C++ STL

STL 是 $Standard\ Template\ Library$ 的简称，中文名称为标准模板库，从根本上讲，STL 就是各种容器的集合，容器可以理解为能够实现很多功能的系统的函数。常见的容器有 `vector`，`stack`，`queue`，`map`，`set` 等。

迭代器（`iterators`）是用来访问容器中的元素，类似于指针。迭代器包含两个函数：

1. `begin()`，返回一个容器开头的迭代器；
2. `end()`，返回一个容器末尾的迭代器，但是**不包括最后一个元素，而是最后一个元素的下一个地址。**

定义：

```
containers <typename>::iterators name;
//containers为容器类型，typename为容器内的数据类型，name为迭代器名
```

string

在 C 语言中，一般使用字符数组 `char str[ ]` 来存放字符串，但是操作起来非常麻烦，容易出错。$C++$ 在 STL 中加入了 `string` 类型用来存放字符串，使用起来更加方便。

使用时，需要添加头文件 `string`，即 `#include<string>`。

1.string的定义

```
string name;                  // name是变量名
string name2=”abcde”;         //可直接赋值
```

如果需要存放多个字符串，定义 `string` 类型数组：

```
string name [N];
name[N]可以存放N个字符串。
```

2.string的输入输出

如果要输入输出，只能使用 `cin` 和 `cout`。使用 `cin` 读入 `string` 类型，就像用 `scanf()` 读字符数组一样，忽略开头的(制表符、换行符、空格），当再次碰到空字符就停止（并不会读取空字符）。

```
string s;
cin>>s;
```

读入时忽略了空字符，可以将其记作 `cin` 读字符串读的是单词。当然，有时我们更希望读取的是句子。$C++$ 提供了 `getline` 函数以供使用。

`getline` 的原型是：`getline(cin,s);`，`cin` 指的是读入流，一般情况下我们直接写 `cin` 即可，`s` 是字符串，即我们读入的东西要存放的字符串。

```
string s;
getline(cin,s);      // getline()读入一行的字符，会舍弃换行符
```

3.string的访问

（1）通过下标访问

对于 `string` 类型的字符串进行访问，与字符数组访问一样，使用下标。如果有 `string` 类型变量名为 `s` ，可直接访问对应字符 `s[0]，s[1]，s[2]……，s[s.size()-1]` ， `s.size()` 是用来求 `string` 类型的长度。

（2）通过迭代器访问

```
string::iterator it;
```

定义了 `string` 的迭代器 `it` ，可以通过 `*it` 访问 `string` 中的每一个字符，在常用的 STL 容器中，只有 `vector` 和 `string` 允许使用“`v.begin()+5` ”这种迭代器加上整数的写法，等价于访问 `s[5]` 。同时，迭代器可以进行自加、自减操作，即 `it++` ， `++it` ， `it--` ， `--it` 。
例如：

```
string s="abcdef";
for(string::iterator it=s.begin();it!=s.end();it++)//输出abcdef
        cout<<*it<<" ";
cout<<endl;
for(string::iterator it=s.begin()+3;it!=s.end();it++)//输出def
        cout<<*it<<" ";
```

`s.end()` 不是取 `s` 的尾元素地址，而是尾元素地址的下一个地址，不存储任何元素，不支持 `it<s.end()` 的写法。

（4）string的运算

① `string` 类型可以直接进行加法运算。但是加法是将两个字符串拼接起来。

例如：

```
string s1=”abc”,s2=”efgh”;
cout<<s1+s2;          //输出abcefgh
```

② `string` 类型可以互相进行复制。
例如：

```
string s3;
s3=s2;
cout<<s3;        //输出efgh
```

③ `string` 类型可以直接进行关系运算。直接比较大小，按照字典序进行比较。

```
if(s1>s2)  cout<<1;
else cout<<0;        //最后结果为0，“efgh”字典序大于“abc”
//判断是否相等。
string s1=”abc”,s2=”abc”;
if(s1==s2)  cout<<1;         //会输出1
```

(5) string的常用函数

① size()和length()

这两个函数都是返回 `string` 类型的长度（字符个数）。时间复杂度 $O(1)$ 。

② clear()和empty()

`clear()` 用来清空 `string` 中的所有元素，时间复杂度为 $O(1)$ 。

`empty()` 用来判断 `string` 是否为空，是返回 `true` ，否则返回 `false` 。

例如：

```
string s=”abcde”;
cout<<s.size ();         //输出5
s.clear();
cout<<s.length();        //输出0
if(s.empty())   cout<<1;
else cout<<0;            //输出1
```

