# 序列式容器

## string

迭代器：Random Access Iterator

#include <string>

### 元素访问

可以通过迭代器或者下标法 str[i]或\*iterator

### 初始化

① string str：生成空字符串

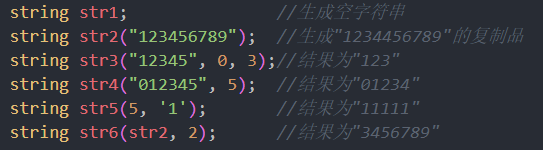
② string s(str)：生成字符串为str的复制品

③ string s(str, strbegin, strlen)：将字符串str中从下标strbegin开始、长度为strlen的部分作为字符串初值

④ string s(cstr, char\_len)：以C\_string类型cstr的前char\_len个字符串作为字符串s的初值

⑤ string s(num ,c)：生成num个c字符的字符串

⑥ string s(str, stridx)：将字符串str中从下标stridx开始到字符串结束的位置作为字符串初值



### 常用成员函数

#### begin

string.begin() 返回首元素地址

#### end

string.end() 返回最后一个元素后的地址

#### []

string [n]：返回第n个元素

#### size\length

string.size() 返回string中的元素个数，返回值是unsigned类型

#### capacity

string.capacity() 返回当前string的总的空间（包括已使用和未使用）

#### empty

string.empty() 判断string是否为空

#### max\_size

返回string对象最多包含的字符数，是一个特别大的值

#### compare

string除了可以通过比较操作符（<、>、<=、>=、==、!=）还有一个成员函数compare，返回一个int值来表示结果：0：相等、1：大于、-1：小于

str1.compare(B)：比较str1和str2

str1.compare(f1,n1,str2)：str1的第f1后的n1位与str2比较

str1.compare(f1,n1,str2,f2,n2)：str1的第f1后的n1位与str2的第f2后的n2位比较

#### push\_back

string.push\_back(x) 将元素x插入最尾端

#### insert

string.insert(pos,str) 在pos号位置插入字符串str

string.insert(it,it2,it3) 串[it2,it3)插在it位置上

#### append

string.append(str) 将str拼接在后面，还可以直接使用“+”号。append只能加字符串，而“+”号可以加字符。

string.append(n,c) 在string后面加入n个字符c

string.append(str,n) 将C型字符串str的前n个字符拼接到后面

#### clear

string.clear() 清空所有元素

#### swap

string.swap(str) 交换两个字符串

#### erase

string.erase(it) 删除it处的单个字符

string.erase(first,last) 删除[first,last)区间所有的字符

string.erase(pos,n) 删除pos位置开始的n个字符

#### replace

string.replace(pos,n,str) 将pos位置开始的n个字符替换成字符串str

string.replace(pos,n,m,c) 将pos位置开始的n个字符替换成m个字符c

string.replace(first,last,str) 将[first,last)区间所有的字符替换成字符串str

#### substr

string.substr(pos) 返回从pos位开始，到行尾的子串

string.substr(pos,len) 返回从pos位开始、长度为len的子串

#### find

string.find(str) 查找子串str，返回找到的第一次出现的位置索引，若没有找到，返回string::npos（即-1）

string.find(str,pos) 从pos位置开始查找子串str，返回找到的位置索引

#### rfind

string.find(str) 反向查找子串str，返回找到的第一次出现的位置索引，若没有找到，返回string::npos（即-1）

string.find(str,pos) 从pos位置开始反向查找子串str，返回找到的位置索引

#### find\_first\_of

string.find\_first\_of(str) 查找str里的字符位于string中的第一个位置，返回找到的位置索引

string.find\_first\_of(str,pos) 从pos位置开始查找str里的字符位于string中的第一个位置，返回找到的位置索引

#### find\_first\_not\_of

string.find\_first\_not \_of(str) 查找不是str里的字符位于string中的第一个位置，返回找到的位置索引

string.find\_first\_not \_of(str,pos) 从pos位置开始查找不是str里的字符位于string中的第一个位置，返回找到的位置索引

#### find\_last\_of

string.find\_last\_of(str) 查找str里的字符位于string中的最后一个位置，返回找到的位置索引

string.find\_last\_of(str,pos) 从pos位置开始查找str里的字符位于string中的最后一个位置，返回找到的位置索引

#### find\_last\_not \_of

string.find\_last\_not \_of(str) 查找不是str里的字符位于string中的最后一个位置，返回找到的位置索引

string.find\_last\_not \_of(str,pos) 从pos位置开始查找不是str里的字符位于string中的最后一个位置，返回找到的位置索引

#### resize

string.resize(new\_size,x)：调整string大小为new\_size，若当前大小超过new\_size，删掉超过的元素，若不超过，在末尾加上x直至大小调整为new\_size

## vector

迭代器：Random Access Iterator，支持++、--、+=、-=、+n、-n

#include <vector>

### 元素访问

可以通过迭代器或者下标法 vector[i]或\*iterator

### 初始化

① vector<T> v1：生成空的容器

② vector<T> v1(v2)：创建v1，包含v2的所有元素

③ vector<T> v1 = v2：创建v1，包含v2的所有元素

④ vector<T> v1(n)：创建v1，里面包含n个0

⑤ vector<T> v1(n, val)：创建v1，里面包含n个val

⑥ vector<T> v1{a,b,c,…}：创建v1，每个元素被赋予了初值

⑦ vector<T> v1 = {a,b,c,…}：创建v1，每个元素被赋予了初值

⑧ vector<T> v1(first, last)：声明一个vector，其元素的初始值来源于由区间所指定的序列中的元素，first和last是迭代器

### 常用成员函数

#### begin

vector.begin() 返回首元素地址

#### end

vector.end() 返回最后一个元素后的地址

#### rbegin

vector的最后一个元素的位置，执行++操作相当于变为后退--

#### rend

vector的第一个元素的前一个位置，类似于end()，没有元素

#### 比较操作符

比较两个vector

#### []

vector[n]：返回第n个元素

#### size

vector.size() 返回vector中的元素个数，返回值是unsigned类型

#### capacity

vector.capacity() 返回当前vector的总的空间（包括已使用和未使用）

#### empty

vector.empty() 判断vector是否为空

#### front

vector.front() 返回第一个元素

#### back

vector.back() 返回最后一个元素

#### push\_back

vector.push\_back(x) 将元素x插入最尾端

#### pop\_back

vector.pop\_back()将最尾端元素取出

#### clear

vector.clear() 清空所有元素

#### insert

vector.insert(it,x)：在it处插入元素x

vector.insert(it,n,x)：在it处插入n个元素x

#### erase

vector.erase(it)：删除it处的单个元素

vector.erase(first,last)：删除[first,last)区间所有的元素

#### resize

vector.resize(new\_size,x)：调整容器大小为new\_size，若当前大小超过new\_size，删掉超过的元素，若不超过，在末尾插入x直至大小调整为new\_size

