C++

the programming language
from cplusplus.com

Only you

2015 年 4 月 17 日

目录

第一章　容器

1.1　array

包含头文件:<array>　　C++11

array 是固定大小的顺序容器，用来维护一个线性存储的序列。

除了数据元素之外，array 中不保存任何数据。这个类只是增加了一些成员和全局函数，所以 array 可以作为标准容器。大小为零的 array 是合法的，但是不能做任何引用。和其他标准容器不同的是，对 array 做 swap 操作复杂度是线性的，效率非常低，这也是为了在 swap 操作之后保持其迭代器和数据的关联。

array 容器的另一个特点是它可以被看做元组对象：<array> 重载了 get 函数来访问其中的元素，就像一个特殊的元组类型一样。

1.1.1　迭代器

begin a.begin() 返回指向第一个元素的迭代器

end a.end() 返回指向最后一个元素后面的迭代器

rbegin a.rbegin() 返回反向的迭代器，指向最后一个元素

rend a.rend() 返回反向的迭代器，指向第一个元素的前面

cbegin a.cbegin() 返回指向第一个元素的常量迭代器

cend a.cend() 返回指向最后一个元素后面的常量迭代器

crbegin a.crbegin() 返回反向的常量迭代器，指向最后一个元素

crend a.crend() 返回反向的常量迭代器，指向第一个元素的前面

1.1.2　容量

size a.size() 返回容器中的元素个数

max_size a.max_size() 返回容器可以容纳的最大元素个数，此时和 size 一样

empty a.empty() 测试容器是否为空

1.1.3 元素访问

operator [ ] a[size_type] 访问元素

at a.at(int) 访问元素

front a.front() 访问第一个元素

back a.back() 访问最后一个元素

data a.data() 返回数组的头指针

1.1.4 修改器

fill a.fill(value_type val) 用 val 填充

swap a.swap(deque<value_type>) 交换内容

样例：

```
#include <iostream>
#include <array>
using namespace std;
int main()
{
    array<int,5> a1={1,3,5,7,9};
    array<int,5> a2={0,2,4,6,8};
    for(auto it=a1.begin(); it!=a1.end(); ++it)
        cout<<*it<<' ';
    cout<<'\n';

    for(auto rit=a2.rbegin(); rit!=a2.rend(); ++rit)
        cout<<*rit<<' ';
    cout<<'\n';

    cout<<'\n';
    cout<<"size of a1: "<<a1.size()<<endl;
    cout<<"max_size of a2: "<<a2.max_size()<<endl;
    cout<<"a1 is empty? "<<a1.empty()<<endl;

    cout<<'\n';
    a1.front()=a1.back();
    a1[1]=a1[3];
    a1.at(2)=0;
    a1.swap(a2);
    a1.fill(0);
```

```
27      for(int &x : a1)
28          cout<<x<<' ';
29      cout<<'\n';
30      for(int &x : a2)
31          cout<<x<<' ';
32      cout<<'\n';
33      return 0;
34  }
35  /*
36  output:
37  1 3 5 7 9
38  8 6 4 2 0
39
40  size of a1: 5
41  max_size of a2: 5
42  a1 is empty? 0
43
44  0 0 0 0 0
45  9 7 0 7 9
46  */
```

1.2 deque

包含头文件：<deque>

deque(double-ended queue，双端队列)，发音似“deck”。它是一个可以在两端进行添加和删除，并可以动态改变大小的容器。和 vector 不同，deque 不保证存储数据的内存是连续的，因此，对一个指针进行偏移可能导致未知的错误。它的元素被保存在不同的内存块中，实现起来比 vector 复杂一些，对于频繁的不在队列两端的插入、删除操作，deque 花费的代价是很高的。

1.2.1 构造与析构

空容器构造函数 deque()

构造一个空的容器，不含任何元素

填充构造函数 deque(size_type n, value_type& val)

构造一个含有 n 个元素的容器，每个元素是 val 的一个拷贝

范围构造函数 deque(InputIterator first, InputIterator last)

构造一个包含 [first, last) 的容器，保持原来的顺序

复制构造函数 deque(const deque& x)

构造一个容器 x 的拷贝，保持原来的顺序

移动构造函数 C++11 deque(deque&& x)

构造一个包含 x 中元素的容器；如果指定的分配器与 x 不同，则重新分配内存，否则直接引用 x 的内容，返回时 x 的状态是不确定的但是有效的。

初始化列表构造函数 C++11 deque(initializer_list<value_type> il)

构造一个和 il 内容相同的容器，保持原来的顺序

析构函数 ~deque()

释放所有用分配器分配的内存，并销毁这个对象

赋值运算符 operator =

deque& operator= (const deque& x)

deque& operator= (deque&& x)

deque& operator= (initializer_list<value_type> il)

使用当前的分配器，将 x 中所有的元素复制到容器中，复制之前容器中的内容被销毁

样例：

```
#include <deque>
#include <iostream>
using namespace std;
int main()
{
    int myints[5] = {0, 2, 4, 6, 8};
    deque<int> a1; // empty container
    deque<int> a2(4, 20); // 4 number with value 20
    deque<int> a3(a2.begin(), a2.end());
    deque<int> a4(a3);
    deque<int> a5(myints, myints+5);
    deque<int> a6(std::move(a5)); // a5 wasted
    cout<<"a5: ";
    for(int &x : a5) cout<<x<<' ';
    cout<<endl;

    cout<<"a6: ";
    for(int &x : a6) cout<<x<<' ';
    cout<<endl;
    return 0;
}
/*
output:

a5:
a6: 0 2 4 6 8
```

```
27 */
```

1.2.2 迭代器

begin a.begin() 返回指向第一个元素的迭代器

end a.end() 返回指向最后一个元素后面的迭代器

rbegin a.rbegin() 返回反向的迭代器，指向最后一个元素

rend a.rend() 返回反向的迭代器，指向第一个元素的前面

cbegin C++11 a.cbegin() 返回指向第一个元素的常量迭代器

cend C++11 a.cend() 返回指向最后一个元素后面的常量迭代器

crbegin C++11 a.crbegin() 返回反向的常量迭代器，指向最后一个元素

crend C++11 a.crend() 返回反向的常量迭代器，指向第一个元素的前面

1.2.3 容量

size a.size() 返回容器中元素个数

max_size a.max_size()
返回容器可容纳的最大元素个数，这是根据已知的系统现状计算出来的最大可能性，但这不保证一定可以分配这么多空间来保存元素

resize a.resize(size_type n, value_type val)
将容器容量置为 n，如果当前容量小于 n，其余位置用 val 填充

empty a.empty() 测试容器是否为空

shrink_to_fit C++11 a.shrink_to_fit()
减少容器的内存占用使之与当前包含的元素占用空间相同

1.2.4 元素访问

operator [ ] a[size_type] 访问元素

at a.at(int) 访问元素

front a.front() 访问第一个元素

back a.back() 访问最后一个元素

1.2.5 修改器

assign

1. a.assign(InputIterator first, InputIterator second)
   将容器置为 [first, last) 中的元素，顺序不变
2. a.assign(size_type n, const value_type& val)
   将容器置为 n 个值为 val 的元素
3. a.assign(initializer_list<value_type> il) C++11
   将容器置为 il 中的内容，顺序不变

push_back a.push_back(value_type val) 在容器尾部追加元素

push_front a.push_front(value_type val) 在容器头部增加元素

pop_back a.pop_back() 在容器尾部删除元素

pop_front a.pop_front() 在容器头部删除元素

insert a.insert(...)

在当前位置插入元素，返回一个指向插入的第一个元素的迭代器
C++11 的位置迭代器为 const_iterator

1. a.insert(iterator pos, const value_type &val)
2. a.insert(iterator pos, size_type n, const value_type &val)
3. a.insert(iterator pos, InputIterator first, InputIterator last)
4. a.insert(const_iterator pos, value_type&& val) C++11
5. a.insert(const_iterator pos, initializer_list<value_type> il) C++11

erase a.erase(...)

删除当前位置的元素，返回下一个元素的迭代器
C++11 的位置迭代器为 const_iterator

1. a.erase(iterator pos)
2. a.erase(iterator first, iterator last)

swap a.swap(deque<value_type>& b)

交换两个容器的内容

clear a.clear() 清空容器

emplace C++11 a.emplace(const_iterator pos, Args&&... args)

使用 args 构造并安放元素，这种安放是直接在目标位置放置，不经过任何的复制和移动，这也是它和 insert 的不同之处，经测试，emplace 的效率比 insert 高 20% 左右。返回值为指向新安放的元素的迭代器

emplace_front C++11 a.emplace_front(Args&& ... args)

函数没有返回值

emplace_back C++11 a.emplace_back(Args&& ... args)

函数没有返回值

样例：

```
#include <iostream>
#include <deque>
using namespace std;
class Node
{
public:
    int x,y;
    Node(int _x=0,int _y=0)
    {
        x=_x,y=_y;
    }
    friend ostream& operator << (ostream& out, Node &p)
    {
        out<<p.x<<' '<<p.y<<endl;
        return out;
    }
};
int main()
{
    int myints[5] = {1, 3, 5, 7, 9};
    deque<int> a1;
    a1.assign(myints,myints+5);
    a1.pop_back();
    a1.pop_front();
    for(int &x : a1)
        cout<<x<<' ';
    cout<<endl;

    a1.assign(5, 4);
    a1.push_back(9);
    a1.push_front(1);

    deque<int>::iterator it=a1.begin()+2;
    it = a1.insert(it, 2);
    a1.insert(it, myints, myints+2);

    for(int &x : a1)
        cout<<x<<' ';
```

```
39      cout<<endl;
40
41      deque<Node> a2;
42      a2.emplace(a2.begin(), 1);
43      a2.emplace(a2.begin(), 3, 4);
44      a2.emplace_front(0, -1);
45      a2.emplace_back(-1, 0);
46      for(auto &x : a2)
47          cout<<x;
48      return 0;
49  }
50  /*
51  output:
52  3 5 7
53  1 4 1 3 2 4 4 4 4 9
54  0 -1
55  3 4
56  1 0
57  -1 0
58  */
```

1.3 forward_list

包含头文件:<forward_list> C++11

forward_list, 前向链表，是一种序列容器，在任何位置的插入和删除操作复杂度都是常数，它和 list 的区别就是它只有向前的指针，而 list 维护了向前和向后的指针。对于只在一个方向上的操作，forward_list 比 list 效率略高。和 array、vector 等容器相比，forward_list 可以高效地插入删除，但随机访问的复杂度很高。

forward_list 被设计得非常高效，如同 c 语言的单向链表，因而有意不提供 size 成员函数，否则不但会占用额外的空间，还会影响插入和删除的效率，如果需要获得它的大小，可以使用 distance 算法来计算 begin 和 end 之间的距离，但这个操作是线性的。

1.3.1 构造与析构

空容器构造函数 forward_list()

构造一个空的容器，不含任何元素

填充构造函数 forward_list(size_type n, value_type& val)

构造一个含有 n 个元素的容器，每个元素是 val 的一个拷贝

范围构造函数 forward_list(iterator first, iterator last)

构造一个包含 [first, last) 的容器，保持原来的顺序

复制构造函数 forward_list(const forward_list& x)

构造一个容器 x 的拷贝，保持原来的顺序

移动构造函数 forward_list(forward_list&& x)

构造一个包含 x 中元素的容器；如果指定的分配器与 x 不同，则重新分配内存，否则直接引用 x 的内容，返回时 x 的状态是不确定的但是有效的。

初始化列表构造函数 forward_list(initializer_list<value_type> il)

构造一个和 il 内容相同的容器，保持原来的顺序

析构函数 ~forward_list()

释放所有用分配器分配的内存，并销毁这个对象

赋值运算符 operator =

forward_list& operator= (const forward_list& x)

forward_list& operator= (forward_list&& x)

forward_list& operator= (initializer_list<value_type> il)

使用当前的分配器，将 x 中所有的元素复制到容器中，复制之前容器中的内容被销毁

1.3.2 迭代器

before_begin a.before_begin() 返回第一个元素之前的迭代器

begin a.begin() 返回指向第一个元素的迭代器

end a.end() 返回指向最后一个元素后面的迭代器

cbefore_begin a.cbefore_begin() 返回第一个元素之前的常量迭代器

cbegin a.cbegin() 返回指向第一个元素的常量迭代器

cend a.cend() 返回指向最后一个元素后面的常量迭代器

1.3.3 容量

max_size a.max_size()

返回容器可容纳的最大元素个数，这是根据已知的系统现状计算出来的最大可能性，但这不保证一定可以分配这么多空间来保存元素

empty a.empty() 测试容器是否为空

1.3.4 元素访问

front a.front() 访问第一个元素

1.3.5 修改器

assign

1. a.assign(InputIterator first, InputIterator second)
   将容器置为 [first, last) 中的元素，顺序不变

2. a.assign(size_type n, const value_type& val)
   将容器置为 n 个值为 val 的元素
3. a.assign(initializer_list<value_type> il)
   将容器置为 il 中的内容，顺序不变

emplace_front a.emplace_front(Args&&... args)
在第一个位置安放一个元素，没有返回值

push_front a.push_front(value_type& val)
向第一个位置插入一个元素

pop_front a.pop_front() 删除第一个元素

emplace_after a.emplace_after(const_iterator pos, Args&&... args)
在给定的位置之后安放一个元素，返回指向新元素的迭代器

insert_after a.insert_after(...)
在给定的位置之后插入元素，返回一个指向插入的最后一个元素的迭代器，如果没有插入元素则返回给定的位置

1. a.insert(const_iterator pos, const value_type& val)
2. a.insert(const_iterator pos, value_type&& val)
3. a.insert(const_iterator pos, size_type n, const value_type& val)
4. a.insert(const_iterator pos, InputIterator first, InputIterator last)
5. a.insert(const_iterator pos, initializer_list<value_type> il)

erase a.erase_after(...)