③ insert(pos,s2)

在 `s` 下标为 `pos` 的元素前插入 `string` 类型 `s2` 。

例如：

```
string s=”abcde”,s2=”opq”;
s.insert(3,s2);
cout<<s;         //输出abcopqde
```

④ erase(pos,len)

删除 `s` 中下标为 `pos` 开始的 `len` 个字符 。

例如：

```
string s=”abcdefgh”;
s.erase(3,3);
cout<<s;         //输出abcgh
```

⑤ find(s2)

当 `s2` 是 `s` 子串时，返回在s中第一次出现的位置，否则，返回 `string::npos` 。

例如：

```
string s=”abcdefabcde”,s2=”cde”;
if(s.find(s2)!=string::npos) cout<<s.find(s2);          //输出2
s.find(s2,pos)，是在 s 中以 pos 位置起查找 s2 第一次出现的位置，返回值s.find(s2)相同。
```

⑥ replace(pos,len,s2)

删除 `s` 中下标为 `pos` 开始的 `len` 个字符，并在下标为 `pos` 处插入 `s2` 。

例如：

```
string s1="asfgg",s2="ad";
s1.replace(1,1,s2);
cout<<s1<<endl;          //输出aadfgg
```

vector

`vector` 为变成数组，即长度可以根据需要进行改变的数组。在信息学竞赛中，有些题目需要定义很大的数组，这样会出现"超出内存限制"的错误，使用 `vector` 简洁方便，还可以节省空间。

使用时，需要添加头文件 `vector` ，即 `#include<vector>` 。

1.vector的定义

```
vector<typename> name;
```

以上定义相当于定义了一个一维数组 `name[size]` ，只是 `size` 不确定，大小可以根据需要而变化。其中 `tyepename` 为基本类型，如 `int` 、 `double` 、 `char` 、结构体等，也可以是 STL 的容器，如 `string` 、 `queue` ， `vector` 等。

例如：

```
vector<int> a;
vector<double> score;
vector<node> stu; //node为已经定义了得结构体
```

但是，如果 `typename` 也是一个 `STL` 容器，那么定义时，需要在两个 "$>$" 之间加上一个空格，因为 "$>>$" 会被当作右移运算，从而导致编译出错，例如：

```
vector<int> a[100];      //定义一个一维长度固定为100，另一维不确定的二维数组a[100][size]
vector<vector<int> > a;  //定义一个两维都可变的二维数组a[size][size]
```

2.vector的常用函数

(1) push_back(x)

在 `vector` 数组后面添加一个元素 `x` ，下标从 `0` 开始，时间复杂度为 `O(1)` 。

(2) size()

如果是一维数组，`size()` 获得 `vector` 中元素的个数；如果是二维数组，`size()` 获得 `vector` 中的第二维的元素个数，时间复杂度为 $O(1)$ 。

(3) pop_back()

删除 `vector` 的末尾元素，时间复杂度为 $O(1)$.

(4) clear()

清空 `vector` 中的元素，时间复杂度为 $O(n)$，`n` 为 `vector` 中元素的个数。

(5) insert(it,x)

在 `vector` 迭代器 `it` 处插入一个元素 `x`，`x` 后的元素后移，时间复杂度为 $O(n)$.

(6) erase(it)、erase(first,last)

删除 `vector` 中的元素元素。`erase(it)`，删除迭代器 `it` 处的元素；`erase(first,last)`，删除左闭右开区间 $[first, last)$ 内的所以元素。

例如：

```
    vector<int> v;
    for(int i=1;i<=5;i++) v.push_back(i);       //数组元素为1 2 3 4 5
    for(int i=0;i<v.size();i++) cout<<v[i]<<" ";//输出1 2 3 4 5
    cout<<endl;
    v.pop_back();
    for(int i=0;i<v.size();i++) cout<<v[i]<<" ";//输出1 2 3 4
    cout<<endl;
    v.insert(v.begin()+2,10);                   //将10插入到v[2]处
    for(int i=0;i<v.size();i++) cout<<v[i]<<" ";//输出1 2 10 3 4
    cout<<endl;
    v.erase(v.begin()+1,v.begin()+3);           //删除v[1],v[2],
    for(int i=0;i<v.size();i++) cout<<v[i]<<" ";//输出1 3 4
    cout<<endl;
    v.clear();
    cout<<v.size();                             //输出0
```