## list

迭代器：Bidirectional Iterator，支持++、--，不支持+=、-=、+n、-n

#include <list>

### 元素访问

只能通过迭代器来访问

### 初始化

① list<T> l：创建一个空的list

② list<T> l(n)：创建一个有n个元素的list，每一个元素都为0

③ list<T> l(n,m)：创建一个有n个元素的list，每一个元素都为m

④ list<T> l(first,last)：声明一个list，其元素的初始值来源于由区间所指定的序列中的元素，first和last是迭代器

⑤ list<T> l{a,b,c,…}：创建一个list，每一个元素都被赋予了初值

⑥ list<T> l(li)：创建一个list的复制

### 常用成员函数

#### begin

list.begin() 返回首元素地址

#### 比较操作符

比较两个list

#### end

list.end() 返回最后一个元素后的地址

#### rbegin

list的最后一个元素的位置，执行++操作相当于变为后退--

#### rend

list的第一个元素的前一个位置，类似于end()，没有元素

#### empty

list.empty() 判断list是否为空

#### size

list.size() 返回list中的元素个数

#### front

list.front() 返回第一个元素

#### back

list.back() 返回最后一个元素

#### push\_front

list.push\_front(x) 将元素x插入到list头端（通过调用insert实现）

#### push\_back

list.push\_back(x) 将元素x插入到list尾端（通过调用insert实现）

#### insert

list.insert(it,x)：在it处插入元素x

list.insert(it,n,x)：在it处插入n个元素x

list.insert(it, first,last)：在it处插入[first,last)之间的元素

#### erase

list.erase(it) 移除迭代器it所指节点

#### pop\_front

list.pop\_front() 移除头节点（通过调用erase实现）

#### pop\_back

list.pop\_back() 移除尾节点（通过调用erase实现）

#### clear

list.clear() 清除整个链表

#### remove

list.remove(x) 将数值为x的所有元素移除

#### unique

list.unique() 移除数值相同的连续元素（只有“连续而相同的元素”才会被移除至只剩一个）

#### splice

list.splice(it, li) 将列表li接合于it所指位置之前；

list.splice(it, li, first, last) 将列表li的[first, last)内的所有元素接合于it所指位置之前，it不能位于[first, last)内；

list.splice(it, li, po) 将列表li的po所指元素接合于it所指位置之前；

#### merge

list.merge(li) 将列表li合并到列表list里面，前提是两个列表均递增有序

#### reverse

list.reverse() 将列表内的元素逆向重置

#### sort

list.sort() 将列表内的元素排序，采用的是快排

#### swap

list.swap(li) 实现两个列表内容互换

## deque

迭代器：Random Access Iterator，支持++、--、+=、-=、+n、-n

#include <deque>

### 元素访问

可以通过迭代器或者下标法 deque[i]或\*iterator

### 初始化

① deque<T> dq：创建一个空的deque

② deque<T> dq(dq1)：创建一个deque，dq1的复制

③ deque<T> dq = dq1：创建一个deque，dq1的复制

④ deque<T> dq(n)：创建一个包含n个0元素的deque

⑤ deque<T> dq(n,m)：创建一个包含n个m的deque

⑥ deque<T> dq(first,last)：创建一个包含[first,last)区间元素的deque

⑦ deque<T> dq({a,b,c,…})：创建一个deque，每个元素都被赋予了初值

⑧ deque<T> dq = {a,b,c,…}：创建一个deque，每个元素都被赋予了初值

### 常用成员函数

#### begin

deque.begin() 返回首元素地址

#### 比较操作符

比较两个deque

#### end

deque.end() 返回最后一个元素后的地址

#### rbegin

deque的最后一个元素的位置，执行++操作相当于变为后退--

#### rend

deque的第一个元素的前一个位置，类似于end()，没有元素

#### []

deque[n]：返回第n个元素

#### front

deque.front() 返回第一个元素

#### back

deque.back() 返回最后一个元素

#### size

deque.size() 返回deque中的元素个数

#### empty

deque.empty() 判断deque是否为空

#### max\_size

deque.max\_size() deque所允许的最大元素数，是一个很大的数值

#### push\_front

deque.push\_front(x) 将元素x插入到deque头端

#### push\_back

deque.push\_back(x) 将元素x插入到deque尾端

#### pop\_front

deque.pop\_front() 移除头部元素

#### pop\_back

deque.pop\_back() 移除尾部元素

#### clear

deque.clear() 清除整个deque

#### erase

deque.erase(it) 移除迭代器it所指节点

deque.erase(first,last) 移除[first,last)之间所指所有元素

#### insert

deque.insert(it,x)：在it处插入元素x

deque.insert(it,n,x)：在it处插入n个元素x

deque.insert(it, first,last)：在it处插入[first,last)之间的元素

## stack

迭代器：没有迭代器

stack以deque作为底层容器，也可以list作为底层容器

#include <stack>

### 元素访问

只能通过stack.top()来访问栈顶元素

### 初始化

① stack<T> st：创建一个空的stack

② stack<T> st(st1)：创建一个stack，st1的复制

### 常用成员函数

#### empty

stack.empty() 判断stack是否为空

#### 比较操作符

比较两个栈

#### size

stack.size() 返回stack中的元素个数

#### top

stack.top() 返回栈顶元素

#### push

stack.push(x) 将元素x压入栈中

#### pop

stack.pop() 将栈顶元素出栈

## queue

迭代器：没有迭代器

queue以deque作为底层容器，也可以list作为底层容器

#include <queue>

### 元素访问

只能通过queue.front()和queue.back()来访问队首和队尾元素

### 初始化

① queue<T> q：创建一个空的queue

② queue <T> q(Q)：创建一个queue，Q的复制

### 常用成员函数

#### empty

queue.empty() 判断queue是否为空

#### 比较操作符

比较两个队列

#### <

比较两个队列大小

#### size

queue.size() 返回queue中的元素个数

#### front

queue.front() 返回队列首部元素

#### push

queue.push(x) 将元素x加到队尾

#### pop

queue.pop() 将队首元素出队

## priority\_queue

没有迭代器，优先队列，递减排序，由一个大顶堆完成，后者是一个vector表现的完全二叉树

#include <queue>

### 元素访问

只能通过priority\_queue.top()来访问堆顶元素

### 初始化

① priority\_queue<T> pq：创建一个空的priority\_queue

② priority\_queue<T> pq1(pq)：创建一个priority\_queue，pq的复制

③ priority\_queue<T> pq(first,last)：声明一个priority\_queue，其元素的初始值来源于由区间所指定的序列中的元素，first和last是迭代器

④ priority\_queue<T> pq(first,last,comp)：声明一个priority\_queue，其元素的初始值来源于由区间所指定的序列中的元素，first和last是迭代器，排序原则是comp