1. a.erase(const_iterator pos) 删除当前位置之后的元素，返回下一个元素的迭代器
2. a.erase(const_iterator pos, const_iterator last) 删除 (pos, last) 的元素

swap a.swap(forward_list<value_type>& b)
交换两个容器的内容

clear a.clear() 清空容器

resize a.resize(size_type n, const value_type& val)
将容器大小置为 n，如果 n 比当前元素数量小，只保存前 n 个元素；如果 n 比当前元素数量大，则用 val 填充，若不给出 val，则用默认构造函数

样例：

```
1 #include <iostream>
2 #include <forward_list>
3 using namespace std;
4 int main()
5 {
```

```
6     forward_list<int> a1;
7     forward_list<int> a2;
8     a1.assign(4, 1);
9     a2.assign(a1.begin(), a1.end());
10    a2.insert_after(a2.before_begin(), 3);
11    a2.push_front(2);
12    a2.erase_after(a2.before_begin());
13
14    a1.assign({1, 2, 3});
15    a1.emplace_front(0);
16    a1.emplace_after(a1.before_begin(), 9);
17    a1.resize(9, -1);
18
19    cout<<"a1: ";
20    for(int& x : a1) cout<<x<<' ';
21    cout<<endl;
22    cout<<"a2: ";
23    for(int& x : a2) cout<<x<<' ';
24    cout<<endl;
25    return 0;
26 }
27 /*
28 output:
29
30 a1: 9 0 1 2 3 -1 -1 -1 -1
31 a2: 3 1 1 1 1
32 */
```

1.3.6 操作

splice_after a.splice_after(...)

从一个链表剪下一段并拼接在另一个链表上

整个链表

1. a.splice_after(const_iterator pos, forward_list& b)
2. a.splice_after(const_iterator pos, forward_list&& b)

单个元素 在 pos 后面拼接位于迭代器 i 后面的那个元素

1. a.splice_after(const_iterator pos, forward_list& b, const_iterator i)
2. a.splice_after(const_iterator pos, forward_list&& b, const_iterator i)

一个区间 在 pos 后面拼接位于 (first, last) 的元素

1. a.splice_after(const_iterator pos, forward_list& b, const_iterator first, const_iterator last)

2. a.splice_after (const_iterator pos, forward_list&& b, const_iterator first, const_iterator last);

remove a.remove(const value_type& val)
删除所有值为 val 的元素

remove_if a.remove_if(Predicate pred)
删除所有使谓词为真的元素

unique

1. a.unique()
   相同的元素只保留第一个，其余的删除
2. a.unique(BinaryPredicate binary_pred)
   函数会调用 binary_pred(*i, *(i-1)), 如果二元谓词为真，则删除迭代器 i 指向的元素

merge

1. a.merge(forward_list& b) , a.merge(forward_list&& b)
   a 和 b 应当已经是有序的，将 b 合并到 a 中，并使得 a 依然有序，这个过程中不会创建或销毁元素，只有移动，结束后 b 将变成空的容器
2. a.merge(forward_list& b, Compare comp) , a.merge(forward_list&& b, Compare comp)
   使用 comp 定义的“严格弱序”关系进行比较

sort a.sort() , a.sort(Compare comp)
对链表使用 operator< 或 comp 进行排序，须保证对元素的“严格弱序”，排序前相等的元素，排序后相对位置不变

reverse a.reverse() 反转链表

样例：

```
#include <iostream>
#include <forward_list>
using namespace std;
int main()
{
    forward_list<int> a1={1, 2, 3, 4, 5};
    forward_list<int> a2={5, 3, 2, 4, 1};
    a2.splice_after(a2.before_begin(), a1, a1.before_begin(), a1.end());
    cout<<"a2: ";
    for(int& x : a2) cout<<x<<' ';
    cout<<endl;

    a2.sort();
```

```
14      a2.remove(3);
15      cout<<"a2: ";
16      for(int& x : a2) cout<<x<<' ';
17      cout<<endl;
18
19      a2.unique();
20      a2.reverse();
21      cout<<"a2: ";
22      for(int& x : a2) cout<<x<<' ';
23      cout<<endl;
24      return 0;
25  }
26  /*
27  output:
28
29  a2: 1 2 3 4 5 5 3 2 4 1
30  a2: 1 1 2 2 4 4 5 5
31  a2: 5 4 2 1
32  */
```

1.4 list

包含头文件:<list>

list, 双向链表，是一种序列容器，可以高效地在任何位置插入和删除，和 forward_list 相似，但多了反向的指针。

1.4.1 构造与析构

空容器构造函数 list()

构造一个空的容器，不含任何元素

填充构造函数 list(size_type n, value_type& val)

构造一个含有 n 个元素的容器，每个元素是 val 的一个拷贝

范围构造函数 list(iterator first, iterator last)

构造一个包含 [first, last) 的容器，保持原来的顺序

复制构造函数 list(const list& x)

构造一个容器 x 的拷贝，保持原来的顺序

移动构造函数 C++11 list(list&& x)

构造一个包含 x 中元素的容器；如果指定的分配器与 x 不同，则重新分配内存，否则直接引用 x 的内容，返回时 x 的状态是不确定的但是有效的。

初始化列表构造函数 C++11 list(initializer_list<value_type> il)

构造一个和 il 内容相同的容器，保持原来的顺序

析构函数 ~list()

释放所有用分配器分配的内存，并销毁这个对象

赋值运算符 operator =

list& operator= (const list& x)

list& operator= (list&& x)

list& operator= (initializer_list<value_type> il)

使用当前的分配器，将 x 中所有的元素复制到容器中，复制之前容器中的内容被销毁

1.4.2 迭代器

begin a.begin() 返回指向第一个元素的迭代器

end a.end() 返回指向最后一个元素后面的迭代器

rbegin a.rbegin() 返回反向的迭代器，指向最后一个元素

rend a.rend() 返回反向的迭代器，指向第一个元素的前面

cbegin C++11 a.cbegin() 返回指向第一个元素的常量迭代器

cend C++11 a.cend() 返回指向最后一个元素后面的常量迭代器

crbegin C++11 a.crbegin() 返回反向的常量迭代器，指向最后一个元素

crend C++11 a.crend() 返回反向的常量迭代器，指向第一个元素的前面

1.4.3 容量

size a.size() 返回容器中元素个数

max_size a.max_size()

返回容器可容纳的最大元素个数，这是根据已知的系统现状计算出来的最大可能性，但这不保证一定可以分配这么多空间来保存元素

empty a.empty() 测试容器是否为空

1.4.4 元素访问

front a.front() 访问第一个元素

back a.back() 访问最后一个元素

1.4.5 修改器

assign

1. a.assign(InputIterator first, InputIterator second)
   将容器置为 [first, last) 中的元素，顺序不变
2. a.assign(size_type n, const value_type& val)
   将容器置为 n 个值为 val 的元素
3. a.assign(initializer_list<value_type> il) C++11
   将容器置为 il 中的内容，顺序不变

emplace_front C++11 a.emplace_front(Args&&... args)
在第一个位置安放一个元素，没有返回值

push_front a.push_front(value_type val) 在容器头部增加元素

pop_front a.pop_front() 在容器头部删除元素

emplace_back C++11 a.emplace_back(Args&&... args)
在最后一个位置安放一个元素，没有返回值

push_back a.push_back(value_type val) 在容器尾部追加元素

pop_back a.pop_back() 在容器尾部删除元素

emplace C++11 a.emplace(const_iterator pos, Args&&... args)
使用 args 构造并在 pos 位置安放元素，返回值为指向新安放的元素的迭代器

insert a.insert(...)
在当前位置插入元素，返回一个指向插入的第一个元素的迭代器
C++11 的位置迭代器为 const_iterator

1. a.insert(iterator pos, const value_type& val)
2. a.insert(iterator pos, size_type n, const value_type& val)
3. a.insert(iterator pos, InputIterator first, InputIterator last)
4. a.insert(const_iterator pos, value_type&& val) C++11
5. a.insert(const_iterator pos, initializer_list<value_type> il) C++11

erase a.erase(...)
删除当前位置的元素，返回下一个元素的迭代器
C++11 的位置迭代器为 const_iterator

1. a.erase(iterator pos)
2. a.erase(iterator first, iterator last)

swap a.swap(list<value_type>& b)
交换两个容器的内容

resize a.resize(size_type n, const value_type& val)

将容器大小置为 n，如果 n 比当前元素数量小，只保存前 n 个元素；如果 n 比当前元素数量大，则用 val 填充，若不给出 val，则用默认构造函数

clear a.clear() 清空容器

1.4.6 操作

splice a.splice(...)

从一个链表剪下一段并拼接在另一个链表上

整个链表

1. a.splice(const_iterator pos, list& b)
2. a.splice(const_iterator pos, list&& b)

单个元素 在 pos 后面拼接迭代器 i 指向的那个元素

1. a.splice(const_iterator pos, list& b, const_iterator i)
2. a.splice(const_iterator pos, list&& b, const_iterator i)

一个区间 在 pos 后面拼接位于 [first, last) 的元素

1. a.splice(const_iterator pos, list& b, const_iterator first, const_iterator last)
2. a.splice (const_iterator pos, list&& b, const_iterator first, const_iterator last);

remove a.remove(const value_type& val)

删除所有值为 val 的元素

remove_if a.remove_if(Predicate pred)

删除所有使谓词为真的元素

unique

1. a.unique()
   相同的元素只保留第一个，其余的删除
2. a.unique(BinaryPredicate binary_pred)
   函数会调用 binary_pred(*i, *(i-1)), 如果二元谓词为真，则删除迭代器 i 指向的元素

merge

1. a.merge(list& b) , a.merge(list&& b)
   a 和 b 应当已经是有序的，将 b 合并到 a 中，并使得 a 依然有序，这个过程中不会创建或销毁元素，只有移动，结束后 b 将变成空的容器
2. a.merge(list& b, Compare comp) , a.merge(list&& b, Compare comp)
   使用 comp 定义的“严格弱序”关系进行比较

sort a.sort() , a.sort(Compare comp)

对链表使用 operator< 或 comp 进行排序，须保证对元素的“严格弱序”，排序前相等的元素，排序后相对位置不变

reverse a.reverse() 反转链表

1.5 map

包含头文件:<map>

map 是一种关联容器，按照特定的顺序保存键 -值对。

```
template < class Key,                                    // map::key_type
           class T,                                      // map::mapped_type
           class Compare = less<Key>,                    // map::key_compare
           class Alloc = allocator<pair<const Key,T> >   // map::allocator_type
           > class map;
```

map 容器通常比 unordered_map 慢，但它允许基于顺序的迭代。map 可以通过下标访问运算符直接访问键的值。

1.5.1 构造与析构

空容器构造函数 map(const key_compare& comp=key_compare())

构造一个空的容器，不含任何元素

范围构造函数 map(iterator first, iterator last, const key_compare& comp=key_compare())

构造一个包含 [first, last) 的容器，保持原来的顺序

复制构造函数 map(const map& x)

构造一个容器 x 的拷贝，保持原来的顺序

移动构造函数 C++11 map(map&& x)

构造一个包含 x 中元素的容器；如果指定的分配器与 x 不同，则重新分配内存，否则直接引用 x 的内容，返回时 x 的状态是不确定的但是有效的。

初始化列表构造函数 C++11 map(initializer_list<value_type> il, const key_compare& comp=key_compare())

构造一个和 il 内容相同的容器，保持原来的顺序

析构函数 ~map()

释放所有用分配器分配的内存，并销毁这个对象

赋值运算符 operator =

map& operator= (const map& x)

map& operator= (map&& x)

map& operator= (initializer_list<value_type> il)

使用当前的分配器，将 x 中所有的元素复制到容器中，复制之前容器中的内容被销毁

1.5.2 迭代器

begin a.begin() 返回指向第一个元素的迭代器

end a.end() 返回指向最后一个元素后面的迭代器

rbegin a.rbegin() 返回反向的迭代器，指向最后一个元素

rend a.rend() 返回反向的迭代器，指向第一个元素的前面

cbegin C++11 a.cbegin() 返回指向第一个元素的常量迭代器

cend C++11 a.cend() 返回指向最后一个元素后面的常量迭代器

crbegin C++11 a.crbegin() 返回反向的常量迭代器，指向最后一个元素

crend C++11 a.crend() 返回反向的常量迭代器，指向第一个元素的前面

1.5.3 容量

size a.size() 返回容器中元素个数

max_size a.max_size()

返回容器可容纳的最大元素个数，这是根据已知的系统现状计算出来的最大可能性，但这不保证一定可以分配这么多空间来保存元素

empty a.empty() 测试容器是否为空

1.5.4 元素访问

operator [ ] a[const key_type& k], a[key_type&& k] 访问元素，返回键为 k 的元素的引用

at a.at(const key_type& k), a.at(const key_type& k) 访问键的值，返回这个元素的引用

样例：

```
#include <iostream>
#include <map>
using namespace std;
int main()
{
    map<int, char> m;
    m[0]='a';
    m[1]='b';
    cout<<"m[0] is: "<<m[0]<<endl;
    cout<<"m[1] is: "<<m[1]<<endl;
    cout<<"m[2] is: "<<m[2]<<endl;
    // insert an elements, a[2] is initialized to char(0)
    cout<<"m has "<<m.size()<<" elements."<<endl;
    m.at(2)='c';
    // if the key does not match, the function throw an out_of_range exception.
    for(auto& x : m) cout<<'('<<x.first<<", "<<x.second<<")"<<endl;
    return 0;
}
/*
```

```
output:

m[0] is: a
m[1] is: b
m[2] is:
m has 3 elements.
(0, a)
(1, b)
(2, c)
*/
```

1.5.5 修改器

insert a.insert(...)

