标准模板库包含了序列容器（sequence containers）、关系容器（associative containers）和无序关联容器。

序列容器

一 向量 vector

vector 是向量类型，它可以容纳许多类型的数据，如若干个整数，所以称其为容器。vector 是C++ STL的一个重要成员，是单向的开口的连续线性空间，使用它时需要包含头文件：

```
#include <vector>
```

1.1 vector的初始化

- vector a(10);
  定义容器长度，10个整型元素
  （尖括号中为元素类型名，它可以是任何合法的数据类型），但没有给出初值，

其值是不确定的。

- vector a(10,1);
  定义了10个整型元素的向量,且给出每个元素的初值为1
- vector a(b);
  用b向量来创建a向量，整体复制性赋值
- vector a(b.begin(),b.begin+3);
  定义了a值为b中第0个到第2个（共3个）元素
- int b[7]={1,2,3,4,5,9,8}; vector a(b,b+7);
  从数组中获得初值
- vector<vector > nums ={{1, 2, 3, 4}, { 5, 6, 7, 8}, { 9, 10, 11, 12}};
- vector<vector > res(n, vector(n, 0));

1.2 vector对象的几个重要操作

- a.assign(b.begin(), b.begin()+3);
  b为向量，将b的0~2个元素构成的向量赋给a
- a.assign(4,2);
  是a只含4个元素，且每个元素为2
- a.back();
  返回a的最后一个元素
- a.front();
  返回a的第一个元素
- a[i];
  返回a的第i个元素，当且仅当a[i]存在
- a.clear();
  清空a中的元素
- a.empty();
  判断a是否为空，空则返回ture,不空则返回false
- a.pop_back();
  删除a向量的最后一个元素
- a.erase(a.begin()+1,a.begin()+3);
  删除a中第1个（从第0个算起）到第2个元素，也就是说删除的元素从a.begin()+1算起（包括它）一直到a.begin()+ 3（不包括它）
- a.push_back(5);

在a的最后一个向量后插入一个元素，其值为5

- a.insert(a.begin()+1,5);
  在a的第1个元素（从第0个算起）的位置插入数值5，如a为1,2,3,4，插入元素后为1,5,2,3,4
- a.insert(a.begin()+1,3,5);
  在a的第1个元素（从第0个算起）的位置插入3个数，其值都为5
- a.insert(a.begin()+1,b+3,b+6);
  b为数组，在a的第1个元素（从第0个算起）的位置插入b的第3个元素到第5个元素（不包括b+6），如b为1,2,3,4,5,9,8 ，插入元素后为1,4,5,9,2,3,4,5,9,8
- a.size();
  返回a中元素的个数；
- a.capacity();
  返回a在内存中总共可以容纳的元素个数
- a.resize(10);
  将a的现有元素个数调至10个，多则删，少则补，其值随机
- a.resize(10,2);
  将a的现有元素个数调至10个，多则删，少则补，其值为2
- a.reserve(100);
  将a的容量（capacity）扩充至100，也就是说现在测试a.capacity();的时候返回值是100.这种操作只有在需要给a添加大量数据的时候才 显得有意义，因为这将避免内存多次容量扩充操作（当a的容量不足时电脑会自动扩容，当然这必然降低性能）
- a.swap(b);
  b为向量，将a中的元素和b中的元素进行整体性交换
- a==b;
  b为向量，向量的比较操作还有!=,>=,<=,>,<

1.3 vector添加元素的几种方式

- 向向量a中添加元素

```
vector<int> a;
for(int i=0;i<10;i++)
    a.push_back(i);
```

- 从现有向量中选择元素向向量中添加

```
int a[6]={1,2,3,4,5,6};
vector<int> b;
vector<int> c(a,a+4);
for(vector<int>::iterator it=c.begin();it<c.end();it++)
    b.push_back(*it);
```

- 从文件中读取元素向向量中添加

```
ifstream in("data.txt");
vector<int> a;
for(int i; in>>i)
    a.push_back(i);
```