### 常用成员函数

#### empty

priority\_queue.empty() 判断priority\_queue是否为空

#### size

priority\_queue.size() 返回priority\_queue中的元素个数

#### push

priority\_queue.push(x) 将元素x加入队列

#### pop

priority\_queue.pop() 将队首元素出队

#### top

priority\_queue.top() 返回队首元素

## slist(forward\_list)

迭代器：Forward Iterator，只支持++，没有--

#include < forward\_list>

### 元素访问

只能通过迭代器来访问

### 初始化

① forward\_list <T> l：创建一个空的forward\_list

② forward\_list <T> l(n)：创建一个有n个元素的forward\_list，每一个元素都为0

③ forward\_list <T> l(n,m)：创建一个有n个元素的forward\_list，每一个元素都为m

④ forward\_list <T> l(first,last)：声明一个forward\_list，其元素的初始值来源于由区间所指定的序列中的元素，first和last是迭代器

⑤ forward\_list <T> l{a,b,c,…}：创建一个forward\_list，每一个元素都被赋予了初值

⑥ forward\_list <T> l(li)：创建一个forward\_list的复制

### 常用成员函数

#### begin

slist.begin() 返回首元素地址

#### end

slist.end() 返回最后一个元素后的地址

#### empty

slist.empty() 判断slist是否为空

#### size

slist.size() 返回slist中的元素个数

#### swap

slist.swap(sl) 两个slist互换，（本质是交换的头结点）

#### front

slist.front() 返回列表头部元素

#### push\_front

slist.push\_front(x) 将元素x插入到slist头部

#### pop\_front

slist.pop\_front() 删除头部元素

#### insert

slist.insert(it,x)：在it处插入元素x

slist.insert(it,n,x)：在it处插入n个元素x

slist.insert(it, first,last)：在it处插入[first,last)之间的元素

# 二、关联式容器

关联式容器地内部结构一般是一个平衡二叉树

## set

所有元素会根据键值自动排序，不允许通过set的迭代器改变set的元素值，set iterator是一种constant iterator，set使用RB-tree作为底层容器，也有使用hash-table作为底层容器的，以hash-table作为底层的hashset元素并非自动排序的。

#include <set>（RB-tree）

#include <unordered\_set>（hash\_table）

### 初始化

① set<T> s：初始化一个空的set

② set<T> s(first,last)：声明一个set，其元素的初始值来源于由区间所指定的序列中的元素

③ set<T> s(first,last,comp)：声明一个set，其元素的初始值来源于由区间所指定的序列中的元素，排序规则为comp

④ set<T> s(s1)：初始化一个set，是s1的复制

### 常用成员函数

#### begin

set.begin() 返回首元素地址

#### end

set.end() 返回最后一个元素后的地址

#### empty

set.empty() 判断set是否为空

#### size

set.size() 返回set中的元素个数

#### swap

set.swap(sl) 两个set互换，（本质是交换的头结点）

#### max\_size

set.max\_size() set所允许的最大元素数，是一个很大的数值

#### insert

set.insert(x)：插入元素x，并返回pair<set<T>::iterator,bool>的结果，而multiset返回迭代器

set.insert(it, x)：插入元素x，并返回一个迭代器

set.insert(first,last)：插入[first,last)之间的元素

#### equal\_range()

set.equal\_range(x) 返回集合中与给定值x相等的上下限的两个迭代器

#### erase

set.erase(it) 移除迭代器it处的元素

set.erase(first,last) 移除[first,last)之间所有元素

set.erase(x) 移除键值为x的元素，并返回个数

#### clear

set.clear() 清除所有元素

#### count()

set.count(x) 返回set中x的个数

#### find()

set.find(x) 返回值为x的元素的迭代器

#### lower\_bound()

set. lower\_bound(x) 返回指向大于（或等于）x的第一个元素的迭代器

#### upper\_bound()

set. upper\_bound() 返回指向大于x的第一个元素的迭代器

#### rbegin()

set. rbegin() 返回集合最后一个元素的反向迭代器

#### rend()

set.rend() 返回集合第一个元素的反向迭代器

#### key\_comp()

set.key\_comp() 返回一个用于元素间键比较的函数

#### value\_comp()

set.value\_comp() 返回一个用于元素间值比较的函数

## map

map不允许两个元素具有相同的键值，可以修改map的元素实值value，不能修改键值key。map使用RB-tree作为底层容器，也有使用hash-table作为底层容器的，以hash-table作为底层的hashmap元素并非自动排序的。