单元素 a.insert(const value_type& val)

返回一个 pair<iterator, bool>，如果插入成功，迭代器指向新元素，布尔值为真；如果插入失败，迭代器指向键相同的元素，布尔值为假

区间 a.insert(InputIterator first, InputIterator last)

带线索 a.insert(const_iterator pos, value_type&& val)

返回一个迭代器指向新元素或者键相同的元素

初始化列表 a.insert(initializer_list<value_type> il) C++11

erase a.erase(...)

删除当前位置的元素

C++11 的位置迭代器为 const_iterator

1. a.erase(iterator pos) C++11 返回下一个元素的迭代器
2. a.erase(iterator first, iterator last) C++11 返回下一个元素的迭代器
3. a.erase(const key_type& k) 返回删除的元素个数

swap a.swap(map<value_type>& b)

交换两个容器的内容

clear a.clear() 清空容器

emplace C++11 a.emplace(Args&&... args)

使用 args 构造并安放元素，返回 pair<iterator, bool>

emplace_hint C++11 a.emplace_hint(const_iterator pos, Args&&... args)

根据提示加速插入过程，使用 args 构造并安放元素，返回迭代器，插入成功则指向新元素，否则指向键相同的元素

样例：

```cpp
#include <iostream>
#include <map>
using namespace std;
int main()
{
    map<int, int> m;
    auto p=m.insert(pair<int, int>(1,1));
    m.insert(p.first, pair<int,int>(2,2));
    m.insert({{3,3},{0,0}});
    cout<<m.erase(2)<<endl;
    m.emplace(4,4);
    m.emplace_hint(m.end(),5,5);
    for(auto& x : m) cout<<x.first<<", "<<x.second<<endl;
    return 0;
}
/*
output:

1
0, 0
1, 1
3, 3
4, 4
5, 5
*/
```

1.5.6 观察者

key_comp a.key_comp(key_type x, key_type y)

返回键的比较器，默认是 less

value_comp a.value_comp(value_type x, value_type y)

返回元素的比较器，比较两个元素的键的大小

样例：

```cpp
#include <iostream>
#include <map>
using namespace std;
int main()
{
    map<char, int> m;
```

```
7     m['a']=1;
8     m['b']=0;
9     map<char, int>::key_compare comp_k=m.key_comp();
10    map<char, int>::value_compare comp_v=m.value_comp();
11    map<char, int>::iterator it1,it2;
12    it1=it2=m.begin();
13    it2++;
14    cout<<comp_k(it1->first, it2->first)<<endl;
15    cout<<comp_v(*it1, *it2)<<endl;
16    return 0;
17 }
18 /*
19 output:
20
21 1
22 1
23 */
```

1.5.7 操作

find a.find(const key_type& k)
返回键为 k 的迭代器，没有则返回 map::end

count a.count(const key_type& k)
返回键为 k 的元素的个数

lower_bound a.lower_bound(const key_type& k)
返回键不小于 k 的最小的元素的迭代器

upper_bound a.upper_bound(const key_type& k)
返回键大于 k 的最小的元素的迭代器

equal_range a.equal_range(const key_type& k)
返回 pair<iterator, iterator>，(a.lower_bound(k), a.upper_bound(k))

样例：

```
1  #include <iostream>
2  #include <map>
3  using namespace std;
4  int main()
5  {
6     map<int, int> m;
7     m[1]=1;
```

```
8     m[2]=2;
9     m[3]=3;
10    m[4]=4;
11    map<int, int>::iterator low,up;
12    low=m.lower_bound(2);
13    up=m.upper_bound(3);
14    m.erase(low, up);
15    for(auto& x : m) cout<<x.first<<' ';
16    cout<<endl;
17    return 0;
18 }
19 /*
20 output:
21
22 1 4
23 */
```

1.5.8 关于 multimap

multimap 与 map 基本相同，但在插入元素时不会出现因键相同而插入失败的情况。对于键相同的元素，C++11 保证后插入的元素放在已经存在的元素的后面，而 C++98 未做这样的保证。

1.6 queue

包含头文件:<queue>

这个头文件包含先进先出的队列和优先队列：

queue，队列，是一种容器适配器，只允许从一端插入，从另一端删除的容器，list 和 deque 对这种操作效率较高，如果没有指定，则默认用 deque 实现。

priority_queue，优先队列，谁一种容器适配器，只允许优先级最高的元素先取出来的容器，可用 deque 和 vector 实现，如果没有指定，默认用 vector 实现。

1.6.1 queue 成员函数

构造函数

初始化构造函数 queue (const container_type& ctnr)

移动构造函数 C++11 queue (container_type&& ctnr = container_type())

指定分配器的构造函数 C++11 queue (const Alloc& alloc)

指定分配器的构造函数 C++11 queue (const container_type& ctnr, const Alloc& alloc)

指定分配器的移动构造函数 C++11 queue (container_type&& ctnr, const Alloc& alloc)

指定分配器的复制构造函数 C++11 queue (const queue& x, const Alloc& alloc)

指定分配器的移动构造函数 C++11 queue (queue&& x, const Alloc& alloc)

empty a.empty() 容器是否为空

size a.size() 返回容器元素个数

front a.front() 返回第一个元素的引用

back a.back() 返回最后一个元素的引用

push a.push(const value_type& val), a.push(value_type&& val) C++11
在最后插入一个元素

emplace C++11 a.emplace(Args&&... args) 在最后安放一个元素

pop a.pop() 删除第一个元素

swap C++11 a.swap(queue& b) 交换两个容器的内容

样例：

```
#include <iostream>
#include <deque>
#include <list>
#include <queue>
using namespace std;
int main()
{
    deque<int> d(4,10);
    list<int> li(5,9);
    queue<int> a1;
    queue<int> a2(d);
    queue<int,list<int> >a3;
    queue<int,list<int> >a4(li);
    return 0;
}
```

1.6.2 priority_queue 成员函数

构造函数

初始化构造函数 priority_queue (const Compare& comp, const Container& ctnr)

范围构造函数 priority_queue (InputIterator first, InputIterator last,
const Compare& comp, const Container& ctnr)

移动构造函数 C++11 priority_queue (const Compare& comp = Compare(),
Container&& ctnr = Container());

移动范围构造函数 C++11 priority_queue (InputIterator first, InputIterator last, const Compare& comp, Container&& ctnr = Container());

指定分配器的构造函数 C++11

1. priority_queue (const Alloc& alloc);
2. priority_queue (const Compare& comp, const Alloc& alloc);
3. priority_queue (const Compare& comp, const Container& ctnr, const Alloc& alloc);
4. priority_queue (const Compare& comp, Container&& ctnr, const Alloc& alloc);
5. priority_queue (const priority_queue& x, const Alloc& alloc);
6. priority_queue (priority_queue&& x, const Alloc& alloc);

empty a.empty() 容器是否为空

size a.size() 返回容器元素个数

top a.top() 返回优先级最高的元素的引用

push a.push(const value_type& val), a.push(value_type&& val) C++11
插入一个元素

emplace C++11 a.emplace(Args&&... args) 安放一个元素

pop a.pop() 删除优先级最高的元素

swap C++11 a.swap(priority_queue& b) 交换两个容器的内容

样例：

```
#include <iostream>
#include <queue>
#include <vector>
#include <functional>
using namespace std;
class mycomparison
{
    bool reverse;
public:
    mycomparison(const bool& revparam=false)
    {
        reverse=revparam;
    }
    bool operator() (const int& lhs, const int&rhs) const
    {
        if (reverse) return (lhs>rhs);
        else return (lhs<rhs);
```

```
18      }
19  };
20
21  int main ()
22  {
23      int myints[]= {10,60,50,20};
24
25      priority_queue<int> first;
26      priority_queue<int> second (myints,myints+4);
27      priority_queue<int, vector<int>, greater<int> >
28          third (myints,myints+4);
29      typedef priority_queue<int,vector<int>,mycomparison> mypq_type;
30
31      mypq_type fourth;
32      mypq_type fifth (mycomparison(true));
33
34      return 0;
35  }
```

1.7 set

包含头文件:<set>

set，集合，是一种按照特定的顺序保存元素容器，其中每个元素都是独特的。由于有顺序，一旦插入便不能修改，但可以删除。

```
template < class T,                        // set::key_type/value_type
           class Compare = less<T>,        // set::key_compare/value_compare
           class Alloc = allocator<T>      // set::allocator_type
           > class set;
```

set 容器通常比 unordered_set 慢，但它允许基于顺序的迭代。set 通常用二叉搜索树实现。

1.7.1 构造与析构

空容器构造函数 set(const key_compare& comp=key_compare())

构造一个空的容器，不含任何元素

范围构造函数 set(iterator first, iterator last, const key_compare& comp=key_compare())

构造一个包含 [first, last) 的容器，保持原来的顺序

复制构造函数 set(const set& x)

构造一个容器 x 的拷贝，保持原来的顺序

移动构造函数 C++11 set(set&& x)

构造一个包含 x 中元素的容器；如果指定的分配器与 x 不同，则重新分配内存，否则直接引用 x 的内容，返回时 x 的状态是不确定的但是有效的。

初始化列表构造函数 C++11 set(initializer_list<value_type> il, const key_compare& comp=key_compare())

构造一个和 il 内容相同的容器，保持原来的顺序

析构函数 ~set()

释放所有用分配器分配的内存，并销毁这个对象

赋值运算符 operator =

set& operator= (const set& x)

set& operator= (set&& x)

set& operator= (initializer_list<value_type> il)

使用当前的分配器，将 x 中所有的元素复制到容器中，复制之前容器中的内容被销毁

1.7.2 迭代器

begin a.begin() 返回指向第一个元素的迭代器

end a.end() 返回指向最后一个元素后面的迭代器

rbegin a.rbegin() 返回反向的迭代器，指向最后一个元素

rend a.rend() 返回反向的迭代器，指向第一个元素的前面

cbegin C++11 a.cbegin() 返回指向第一个元素的常量迭代器

cend C++11 a.cend() 返回指向最后一个元素后面的常量迭代器

crbegin C++11 a.crbegin() 返回反向的常量迭代器，指向最后一个元素

crend C++11 a.crend() 返回反向的常量迭代器，指向第一个元素的前面

1.7.3 容量

size a.size() 返回容器中元素个数

max_size a.max_size()

返回容器可容纳的最大元素个数，这是根据已知的系统现状计算出来的最大可能性，但这不保证一定可以分配这么多空间来保存元素

empty a.empty() 测试容器是否为空

1.7.4 修改器

insert a.insert(...)

单元素 a.insert(const value_type& val)

返回一个 pair<iterator, bool>，如果插入成功，迭代器指向新元素，布尔值为真；如果插入失败，迭代器指向键相同的元素，布尔值为假

区间 a.insert(InputIterator first, InputIterator last)

带线索 a.insert(const_iterator pos, value_type&& val)
返回一个迭代器指向新元素或者键相同的元素

初始化列表 a.insert(initializer_list<value_type> il) C++11

erase a.erase(...)
删除当前位置的元素
C++11 的位置迭代器为 const_iterator

1. a.erase(iterator pos) C++11 返回下一个元素的迭代器
2. a.erase(iterator first, iterator last) C++11 返回下一个元素的迭代器
3. a.erase(const key_type& k) 返回删除的元素个数

swap a.swap(set<value_type>& b)
交换两个容器的内容

clear a.clear() 清空容器

emplace C++11 a.emplace(Args&&... args)
使用 args 构造并安放元素，返回 pair<iterator, bool>

emplace_hint C++11 a.emplace_hint(const_iterator pos, Args&&... args)
根据提示加速插入过程，使用 args 构造并安放元素，返回迭代器，插入成功则指向新元素，否则指向键相同的元素

1.7.5 观察者

key_comp a.key_comp(key_type x, key_type y)
返回键的比较器，默认是 less

value_comp a.value_comp(value_type x, value_type y)
返回元素的比较器，比较两个元素的键的大小

1.7.6 操作

find a.find(const value_type& k)
返回元素 k 的迭代器，没有则返回 set::end

count a.count(const value_type& k)
返回值为 k 的元素的个数

lower_bound a.lower_bound(const key_type& k)
返回键不小于 k 的最小的元素的迭代器

upper_bound a.upper_bound(const key_type& k)
返回键大于 k 的最小的元素的迭代器

equal_range a.equal_range(const key_type& k)
返回 pair<iterator, iterator>，(a.lower_bound(k), a.upper_bound(k))

1.7.7 关于 multiset

multiset 与 set 基本相同，但在插入元素时不会出现因元素相同而插入失败的情况。对于键相同的元素，C++11 保证后插入的元素放在已经存在的元素的后面，而 C++98 未做这样的保证。

1.8 stack

包含头文件:<stack>

stack，栈，是一种先进先出容器适配器，只允许在一端进行插入和删除操作。可以用 vector、deque、list 来实现，如果没有指定，默认使用 deque 实现。

1.8.1 成员函数

构造函数

初始化构造函数 stack (const Container& ctnr)

移动构造函数 C++11 stack (Container&& ctnr = Container());

复制构造函数 C++11 stack (const stack& x)

指定分配器的构造函数 C++11 以上构造函数的指定分配器的构造函数

empty a.empty() 容器是否为空

size a.size() 返回容器元素个数

top a.top() 返回栈顶元素的引用

push a.push(const value_type& val), a.push(value_type&& val) C++11
在栈顶插入一个元素

emplace C++11 a.emplace(Args&&... args) 在栈顶安放一个元素

pop a.pop() 删除栈顶的元素

swap C++11 a.swap(stack& b) 交换两个容器的内容

1.9 unordered_map

包含头文件:<unordered_map> C++11

unordered_map，无序映射，是一种关联容器，存储键和值的映射，并允许基于键的快速访问。其中的元素并不是有序的。

```
template < class Key,                                    // unordered_map::key_type
         class T,                                       // unordered_map::mapped_type
          class Hash = hash<Key>,                         // unordered_map::hasher
```

```
    class Pred = equal_to<Key>,                     // unordered_map::key_equal
  class Alloc = allocator< pair<const Key,T> > // unordered_map::allocator_type
    > class unordered_map;
```