- 错误的用法【误区】

```
vector<int> a;
for(int i=0;i<10;i++)
    a[i]=i; //error
```

> 初始化的a并没有分配内存，还是个空对象，利用下标访问空对象肯定是错误的用法。但是可以利用下标访问已存在的元素。

1.4 vector读取元素的几种方式

- 通过下标方式读取

```
int a[6]={1,2,3,4,5,6};
vector<int> b(a,a+4);
for(int i=0;i<=b.size()-1;i++)
    cout<<b[i]<<" ";
```

- 通过遍历器方式读取

```
int a[6]={1,2,3,4,5,6};
```

```
vector<int> b(a,a+4);
for(vector<int>::iterator it=b.begin();it!=b.end();it++)
    cout<<*it<<" ";
```

1.5 常用到的算法

所需的头文件

```
#included <algorithm>
```

- sort(a.begin(),a.end());
  对a中的从a.begin()（包括它）到a.end()（不包括它）的元素进行从小到大排列
- reverse(a.begin(),a.end());
  对a中的从a.begin()（包括它）到a.end()（不包括它）的元素倒置，但不排列，如a中元素为1,3,2,4,倒置后为4,2,3,1
- copy(a.begin(),a.end(),b.begin()+1);
  把a中的从a.begin()（包括它）到a.end()（不包括它）的元素复制到b中，从b.begin()+1的位置（包括它）开始复制，覆盖掉原有元素
- find(a.begin(),a.end(),10);
  在a中的从a.begin()（包括它）到a.end()（不包括它）的元素中查找10，若存在返回其在向量中的位置

1.6 其他（二维数组）

vectot<vector> array

二 队列

queue

队列使用deque 或 list 为底层容器，是实现先进先出。queue 模板类也需要两个模板参数，一个是元素类型，一个容器类型，元素类型是必要的，容器类型是可选的，默认为deque 类型。
这里稍微介绍下priority_queue，是一个权值的队列，即队列中的顺序是按权值排序的，权值最高的在队头，并且只有队头有机会被外界取用。处理规则用的heap。

定义queue 对象

```
#include <queue>
queue<int> q1;
queue<double> q2;
```

queue 的基本操作有：

- 入队
  如例：q.push(x); 将x 接到队列的末端。
- 出队
  如例：q.pop(); 弹出队列的第一个元素，注意，并不会返回被弹出元素的值。
- 访问队首元素
  如例：q.front()，即最早被压入队列的元素。
- 访问队尾元素
  如例：q.back()，即最后被压入队列的元素。
- 判断队列空
  如例：q.empty()，当队列空时，返回true。
- 访问队列中的元素个数
  如例：q.size()

优先队列priority_queue

优先队列中，元素被赋予优先级。当访问元素时，具有最高优先级的元素最先删除。优先队列具有最高级先出 （ first in, largest out ）的行为特征。

> 优先队列具有队列的所有特性，包括队列的基本操作，只是在这基础上添加了内部的一个排序，它本质是一个堆实现的。

基本操作

```
top 访问队头元素
empty 队列是否为空
size 返回队列内元素个数
push 插入元素到队尾 (并排序)
emplace 原地构造一个元素并插入队列
pop 弹出队头元素
```

```
swap 交换内容
```

定义

```
priority_queue<Type, Container, Functional>
```

Type 就是数据类型，Container 就是容器类型（Container必须是用数组实现的容器，比如vector,deque等等，但不能用 list。STL里面默认用的是vector），Functional 就是比较的方式。

- 当需要用自定义的数据类型时才需要传入这三个参数，**使用基本数据类型时，只需要传入数据类型，默认是大顶堆。**

```
priority_queue <int>res(res.begin(), res.end());  //默认的大顶堆
priority_queue <int,vector<int>,less<int> >q; //大顶堆，出队列是权值最大的（默认情况下就是大顶堆），降序队列
priority_queue <int,vector<int>,greater<int> > q; 小顶堆，出队列的是权值最小的，升序队列
```