map的iterator既不是constant iterator，也不是mutable iterator

#include <map>（RB-tree）

#include <unordered\_map>（hash\_table）

### 初始化

① map<T> m：初始化一个空的map

② map <T> m(first,last)：声明一个map，其元素的初始值来源于由区间所指定的序列中的元素

③ map <T> m(first,last,comp)：声明一个map，其元素的初始值来源于由区间所指定的序列中的元素，排序规则为comp

④ map <T> m(m1)：初始化一个map，是m1的复制

### 常用成员函数

#### begin

map.begin() 返回首元素地址

#### end

map.end() 返回最后一个元素后的地址

#### []

map [key]：返回key的value值

#### empty

map.empty() 判断map是否为空

#### size

map.size() 返回map中的元素个数

#### swap

map.swap(sl) 两个map互换，（本质是交换的头结点）

#### max\_size

map.max\_size() map所允许的最大元素数，是一个很大的数值

#### insert

map.insert(x)：插入元素x，并返回pair<set<T>::iterator,bool>的结果，而multimap返回迭代器

map.insert(it, x)：插入元素x，并返回一个迭代器

map.insert(first,last)：插入[first,last)之间的元素

#### equal\_range()

map.equal\_range(x) 返回集合中与给定值x相等的上下限的两个迭代器

#### erase

map.erase(it) 移除迭代器it处的元素

map.erase(first,last) 移除[first,last)之间所有元素

map.erase(x) 移除键值为x的元素，并返回个数

#### clear

map.clear() 清除所有元素

#### count()

map.count(x) 返回map中x的个数

#### find()

map.find(x) 返回值为x的元素的迭代器

#### lower\_bound()

map. lower\_bound(x) 返回指向大于（或等于）x的第一个元素的迭代器

#### upper\_bound()

map. upper\_bound() 返回指向大于x的第一个元素的迭代器

#### rbegin()

map. rbegin() 返回集合最后一个元素的反向迭代器

#### rend()

map.rend() 返回集合第一个元素的反向迭代器

#### key\_comp()

set.key\_comp() 返回一个用于元素间值比较的函数

#### value\_comp()

map.value\_comp() 返回一个用于元素间值比较的函数

## multiset、multimap

特性及用法与set、map完全相同，唯一差别在于允许键值重复，insert采用的是RB-tree里的insert\_equal。

#include <multiset>

#include <multimap>

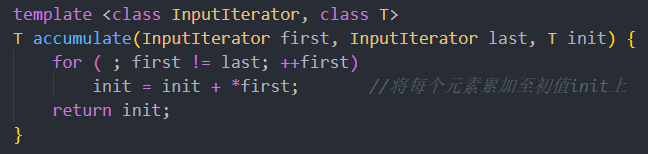
# 三、算法

## 数值算法

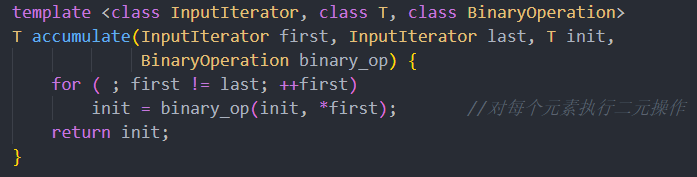
#include <numeric>

### accumulate元素累计

//版本1



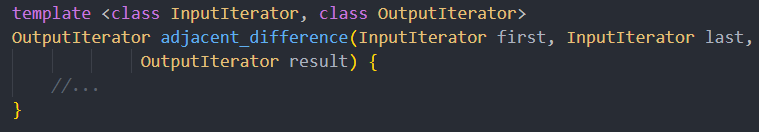
//版本2



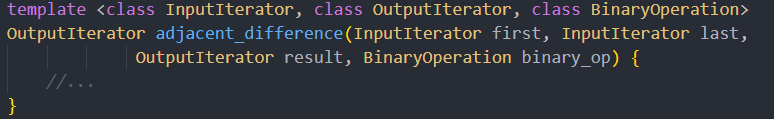
算法accumulate用来计算init和[first, last)内所有元素的总和，先将**init初始化**，然后针对[first, last)区间中的每一个迭代器i，依序执行init = init + \*i（第1版本）或init = binary\_op(init, \*i)（第2版本）。

### adjacent\_difference相邻元素差额

//版本1



//版本2

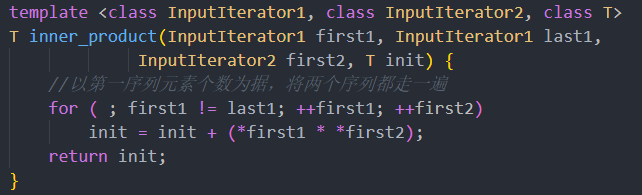


**质变**算法adjacent\_difference计算[first, last)内相邻元素的差额。第1版本使用operator-来计算差额，第2版本采用外界提供的二元仿函数。第1版本针对[first, last)区间中的每一个迭代器i，将\*i - \*(i - 1)赋值给\*(result + (i - first))，第2个版本则是将binary\_op(\*i, \*(i - 1))的运算结果赋值给\*(result + (i - first))。

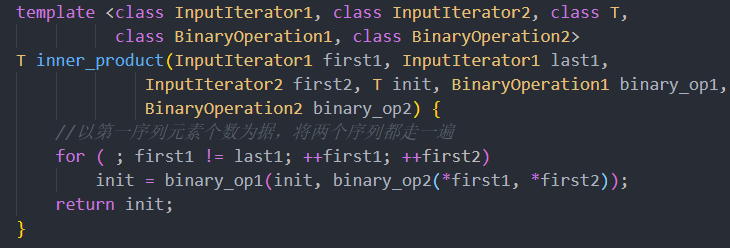
**注意：**adjacent\_difference返回的是一个输出迭代器，比可以设置这样一个输出迭代器传入adjacent\_difference中（ostream\_iterator<int> oite(cout, “ ”)）。而且first表示的第1个元素保留原值，从第2个元素开始进行差额处理

### inner\_product内积

//版本1



//版本2



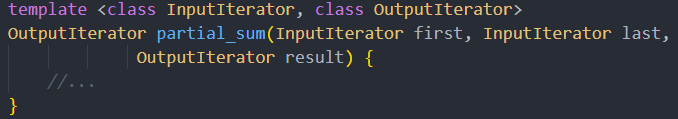
算法inner\_product计算[first1, last1)和[first2, first2 + (last1 – first1))的一般内积。注意，一定得提供初值init。

第1个版本会将两个区间的内积结果加上init。也就是说，先将结果初始化为init，然后针对[first1, last1)的每一个迭代器i，由头至尾依序执行init = init + (\*i) × \*( first2 + (last1 – first1))。

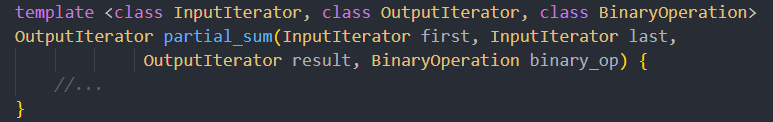
第2个版本首先将结果初始化为init，然后针对[first1, last1)的每一个迭代器i，由头至尾依序执行init = binary\_op1(init , binary\_op2( (\*i), \*( first2 + (last1 – first1) ) ) )。

### partial\_sum局部求和

//版本1



//版本2

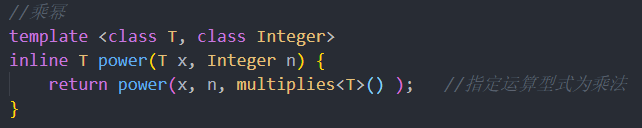


**质变**算法partial\_sum计算局部总和，它与adjacent\_difference一样，返回一个输出迭代器， 且第一个元素均保留原值。运算总和首先初始化为\*first，然后赋值给\*result，对于[first + 1, last)中每个迭代器i，从头至尾依序执行sum = sum + \*i（第1个版本）或sum = binary\_op(sum, \*i)（第2个版本），再将sum赋值给\*(result + (i - first))。

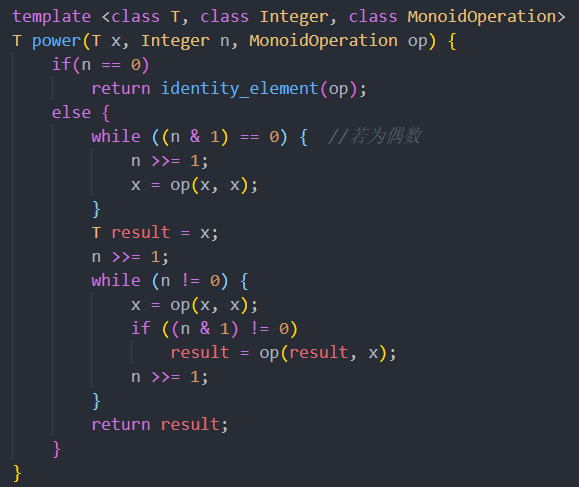
如果加法和减法的定义一如常规定义，那么partial\_sum与之前的adjacent\_difference互为逆运算。这里的意思是，如果对区间1,2,3,4,5执行partial\_sum，获得结果1,3,6,10,15，再对此结果执行adjacent\_difference，便会获得原始区间1,2,3,4,5。