1.9.1 构造与析构

空的构造函数 unordered_map(size_type n,
const hasher& hf = hasher(),
const key_equal& eql = key_equal(),
const allocator_type& alloc = allocator_type())
构造一个空的容器，n 为散列桶的最小值，hf 为一个散列函数，eql 为判断两个元素是否相等的函数

范围构造函数 unordered_map(InputIterator first, InputIterator last,
size_type n,
const hasher& hf = hasher(),
const key_equal& eql = key_equal(),
const allocator_type& alloc = allocator_type())

复制构造函数 unordered_map (const unordered_map& ump)

移动构造函数 unordered_map (unordered_map&& ump)

初始化列表构造函数 unordered_map(initializer_list<value_type> il,
size_type n,
const hasher& hf = hasher(),
const key_equal& eql = key_equal(),
const allocator_type& alloc = allocator_type())

析构函数 ~unordered_map() 释放所有用分配器分配的内存，并销毁这个对象

赋值运算符 operator =
unordered_map& operator= (const unordered_map& x)
unordered_map& operator= (unordered_map&& x)
unordered_map& operator= (initializer_list<value_type> il)
使用当前的分配器，将 x 中所有的元素复制到容器中，复制之前容器中的内容被销毁

1.9.2 容量

size a.size() 返回容器中元素个数

max_size a.max_size()
返回容器可容纳的最大元素个数，这是根据已知的系统现状计算出来的最大可能性，但这不保证一定可以分配这么多空间来保存元素

empty a.empty() 测试容器是否为空

1.9.3 迭代器

begin a.begin(size_type n) 返回指向第一个元素的迭代器，参数 n 表示第 n 个散列桶

end a.end(size_type n) 返回指向最后一个元素后面的迭代器，参数 n 表示第 n 个散列桶

cbegin a.cbegin(size_type n) 返回指向第一个元素的常量迭代器，参数 n 表示第 n 个散列桶

cend a.cend(size_type n) 返回指向最后一个元素后面的常量迭代器，参数 n 表示第 n 个散列桶

样例：

```
#include <iostream>
#include <unordered_map>
using namespace std;
int main ()
{
    unordered_map<string, string> mymap;
    mymap={{"Australia","Canberra"},{"U.S.","Washington"},{"France","Paris"}};
    cout<<"mymap contains:";
    for(auto it=mymap.begin(); it!=mymap.end(); ++it)
        cout<<" "<<it->first<<":"<<it->second;
    cout<<endl;

    cout << "mymap's buckets contain:\n";
    for(unsigned i=0; i<mymap.bucket_count(); ++i)
    {
        cout << "bucket #" << i << " contains:";
        for(auto local_it=mymap.begin(i); local_it!=mymap.end(i); ++local_it )
            cout << " " << local_it->first << ":" << local_it->second;
        cout<<endl;
    }

    return 0;
}
/*
output:

mymap contains: France:Paris U.S.:Washington Australia:Canberra
mymap's buckets contain:
bucket #0 contains:
bucket #1 contains:
bucket #2 contains:
bucket #3 contains:
bucket #4 contains:
```

```
bucket #5 contains: France:Paris
bucket #6 contains:
bucket #7 contains: Australia:Canberra
bucket #8 contains: U.S.:Washington
bucket #9 contains:
bucket #10 contains:
*/
```

1.9.4 元素访问

operator [ ] a[const key_type& k], a[key_type&& k]
访问键的值，返回这个值的引用（而不是这个元素）

at a.at(const key_type& k), a.at(const key_type& k)
访问键的值，返回这个值的引用（而不是这个元素）

1.9.5 元素查找

find a.find(const key_type& k)
查找键为 k 的元素，返回迭代器，找不到则返回 unordered_map::end

count a.count(const key_type& k)
返回键为 k 的元素的个数

equal_range a.equal_range(const key_type& k)
返回 pair<iterator low, iterator high>

1.9.6 修改器

insert a.insert(...)

单元素 a.insert(const value_type& val), a.insert(P&& val)
返回一个 pair<iterator, bool>，如果插入成功，迭代器指向新元素，布尔值为真；如果插入失败，迭代器指向键相同的元素，布尔值为假

区间 a.insert(InputIterator first, InputIterator last)

带线索 a.insert(const_iterator hint, value_type& val)
a.insert(const_iterator hint, P&& val) 返回一个迭代器指向新元素或者键相同的元素

初始化列表 a.insert(initializer_list<value_type> il)

erase a.erase(...)
删除当前位置的元素

1. a.erase(const_iterator pos) 返回下一个元素的迭代器
2. a.erase(const_iterator first, const_iterator last) 返回下一个元素的迭代器
3. a.erase(const key_type& k) 返回删除的元素个数

swap a.swap(unordered_map<value_type>& b)
交换两个容器的内容

clear a.clear() 清空容器

emplace a.emplace(Args&&... args)
使用 args 构造并安放元素，返回 pair<iterator, bool>

emplace_hint a.emplace_hint(const_iterator pos, Args&&... args)
根据提示加速插入过程（容器可能不使用提示），使用 args 构造并安放元素，返回迭代器，插入成功则指向新元素，否则指向键相同的元素

1.9.7 散列桶

bucket_count a.bucket_count()
返回散列桶的个数，散列桶的个数直接影响散列表的使用率，容器会在必要的时候增加这个值，从而引起重新计算散列值

max_bucket_count a.max_bucket_count()
返回散列桶的最大可能的值

bucket_size a.bucket_size(size_type n)
返回散列桶 n 中的元素个数

bucket a.bucket(const key_type& k)
返回键 k 被散列到的桶的标号

样例：

```
#include <iostream>
#include <unordered_map>
using namespace std;
int main()
{
    unordered_map<int, int> m;
    for(int i=0; i<12; i++)
        m.insert({i,i});
    cout<<"bucket_count: "<<m.bucket_count()<<endl;
    cout<<"max_bucket_count: "<<m.max_bucket_count()<<endl;
    cout<<"bucket_size: "<<m.bucket_size(0)<<endl;
    cout<<"bucket of {3, 3}: "<<m.bucket(3)<<endl;
    return 0;
}
/*
output:

```

```
bucket_count: 23
max_bucket_count: 357913941
bucket_size: 1
bucket of {3, 3}: 3
*/
```

1.9.8 散列策略

load_factor a.load_factor()

返回 (float)load_factor = size / bucket_count

max_load_factor a.max_load_factor() 获取 load factor 最大值

a.max_load_factor(float z) 设置最大 load factor

rehash a.rehash(size_type n)

设置散列桶的个数为 n 或更多，如果 n 比当前散列桶的个数多，容器强制重散列；如果 n 比当前散列桶的个数少，则对容器没有影响

reserve a.reserve(size_type n)

设置适当的散列桶个数以适应至少 n 个元素，如果 n 大于散列桶的个数与最大负载因子的乘积，则容器会增加散列桶的个数，否则没有什么影响

1.9.9 观察者

hash_function a.hash_function()

返回散列函数

key_eq key_equal key_eq()

返回判断相等的函数

get_allocator alloctor_type get_allocator()

返回分配器

1.9.10 关于 unordered_multimap

unordered_multimap 与 unordered_map 基本相同，但在插入元素时不会出现因键相同而插入失败的情况。

1.10 unordered_set

包含头文件:<unordered_set> C++11

unordered_set，无序集合，和 set 相比，存储在其中的元素并不是有序的。

```
template < class Key,                        // unordered_set::key_type/value_type
           class Hash = hash<Key>,           // unordered_set::hasher
```

```
    class Pred = equal_to<Key>,       // unordered_set::key_equal
    class Alloc = allocator<Key>      // unordered_set::allocator_type
    > class unordered_set;
```

1.10.1 构造与析构

空的构造函数 unordered_set(size_type n,
const hasher& hf = hasher(),
const key_equal& eql = key_equal(),
const allocator_type& alloc = allocator_type())
构造一个空的容器，n 为散列桶的最小值，hf 为一个散列函数，eql 为判断两个元素是否相等的函数

范围构造函数 unordered_set(InputIterator first, InputIterator last,
size_type n,
const hasher& hf = hasher(),
const key_equal& eql = key_equal(),
const allocator_type& alloc = allocator_type())

复制构造函数 unordered_set (const unordered_set& ust)

移动构造函数 unordered_set (unordered_set&& ust)

初始化列表构造函数 unordered_set(initializer_list<value_type> il,
size_type n,
const hasher& hf = hasher(),
const key_equal& eql = key_equal(),
const allocator_type& alloc = allocator_type())

析构函数 ~unordered_set() 释放所有用分配器分配的内存，并销毁这个对象

赋值运算符 operator =
unordered_set& operator= (const unordered_set& x)
unordered_set& operator= (unordered_set&& x)
unordered_set& operator= (initializer_list<value_type> il)
使用当前的分配器，将 x 中所有的元素复制到容器中，复制之前容器中的内容被销毁

1.10.2 容量

size a.size() 返回容器中元素个数

max_size a.max_size()
返回容器可容纳的最大元素个数，这是根据已知的系统现状计算出来的最大可能性，但这不保证一定可以分配这么多空间来保存元素

empty a.empty() 测试容器是否为空

1.10.3 迭代器

begin a.begin(size_type n) 返回指向第一个元素的迭代器，参数 n 表示第 n 个散列桶

end a.end(size_type n) 返回指向最后一个元素后面的迭代器，参数 n 表示第 n 个散列桶

cbegin a.cbegin(size_type n) 返回指向第一个元素的常量迭代器，参数 n 表示第 n 个散列桶

cend a.cend(size_type n) 返回指向最后一个元素后面的常量迭代器，参数 n 表示第 n 个散列桶

1.10.4 元素查找

find a.find(const key_type& k)
查找键为 k 的元素，返回迭代器，找不到则返回 unordered_set::end

count a.count(const key_type& k)
返回键为 k 的元素的个数

equal_range a.equal_range(const key_type& k)
返回 pair<iterator low, iterator high>

1.10.5 修改器

insert a.insert(...)

单元素 a.insert(const value_type& val), a.insert(P&& val)
返回一个 pair<iterator, bool>，如果插入成功，迭代器指向新元素，布尔值为真；如果插入失败，迭代器指向 val 相等的元素，布尔值为假

区间 a.insert(InputIterator first, InputIterator last)

带线索 a.insert(const_iterator hint, value_type& val)
a.insert(const_iterator hint, P&& val) 返回一个迭代器指向新元素或者 val 相等的元素

初始化列表 a.insert(initializer_list<value_type> il)

erase a.erase(...)
删除当前位置的元素

1. a.erase(const_iterator pos) 返回下一个元素的迭代器
2. a.erase(const_iterator first, const_iterator last) 返回下一个元素的迭代器
3. a.erase(const key_type& k) 返回删除的元素个数

swap a.swap(unordered_set<value_type>& b)
交换两个容器的内容

clear a.clear() 清空容器

emplace a.emplace(Args&&... args)
使用 args 构造并安放元素，返回 pair<iterator, bool>

emplace_hint a.emplace_hint(const_iterator pos, Args&&... args)
根据提示加速插入过程（容器可能不使用提示），使用 args 构造并安放元素，返回迭代器，插入成功则指向新元素，否则指向值相等的元素

1.10.6 散列桶

bucket_count a.bucket_count()
返回散列桶的个数，散列桶的个数直接影响散列表的使用率，容器会在必要的时候增加这个值，从而引起重新计算散列值

max_bucket_count a.max_bucket_count()
返回散列桶的最大可能的值

bucket_size a.bucket_size(size_type n)
返回散列桶 n 中的元素个数

bucket a.bucket(const key_type& k)
返回值 k 被散列到的桶的标号

1.10.7 散列策略

load_factor a.load_factor()
返回 (float)load_factor = size / bucket_count

max_load_factor a.max_load_factor() 获取 load factor 最大值
a.max_load_factor(float z) 设置最大 load factor

rehash a.rehash(size_type n)
设置散列桶的个数为 n 或更多，如果 n 比当前散列桶的个数多，容器强制重散列；如果 n 比当前散列桶的个数少，则对容器没有影响

reserve a.reserve(size_type n)
设置适当的散列桶个数以适应至少 n 个元素，如果 n 大于散列桶的个数与最大负载因子的乘积，则容器会增加散列桶的个数，否则没有什么影响

1.10.8 观察者

hash_function a.hash_function()
返回散列函数

key_eq key_equal key_eq()
返回判断相等的函数

get_allocator alloctor_type get_allocator()
返回分配器

1.10.9 关于 unordered_multiset

unordered_multiset 与 unordered_set 基本相同，但在插入元素时不会出现因键相同而插入失败的情况。

1.11 vector

包含头文件:<vector>

vector，向量，是一个序列容器，动态数组，它的存储空间是连续的，当数组增大时，容器通过重新分配空间来保存更多的元素。对于随机访问和在尾部添加和删除的操作，效率是很高的。

1.11.1 构造与析构

空容器构造函数 vector()

构造一个空的容器，不含任何元素

填充构造函数 vector(size_type n, value_type& val)

构造一个含有 n 个元素的容器，每个元素是 val 的一个拷贝

范围构造函数 vector(InputIterator first, InputIterator last)

构造一个包含 [first, last) 的容器，保持原来的顺序

复制构造函数 vector(const deque& x)

构造一个容器 x 的拷贝，保持原来的顺序

移动构造函数 C++11 vector(vector&& x)

构造一个包含 x 中元素的容器；如果指定的分配器与 x 不同，则重新分配内存，否则直接引用 x 的内容，返回时 x 的状态是不确定的但是有效的。

初始化列表构造函数 C++11 vector (initializer_list<value_type> il)