3.vector的访问

访问 `vector` 中的元素一般有两种方式。

（1）通过下标进行访问，对于容器vector v，可以使用v[i]来访问第i个元素。

（2）通过迭代器访问

定义：

```
vector<int>::iterator it;
```

定义一个迭代器 `it`，通过 `*it` 访问 `int` 类型的 `vector` 中的元素。

在常用的 STL 容器中，只有 `vector` 和 `string` 允许使用“`v.begin()+5`”这种迭代器加上整数的写法。

```
vector<int> v;
for(int i=1;i<=5;i++) v.push_back(i);//数组元素为1 2 3 4 5
vector<int>::iterator it=v.begin();
for(int i=0;i<v.size();i++)
    cout<<*(it+i)<<" ";//输出1 2 3 4 5
```

同时，迭代器可以进行自加、自减操作，即 `it++`，`++it`，`it--`，`--it`。

```
vector<int> v;
for(int i=1;i<=5;i++) v.push_back(i);//数组元素为1 2 3 4 5
for(vector<int>::iterator it=v.begin();it!=v.end();it++)
    cout<<*it<<" ";//输出1 2 3 4 5
```

stack

`stack` 翻译为栈，实现先进后出的容器。

使用时，需要添加头文件 `stack` ，即 `#include<stack>` 。

1.stack的定义

```
stack<typename> name;
```

例如：

```
stack<int> s;   //定义一个int类型的栈s
```

2.stack的常用函数

(1) psuh(x)

将元素 `x` 压栈，时间复杂度为 $O(1)$ 。

(2) top()

获取栈顶元素，时间复杂度为 $O(1)$ 。

(3) pop()

弹出栈顶元素，时间复杂度为 $O(1)$ 。

(4) empty()

检测 `stack()` 是否为空，空返回 `true` ，否则返回 `false` ，时间复杂度为 $O(1)$ 。在使用 `top()` 和 `pop()` 之前，须用 `q.empty()` 判断是否为空，否则可能因为栈空出现错误。

(5) size()

返回 `stack` 内的元素个数，时间复杂度为 $O(1)$ 。

例如：

```
stack<int> s;
for(int i=1;i<=5;i++) s.push(i);//栈元素为1 2 3 4 5
s.pop();    //删除栈顶元素5
cout<<s.top()<<endl;    //输出4
cout<<s.size()<<endl;   //输出4
while(!s.empty())
{
    cout<<s.top();  //输出 4 3 2 1
    s.pop();
}
```

queue

`queue` 翻译为队列，是一个“先进先出”的容器。

使用时，需要添加头文件 `queue` ，即 `#include<queue>` 。

1.queue的定义

```
queue <typename> name;      // name是变量名
```

例如：

```
queue <int> q;//定义一个int类型，名为q的队列
```

2.queue的常用函数

（1）push(x)

将 `x` 入队，时间复杂度 $O(1)$ 。

（2）front()和back()

分别用来访问队首和队尾元素，时间复杂度 $O(1)$ 。

（3）pop()

删除队首元素。

（4）empty()

用来检查队首是否为空，返回 `true` 或者 `false` ，时间复杂度 $O(1)$ 。在使用 `front()` 和 `pop()` 之前，须用 `empty()` 判断是否为空，否则可能因为队空出现错误。

（5）size()

返回中的元素个数，时间复杂度 $O(1)$ 。

priority_queue

`priority_queue` 翻译为优先队列，其底层是用堆实现的。

在优先队列中，任何时刻，队首元素一定是当前优先级最高（值最大）的一个（大根堆），也可以是值最小的一个（小根堆）。你可以不断往队列中添加或删除优先级不同的元素，每次操作队列都会自动调整，始终保证队首优先级最高。

优先队列的优先级设置一般是数字越大优先级越大，对于char，字典序越大优先级越大。

使用时，需要添加头文件 `queue` ，即 `#include<queue>` 。

1.priority_queue的定义

```
priority_queue <typename> name;
```

例如：

```
priority_queue <int> q;//定义一个int类型，名为q的优先队列：
```

这样定义的是一个大根堆，它的原型其实是：

```
priority_queue <int,vector<int>,less<int> > q;
```

尖括号中多了两个参数，`vector<int>` ，表示的是承载底层数据结构——堆的容器，类型与第一个参数一致； `less<int>` ，是对第一个参数的比较类，表示数字越大优先级越大（大根堆），而如果用 `greater<int>` ，则表示数字越小优先级越大。

