所提及

存储结构体

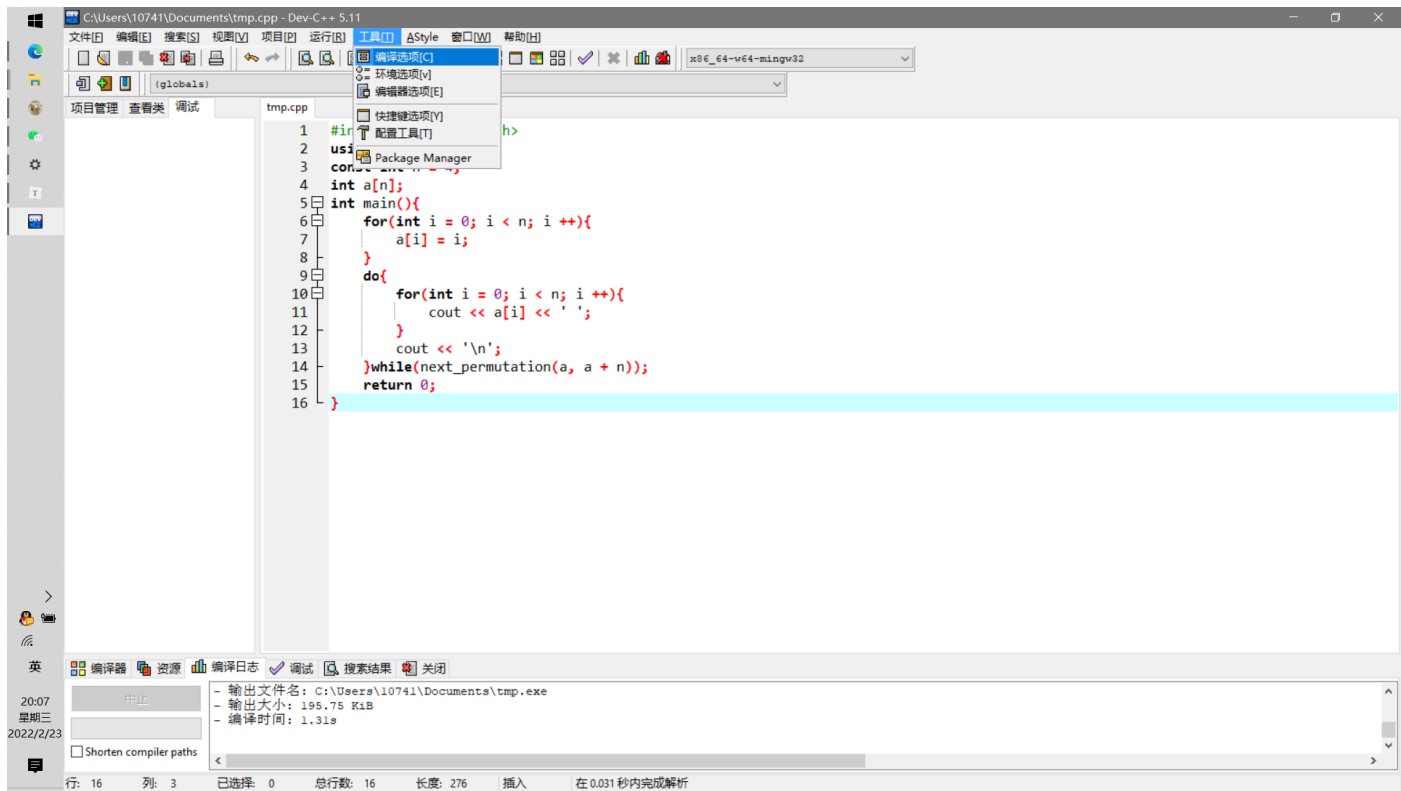
如果要存储结构体，需要提供一个比较方法。最简单的方法就是给结构体重载小于号

附录

关于dev c++的c++11的开启

dev-c++默认不开启c++11。但我们有时候希望使用c++11的新特性（同时又想用dev c++），就需要手动开启

工具-编译选项



代码生成/优化-代码生成-语言标准

设定编译器配置

x86_64-w64-mingw32



编译器 代码生成/优化 目录 程序

C 编译器: 代码生成: 代码警告 代码性能: 连接器 输出

生成特定机器的专用指令

最少优化, 保持全部兼容性 (-mtune)

使用处理器内建函数

优化级别 (-Ox)

使用下列位宽编译 (-mx)

语言标准 (-std)

ISO C++11

获得更多关于GCC的选项, 请参阅

<http://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html>

✓ 确定[O]

✕ 取消[C]

? 帮助[H]