构造一个和 il 内容相同的容器，保持原来的顺序

析构函数 ~vector()

释放所有用分配器分配的内存，并销毁这个对象

赋值运算符 operator =

vector& operator= (const vector& x)

vector& operator= (vector&& x)

vector& operator= (initializer_list<value_type> il)

使用当前的分配器，将 x 中所有的元素复制到容器中，复制之前容器中的内容被销毁

1.11.2 迭代器

begin a.begin() 返回指向第一个元素的迭代器

end a.end() 返回指向最后一个元素后面的迭代器

rbegin a.rbegin() 返回反向的迭代器，指向最后一个元素

rend a.rend() 返回反向的迭代器，指向第一个元素的前面

cbegin C++11 a.cbegin() 返回指向第一个元素的常量迭代器

cend C++11 a.cend() 返回指向最后一个元素后面的常量迭代器

crbegin C++11 a.crbegin() 返回反向的常量迭代器，指向最后一个元素

crend C++11 a.crend() 返回反向的常量迭代器，指向第一个元素的前面

1.11.3 容量

size a.size() 返回容器中元素个数

max_size a.max_size()
返回容器可容纳的最大元素个数，这是根据已知的系统现状计算出来的最大可能性，但这不保证一定可以分配这么多空间来保存元素

resize a.resize(size_type n, value_type val)
将容器容量置为 n，如果当前容量小于 n，其余位置用 val 填充

empty a.empty() 测试容器是否为空

reserve a.reserve(size_type n)
改变容器容量，使得容器至少可以容纳 n 个元素

shrink_to_fit C++11 a.shrink_to_fit()
减少容器的内存占用使之与当前包含的元素占用空间相同

1.11.4 元素访问

operator [ ] a[size_type] 访问元素

at a.at(size_type) 访问元素

front a.front() 访问第一个元素

back a.back() 访问最后一个元素

data C++11 a.data()
返回向量存储元素的首地址

1.11.5 修改器

assign

1. a.assign(InputIterator first, InputIterator second)
   将容器置为 [first, last) 中的元素，顺序不变
2. a.assign(size_type n, const value_type& val)
   将容器置为 n 个值为 val 的元素
3. a.assign(initializer_list<value_type> il) C++11
   将容器置为 il 中的内容，顺序不变

push_back a.push_back(value_type val) 在容器尾部追加元素

pop_back a.pop_back() 在容器尾部删除元素

insert a.insert(...)

在当前位置插入元素，返回一个指向插入的第一个元素的迭代器

C++11 的位置迭代器为 const_iterator

1. a.insert(iterator pos, const value_type &val)
2. a.insert(iterator pos, size_type n, const value_type &val)
3. a.insert(iterator pos, InputIterator first, InputIterator last)
4. a.insert(const_iterator pos, value_type&& val) C++11
5. a.insert(const_iterator pos, initializer_list<value_type> il) C++11

erase a.erase(...)

删除当前位置的元素，返回下一个元素的迭代器

C++11 的位置迭代器为 const_iterator

1. a.erase(iterator pos)
2. a.erase(iterator first, iterator last)

swap a.swap(vector<value_type>& b)

交换两个容器的内容

clear a.clear() 清空容器

emplace C++11 a.emplace(const_iterator pos, Args&&... args)

使用 args 构造并安放元素，返回值为指向新安放的元素的迭代器

emplace_back C++11 a.emplace_back(Args&& ... args)

函数没有返回值

1.11.6 vector<bool>

这是 vector 模板的一个特例，用来保存 bool 类型并高效利用空间。

它在内存中并不是以一个 bool 数组的形式存在，而是用一个比特来表示一个 bool 值；

新的成员函数 flip 和 swap；

迭代器并不是一个指针，而是对比特的模拟访问。

除了 data()、emplace()、emplace_back()，包含 vector 的其他成员。

a.flip() 对每个 bool 值取反

a.swap(vector& x) 交换两个容器的内容

a.swap(reference b1, reference b2) 交换两个值

样例：

```
1 #include <iostream>
2 #include <vector>
3 using namespace std;
4 int main()
```

```
{
    vector<bool> a1, a2;
    a1.push_back(true);
    a1.push_back(false);
    a1.push_back(true);
    a2.push_back(false);
    a2.push_back(true);
    a1.flip();
    a1.swap(a1[0], a1[1]);
    a1.swap(a2);

    cout<<boolalpha;
    cout<<"a1 contains: ";
    for(int i=0; i<a1.size();i++) cout<<a1[i]<<' ';
    cout<<endl;
    cout<<"a2 contains: ";
    for(int i=0; i<a2.size();i++) cout<<a2[i]<<' ';
    cout<<endl;
    return 0;
}
/*
output:

a1 contains: false true
a2 contains: true false false
*/
```

第二章　类和类模板

2.1　bitset

包含头文件：<bitset>

一个 bitset 保存一个比特，只有 0 或 1 两种可能的值，这个类模拟一个定长的 bool 数组，但为了节省空间，每个元素只占一个比特，一个 bitset 的元素个数必须是在编译器可以确定的，对于可以动态改变的可以参考vector<bool> 。

```
template <size_t N> class bitset;
```

2.1.1　构造函数

默认构造函数 bitset()

每个元素都为 0 的默认构造函数

用整数初始化 bitset (unsigned long long val)

用 val 的二进制串初始化

用字符串初始化

```
bitset (const basic_string<charT,traits,Alloc>& str,
    typename basic_string<charT,traits,Alloc>::size_type pos = 0,
    typename basic_string<charT,traits,Alloc>::size_type n =
    basic_string<charT,traits,Alloc>::npos,
    charT zero = charT('0'), charT one = charT('1'))
```

后两个参数用来指定字符串中哪个字符代表 0，哪个字符代表 1

用 C 风格字符串初始化

```
bitset (const charT* str,
  typename basic_string<charT>::size_type n = basic_string<charT>::npos,
    charT zero = charT('0'), charT one = charT('1'))
```

2.1.2 访问及操作

operator [ ] a[size_type] 获取某个位置的值

count a.count() 获取 1 的个数

size a.size() 保存的比特的个数

test a.test(pos) 这个位置是否被设为 1

any a.any() 测试是否至少有一个为 1

none a.none() 测试是否都不为 1

all C++11 a.all() 测试是否都为 1

set set(size_t pos, bool val = true)
将 pos 位置的数设置为 val，不指定参数则将所有位设置为 1

reset reset(size_t pos)
将 pos 位置的数设为 0，不指定位置则将所有位设置为 0

flip flip(size_t pos)
将 pos 位置的数取反，不指定位置则所有位取反

to_string to_string(charT zero = charT('0'), charT one = charT('1'))

to_ulong a.to_ulong() 转换为 unsigned long

to_ullong a.to_ullong() 转换为 unsigned long long

位运算 重载了类似整形数的位运算的操作以及输入输出流操作，不再一一列举

样例：

```
#include <iostream>
#include <bitset>
#include <string>
using namespace std;
int main()
{
    bitset<16> a, b(0x7fff), c(string("11110000"));
    cout<<"test all: "<<a.all()<<endl;
    a[0]=1;
    a.flip();
    cout<<"test count: "<<a.count()<<endl;
    cout<<"b to_ulong: "<<b.to_ulong()<<endl;
    cout<<"b & c: "<< (b&c) <<endl;
    return 0;
}
```

```
/*
output:

test all: 0
test count: 15
b to_ulong: 32767
b & c: 0000000011110000
*/
```

2.2 complex

包含头文件：<complex>

这个头文件实现了直角坐标系下的复数类，并重载了一些函数和操作符。complex<float>、complex<double>、complex<long double> 这些实例只能显式转换到低精度。

2.2.1 成员函数

构造函数 complex(const T& re = T(), const T& im = T())
complex(const complex& x)
complex(const complex<U>& x)

imag a.imag() 获取虚部 a.imag(val) C++11 修改虚部

real a.real() 获取实部 a.real(val) C++11 修改实部

2.2.2 复数数值

real real(const complex<T>& x) 获取 x 的实部
real(ArithmeticType x)C++11 以 double 返回（可以看做是复数 complex<T>(x, 0)），若 x 为 float 或 long double 则为 complex<float>、complex<long double>

imag imag(const complex<T>& x) 获取 x 的虚部
imag(ArithmeticType x)C++11 返回 (double) 0

abs abs(const complex<T>& x) 返回 x 的绝对值

arg arg(const complex<T>& x)，arg(ArithmeticType x)C++11 返回复数的辅角

norm norm(const complex<T>& x)，norm(ArithmeticType x)C++11 返回绝对值的平方

conj conj(const complex<T>& x)，conj(ArithmeticType x)C++11 返回共轭复数

polar polar(const T& rho, const T& theta = 0) 返回极坐标 (rho, theta) 转换成直角坐标下的复数

proj proj(const complex<T>& x)，conj(ArithmeticType x)C++11 返回 x 在黎曼球面的投影

2.2.3 函数重载

以下函数可以应用到复数上：

cos, cosh, exp, log, log10, pow, sin, sinh, sqrt, tan, tanh

acos, acosh, asin, asinh, atan, atanh C++11

不再一一详述。

2.2.4 操作符重载

复数类重载了四则运算，四则复合赋值运算符，赋值运算符，相等和不等运算符，输入输出流操作符。复数没有大于小于的关系运算。

样例：

```
#include <iostream>
#include <complex>
using namespace std;
int main()
{
    std::complex<double> a(1,2);
    std::complex<double> b(a);
    a.imag(3);
    cout<<"a: "<<a<<endl;
    cout<<"a + b: "<< (a+b) <<endl;
    cout<<"a * b: "<< a*b <<endl;
    cout<<"norm(a): "<<norm(a)<<endl;
    cout<<"pow(a, 2): "<<pow(a,2)<<endl;
    cout<<"pow(a, b): "<<pow(a, b)<<endl;
    return 0;
}
/*
output:

a: (1,3)
a + b: (2,5)
a * b: (-5,5)
norm(a): 10
pow(a, 2): (-8,6)
pow(a, b): (-0.238513,-0.103676)
*/
```

2.3 initializer_list

C++11

包含头文件：<initializer_list>

这个类用来处理 C++ 中的初始化列表，即某个类的一组常量值组成的列表，比如：auto il = {1, 2, 3} 是整型数据的一个初始化列表。初始化列表对象可以看做是自动构造的数组，使用复制初始化构造并采用必要的隐式转换。初始化列表是对其包含的元素的一个引用，所以对它的拷贝并不会复制这些元素，而仍然是原数组的一个引用。

2.3.1 成员函数

构造函数 initializer_list() 构造空的初始化列表

size a.size() 返回列表中的元素个数

begin a.begin() 返回第一个元素的指针

end a.end() 返回最后一个元素后面位置的指针

样例：

```
#include <iostream>
#include <initializer_list>
using namespace std;
int main()
{
    initializer_list<int> il;
    il = {1, 2, 3};
    cout<<"il contains:";
    for(int x: il) cout<<' '<<x;
    cout<<endl;
    return 0;
}
/*
output:

il contains: 1 2 3
*/
```

2.4 string

包含头文件：<string>

这个类包括字符串类型，字符特征和一些转换函数。

2.4.1 类模板 basic_string

```
template < class charT,
    class traits = char_traits<charT>,    // basic_string::traits_type
    class Alloc = allocator<charT>        // basic_string::allocator_type
    > class basic_string;
```

这是对字符串类型的泛化。

2.4.1.1 构造与析构

空的构造函数 basic_string(const allocator_type& alloc = allocator_type())

复制构造函数 basic_string(const basic_string& str)
basic_string(const basic_string& str, const allocator_type& alloc)

子串构造函数 basic_string(const basic_string& str, size_type pos, size_type len = npos,
const allocator_type& alloc = allocator_type())

从 c 字符串构造 basic_string(const charT* s, const allocator_type& alloc = allocator_type())

从字符串指针构造 basic_string(const charT* s, size_type n,
const allocator_type& alloc = allocator_type())

填充构造函数 basic_string(size_type n, charT c, const allocator_type& alloc = allocator_type())

范围构造函数 basic_string(InputIterator first, InputIterator last,
const allocator_type& alloc = allocator_type())

初始化列表构造 C++11 basic_string(initializer_list<charT> il,
const allocator_type& alloc = allocator_type())

移动构造函数 C++11 basic_string(basic_string&& str)
basic_string(basic_string&& str, const allocator_type& alloc)

析构 *sim*basic_string()

赋值运算符 basic_string& operator=(const basic_string& str)
basic_string& operator= (const charT* s)
basic_string& operator= (charT c)
basic_string& operator= (initializer_list<charT> il)
basic_string& operator= (basic_string&& str)

2.4.1.2 迭代器

begin a.begin() 返回指向第一个元素的迭代器

end a.end() 返回指向最后一个元素后面的迭代器

rbegin a.rbegin() 返回反向的迭代器，指向最后一个元素

rend a.rend() 返回反向的迭代器，指向第一个元素的前面

cbegin C++11 a.cbegin() 返回指向第一个元素的常量迭代器

cend C++11 a.cend() 返回指向最后一个元素后面的常量迭代器

crbegin C++11 a.crbegin() 返回反向的常量迭代器，指向最后一个元素

crend C++11 a.crend() 返回反向的常量迭代器，指向第一个元素的前面

2.4.1.3 容量

size a.size() 返回字符的个数

length a.length() 返回字符串长度

max_size a.max_size()
返回容器可容纳的最大元素个数，这是根据已知的系统现状计算出来的最大可能性，但这不保证一定可以分配这么多空间来保存元素

resize a.resize(size_type n, charT c)
将容器容量置为 n，如果当前容量小于 n，其余位置用 c 填充

capcity a.capcity() 返回当前已经分配的空间大小

empty a.empty() 测试容器是否为空

clear a.clear() 清空字符串

reserve a.reserve(size_type n = 0)
改变容器容量，使得容器至少可以容纳 n 个元素

shrink_to_fit C++11 a.shrink_to_fit()
减少容器的内存占用使之与当前包含的元素占用空间相同

2.4.1.4 元素访问

operator [ ] a[size_type] 访问元素，返回元素的引用

at a.at(size_type) 访问元素，返回元素的引用

front C++11 a.front() 访问第一个元素，返回元素的引用

back C++11 a.back() 访问最后一个元素，返回元素的引用

2.4.1.5 修改器

operator+= 在字符串后面添加字符串，返回自身的引用

append 在字符串后面追加字符串，返回自身的引用

1. append(const basic_string& str)
2. append(const basic_string& str, size_type subpos, size_type sublen = npos)
3. append(const charT* s)