### power 幂运算

//版本1，不指定运算类型，默认为乘幂

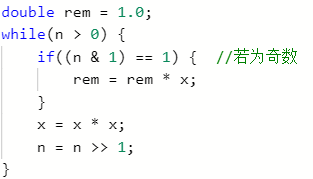


//版本2，指定运算类型为MonoidOperation



默认计算x的n次幂，运算类型可由外界指定，版本2可作为自己实现乘幂运算的模板

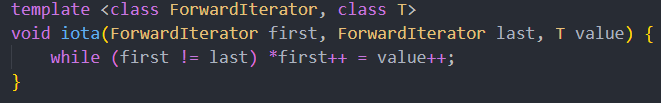
更简单明了的实现乘幂的代码可以如下



如果有负次幂，那么取倒数即可。

### iota

//版本



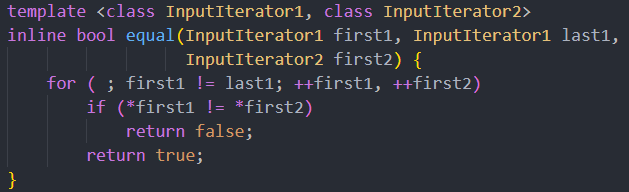
**质变**算法iota用来设定某个区间的内容，使其内的每一个元素从指定的value值开始，呈现递增状态。

## 基本算法

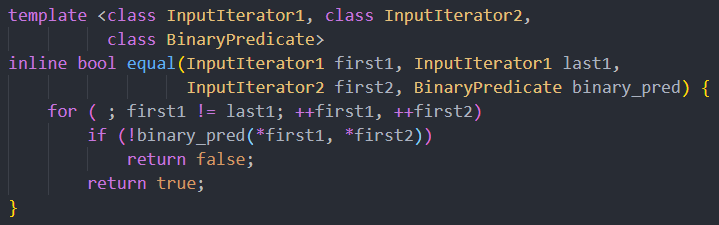
#include <algorithm>

### equal相等

//版本1



//版本2

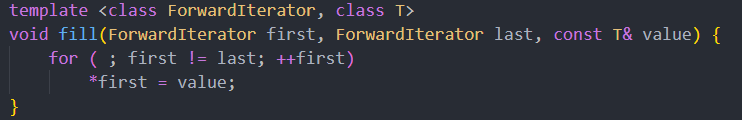


算法equal判断两个区间相等于否。如果两个序列在[first, last)区间内相等就返回true，如果第二个序列元素较多，多出来的元素就不予考虑。如果第二序列元素个数比第一序列元素个数少，会造成不可预知的危险。

第1个版本缺省采用equality操作符来进行大小比较。第2个版本允许我们指定仿函数pred作为比较依据。

### fill改填元素值

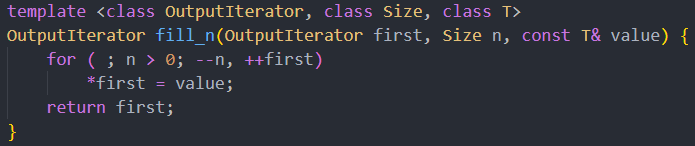
//版本



算法fill将[first, last)区间内的所有元素改填新值value。

### fill\_n改填元素值，n次

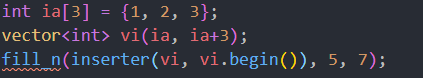
//版本



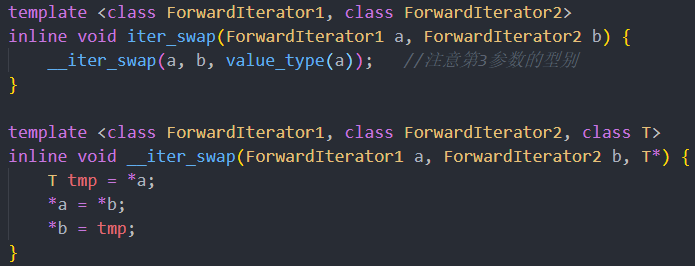
算法fill\_n将[first, last)区间内的前n个元素改填为新值value，返回的迭代器为被填入的最后一个元素的下一个位置。

如果n超过了容器大小，会造成怎样的结果呢？

通过源码可知，每次迭代进行的是assignment操作，是一种覆写（overwrite）操作，所以一旦操作区间超过容器大小，就会造成不可预知的结果。**解决办法之一**是，利用inserter()产生一个具有插入（insert）而非覆写（overwrite）能力的迭代器。inserter()可产生一个来修饰迭代器的配接器，用法如下：

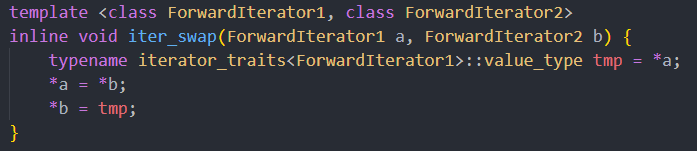


### iter\_swap元素互换



算法iter\_swap将两个ForwardIterator所指的对象对调。

注意：value\_type(a)是获取迭代器a的型别，然后\_\_iter\_swap函数第三个参数是T\*，这刚好对应迭代器型别，在value\_type函数的定义里面，实际上是取iterator\_traits<Iterator>::value\_type\*，也就是说迭代器的value\_type实际上是萃取机萃取出的value\_type的指针形式。所以上述双层构造可以改写成如下形式：



### lexicographical\_compare

以“字典排列方式”对两个序列[first1, last1)和[first2, last2)进行比较。比较操作针对两序列中的对应位置上的元素进行，并持续到（1）某一组对应元素彼此不相等；（2）同时到达last1和last2（即两序列元素个数相同，且一样）；（3）到达last1或last2（即两序列元素个数不同，但长度较短的部分元素相同）。

1. 如果到达last1而未到达last2，返回true；
2. 如果到达last2而未到达last1，返回false；
3. 如果同时到达last1和last2，返回false；
4. 如果出现某一组元素不相等，若第一序列元素较小，返回true，否则false。

### mismatch

用来平行比较两个序列，指出两者之间的第一个不匹配点。返回一对迭代器，分别指向两序列中的不匹配点。如果第二序列的元素个数比第一序列多，多出来的元素忽略不计。如果第二序列的元素个数比第一序列少，发生不可预期的行为。



### swap

用来交换两个对象的内容。

### copy

将输入区间[first, last)内的元素复制到输出区间[result, result + (last – first))。返回一个迭代器：result + (last – first)。如果输出区间的起点与输入区间重叠，有可能出现错误结果。但只是“有可能”，因为copy算法根据接收的迭代器特性决定调用memmove()来执行任务，就不会出现错误，因为memmove()会先将整个输入区间的内容复制下来，没有被覆盖的危险。