三 栈 stack

可以使用deque 和 list 为底部容器，实现先进后出的数据结构。

```
    stack<int> arr;//声明
    arr.push(5);//5入栈
    arr.top();//获取栈顶值
    arr.pop();//出栈，没有返回值
    arr.empty() //为空返回1，不为空返回0
```

四 堆 heap

STL提供的是 max-heap，heap默认的也是max-heap，实现方式是：heap 和 vector，heap为一个完全二叉树，数据按节点对应关系存储在vector容器内。heap没有迭代器，不提供遍历功能，因为所有的元素按照特别的排列规则存在vector中。建立堆make_heap()，在堆中添加数据push_heap()，在堆中删除数据pop_heap()和

堆排序sort_heap()

- 所需头文件

```
#include <algorithm>
```

4.1 函数介绍

下面的_First与_Last为可以随机访问的迭代器（指针），_Comp为比较函数（仿函数），其规则为：如果函数的第一个参数小于第二个参数应返回true，否则返回false。

- 建立堆
  make_heap(_First, _Last, _Comp)
  将现有数据转化为一个heap。默认是建立最大堆的。对int类型，可以在第三个参数传入greater()得到最小堆。
- 在堆中添加数据
  push_heap (_First, _Last)
  要先在容器中加入数据，再调用push_heap ()
- 在堆中删除数据
  pop_heap(_First, _Last)
  要先调用pop_heap()再在容器中删除数据
- 堆排序
  sort_heap(_First, _Last)
  排序之后就不再是一个合法的heap了。（我理解是底层因为是vector容器，相当于在vector中数据变有序了）

4.2 代码实现

```
#include <algorithm>

vector <int> res = {1,3,5,4,2,7};
/* 建大顶堆 */
make_heap(res.begin(), res.end()); //默认建大顶堆
//打印结果：7 4 5 3 2 1 （7位堆顶值）
/- 添加元素 -/
res.push_back(8);
```

```
push_heap(res.begin(), res.end());
//打印结果：8 4 7 3 2 1 5
/- 删除元素 -/
pop_heap(res.begin(), res.end());
res.pop_back();
//打印结果：7 4 5 3 2 1
/- 排序 -/
sort_heap(res.begin(), res.end());
//打印结果：1 2 3 4 5 7

//建小顶堆堆
make_heap(res.begin(), res.end(), greater<int>());  //注意第三个参数greater<int>() （需要头文件functional），后面加入元素删除元素都需要改参数
//打印结果：1 2 5 4 3 7  （1为堆顶值）
res.push_back(8);
push_heap(res.begin(), res.end(), greater<int>());
//打印结果：1 2 5 4 3 7 8
 pop_heap(res.begin(), res.end(), greater<int>());
res.pop_back();
//打印结果：2 3 5 4 8 7
 sort_heap(res.begin(), res.end(), greater<int>());//从大到小排序
//打印结果：8 7 5 4 3 2
```

五 list 双向链表

5.1 List定义

List是stl实现的双向链表，与向量(vectors)相比, 它允许快速的插入和删除，但是随机访问却比较慢。使用时需要添加头文件

```
#include <list>
```

这里再介绍下 slist，为单向链表，只有单向的迭代器。

```
#include <slist>
slist <int> islist;
```

5.2 List定义和初始化：

- listlst1; //创建空list
- list lst2(5); //创建含有5个元素的list
- listlst3(3,2); //创建含有3个元素的list
- listlst4(lst2); //使用lst2初始化lst4
- listlst5(lst2.begin(),lst2.end()); //同lst4