4. append(const charT* s, size_type n)
5. append(size_type n, charT c)
6. append(InputIterator first, InputIterator last)
7. append(initializer_list<charT> il) C++11

push_back a.push_back(charT c)

assign a.assign(...) 将字符串置为另一个字符串，返回自身的引用，参数包含 append 的所有类型且 assign(basic_string&& str)

insert 在特定位置插入字符串，第 6 到第 8 个返回插入字符串的第一个位置的迭代器，其他返回自身的引用

1. insert(size_type pos, const basic_string& str)
2. insert(size_type pos, const basic_string& str, size_type subpos, size_type sublen)
3. insert(size_type pos, const charT* s)
4. insert(size_type pos, const charT* s, size_type n)
5. insert(size_type pos, size_type n, charT c)
6. insert(const_iterator p, size_type n, charT c)
7. insert(const_iterator p, charT c)
8. insert(iterator p, InputIterator first, InputIterator last);
9. insert(const_iterator p, initializer_list<charT> il) C++11

erase erase(pos = 0, len = npos) 删除从 pos 开始的 len 个字符，返回自身的引用
erase(const_iterator p) 删除 p 指向的字符，返回删除的第一个字符的位置的迭代器
erase(const_iterator first, const_iterator last) 删除这个区间的字符，返回删除的第一个字符的位置的迭代器

replace 字符串替换，返回自身的引用

1. replace(size_type pos, size_type len, const str)
2. replace(const_iterator i1, const_iterator i2, const str)
3. replace(size_type pos, size_type len, const str, size_type subpos, size_type sublen)
4. replace(size_type pos, size_type len, const charT* s)
5. replace(const_iterator i1, const_iterator i2, const charT* s)
6. replace(size_type pos, size_type len, const charT* s, size_type n)
7. replace(const_iterator i1, const_iterator i2, const charT* s, size_type n)
8. replace(size_type pos, size_type len, size_type n, charT c)
9. replace(const_iterator i1, const_iterator i2, size_type n, charT c)
10. replace(const_iterator i1, const_iterator i2, InputIterator first, InputIterator last)

11. replace(const_iterator i1, const_iterator i2, initializer_list<charT> il) C++11

swap swap(basic_string& str) 交换两个字符串

pop_back C++11 pop_back() 删除最后一个字符

2.4.1.6 字符串操作

c_str a.c_str() 返回 C 类型字符串的首地址

data a.data() 返回 C 类型字符串的首地址

get_allocator a.get_allocator() 返回分配器的拷贝

copy a.copy(charT* s, size_type len, size_type pos = 0)
将子串拷贝到 s 开始的位置，返回实际拷贝的长度，不会在字符串末尾添加结束符

find 在字符串中 pos 的位置开始查找，返回找到的第一个匹配的起始位置，未找到则返回 basic_string::pos

1. find(const basic_string& str, size_type pos = 0)
2. find(const charT* s, size_type pos = 0)
3. find(const charT* s, size_type pos, size_type n) 查找字符串 s 的前 n 个字符
4. find(charT c, size_type pos = 0)

rfind 在字符串中 pos 的位置开始查找，返回找到的最后一个匹配的起始位置，未找到则返回 basic_string::pos

1. rfind(const basic_string& str, size_type pos = 0)
2. rfind(const charT* s, size_type pos = 0)
3. rfind(const charT* s, size_type pos, size_type n) 查找字符串 s 的前 n 个字符
4. rfind(charT c, size_type pos = 0)

find_first_of 在字符串中 pos 的位置开始查找，返回第一个和给定字符串中任何一个字符相等的字符的位置，未找到则返回 basic_string::pos

1. find_first_of(const basic_string& str, size_type pos = 0)
2. find_first_of(const charT* s, size_type pos = 0)
3. find_first_of(const charT* s, size_type pos, size_type n) 给定字符串 s 的前 n 个字符
4. find_first_of(charT c, size_type pos = 0)

find_last_of 在字符串中 pos 的位置开始查找，返回最后一个和给定字符串中任何一个字符相等的字符的位置，未找到则返回 basic_string::pos

1. find_last_of(const basic_string& str, size_type pos = 0)
2. find_last_of(const charT* s, size_type pos = 0)
3. find_last_of(const charT* s, size_type pos, size_type n) 给定字符串 s 的前 n 个字符

4. find_last_of(charT c, size_type pos = 0)

find_first_not_of 在字符串中 pos 的位置开始查找，返回第一个不和给定字符串中任何一个字符相等的字符的位置，未找到则返回 basic_string::pos

1. find_first_not_of(const basic_string& str, size_type pos = 0)
2. find_first_not_of(const charT* s, size_type pos = 0)
3. find_first_not_of(const charT* s, size_type pos, size_type n)
   给定字符串 s 的前 n 个字符
4. find_first_not_of(charT c, size_type pos = 0)

find_last_not_of 在字符串中 pos 的位置开始查找，返回最后一个不和给定字符串中任何一个字符相等的字符的位置，未找到则返回 basic_string::pos

1. find_last_not_of(const basic_string& str, size_type pos = 0)
2. find_last_not_of(const charT* s, size_type pos = 0)
3. find_last_not_of(const charT* s, size_type pos, size_type n)
   给定字符串 s 的前 n 个字符
4. find_last_not_of(charT c, size_type pos = 0)

substr substr(size_type pos = 0, size_type len = npos)
返回从 pos 开始的长度为 npos 的字符串

compare 比较两个字符串，返回 0 则相等，小于 0 则第一个字符串小，大于 0 则第二个字符串小

1. compare (const basic_string& str)
2. compare (size_type pos, size_type len, const basic_string& str)
3. compare (size_type pos, size_type len, const basic_string& str,
   size_type subpos, size_type sublen)
4. compare (const charT* s)
5. compare (size_type pos, size_type len, const charT* s)
6. compare (size_type pos, size_type len, const charT* s, size_type n)

2.4.1.7 函数重载

operator + 拼接字符串

关系运算符 ==, !=, <, <=, >, >=

swap swap(basic_string<charT, traits, Alloc>& x, basic_string<charT, traits, Alloc>& y)

operator >> << 输入输出流操作

getline getline(basic_istream<charT, traits>& is,
basic_string<charT, traits, Alloc>& str, charT delim)
getline(basic_istream<charT, traits>&& is,
basic_string<charT, traits, Alloc>& str, charT delim)
从 is 获取字符串直到遇到 delim（不指定则为换行符）或发生错误（如 eof），赋值给 str，返回 is

2.4.1.8 成员常量

static const size_type npos = -1
样例：

```
#include <iostream>
#include <string>
int main ()
{
    std::basic_string<
        char,
        std::char_traits<char>,
        std::allocator<char>
        > str ("Test");
    std::cout<<str<<std::endl;
    std::cout<<"find_first_of \"aeiou\": ";
    std::cout<<str.find_first_of("aeiou",0)<<std::endl;
    std::cout<<"compare with Tesw: ";
    std::cout<<str.compare("Tesw")<<std::endl;
    return 0;
}
/*
output:

Test
find_first_of "aeiou": 1
compare with Tesw: -1
*/
```

2.4.2 类模板 char_traits

这个类用来指定字符的属性并为某些字符序列的操作提供具体语义。标准库包含以下实例：
char_traits<char>
char_traits<wchar_t>
char_traits<char16_t> C++11
char_traits<char32_t> C++11

它们被 basic_string 对象和输入输出对象作为默认的字符属性，当然也可以自定义这些属性。

2.4.2.1 成员函数

eq bool eq(char_type c, char_type d) 判断两个字符是否相等

lt bool lt(char_type c, char_type d) 判断是否为小于关系

length size_t length(const char_type* s) 获取字符串长度

assign void assign(char_type& r, const char_type& c) 将 c 赋值给 r
char_type assign(char_type* p, site_t n, char_type c) 用 n 个 c 填充 p 开始的字符数组

compare int compare(const char_type* p, const char_type* q, size_t n)
用 eq() 和 lt() 比较两个字符数组，返回大于 0、小于 0、等于 0 的值表示二者为大于、小于、等于的关系

find char_type* find(const char_type* p, size_t n, const char_type& c)
使用 eq() 在 p 的前 n 个字符中查找等于 c 的第一个字符，返回指向它的指针，未找到则返回空指针

move char_type* move(char_type* dest, const char_type* src, size_t n)
复制 src 开始的 n 个字符到 dest（区间可以覆盖），返回 dest

copy char_type* copy(char_type* dest, const char_type* src, size_t n)
复制 src 开始的 n 个字符到 dest（区间不可以覆盖），返回 dest

eof int_type eof() 返回表示文件尾的值

not_eof int_type not_eof(int_type c)
如果 c 不是 eof 则返回 c，否则返回其它非 eof 的值

to_char_type char_type to_char_type (int_type c)
返回和 c 等效的字符型数据

to_int_type int_type to_int_type (char_type c)
返回和 c 等效的整型数据

eq_int_type bool eq_int_type (int_type x, int_type y)
如果认为 x 等于 y 返回 true，否则返回 false

样例：

```
1 #include <iostream>
2 #include <string>
3 #include <cctype>
4 #include <cstddef>
5
```

```
// traits with case-insensitive eq:
class my_traits: public std::char_traits<char>
{
public:
    static bool eq (char c, char d){
        return std::tolower(c)==std::tolower(d);
    }
    static const char* find (const char* s, std::size_t n, char c){
        while( n-- && (!eq(*s,c)) ) ++s;
        return s;
    }
};

int main ()
{
    std::basic_string<char,my_traits> str ("Test");
    std::cout << "T found at position " << str.find('t') << '\n';
    const char* p=my_traits::find("Test", 4, 't');
    std::cout << "found " << *p << std::endl;
    return 0;
}
/*
output:

T found at position 0
found T
*/
```

2.4.3 类的实例

string 即 basic_string<char>

u16string C++11 basic_string<char16_t>

u32string C++11 basic_string<char32_t>

wstring basic_string<wchar_t>

2.4.4 转换函数

C++11

stoi 将 base 进制的 str 转换为整型数，如果 idx 不为空，则将 str 中整数后面的第一个位置赋值给 idx 指向的值

int stoi(const string& str, size_t* idx = 0, int base = 10)

int stoi(const wstring& str, size_t* idx = 0, int base = 10)

stol 将 base 进制的 str 转换为长整型数，如果 idx 不为空，则将 str 中整数后面的第一个位置赋值给 idx 指向的值

long stol(const string& str, size_t* idx = 0, int base = 10)

long stol(const wstring& str, size_t* idx = 0, int base = 10)

stoul 将 base 进制的 str 转换为无符号长整型数，如果 idx 不为空，则将 str 中整数后面的第一个位置赋值给 idx 指向的值

unsigned long stoul(const string& str, size_t* idx = 0, int base = 10)

unsigned long stoul(const wstring& str, size_t* idx = 0, int base = 10)

stoll 将 base 进制的 str 转换为 64 位长整型数，如果 idx 不为空，则将 str 中整数后面的第一个位置赋值给 idx 指向的值

long long stoll(const string& str, size_t* idx = 0, int base = 10)

long long stoll(const wstring& str, size_t* idx = 0, int base = 10)

stoull 将 base 进制的 str 转换为无符号 64 位长整型数，如果 idx 不为空，则将 str 中整数后面的第一个位置赋值给 idx 指向的值

unsigned long long stoull(const string& str, size_t* idx = 0, int base = 10)

unsigned long long stoull(const wstring& str, size_t* idx = 0, int base = 10)

stof 将 str 转换为浮点型，如果 idx 不为空，则将 str 中浮点数后面的第一个位置赋值给 idx 指向的值

float stof(const string& str, size_t* idx = 0)

float stof(const wstring& str, size_t* idx = 0)

stod 将 str 转换为双精度浮点型，如果 idx 不为空，则将 str 中浮点数后面的第一个位置赋值给 idx 指向的值

double stod(const string& str, size_t* idx = 0)

double stod(const wstring& str, size_t* idx = 0)

stold 将 str 转换为长精度浮点型，如果 idx 不为空，则将 str 中浮点数后面的第一个位置赋值给 idx 指向的值

long double stold(const string& str, size_t* idx = 0)

long double stold(const wstring& str, size_t* idx = 0)

to_string 数据转换为字符串

1. string to_string(int val)
2. string to_string(long val)
3. string to_string(long long val)
4. string to_string(unsigned val)
5. string to_string(unsigned long val)
6. string to_string(unsigned long long val)

7. string to_string(float val)
8. string to_string(double val)
9. string to_string(long double val)

to_wstring 数据转换为宽字符串

1. wstring to_wstring(int val)
2. wstring to_wstring(long val)
3. wstring to_wstring(long long val)
4. wstring to_wstring(unsigned val)
5. wstring to_wstring(unsigned long val)
6. wstring to_wstring(unsigned long long val)
7. wstring to_wstring(float val)
8. wstring to_wstring(double val)
9. wstring to_wstring(long double val)