### copy\_backward

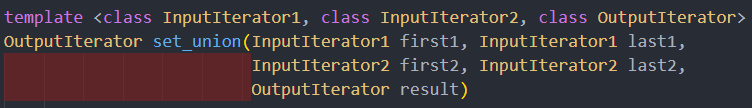
将输入区间[first, last)内的元素以逆行的方向复制到以result – 1为起点，方向亦是逆行的区间上。返回一个迭代器：result - (last – first)。copy\_backward算法可以解决copy算法的问题，因为copy算法输出区间的起点如果与输入区间重叠，就意味着输出区间的终点就一定不在输入区间内，而copy算法输出区间的终点可以视为copy\_backward算法输出区间的起点。

## set相关算法

### set\_union

算法set\_union可构造S1、S2的并集，但这个并集与直观的不太一样，返回的是一个迭代器，指向输入区间的尾端。由于S1和S2内每个元素都不需唯一，因此，如果某个值在S1出现n次，在S2出现m次，那么该值在输出区间会出现max(m, n)次。

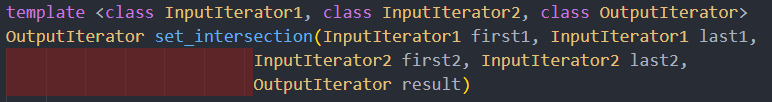
例如S1 = {1，3，5，7，9，11}，S2 = {1，1，2，3，5，8，13}，并集的结果是{1，1，2，3，5，7，8，9，11，13}。



### set\_intersection

算法set\_ intersection可构造S1、S2的交集，但这个交集与直观的不太一样，返回的是一个迭代器，指向输入区间的尾端。由于S1和S2内每个元素都不需唯一，因此，如果某个值在S1出现n次，在S2出现m次，那么该值在输出区间会出现min(m, n)次。

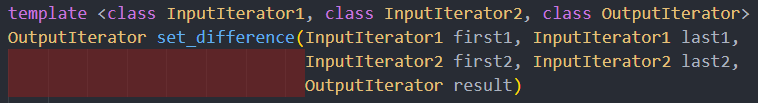
例如S1 = {1，3，5，7，9，11}，S2 = {1，1，2，3，5，8，13}，并集的结果是{1，3，5}。



### set\_difference

算法set\_ difference可构造S1、S2的差集，即S1 – S2。但这个差集与直观的不太一样。返回的是一个迭代器，指向输入区间的尾端。由于S1和S2内每个元素都不需唯一，因此，如果某个值在S1出现n次，在S2出现m次，那么该值在输出区间会出现max(n - m, 0)次。

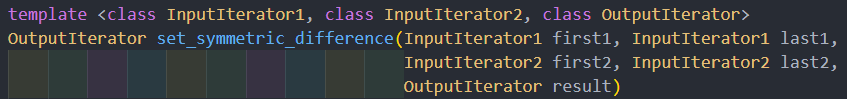
例如S1 = {1，3，5，7，9，11}，S2 = {1，1，2，3，5，8，13}，S1 – S2的结果是{7，9，11}，S2 – S1的结果是{1，2，8，13}。



### set\_symmetric\_difference

算法set\_ symmetric\_difference可构造S1、S2的对称差集（**去掉两个集合相等的元素**），即出现在，但不出现在中的集合。这个对称差集与直观的不太一样，返回的是一个迭代器，指向输入区间的尾端。由于S1和S2内每个元素都不需唯一，因此，如果某个值在S1出现n次，在S2出现m次，那么该值在输出区间会出现|m - n|次。

例如S1 = {1，3，5，7，9，11}，S2 = {1，1，2，3，5，8，13}，对称差集的结果是{1，2，7，8，9，11，13}。



## 其他算法

### 单纯的数据处理

#### adjacent\_find

找出[first, last)区间内的第一组满足条件的相邻元素。这里所谓的“条件”，在版本1中是指“两元素相等”，在版本2中允许用户指定一个二元运算。返回的是第一个元素的迭代器。如果first == last，则返回last。



#### count

运用equality操作符，将[first, last)区间内的每一个元素拿来和指定值value比较，并返回与value相等的元素个数。

#### count\_if

将指定操作pred实施于[first, last)区间内的每一个元素上，并返回“造成pred计算结果为true”的所有元素的个数，即返回满足pred条件的所有元素的个数。

#### find

运用equality操作符，循序查找[first, last)区间内的每一个元素，找出满足第一个匹配等同条件的元素，并返回该元素的迭代器，否则，返回last。

#### find\_if

根据指定的pred运算条件，循序查找[first, last)区间内的每一个元素，找出第一个零pred运算结果为true的元素，并返回该元素的迭代器，否则，返回last。

#### find\_end

在序列一[first1, last1)的区间内，查找序列二[first2, last2)的最后一次出现的位置，返回序列二最后一次出现的位置中第一个元素的位置，如果找不到，返回last1。

版本一使用equality操作符比较元素是否相等，版本二使用用户指定的某个二元运算作为判断元素相等与否的依据。

#### find\_first\_of

以序列二[first2, last2)区间内的某些元素作为查找目标，寻找它们在序列一[first1, last1)区间内的第一次出现（满足条件）的地点，即序列二的任意元素第一次出现（满足条件）在序列一中的位置，找不到则返回last1。版本一是出现，版本二是满足给定条件。

#### for\_each

将仿函数f施行于[first, last)区间内每一个元素身上。f不可以改变元素内容，因为first和last都是InputIterator，不保证接受赋值行为。如果想要修改内容，应该使用算法transform()，f可以返回一个值，但该值会被忽略。



#### generate

将仿函数gen的运算结果填写在[first, last)区间内的所有元素身上。所谓填写，用的是迭代器所指元素之assignment操作符。

//generate(first, last, gen)

#### generate\_n

将仿函数gen的运算结果填写在first迭代器开始的n个元素身上。所谓填写，用的是迭代器所指元素之assignment操作符。

//generate(first, n, gen)

#### includes（应用于有序区间）

判断序列二S2是否“涵盖于”序列一S1。S1和S2必须都是有序序列，且元素可重复。所谓涵盖就是指“S2的每一元素都出现在S1中”，注意这里的涵盖的意思是说只要S1有S2的所有元素即可，而不必关心元素是否是连续的，即S1有S2的所有元素，却不一定有S2这一序列。

假设某元素在S1中出现了m次，在S2中出现了n次，若m < n，则必定返回false。

// includes(s1.begin(), s1.end(), s2.begin(), s2.end())

// includes(s1.begin(), s1.end(), s2.begin(), s2.end(), less<int>())

// includes(s1.begin(), s1.end(), s2.begin(), s2.end(), less<greater>())

#### max\_element

返回一个迭代器，指向序列之中数值最大的元素。

max\_element(first, last)