5.3 List常用操作函数：

- Lst1.assign() 给list赋值
- Lst1.back() 返回最后一个元素
- Lst1.begin() 返回指向第一个元素的迭代器
- Lst1.clear() 删除所有元素
- Lst1.empty() 如果list是空的则返回true
- Lst1.end() 返回末尾的迭代器
- Lst1.erase() 删除一个元素
- Lst1.front() 返回第一个元素
- Lst1.get_allocator() 返回list的配置器
- Lst1.insert() 插入一个元素到list中
- Lst1.max_size() 返回list能容纳的最大元素数量
- Lst1.merge() 合并两个list
- Lst1.pop_back() 删除最后一个元素
- Lst1.pop_front() 删除第一个元素
- Lst1.push_back() 在list的末尾添加一个元素
- Lst1.push_front() 在list的头部添加一个元素
- Lst1.rbegin() 返回指向第一个元素的逆向迭代器
- Lst1.remove() 从list删除元素
- Lst1.remove_if() 按指定条件删除元素
- Lst1.rend() 指向list末尾的逆向迭代器
- Lst1.resize() 改变list的大小
- Lst1.reverse() 把list的元素倒转

- Lst1.size() 返回list中的元素个数
- Lst1.sort() 给list排序
- Lst1.splice() 合并两个list

```
//初始化状态：  c1(10,20,30), c2(40,50,60)

c1.splice(++c1.begin(), c2);  //c1(10,40,50,60,20,30) c2变为空。全合并（从第二个位置全插入）
c1.splice(++c1.begin(),c2,++c2.begin()); //c1(10,50,20,30) ; c2(40,60)  指定元素合并（c2的第二个元素插入）
c1.splice(++c1.begin(),c2,++c2.begin(),c2.end());
//c1(10,50,60,20,30); c2(40)  指定范围合并（c2的第二个元素到末尾插入到c1的第二个位置）
```

- Lst1.swap() 交换两个list
- Lst1.unique() 删除list中重复的元素

关联容器

实现能快速查找（ O(log n) 复杂度）的数据结构。

- set（类似树状，唯一键的集合，按照键排序 ）

  > 不重复元素的集合。

- multiset（类似树状，键的集合，按照键排序 ）

  > 跟set具有一样的功能，但允许重复的元素。

- **map**（两个键值的树状）

  > 关联数组，每个元素含有两个数据项，map将一个数据项映射到另一个数据项中。

- multimap（两个键值的树状）

  > 跟map具有相同功能，但允许重复的键值。

- 总结：
  关联容器和非关联容器的区别在于是否根据键值进行排序。
  关联容器中含有multi则表示集合元素的键值可重复。map为键值对，set为一个键

值。

六 map

C++中map提供的是一种键值对容器，里面的数据都是成对出现的,如下图：每一对中的第一个值称之为关键字(key)，每个关键字只能在map中出现一次；第二个称之为该关键字的对应值。

6.1 声明

```
//头文件
    #include<map> //map是会根据关键字key进行排序的，通常比unordered_map容器慢
    map<int, string> ID_Name;

    // 使用{}赋值是从c++11开始的，因此编译器版本过低时会报错，如visual studio 2012
    map<int, string> ID_Name = {
    { 2015, "Jim" },
    { 2016, "Tom" },
    { 2017, "Bob" } };

#include<unordered_map>
//unordered_map不根据关键字key排序的
unordered_map<int, int> res;
res[1] = 2;
res.count(key)  //返回值为1，没有关键字key的话返回0
```

6.2 插入操作

6.2.1 使用[ ]进行单个插入

```
    map<int, string> ID_Name;
    // 如果已经存在键值2015，则会作赋值修改操作，如果没有则插入
    ID_Name[2015] = "Tom";
```

6.2.2 使用insert进行单个和多个插入

insert共有4个重载函数：

```
    // 插入单个键值对，并返回插入位置和成功标志，插入位置已经存在值时，插入失败
    pair<iterator,bool> insert (const value_type& val);

    //在指定位置插入，在不同位置插入效率是不一样的，因为涉及到重排
    iterator insert (const_iterator position, const value_type& val);

    // 插入多个
    void insert (InputIterator first, InputIterator last);

    //c++11开始支持，使用列表插入多个
    void insert (initializer_list<value_type> il);
```