因此，定义大根堆有两种方法，这两种方法是等价的：

```
priority_queue <int> q;
priority_queue <int,vector<int>,less<int> > q;
```

定义小根堆：

```
priority_queue <int,vector<int>,greater<int> > q;
```

注意，最后的“>>”，两个“>”之间是有空格的，没有空格会被当作右移运算，会出现编译错误。

2.priority_queue的常用函数

（1）push(x)

将 `x` 入队，时间复杂度 $O(\log_2 n)$ ，`n` 为当前优先队列中的元素个数。加入后，会自动调整整个优先队列内部结构，保证队首（堆顶）优先级最高。

（2）top()

获取队首元素（堆顶元素），时间复杂度 $O(1)$ 。

（3）pop()

删除队首元素（堆顶元素），时间复杂度 $O(log_2 n)$ ，`n` 为当前优先队列中的元素个数。加入后，会自动调整整个优先队列内部结构，保证队首（堆顶）优先级最高。

（4）empty()

用来检查队首是否为空，返回 `true` 或者 `false` ，时间复杂度 $O(1)$ 。在使用 `top()` 和 `pop()` 之前，须用 `empty()` 判断是否为空，否则可能因为队空出现错误。

例如：

```
priority_queue <int,vector<int>,greater<int> > q;        //定义小根堆
q.push(4);
q.push(6);
q.push(3);
q.push(9);
if(!q.empty())  cout<<q.top();       //输出3
q.pop();
if(!q.empty())  cout<<q.top();       //输出4
```

3.priority_queue结构体

如果优先队列的元素是结构体。

现在读入若干学生的语文、数学成绩和姓名，按照按语文从大到小排序，语文相同按数学大到小排序，数学相同，按名字字典序排序。

也可以使用 `pair` ，更方便。

例如：

```
struct stu
{
    int chinese;
    int math;
    string name;
    bool operator<(const stu &x) const{
        if(chinese<x.chinese) return 1;//语文大的在前,和sort相反
        if(chinese>x.chinese) return 0;
        if(math<x.math) return 1;   //语文相等，数学大的在前
        if(math>x.math) return 0;
        if(name>x.name) return 1;//语文数学都相等，字典序大的在前
        return 0;
    }
};
priority_queue<stu> q;
int main()
{
    stu a;
    a.chinese=90;a.math=80;a.name="abc";
    q.push(a);//放入
    a.chinese=90;a.math=85;a.name="xyz";
    q.push(a);
    a.chinese=90;a.math=85;a.name="yyy";
    q.push(a);
    a.chinese=92;a.math=85;a.name="bcd";
    q.push(a);
    a=q.top();
    cout<<a.chinese<<" "<<a.math<<" "<<a.name<<endl;//输出 92 85 bcd
    q.pop();//删除第一个
    a=q.top();
    cout<<a.chinese<<" "<<a.math<<" "<<a.name<<endl;//输出 90 85 xyz
    return 0;
```

```
}
```

4.Dijkstra最短路算法中的优化

最短路中的 Dj 算法也可以使用优先队列进行优化。

pair

`pair` 是二元结构体，将两个元素捆绑在一起，相当于：

```
struct pair
{
    typename1 first;
    typename2 second;
}
```

使用时，需要添加头文件 `#include<utility>` 。

1.pair的定义

`pair` 有两个参数，可以是任意基本类型或容器。

```
pair <typename1,typename2> name;
```

2.pair的初始化

`pair` 使用 `first` 和 `second` 访问第一，第二个元素。

`pair` 有三种初始化的方式，如下：

```
pair<string,int>  p("abc",1);//初始化1
cout<<p.first<<" "<<p.second<<endl;//输出abc 1
p.first="bcd";//初始化2
p.second=5;
cout<<p.first<<" "<<p.second<<endl;//输出bcd 5
p=make_pair("xyz",9);//初始化3
cout<<p.first<<" "<<p.second<<endl;//输出xyz 9
```

如果有三个元素，也可以用 `pair` 实现：

```
pair<int,pair<int,int> > p[100];//> >中间有空格，否则会被当作位移运算
p[1].first=1;
p[1].second.first=5;
p[1].second.second=2;
cout<<p[1].first<<" "<<p[1].second.second;//输出1 2
```