#### merge（应用于有序区间）

将两个经过排序的集合S1和S2，合并起来置于另一段空间，所得结果也是一个有序序列。返回一个迭代器，指向最后结果的最后一个元素的下一位置。

merge(first1, last1, first2, last2, output)

#### min\_element

返回一个迭代器，指向序列之中数值最小的元素。

#### partition

会将区间[first, last)中的元素重新排列，所有被一元条件pred判定为true的元素，都会被放在区间的前段，被判定为false的元素都会被放在区间的后段。这个算法不保证保留元素的原始相对位置。如果需要保证原始相对位置，应使用stable\_partition。

// partition(first, last, pred)

#### remove移除（但不删除）

移除[first, last)中所有与value相等的元素。这一算法并不是真正从容器中删除那些元素（换句话说容器大小并未改变），而是将每一个不与value相等的元素番赋值给first之后的空间。返回值ForwardIterator标示出重新整理后的最后元素的下一位置。

例如序列{0，1，0，2，0，3，0，4}，如果我们执行remove()，希望移除所有0值元素，执行结果将是{1，2，3，4，0，3，0，4}，返回值ForwardIterator指向第5个元素。如果想要删除那些残余的数据，可以将返回的迭代器交给erase()函数。

注意array不适合使用remove()和remove\_if()，因为array无法缩小尺寸，导致残余数据永远存在。对array而言，比较受欢迎的算法是remove\_copy()和remove\_copy\_if()。

#### remove\_copy

移除[first, last)区间内所有与value相等的元素。这一算法并不是真正从容器中删除那些元素（换句话说原容器没有任何改变），而是将结果复制到一个以result标识起始位置的容器身上。新容器可以与原容器重叠（将result设置为first即可），但如果对新容器实际给值时，超越了旧容器的大小，会产生无法预知的错误。返回值OutputIterator指出被复制的最后元素的下一位置。

#### remove\_if

移除[first, last)区间内所有被仿函数pred判定为true的元素。这一算法并不是真正从容器中删除那些元素（换句话说容器大小并未改变），而是将每一个不符合pred条件的元素番赋值给first之后的空间。返回值ForwardIterator标示出重新整理后的最后元素的下一位置。

#### remove\_copy\_if

移除[first, last)区间内所有被仿函数pred判定为true的元素。这一算法并不是真正从容器中删除那些元素（换句话说原容器没有任何改变），而是将结果复制到一个以result标识起始位置的容器身上。新容器可以与原容器重叠（将result设置为first即可），但如果对新容器实际给值时，超越了旧容器的大小，会产生无法预知的错误。返回值OutputIterator指出被复制的最后元素的下一位置。

#### replace

将[first, last)区间内所有的old\_value都以new\_value替代。没有返回值。

// replace(first, last, old, new)

#### replace\_copy

行为与replace类似，唯一不同的是新序列会被复制到result所指的容器中，返回值OutputIterator指向被复制的最后元素的下一位置。原序列没有任何改变。

#### replace\_if

将[first, last)区间内所有“被pred评估为true”的元素都以new\_value替代。没有返回值。

#### replace\_copy\_if

行为与replace\_if类似，唯一不同的是新序列会被复制到result所指的容器中，返回值OutputIterator指向被复制的最后元素的下一位置。原序列没有任何改变。

#### reverse

将序列[first, last)的元素在原容器中颠倒重排。

#### reverse\_copy

行为类似reverse，但产生出来的新序列会被置于result指出的容器中，返回值OutputIterator指向被复制的最后元素的下一位置。原序列没有任何改变。

#### rotate

将[first,middle)和[middle, last)内的元素互换，middle所指的元素会成为容器的第一个元素。看起来和swap\_ranges()功能近似，但swap\_ranges()只能交换两个长度相等的区间，rotate()可以交换两个长度不同的区间。

#### rotate\_copy

行为类似rotate，但产生出来的新序列会被置于result指出的容器中，返回值OutputIterator指向被复制的最后元素的下一位置。原序列没有任何改变。

由于它不需要就地在原容器中调整内容，实现上也就简单得多。旋转操作其实只是两段元素彼此交换，所以只要先把后段的复制到新容器的前端，再把前段接续复制到新容器即可。

#### search

在序列一[first1, last1)所涵盖的区间中，查找序列二[first2, last2)的首次出现地点。如果查找到，返回序列二首元素位置，如果序列一内不存在与序列二完全匹配的子序列，便返回迭代器last1。

版本一使用equality操作符，版本二使用用户指定的某个二元运算作为判断相等与否的依据。

#### search\_n

在序列[first, last)所涵盖的区间中，查找“连续count个符合条件之元素”所形成的子序列，并返回一个迭代器指向该孩子序列起始处。如果找不到就返回last。版本一中式相等条件，版本二是某个指定的二元运算。

比如在序列iv{10，8，8，7，2，8，7，2，2，8，7，0}中查找“连续两个8”所形成的子序列起点，可以这么写：

iter1 = search\_n(iv.begin(), iv.end(), 2, 8)；

查找“连续3个小于8的元素”所形成的子序列起点可以这么写：

iter1 = search\_n(iv.begin(), iv.end(), 3, 8, less<int>())；

#### swap\_ranges

将[first1, last1)区间内的元素与“从first2开始，个数相同”的元素相互交换。这两个序列可位于同一容器，也可位于不同容器中。如果第二序列的长度小于第一序列，或是两序列在同一容器中彼此重叠，执行结果未可预期。此算法返回一个迭代器，指向第二序列中的最后一个被交换元素的下一位置。实质就是将两段等长区间内的元素互换。

#### transform

transform()的第一版本以仿函数op作用于[first1, last1)中的每一个元素，并以其结果产生出一个新序列。第二版本以仿函数binary\_op作用于一双元素身上（其中一个元素来自[first1, last1)，另一个元素来自“从first2开始的序列”），并以其结果产生一个新序列。如果第二序列的元素个数少于第一序列，结果不可预期。

transform()的两个版本都把执行结果放进result所标示的容器中。result也可指向源端容器，那么transform()的运算结果就会取代该容器的元素。返回值OutputIterator将指向结果序列的最后元素的下一位置。

#### unique

移除[first, last)内的重复元素，但是只会移除相邻的重复元素，如果想要移除所有的重复元素，必须先将序列排序，使所有重复元素都相邻。

unique会返回一个迭代器指向新区间的尾端，新区间内不含相邻的重复元素。这个算法是稳定的，亦即所有保留下来的元素，其原始相对次序不变。

事实上，unique并不会改变[first, last)的元素个数，有一些残留数据会留下来，情况类似于remove。

unique有两个版本，版本一使用简单的相等，版本二使用一个二元运算。

#### unique\_copy

unique\_copy可从[first, last)中将元素复制到以result开头的区间上；如果面对相邻重复元素群，只会复制第一个元素。返回的迭代器指向以result开头的区间的尾端。