下面是具体使用示例：

```
    #include <iostream>
    #include <map>
    using namespace std;
    typedef pair<char, int> pii;
    int main(){
    pii mymap;

    // 插入单个值
    mymap.insert(pii ('a', 100));
    mymap.insert(pii('z', 200));

    //返回插入位置以及是否插入成功
    std::pair<std::map<char, int>::iterator, bool> ret;
    ret = mymap.insert(pii ('z', 500));
    if (ret.second == false)
    std::cout << "element 'z' already existed" << " with a value of " << ret.first->second << '\n';

    //指定位置插入
    std::map<char, int>::iterator it = mymap.begin();
    mymap.insert(it, pii('b', 300)); //效率更高
    mymap.insert(it, pii('c', 400)); //效率非最高

    //范围多值插入
```

```
    std::map<char, int> anothermap;
    anothermap.insert(mymap.begin(), mymap.find('c'));

    // 列表形式插入
    anothermap.insert({ { 'd', 100 }, {'e', 200} });

    return 0;
    }
```

unordered_map

与map的区别

内部实现机理

- map
  map内部实现了一个红黑树，该结构具有自动排序的功能，因此map内部的所有元素都是**有序**的，红黑树的每一个节点都代表着map的一个元素，因此，对于map进行的查找，删除，添加等一系列的操作都相当于是对红黑树进行这样的操作，故红黑树的效率决定了map的效率。
- unordered_map
  unordered_map内部实现了一个哈希表，因此其元素的排列顺序是杂乱的，**无序**的

优缺点

- map
  - 优点：
    有序性，这是map结构最大的优点，其元素的有序性在很多应用中都会简化很多的操作
    红黑树，内部实现一个红黑树使得map的很多操作在的时间复杂度下就可以实现，因此效率非常的高
  - 缺点：
    空间占用率高，因为map内部实现了红黑树，虽然提高了运行效率，但是因为每一个节点都需要额外保存父节点，孩子节点以及红/黑性质，使得每一个节点都占用大量的空间
  - 适用处：

对于那些有**顺序要求**的问题，用map会更高效一些

- unordered_map
  - 优点：
    因为内部实现了哈希表，因此其查找速度非常的快
  - 缺点：
    哈希表的建立比较耗费时间
  - 适用处：
    对于**查找问题**，unordered_map会更加高效一些，因此遇到查找问题，常会考虑一下用unordered_map

代码实现

```
#include<unordered_map>
//1. 检查unordered_map<int, int> mp 是否存在键值x
mp.find(x)!=mp.end() 或 mp.count(x)!=0 //存在满足的条件

//2. 插入数据
mp.insert(Map::value_type(1,"Raoul"));

//3. 遍历mp
unordered_map <key,T>::iterator it;
(*it).first; //the key value
(*it).second; //the mapped value
for(unordered_map<key,T>::iterator iter=mp.begin(); iter!=mp.end(); iter++)
       cout<< iter->first << " " <<  iter->second;

for(auto& v : mp)
        cout << v.first <<"\t" << v.second ;
```

6.3 取值

Map中元素取值主要有at和[ ]两种操作，at会作下标检查，而[]不会。

```
    map<int, string> ID_Name;

```

```
    //ID_Name中没有关键字2016，使用[]取值会导致插入。因此，下面语句不会报错，但打印结果为空
    cout<<ID_Name[2016].c_str()<<endl;

    //使用at会进行关键字检查，因此下面语句会报错
    ID_Name.at(2016) = "Bob";
```

6.4 容量查询

```
    bool empty();// 查询map是否为空
    size_t size();// 查询map中键值对的数量
    size_t max_size(); // 查询map所能包含的最大键值对数量，和系统和应用库有关。 此外，这并不意味着用户一定可以存这么多，很可能还没达到就已经开辟内存失败了
    size_t count( const Key& key ) const;   // 查询关键字为key的元素的个数，在map里结果非0即1
```