样例：

```
#include <iostream>
#include <string>
int main()
{
    std::string s1("1234asdf"), s2(s1);
    int x1, x2;
    std::string::size_type t1,t2;
    x1=std::stoi(s1,&t1,10);
    x2=std::stoi(s2,&t2,16);
    std::cout<<s1<<": "<<x1<<" and "<<s1.substr(t1)<<std::endl;
    std::cout<<s2<<": "<<x2<<" and "<<s2.substr(t2)<<std::endl;
    return 0;
}
/*
output:

1234asdf: 1234 and asdf
1234asdf: 74570 and sdf
*/
```

2.5 tuple

包含头文件：<tuple>

tuple 是可以将不同的几个数据类型打包在一起的一种类型，类似于 C 风格的 struct，但每个数据并没有自己的名字，而是通过它们的位置来访问，但这些类型必须在编译器就要确定下来。

2.5.1 tuple

2.5.1.1 成员函数

构造函数

1. tuple()
2. tuple(const tuple& tpl)
3. tuple(tuple&& tpl)
4. tuple(const tuple<UTypes...>& tpl)
5. tuple(tuple<UTypes...>&& tpl)
6. tuple(const Types&... elems)
7. tuple(UTypes&&... elems)
8. tuple(const pair<U1,U2>& pr)
9. tuple(pair<U1,U2>&& pr)
10. 以上构造函数对应的指定分配器的版本，如：
    tuple(allocator_arg_t aa, const Alloc& alloc)

operator = 返回自身的引用

1. operator= (const tuple& tpl)
2. operator= (tuple&& tpl)
3. operator= (const tuple<UTypes...>& tpl)
4. operator= (tuple<UTypes...>&& tpl)
5. operator= (const pair<U1, U2>& pr)
6. operator= (pair<U1, U2>&& pr)

swap a.swap(tuple& b) 交换两个相同类型的元组

2.5.1.2 成员函数

关系运算符 ==, !=, <, <=, >, >=
依次比较元组中的数据

swap swap(tuple<Types...>& x, tuple<Types...>& y)

get std::get<i>(mytuple) 返回元组中的第 i 个数据的引用

2.5.2 tuple_size

用来获取一个元组中元素的个数

2.5.3 tuple_element

用来获取元组中第 i 个数的类型

样例：

```
#include <iostream>
#include <tuple>
int main ()
{
    std::tuple<int,char,double> mytuple (1, 'a', 3.14);
    std::cout << "mytuple has ";
    std::cout << std::tuple_size<decltype(mytuple)>::value;
    std::cout << " elements." << '\n';
    std::tuple_element<0,decltype(mytuple)>::type first;
    first = std::get<0>(mytuple);
    std::cout << "the first element is: " << first << std::endl;
    return 0;
}
/*
output:

mytuple has 3 elements.
the first element is: 1
*/
```

2.5.4 函数

make_tuple make_tuple (Types&&... args)

构造一个元组对象

forward_as_tuple forward_as_tuple(Types&&... args)

用这些参数的右值引用构造一个 tuple

tie tie(Types&... args)

构造一个包含这些参数的左值引用的元组对象

tuple_cat tuple_cat(Tuples&&... tpls)

连接这些元组构成一个元组

样例：

```
#include <iostream>
#include <tuple>
#include <string>
```

```
4
5  void print(std::tuple<std::string&&,int&&> pack)
6  {
7      std::cout << std::get<0>(pack) << ", "
8                << std::get<1>(pack) << '\n';
9  }
10
11 int main()
12 {
13     std::string str ("test");
14     print(std::forward_as_tuple(str+" one",1));
15     print(std::forward_as_tuple(str+" two",2));
16     auto t1=std::make_tuple(1, 1.0, 'a');
17     int myint;
18     char mychar;
19     std::tie(myint,std::ignore,mychar)=t1;
20     std::cout << "myint is: " << myint << '\n';
21     std::cout << "mychar is: " << mychar << '\n';
22     auto t2 = std::tuple_cat(t1, std::tuple<int,char>(2, 'b'));
23     std::cout << "t2 has " << std::tuple_size<decltype(t2)>::value
24               <<" elements" << std::endl;
25     return 0;
26 }
27 /*
28 output:
29
30 test one, 1
31 test two, 2
32 myint is: 1
33 mychar is: a
34 t2 has 5 elements
35 */
```

2.6 valarray

包含头文件：<valarray>

这个头文件定义了 valarray 类和其辅助类和函数。

2.6.1 valarray

一个 valarray（数值数组）对象用来保存一组相同类型的值，并可以方便地对这些值进行数学运算，还可以通过 operator [ ] 对这些数的一个子集进行这些操作。

2.6.1.1 构造函数

空的构造函数 valarray()

含有 n 个元素 valarray (size_t n)

含有 n 个 val valarray (const T& val, size_t n)

数组的前 n 个元素 valarray (const T p, size_t n)

复制构造函数 valarray (const valarray& x)

移动构造函数 valarray (valarray&& x)

子集构造函数 valarray (const slice_array<T>& sub)

子集构造函数 valarray (const gslice_array<T>& sub)

子集构造函数 valarray (const mask_array<T>& sub)

子集构造函数 valarray (const indirect_array<T>& sub)

初始化列表构造函数 valarray (initializer_list<T> il)

2.6.1.2 其他成员函数

操作符 算术运算符、逻辑运算符、关系运算符以及复合赋值运算符

如果两个操作对象都是 valarray，则对两个对象中对应的元素进行操作，如果其中一个是单个的元素，则用数值数组中的每个元素和这个元素进行操作。

apply apply(T func(T)), apply (T func(const T&))

对数值数组中每个元素执行 func 操作，返回操作后的数组

cshift valarray cshift(int n)

返回将数值数组中的元素进行循环左移（n 为负则右移）后的数值数组

max T max() 返回最大的元素的值

min T min() 返回最小的元素的值

operator = 赋值运算符

1. valarray& operator=(const valarray& x)
2. valarray& operator=(valarray&& x)
3. valarray& operator=(const T& val)
4. valarray& operator=(const slice_array<T>& sub)
5. valarray& operator=(const gslice_array<T>& sub)

6. valarray& operator=(const mask_array<T>& sub)
7. valarray& operator=(const indirect_array<T>& sub)

operator [ ] 元素访问

1. operator[ ](size_t n) 访问数值数组中的第 n 个元素
2. 子集访问，如果数值数组由 const 修饰，则返回一个新的数组，否则返回该子集的左值引用
   - valarray<T> operator[ ](slice slc)
   - slice_array<T> operator[ ](slice slc)
   - valarray<T> operator[ ](const gslice& gslc)
   - gslice_array<T> operator[ ](const gslice& gslc)
   - valarray<T> operator[ ](const valarray<bool>& msk)
   - mask_array<T> operator[ ](const valarray<bool>& msk)
   - valarray<T> operator[ ](const valarray<size_t>& ind)
   - indirect_array<T> operator[ ](const valarray<size_t>& ind)

resize resize(size_t sz, T c = T())
数组大小变为 sz，所有的元素改为 c

shift valarray shift(int n)
返回将数值数组中的元素进行（非循环）左移（n 为负则右移）后的数值数组，空余位置用初始化函数构造

size size_t size() 返回数组大小

sum T sum() 返回数组中元素的和

swap swap (valarray& x) 交换两个相同类型的数值数组（大小可以不同）

样例：

```
#include <iostream>
#include <valarray>
int t(int c)
{
    return c/10;
}
int main ()
{
    int init[]= {10,20,30,40};
    std::valarray<int> first;
    std::valarray<int> second (5);
    std::valarray<int> third (10,3); //three 10
    std::valarray<int> fourth (init,4);
```

```
14      std::valarray<int> fifth (fourth);
15      std::valarray<int> sixth (fifth[std::slice(1,2,1)]);
16      first = fourth+fifth;
17      first = first.apply(t);
18      std::cout<<"max value in first is: "<<first.max()<<'\n';
19      std::cout<<"the sum of first is: "<<first.sum()<<'\n';
20      return 0;
21  }
22  /*
23  output:
24
25  max value in first is: 8
26  the sum of first is: 20
27  */
```

2.6.2 slice

这个类表示 valarray 类的一个切片器，它即不包含也不引用任何值，只为 operator [ ] 描述应当操作的值的下标，一个切片用一个起始位置，需要选择的元素个数和一个步长表示。

默认构造函数 slice()

初始化构造函数 slice(size_t start, size_t length, size_t stride)

拷贝构造函数 slice (const slice& x)

start size_t start() 返回起始位置

size size_t size() 返回需要选择的元素个数

srtide size_t stride() 返回步长

2.6.3 gslice

这个类表示 valarray 类的一个广义切片器，它即不包含也不引用任何值，只为 operator [ ] 描述应当操作的值的下标，一个广义切片用一个起始位置，一组（n 个）需要选择的元素个数和一组（n 个，与元素个数对应）步长表示。

默认构造函数 gslice()

初始化构造函数 gslice(size_t start, const valarray<size_t>& lengths,
const valarray<size_t>& strides)

拷贝构造函数 gslice (const gslice& x)

start size_t start() 返回起始位置

size valarray<size_t> size() 返回需要选择的元素个数

srtide valarray<size_t> stride() 返回步长

样例：

```
#include <iostream>
#include <cstddef>
#include <valarray>

int main ()
{
    std::valarray<int> foo (14);
    for (int i=0; i<14; ++i) foo[i]=i;

    std::size_t start=1;
    std::size_t lengths[]= {2,3};
    std::size_t strides[]= {7,2};

    std::gslice mygslice (start,
        std::valarray<std::size_t>(lengths,2),
        std::valarray<size_t>(strides,2));

    std::valarray<int> bar = foo[mygslice];

    std::cout << "gslice:";
    for (std::size_t n=0; n<bar.size(); n++)
        std::cout << ' ' << bar[n];
    std::cout << '\n';
    return 0;
}
/*
output:

gslice: 1 3 5 8 10 12
*/
```

2.6.4 中间类

slice_array 用来表示用分片器取得的元素的引用，是数值数组的下标运算符使用分片器产生的

gslice_array 用来表示用广义分片器取得的元素的引用，是数值数组的下标运算符使用广义分片器产生的

mask_array 数值数组的下标运算符使用 mask(valarray<bool>) 产生

indirect_array 数值数组的下标运算符使用包含下标的数值数组产生

样例：

```
#include <iostream>
#include <cstddef>
#include <valarray>

int main ()
{
    std::valarray<int> foo (10);
    for (int i=0; i<10; ++i) foo[i]=i;

    std::valarray<bool> mymask (10);
    for (int i=0; i<10; ++i)
        mymask[i] = ((i%2)==1);

    foo[mymask] *= std::valarray<int>(10,5);
    foo[!mymask] = 0;

    std::cout << "foo:";
    for (std::size_t i=0; i<foo.size(); ++i)
        std::cout << foo[i] << ' ';
    std::cout << '\n';

    std::size_t sel[] = {3,5,6};
    std::valarray<std::size_t> selection (sel,3);
    foo[selection]=-1;

    std::cout << "foo:";
    for (std::size_t i=0; i<foo.size(); ++i)
        std::cout << foo[i] << ' ';
    std::cout << '\n';
    return 0;
}
/*
output:

foo:0 10 0 30 0 50 0 70 0 90
foo:0 10 0 -1 0 -1 -1 70 0 90
*/
```

2.6.5 全局函数

函数对每个数值进行操作，返回一个新的数值数组，这些函数包括 abs, acos, asin, atan, atan2, cos, cosh,exp, log, log10, ow, sin, sinh, sqrt, tan, tanh。

第三章　算法与函数

3.1　algorithm

包含头文件:<algorithm>

这个头文件定义了一系列应用于一个范围内元素的函数，一个范围是指一个可以用迭代器或指针进行访问的序列，比如标准模板库定义的容器，这些操作通过迭代器直接实现，不影响容器的构造。

3.1.1　不改变序列的操作

all_of C++11 all_of (InputIterator first, InputIterator last, UnaryPredicate pred)
如果 [first, last) 内的元素都满足一元谓词 pred 或 first == last 则返回 true，否则返回 false

any_of C++11 any_of (InputIterator first, InputIterator last, UnaryPredicate pred)
如果至少有一个元素满足一元谓词则返回 true ，否则返回 false

none_of C++11 none_of (InputIterator first, InputIterator last, UnaryPredicate pred)
如果元素都不满足一元谓词则返回 true ，否则返回 false

for_each for_each (InputIterator first, InputIterator last, Function fn)
对每个元素执行函数 fn，返回值为 fn

find find (InputIterator first, InputIterator last, const T& val)
返回第一个等于 val 的元素的迭代器，如果没有则返回 last

find_if find_if (InputIterator first, InputIterator last, UnaryPredicate pred)
返回第一个满足 pred 的元素的迭代器，如果没有则返回 last

find_if_not C++11 find_if_not (InputIterator first, InputIterator last, UnaryPredicate pred)
返回第一个不满足 pred 的元素的迭代器，如果没有则返回 last

find_end find_end (ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2, ForwardIterator2 last2)
find_end (ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate pred)
在 [first1, last1) 查找 [first2, last2) 的最后一次出现，返回第一个元素的迭代器，如果没有则返回 last1

find_first_of find_first_of (InputIterator first1, InputIterator last1,
ForwardIterator first2, ForwardIterator last2)
find_first_of (InputIterator first1, InputIterator last1,
ForwardIterator first2, ForwardIterator last2, BinaryPredicate pred)
如果 [first1, last1) 中含有等于（或符合 pred）[first2, last2) 中任何一个元素，返回第一次出现的元素的迭代器，否则返回 last1

adjacent_find adjacent_find (ForwardIterator first, ForwardIterator last)
adjacent_find (ForwardIterator first, ForwardIterator last, BinaryPredicate pred)
在 [first, last) 中有相邻的两个元素相等或符合二元谓词 pred，返回第一个元素的迭代器，否则返回 last

count count (InputIterator first, InputIterator last, const T& val)
返回 [first, last) 中等于 val 的元素的个数

count_if count_if (InputIterator first, InputIterator last, UnaryPredicate pred)
返回 [first, last) 中令 pred 为真的元素的个数

mismatch mismatch(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2)
mismatch (InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, BinaryPredicate pred)
在区间 [first1, last1) 中顺次与 first2 开始的区间进行对比，如果有不同或不符合 pred 的元素，返回两个元素的迭代器的一个 pair，否则返回 last1 和相应的第二个区间的迭代器的一个 pair

equal equal (InputIterator1 first1, InputIterator1 last1, InputIterator2 first2)
equal (InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, BinaryPredicate pred)
如果 [first1, last1) 和从 first2 开始的区间的元素依次相同，返回 true，否则返回 false

is_permutation C++11 is_permutation(ForwardIterator1 first1,
ForwardIterator1 last1, ForwardIterator2 first2)
is_permutation (ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2, BinaryPredicate pred)
如果 [first1, last1) 和从 first2 开始的区间相等或符合 pred（不考虑顺序），则返回 true，否则返回 false

search search (ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2, ForwardIterator2 last2)
search (ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate pred)
在 [first1, last1) 中查找 [first2, last2) 第一次出现或顺次符合 pred 的第一个元素的迭代器，否则返回 last1

search_n search_n(ForwardIterator first, ForwardIterator last, Size count, const T& val)
search_n(ForwardIterator first, ForwardIterator last, Size count,

const T& val, BinaryPredicate pred)