`pair` 可以直接做比较。比较规则是先以 `first` 的大小作为标准，只有当 `first` 相等时才去判断 `second` 的大小。

例如：

```
pair<int,int>  p1(5,10);
pair<int,int>  p2(5,15);
pair<int,int>  p3(10,5);
if(p1<p3) cout<<"p1<p3"<<endl;//输出p1<p3
if(p1<p2) cout<<"p1<p2"<<endl; //输出p1<p2
if(p1<=p3) cout<<"p1<=p3"<<endl;//输出p1<=p3make<int,int> p1(5,10);
```

map

`map` 翻译为映射。其实，数组就是一种映射。`int a[100]` 定义了一个 `int` 到 `int` 的映射，`a[5]=25` ，把 `5` 映射到 `25` 。数组总是将 `int` 类型映射到其他类型。如果要将 `string` 类型映射到 `int` 类型，数组就很不方便，此时可以使用 `map` ，`map` 可以将任意基本类型（包括 STL 容器）映射到任意基本类型（包括STL 类型）。

`map` 常用情形：

1)建立字符(串)与整数之间的映射

2)判断大整数(几千位)或者其他类型数据是否存在，可以将 `map` 当布尔数组使用，实现类似哈希表的功能。

3)字符串与字符串的映射。

使用时，需要添加头文件 `map` ，即 `#include<map>` 。

1.map的定义

```
map<typename1,typename2> name;
```

`typename1` 是映射前的类型(键 key)，`typename2` 是映射后的类型（值 $value$ ），`name` 为映射名称。

普通 `int` 数组 `a` 就是：`map<int,int> a;`

字符串映射到整型，必须使用 `string` ，不能使用 `char` ：`map<string,int> a;` 。

2.map的访问

（1）通过下标进行访问

下标访问就像访问普通数组元素一样，如定义：`map<char,int> mp` ，就可以通过 `mp['c']` 访问对应元素，如 `mp['c']=24` 。

（2）通过迭代器进行访问

`map` 的每一对映射都有两个 `typename` ，所以用 `it->first` 访问键，使用 `it->second` 来访问值。

3.map的赋值

`map` 有两种种输入方式：

（1）用insert函数插入pair数据，pair可以作为map的键值对来插入。

```
map<string,int> mp;
mp.insert(pair<string,int>("xyz",1));
mp.insert(pair<string,int>("def",2));
mp.insert(pair<string,int>("abc",3));
for(map<string,int>::iterator it=mp.begin();it!=mp.end();it++)
    cout<<it->first<<" "<<it->second<<" ";//输出abc 3 def 2 xyz 1
```

可以看出，`map` 建立映射后，会自动实现按键从小到大排序，这是因为 `map` 内部使用红黑树实现的（`set` 也是如此）。

（2）用数组进行插入

```
map<string,int> mp;
mp["abc"]=1;
mp["def"]=2;
mp["xyz"]=3;
for(map<string,int>::iterator it=mp.begin();it!=mp.end();it++)
    cout<<it->first<<" "<<it->second<<" ";//输出abc 1 def 2 xyz 3
```

但是它们是有区别的，用 `insert` 函数插入数据，在数据的插入上涉及到集合的唯一性这个概念，即当 `map` 中有这个关键字时，`insert` 操作是插入不了数据的，但是用数组方式就不同了，它可以覆盖以前该关键字对 应的值。

4.map的常用函数

（1）find(key)

返回键为 key 的映射的迭代器，时间复杂度为 $O(log_2 n)$，`n` 为 `map` 映射的对数。

如果未找到，返回 `end()` 的迭代器

例如：

```
map<int,int> mp;
mp.insert(pair<int,int>(1,10));
if(mp.find(2)==mp.end()) cout<<"Not exist";//输出Not exist
```

（2）size()

返回 `map` 中映射的对数，时间复杂度为 $O(1)$。

（3）erase(it)和erase(first,last)

`erase(it)` 删除迭代器it的元素，时间复杂度为 $O(1)$，也可以用 `erase(key)`，key 为要删除映射的键，时间复杂度为 $O(log_2 n)$.