6.5 迭代器

共有八个获取迭代器的函数：* begin, end, rbegin,rend* 以及对应的 * cbegin, cend, crbegin,crend*。
二者的区别在于，后者一定返回 const_iterator，而前者则根据map的类型返回iterator或者 const_iterator。const情况下，不允许对值进行修改。如下面代码所示：

```
    map<int,int>::iterator it;
    map<int,int> mmap;
    const map<int,int> const_mmap;

    it = mmap.begin(); //iterator
    mmap.cbegin(); //const_iterator

    const_mmap.begin(); //const_iterator
    const_mmap.cbegin(); //const_iterator
```

找值的使用例子

```
    map<int, int> maptemp;
```

```
    maptemp[2] = 22;
    auto iter = maptemp.find(2);
    if (iter != maptemp.end())
        cout << iter->first <<endl; //输出值2   cout << iter->second << endl;//输出值22
```

返回的迭代器可以进行加减操作，此外，如果map为空，则 begin = end。

6.6 删除交换

6.6.1 删除

```
    // 删除迭代器指向位置的键值对，并返回一个指向下一元素的迭代器
    iterator erase( iterator pos )

    // 删除一定范围内的元素，并返回一个指向下一元素的迭代器
    iterator erase( const_iterator first, const_iterator last );

    // 根据Key来进行删除， 返回删除的元素数量，在map里结果非0即1
    size_t erase( const key_type& key );

    // 清空map，清空后的size为0
    void clear();
```

6.6.2 交换

```
// 就是两个map的内容互换
    void swap( map& other );
```

6.7 顺序比较

```
    // 比较两个关键字在map中位置的先后
    key_compare key_comp() const;
```

示例：

```
    map<char,int> mymap;
    map<char,int>::key_compare mycomp = mymap.key_comp();

    mymap['a']=100;
    mymap['b']=200;
    mycomp('a', 'b'); // a排在b前面，因此返回结果为true
```

6.8 查找

```
    // 关键字查询，找到则返回指向该关键字的迭代器，否则返回指向end的迭代器
    // 根据map的类型，返回的迭代器为 iterator 或者 const_iterator
    iterator find (const key_type& k);
    const_iterator find (const key_type& k) const;
```

举例：

```
    std::map<char,int> mymap;
    std::map<char,int>::iterator it;

    mymap['a']=50; mymap['b']=100;  mymap['c']=150;
    if (mymap.find('b') != mymap.end())
      mymap.erase (it); // b被成功删除
```

6.9 操作符

```
operator: == != < <= > >=
注意 对于==运算符，只有键值对以及顺序完全相等才算成立。
```

无序关联容器

无序关联容器 - 提供能快速查找（均摊 O(1) ，最坏情况 O(n) 的复杂度）的无序（哈希）数据结构。

- unordered_set（唯一键的集合，按照键生成散列 ）
- unordered_multiset（键的集合，按照键生成散列）

- unordered_map（键值对的集合，按照键生成散列，键是唯一的 ）
- unordered_multimap（键值对的集合，按照键生成散列）

其他

字符串 string

- 所需头文件

```
#include <string>
using namespace std;
```

字符串的声明

```
string Str;
```

上面的声明没有传入参数，所以就直接使用了string的默认的构造函数，这个函数所作的就是把Str初始化为一个空字符串。String类的构造函数和析构函数如下：

- string s;
  生成一个空字符串s
- string s(str)
  拷贝构造函数 生成str的复制品
- string s(str,stridx)
  将字符串str内“始于位置stridx”的部分当作字符串的初值
- string s(str,stridx,strlen)
  将字符串str内“始于stridx且长度顶多strlen”的部分作为字符串的初值
- string s(cstr)
  将C字符串作为s的初值
- string s(chars,chars_len)
  将C字符串前chars_len个字符作为字符串s的初值。
- string s(num,c)
  生成一个字符串，包含num个c字符
- string s(beg,end)