### lower\_bound（应用于有序区间）

这是二分查找的一个版本，试图在已排序的[first, last)中寻找元素value。如果[first, last)中具有与value相等的元素，便返回一个迭代器，指向该元素。如果没有这样的元素存在，就返回“假设这样的元素存在时应该出现的位置”。也就是说他会返回的一个迭代器，指向第一个不小于value的元素。如果value大于[first, last)内的任何一个元素，则返回last。

mid < value时 left = mid + 1

mid ≥ value时 right = mid

### upper\_bound（应用于有序区间）

这是二分查找的一个版本，试图在已排序的[first, last)中寻找元素value。更确切地说，它返回可插入value的最后一个合适的位置，即第一个大于value的元素的位置。如果[first, last)中具有与value相等的元素，便返回一个迭代器，指向该元素的下一位置。如果没有这样的元素存在，就返回“假设这样的元素存在时应该出现的位置”。也就是说他会返回的一个迭代器，指向第一个大于value的元素。如果value大于[first, last)内的任何一个元素，则返回last。

mid ≤ value时 left = mid + 1

mid > value时 right = mid

**注意**：lower\_bound和upper\_bound的循环条件是left < right，而普通二分查找循环条件是left <= right。

### binary\_search（应用于有序区间）

算法binary\_search是一种二分查找法，试图在已排序的[first, last)中寻找元素value。如果有等同于value的元素，便返回true，否则返回false。

返回单纯的bool或许不能满足你，前面所介绍的lower\_bound和upper\_bound能够提供额外的信息。事实上，binary\_search便是利用lower\_bound先找出“假设value存在的话应该出现的位置”，然后再对比该位置上的值是否是我们需要查找的目标，并返回对比结果。

### next\_permutation

next\_permutation会取得[first, last)所标示序列的下一个排列组合，如果没有下一个排列组合，便返回false；否则返回true。**注意**，如果序列不存在下一排列组合（即序列是逆序排列的），next\_permutation会将序列变为正序排列，然后返回false。

**具体实现**如下：

* 从最尾端开始往前寻找两个相邻元素，令第1个元素为\*i，第2个元素为\*ii，且满足\*i < \*ii。
* 找到这样一组相邻元素后，再从尾端开始找到第一个大于\*i的元素，令其为\*j。
* 将i，j元素对调，再将ii之后的所有元素颠倒排列，此即所求之“下一个排列组合”。

### prev\_permutation

prev\_permutation会取得[first, last)所标示序列的上一个排列组合，如果没有下一个排列组合，便返回false；否则返回true。**注意**，如果序列不存在上一排列组合（即序列是正序排列的），prev\_permutation会将序列变为逆序排列，然后返回false。

**具体实现**如下：

* 从最尾端开始往前寻找两个相邻元素，令第1个元素为\*i，第2个元素为\*ii，且满足\*i > \*ii。
* 找到这样一组相邻元素后，再从尾端开始找到第一个小于\*i的元素，令其为\*j。
* 将i，j元素对调，再将ii之后的所有元素颠倒排列，此即所求之“上一个排列组合”。

### random\_shuffle

这个算法将[first, last)的元素随机重排。也就是说有N!种可能，此处N为last – first。排列方式有N!种，random\_shuffle会产生一个均匀分布，因此任何一种排列被选中的可能性是1/N！。

random\_shuffle有两个版本，差别在于随机数的获取。版本一使用内部随机数产生器，版本二使用一个会产生随机数的仿函数。注意，该仿函数的传递方式是by reference而非一般的by value，这是因为随机数产生器有一个重要特质：它拥有局部状态，每次被调用时都会有所改变，并因此保障产生出来的随机数能够随机。

### partial\_sort/partial\_sort\_copy

本算法接受一个middle迭代器（位于序列[first, last)之内），然后重新安排[first, last)，使序列中 的middle – first个最小元素以递增顺序排序，置于[first, middle)内，其余last – middle个元素安置于[middle, last)中，不保证有任何特定顺序。

选择partial\_sort而非sort的唯一理由是效率。如果只挑出前N个最小的元素来排序，当然比对整个序列排序快得多。**注意**，只接受Random Access Iterator。

partial\_sort的任务是找出middle – first个最小元素。

**具体实施**如下：

* 首先界定区间[first, middle)，然后将该区间组织成一个max-heap，这样first指定的元素就是该区间的最大值
* 因为我们要让该区间的元素为最小的值，就意味着要将[middle, last)内的小元素替换进来
* 将[middle, last)的每一个元素拿来与max – heap的最大值进行比较（第一个元素）
* 如果小于最大值，就交换位置并重新保持max – heap的状态。
* 当我们走遍[middle, last)后，较大元素都已经被抽离出[first, middle)，这个时候再sort\_heap()将[first, middle)做一次排序，即完成。

partial\_sort\_copy将（last - first）个最小元素排序后的结果置于[result\_first, result\_last)中。

partial\_sort可以当作**堆排序**来使用。

### sort

sort算法，数据量大时使用Quick Sort（快速排序），分段递归排序。一旦数据量小于某个门槛（16），为避免Quick Sort的递归调用带来过大的额外负荷，就改用Insertion Sort（插入排序）。如果递归层次过深，还会使用Heap Sort（堆排序）。

**具体实施**如下：

* 数据量大时会计算Quick Sort的递归层数阈值，计算方式是2\*log2(last - first)
* 先判断元素个数，如果元素个数超过16才会使用Quick Sort，否则直接使用Insertion Sort
* 如果元素个数超过16，接下来判断递归深度，如果不超过阈值，就使用三点中值的Quick Sort进行排序，如果超过阈值，改用Heap Sort。
* 递归结束后，[first, last)内有多个“元素个数少于16”的子序列，这些子序列并没有完全排序（一定程度的排序），此时进行Insertion Sort。

### equal\_range（应用于有序区间）

equal\_range是二分查找的一个版本，在已排序的[first, last)中寻找value，它返回一堆迭代器i和j，其中i是在不破坏次序的前提下，value可插入的第一个位置（lower\_bound），j则是在不破坏次序的前提下，value可插入的最后一个位置（upper\_bound）。以pair的形式返回lower\_bound和upper\_bound这两个迭代器。

### inplace\_merge（应用于有序区间）

如果两个连接在一起的序列[first, middle)和[middle, last)都已排序，那么inplace\_merge可将它们结合成单一一个序列，并仍保持有序性。

### nth\_element

该算法会重新排列[first, last)，是迭代器nth所指的元素，与“整个[first, last)完整排序后，同一位置的元素”同值。此外保证[nth, last)内没有任何一个元素小于[first, nth)内的元素，但对于[first, nth) 和[nth, last)两个子区间内的元素次序则无任何保证。

### merge sort

分而治之的思想，将区间对半分，左右两段各自排序，再利用inplace\_merge重新组合成一个完整的序列。