`erase(first,last)`，删除左闭右开区间 $[first,last)$，$first$ 为起始迭代器，$last$ 为末尾迭代器的下一个地址，时间复杂度为 $O(last-first)$。

（4）clear()

清空 `map`，时间复杂度为 $O(n)$。

```
map<string,int> mp;
mp["xyz"]=1;
```

```
mp["def"]=2;
mp["abc"]=3;
cout<<mp.size()<<endl;  //输出3
map<string,int>::iterator it=mp.find("xyz");
cout<<it->first<<" "<<it->second<<endl;//输出xyz 1
mp.erase(it);
//将这句话与find可以和合并成一句话mp.erase("xyz")，时间复杂度相同
for(map<string,int>::iterator it=mp.begin();it!=mp.end();it++)
    cout<<it->first<<" "<<it->second<<" ";//输出abc 3 def 2
it=mp.find("abc");
mp.erase(it,mp.end());
cout<<mp.size()<<endl;//输出0
```

set

`set` 翻译为集合，是一个内部自动有序切且不含重复元素的容器。`set` 的主要作用就是**自动去重并按升序排序**，因此遇到不方便开数组的情况，比如元素较大或类型不是 `int`，可以是一个 `set` 解决。`set` 内部也是使用红黑树实现的。

使用时，添加 `set` 头文件，即 `#include<set>`。

1.set的定义

```
set<typename> name;
```

`typename` 可以是任意类型或容器，`name` 是集合名称。

```
set<int> st;   //定义int的集合st
set<int> st[100];  //定义int的100个集合st[0],st[1]...st[99]
```

2.set的访问

`set` 只能通过迭代器访问。

```
set<typename>::iterator it;
```

通过 `*it` 访问 `set` 中元素。

3.set的常用函数

(1) inset(x)

将 `x` 插入到 `set` 中，并自动排序去重，时间复杂度为 $O(log_2 n)$。

如果未找到，返回 `end()` 的迭代器。

例如：

```
set<int> s;
s.insert(10);
if(s.find(5) != s.end())  cout<<"exist"<<endl;
else cout<<"Not exist"<<endl;   //输出Not exist
```

(2) size()

返回 `set` 中的个数，时间复杂度为 $O(1)$ 。

(3) find(x)

返回 `set` 中对应值 `x` 的迭代器，时间复杂度为 $O(log_2 n)$ 。

(4) clear()

清空 `set` 中的元素，时间复杂度为 $O(n)$ 。

(5) erase(it)和erase(first,last)

`erase(it)` 删除迭代器it的元素，时间复杂度为 $O(1)$ ，也可以用 `erase(value)` ， `value` 为要删除元素的值，时间复杂度为 $O(log_2 n)$.
`erase(first,last)` ，删除左闭右开区间 $[first, last)$ ， $first$ 为起始迭代器，$last$ 为末尾迭代器的下一个地址，时间复杂度为 $O(last - first)$ 。

```
set<int> st;
for(int i=5;i<=10;i++) st.insert(i);
cout<<*(st.find(2))<<endl;//输出6
set<int>::iterator it=st.find(8);
st.erase(it,st.end());//删除8 9 10
for(it=st.begin();it!=st.end();it++)
    cout<<*it<<" "; //输出5 6 7
```

multiset

`multiset` 与 `set` 类似，区别是 `multiset` 能够保存重复的元素。

1.multiset的定义

```
multiset<typename> name;
```

2.multiset的常用函数

`multiset` 容器和 `set` 容器有相同的成员函数，但是因为 `multiset` 可以保存重复元素，有些函数的表现会有些不同。和 `set` 容器中的成员函数表现不同的是：

(1) insert()

总是可以成功执行。当插入单个元素时，返回的迭代器指向插入的元素。

(2) find()

会返回和参数匹配的第一个元素的迭代器，如果都不匹配，则返回容器的结束迭代器。

(3) count()

返回和参数匹配的元素的个数。

algorithm

`algorithm` 翻译为算法，提供了大量函数。

1.max(x,y)，min(x,y)，abs(x)，swap(x,y)

`max(x,y)`，`min(x,y)` 返回较大值和较小值，可以是整型，也可以是浮点型。

`abs(x)` 返回 `x` 的绝对值，`x` 必须是整数。如果要求浮点数绝对值，可以使用 `math` 头文件下的 `fabs(x)`

`swap(x,y)` 用来交换 `x` 和 `y` 的值。

2.next_permutation()

求一个序列中全排列的下一个序列。例如 123 的全排列为：123，132，213，231，312，321，231 的下一个排列就是 321 .