以区间beg;end(不包含end)内的字符作为字符串s的初值

- s.~string()
  销毁所有字符，释放内存

字符串操作函数

- = , assign()
  赋以新值
- swap()
  交换两个字符串的内容
- += , append() , push_back()
  在尾部添加字符
- insert()
  插入字符
- erase()
  删除字符
- clear()
  删除全部字符
- replace()
  替换字符
- - 串联字符串
- ==,!=,<,<=,>,>=,compare()
  比较字符串
- size(),length()
  返回字符数量
- max_size()
  返回字符的可能最大个数
- empty()
  判断字符串是否为空，是空时返回ture，不是空时返回false
- capacity()
  返回重新分配之前的字符容量
- reserve()
  保留一定量内存以容纳一定数量的字符

- copy()
  将某值赋值为一个C_string
- c_str()
  将内容以C_string返回
- data()
  将内容以字符数组形式返回
- substr()
  返回某个子字符串
- 查找函数
- begin() end()
  提供类似STL的迭代器支持
- rbegin() rend()
  逆向迭代器
- get_allocator()
  返回配置器

```
[ ], at()
   存取单一字符
>>,getline()
   从stream读取某值
<<
   将谋值写入stream
```

C++字符串和C字符串的转换

C++提供的由C++字符串得到对应的C_string的方法是使用data()、c_str()和copy()。

- data()
  以字符数组的形式返回字符串内容，但并不添加'/0'。
- c_str()
  返回一个以‘/0'结尾的字符数组。
- copy()
  把字符串的内容复制或写入既有的c_string或 字符数组内。C++字符串并不以'/0'结尾。

> 我的建议是在程序中能使用C++字符串就使用，除非万不得已不选用c_string。

5.4 提取子串和字符串连接

提取子串的函数是：substr(),形式如下：

- s.substr();
  返回s的全部内容
- s.substr(11);
  从索引11往后的子串
- s.substr(5,6);
  从索引5开始6个字符

两个字符串结合起来的函数是+。

5.5 搜索与查找

查找函数很多，功能也很强大，包括了：

- find()
- rfind()//从后往前
- find_first_of()
- find_last_of()
- find_first_not_of()
- find_last_not_of()

这些函数返回符合搜索条件的字符区间内的第一个字符的索引，没找到目标就返回npos。

- 所有的函数的参数说明如下：
  - 第一个参数：被搜寻的对象
  - 第二个参数（可有可无）：指出string内的搜寻起点索引
  - 第三个参数（可有可无）：指出搜寻的字符个数

例子程序

```
    string s = " i love ssss!";
    int i = s.find("love",2); //i=4,第二个参数表示查找的初始位置
```

string类的迭代器处理

string类提供了向前和向后遍历的迭代器iterator，迭代器提供了访问各个字符的语法，类似于指针操作，迭代器不检查范围。

- string::iterator或string::const_iterator
  声明迭代器变量
- const_iterator
  不允许改变迭代的内容。

常用迭代器函数有：

```
    const_iterator begin()const;
    iterator begin(); //返回string的起始位置
    const_iterator end()const;
    iterator end(); //返回string的最后一个字符后面的位置
    const_iterator rbegin()const;
    iterator rbegin(); //返回string的最后一个字符的位置
    const_iterator rend()const;
    iterator rend(); //返回string第一个字符位置的前面
    rbegin和rend用于从后向前的迭代访问，通过设置迭代器
string::reverse_iterator,string::const_reverse_iterator实现
```

字符串流处理

通过定义ostringstream和istringstream变量实现

```
 #include <sstream>

    string input("hello,this is a test");
    istringstream is(input);
    string s1, s2, s3, s4;
    is >> s1 >> s2 >> s3 >>
s4;//s1="hello,this",s2="is",s3="a",s4="test"
    ostringstream os;
    os << s1 << " " << s2 << " " << s3 << " " << s4;
    cout << os.str();
```