基本格式：`next_permutation` (起始元素地址，结束元素地址的下一个地址)

```
int a[10];
a[1]=1;a[2]=2;a[3]=3;
do{
    cout<<a[1]<<" "<<a[2]<<" "<<a[3]<<endl;
}while(next_permutation(a+1,a+4));//a[1],a[2],a[3]的排列
```

输出：

```
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
```

3.sort()

`sort` 是实现排序的函数。

（1）sort的基本格式：

`sort( 起始元素地址，结束元素地址的下一个地址，比较函数);`

比较函数缺少会默认对区间元素递增排序。

```
bool cmp(int x,int y)//定义比较函数，string、double、char类型均可
{
    if(x>y) return 1;//如果a>b成立，a放前面
    return 0;   //否则放在后面
}
int main()
{
    int a[6]={4,5,9,-2,5,-5};
    sort(a,a+4);//a[0]~a[3]从小到大排序
    for(int i=0;i<6;i++)    cout<<a[i]<<" ";//输出 -2 4 5 9 5 -5
    cout<<endl;
```

```
    sort(a+2,a+6,cmp);//a[2]~a[5]从大到小排序
    for(int i=0;i<6;i++)    cout<<a[i]<<" ";//输出-2 4 9 5 5 -5
    return 0;
}
```

(2) 结构体sort

现在对学生成绩排序，按语文从大到小排序，语文相同按数学大到小排序，数学相同，按名字字典序排序。

```
struct stu  //结构体
{
    int chinese;
    int math;
    string name;
}a[100];
bool cmp(stu x,stu y)//结构体类型
{
    if(x.chinese>y.chinese) return 1;//语文大的在前
    if(x.chinese<y.chinese) return 0;
    if(x.math>y.math) return 1; //语文相等，数学大的在前
    if(x.math<y.math) return 0;
    if(x.name>y.name) return 1;//语文数学都相等，字典序大的在前
    return 0;
}
int main()
{
    for(int i=1;i<=5;i++)
        cin>>a[i].chinese>>a[i].math>>a[i].name;
    sort(a+1,a+6,cmp);
    for(int i=1;i<=5;i++)
        cout<<a[i].chinese<<" "<<a[i].math<<" "<<a[i].name<<endl;
    return 0;
}
```

输入：

```
90 80 abc
90 85 xyz
92 85 ppt
92 85 bcd
100 53 hij
```

输出：

```
100 53 hij
92 85 ppt
92 85 bcd
90 85 xyz
90 80 abc
```

(3) 容器sort

STL 中的容器中，只有 `vector` 、 `string` 可以使用 `sort()` 。其他类型 `map` 、 `set` 等，其中元素本身就是有序的，无法使用。

```
bool cmp(int x,int y)//结构体类型
{
    if(x>y) return 1;
    return 0;
}
int main()
{
    vector<int> v;
    for(int i=1;i<=5;i++)   //输入 7 9 0 2 1
    {
        int t;
        cin>>t;
        v.push_back(t);
    }
    sort(v.begin(),v.end(),cmp);
    for(int i=0;i<v.size();i++)
        cout<<v[i]<<" ";    //输出9 7 2 1 0
    return 0;
}
```

4.lower_bound(first,last,val)和upper_bound(first,last,val)

`lower_bound(first,last,val)` 用来寻找一个有序(从小到大)数组或者容器 $[first, last)$ 中，第一个值大于或等于 val 的位置。如果是数组，返回该位置指针，如果是容器，返回该位置的迭代器。

`upper_bound(first,last,val)` 用来寻找一个有序数组或容器 $[first, last)$ 中，第一个值大于 val 的位置。如果是数组，返回该位置指针，如果是容器，返回该位置的迭代器。

如果数组或者容器中没有需要寻找的元素，则上面两个函数的返回值均为可以插入该位置的指针或迭代器，时间复杂度为 $O(log_2(last - first))$。

例如：

```
int a[10]={1,2,2,3,3,3,5,5,5,5};
int b[10]={5,5,5,5,3,3,3,2,2,1};
int *t=lower_bound(a,a+10,2);
cout<<t<<endl;  //t为找到元素的地址
cout<<t-a<<endl;    //输出1
//a为a[0]的地址，数组存储使用连续的地址，t-a即数组中第几个元素
//a+2为a[2]的地址
t=upper_bound(a+2,a+10,2);  // t为找到元素的地址
cout<<t-(a+2)<<endl;    //输出3
cout<<upper_bound(a,a+10,3)-a<<endl;//也可以不指针，输出 6
```