在 [first, last) 中查找连续 count 次出现 val 或符合 pred 的起始位置的迭代器

3.1.2 改变序列的操作

copy copy(InputIterator first, InputIterator last, OutputIterator result)

将 [first, last) 拷贝到以 result 开始的范围内，返回拷贝后结束位置的迭代器

copy_n C++11 copy_n(InputIterator first, Size n, OutputIterator result)

将以 first 开始的 n 个元素拷贝到以 result 开始的位置上，返回拷贝后结束位置的迭代器

copy_if C++11 copy_if(InputIterator first, InputIterator last,
OutputIterator result, UnaryPredicate pred)

将 [first, last) 内符合 pred 的元素拷贝到以 result 开始的位置，返回拷贝后结束位置的迭代器

copy_backward copy_backward (BidirectionalIterator1 first,
BidirectionalIterator1 last, BidirectionalIterator2 result)

将 [first, last) 的内容拷贝到以 result 为结束位置的范围内，返回拷贝后范围的起始位置迭代器

move C++11 move(InputIterator first, InputIterator last, OutputIterator result)

移动元素，返回结束位置的迭代器

move_backward C++11 move_backward(BidirectionalIterator1 first,
BidirectionalIterator1 last, BidirectionalIterator2 result)

将 [first, last) 的元素移动到以 result 为结束位置的范围内

swap swap (T& a, T& b)

交换两个数的值

C++11 将该函数的定义更改到 utility 头文件下，并且支持两个等长数组的交换

swap_ranges swap_ranges(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2)

交换 [first1, last1) 和以 first2 开始的区间，返回第二个区间的结束位置的迭代器

iter_swap iter_swap (ForwardIterator1 a, ForwardIterator2 b)

交换两个迭代器的内容，无返回值

transform transform(InputIterator first1, InputIterator last1,
OutputIterator result, UnaryOperation op)

transform (InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, OutputIterator result, BinaryOperation binary_op)

对一个区间的元素执行 op 或对两个区间的对应元素执行 binary_op，结果保存到 result 开始的区间内，返回保存元素的结束位置的迭代器

replace replace(ForwardIterator first, ForwardIterator last,
const T& old_value, const T& new_value)

将区间内的 old_value 替换成 new_value，无返回值

replace_if replace_if(ForwardIterator first, ForwardIterator last,
UnaryPredicate pred, const T& new_value)
将区间内符合 pred 的元素替换成 new_value，无返回值

replace_copy replace_copy(InputIterator first, InputIterator last, OutputIterator result,
const T& old_value, const T& new_value)
将 [first, last) 拷贝到以 result 开始的范围内，并将 old_value 替换成 new_value，返回目的区间的结束位置的迭代器

replace_copy_if replace_copy_if(InputIterator first, InputIterator last, OutputIterator result,
UnaryPredicate pred, const T& new_value)
将 [first, last) 拷贝到以 result 开始的范围内，并将符合 pred 的元素替换成 new_value，返回目的区间的结束位置的迭代器

fill fill(ForwardIterator first, ForwardIterator last, const T& val)
将范围用 val 填充，无返回值

fill_n fill_n(OutputIterator first, Size n, const T& val)
将以 first 开始的 n 个元素用 val 填充，C++11 返回结束位置的迭代器

generate generate(ForwardIterator first, ForwardIterator last, Generator gen)
将函数 gen 生成的元素依次写入范围内，无返回值

generate_n generate_n(OutputIterator first, Size n, Generator gen)
将函数 gen 生成的元素依次写入 first 开始的 n 个位置内，C++11 返回范围的结束位置迭代器

remove remove(ForwardIterator first, ForwardIterator last, const T& val)
返回 last2，使得 [first, last2) 不含 val，[last, last2) 的元素合法但未指定

remove_if remove_if (ForwardIterator first, ForwardIterator last, UnaryPredicate pred)
返回 last2，使得 [first, last2) 不含符合 pred 的元素，[last, last2) 的元素合法但未指定

remove_copy remove_copy (InputIterator first, InputIterator last,
OutputIterator result, const T& val)
将范围内除 val 外的元素拷贝到 result 开始的范围内，返回目的范围的结束位置的迭代器

remove_copy_if remove_copy_if (InputIterator first, InputIterator last,
OutputIterator result, UnaryPredicate pred)
将范围内除符合 pred 的元素拷贝到 result 开始的范围内，返回目的范围的结束位置的迭代器

unique unique(ForwardIterator first, ForwardIterator last)
unique(ForwardIterator first, ForwardIterator last, BinaryPredicate pred)
对于连续的相等或符合 pred 的元素，只保留第一个，元素依次向前覆盖，不会改变容器大小，返回新范围的结束位置的迭代器

unique_copy unique_copy(InputIterator first, InputIterator last, OutputIterator result)
unique_copy(InputIterator first, InputIterator last, OutputIterator result, BinaryPredicate pred)
拷贝范围内的元素到以 result 开始的位置，对于连续的相等或符合 pred 的元素，只保留第一个，返回目的范围的结束位置迭代器

reverse reverse(BidirectionalIterator first, BidirectionalIterator last)
使 [first, last) 反向，无返回值

reverse_copy reverse_copy(BidirectionalIterator first, BidirectionalIterator last, OutputIterator result)
反向拷贝 [first, last) 到 result 开始的位置，返回目的范围的结束位置迭代器

rotate rotate(ForwardIterator first, ForwardIterator middle, ForwardIterator last)
交换 [first, middle) 和 [middle, last) 两个范围，C++11 返回指向执行函数前 first 指向的值的迭代器

rotate_copy rotate_copy (ForwardIterator first, ForwardIterator middle, ForwardIterator last, OutputIterator result)
拷贝 [first, last) 到 result 开始的位置，交换 [first, middle) 和 [middle, last) 两个范围，返回目的范围的结束位置迭代器

random_shuffle random_shuffle(RandomAccessIterator first, RandomAccessIterator last)
random_shuffle(RandomAccessIterator first, RandomAccessIterator last, RandomNumberGenerator& gen)
将每个元素随机与另一个元素交换，函数 gen 用于指定某个元素与哪个位置的元素交换

shuffle C++11 shuffle(RandomAccessIterator first, RandomAccessIterator last, URNG&& g)
使用均匀随机数发生器（见<random>）g 对 [first, last) 内的元素重新排列

3.1.3 分界

is_partitioned C++11 is_partitioned(InputIterator first, InputIterator last, UnaryPredicate pred)
如果整个区域可以分为两部分，前一部分（可为空）使 pred 返回真，后一部分（可为空）返回假，则函数返回真，否则函数返回假

partition partition(BidirectionalIterator first, BidirectionalIterator last, UnaryPredicate pred)
将区域分成两部分，分别使 pred 为真和假，返回第二部分的起始位置迭代器

stable_partition stable_partition(BidirectionalIterator first, BidirectionalIterator last, UnaryPredicate pred)
将区域分成两部分，保证使 pred 同为真或假的元素的相对位置不变

partition_copy C++11 partition_copy(InputIterator first, InputIterator last, OutputIterator1 result_true, OutputIterator2 result_false, UnaryPredicate pred)
将区域分成两部分，并分别拷贝到以 result_true、result_false 开始的区域，返回两个区域的结束位置的迭代器的一个 pair

partition_point C++11 partition_point(ForwardIterator first, ForwardIterator last, UnaryPredicate pred)
返回第一个使 pred 为假的位置的迭代器，须保证该范围已经被分界

3.1.4 排序

sort sort (RandomAccessIterator first, RandomAccessIterator last)
sort(RandomAccessIterator first, RandomAccessIterator last, Compare comp)
将 [first, last) 内的元素按照 operator< 或 comp 进行升序排序，不保证相同元素的相对顺序

stable_sort stable_sort(RandomAccessIterator first, RandomAccessIterator last)
stable_sort(RandomAccessIterator first, RandomAccessIterator last, Compare comp)
排序但保证相同元素的相对顺序

partial_sort partial_sort(RandomAccessIterator first, RandomAccessIterator middle, RandomAccessIterator last)
partial_sort (RandomAccessIterator first, RandomAccessIterator middle, RandomAccessIterator last, Compare comp)
进行部分排序，使得 [first, middle) 是范围内最小的元素且已升序排序

partial_sort_copy partial_sort_copy(InputIterator first,InputIterator last, RandomAccessIterator result_first, RandomAccessIterator result_last)
partial_sort_copy (InputIterator first,InputIterator last, RandomAccessIterator result_first, RandomAccessIterator result_last, Compare comp)
将 [first, last) 内最小的那些元素拷贝到 [result_first, result_last) 且已升序排序，返回目的范围的结束位置的迭代器

is_sorted C++11 is_sorted (ForwardIterator first, ForwardIterator last)
is_sorted(ForwardIterator first, ForwardIterator last, Compare comp)
如果范围是已升序排序的，则返回 true，否则返回 false

is_sorted_until C++11 is_sorted_until(ForwardIterator first, ForwardIterator last)
is_sorted_until(ForwardIterator first, ForwardIterator last, Compare comp)
返回第一个未按升序排序的元素的迭代器

nth_element nth_element(RandomAccessIterator first, RandomAccessIterator nth, RandomAccessIterator last)
nth_element(RandomAccessIterator first, RandomAccessIterator nth, RandomAccessIterator last, Compare comp)
重新放置元素，使得对 nth 指向的元素有：前面的元素不比它大，后面的元素不比它小

3.1.5 二分查找

lower_bound lower_bound (ForwardIterator first, ForwardIterator last, const T& val)
lower_bound (ForwardIterator first, ForwardIterator last, const T& val, Compare comp)

返回不小于 val 的第一个元素的迭代器，须保证已排序

upper_bound upper_bound(ForwardIterator first, ForwardIterator last, const T& val)
upper_bound(ForwardIterator first, ForwardIterator last, const T& val, Compare comp)
返回大于 val 的第一个元素的迭代器，须保证已排序

equal_range equal_range(ForwardIterator first, ForwardIterator last, const T& val)
equal_range(ForwardIterator first, ForwardIterator last, const T& val, Compare comp)
返回 lower_bound 和 upper_bound 的 pair

binary_search binary_search(ForwardIterator first, ForwardIterator last, const T& val)
binary_search(ForwardIterator first, ForwardIterator last, const T& val, Compare comp)
如果范围内有和 val 相等的元素则返回 true，否则返回 false，须保证已排序

3.1.6 归并

merge merge(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2, OutputIterator result)
merge (InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2, OutputIterator result, Compare comp)
归并两个范围并保存到 result 开始的范围内，返回目的范围的结束位置的迭代器

inplace_merge inplace_merge(BidirectionalIterator first, BidirectionalIterator middle,
BidirectionalIterator last)
inplace_merge (BidirectionalIterator first, BidirectionalIterator middle,
BidirectionalIterator last, Compare comp)
归并 [first, middle) 和 [middle, last) 保存到 [first, last)

includes includes(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2)
includes(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2, Compare comp)
如果已排序的 [first1, last1) 包含已排序的 [first2, last2) 中所有元素，则返回 true，否则返回 false

set_union set_union (InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2, OutputIterator result)
set_union (InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2,
OutputIterator result, Compare comp)
求两个集合的并，保存到 result 开始的范围，返回目的范围的结束位置的迭代器

set_intersection set_intersection(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result)
set_intersection(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2,

OutputIterator result, Compare comp)
求两个集合的交，保存到 result 开始的范围，返回目的范围的结束位置的迭代器

set_difference set_difference(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result)
set_difference(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2,
OutputIterator result, Compare comp)
求两个集合的差，保存到 result 开始的范围，返回目的范围的结束位置的迭代器

set_symmetric_difference set_symmetric_difference(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2, OutputIterator result)
set_symmetric_difference(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2,
OutputIterator result, Compare comp)
求两个集合的对称差，保存到 result 开始的范围，返回目的范围的结束位置的迭代器

3.1.7 堆操作

push_heap push_heap(RandomAccessIterator first, RandomAccessIterator last)
push_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp)
此时 [first, last-1) 已经是一个堆，将 last-1 指向的元素放进堆中

pop_heap pop_heap(RandomAccessIterator first, RandomAccessIterator last)
pop_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp)
此时 [first, last) 已经是一个堆，将最大的元素放到 last-1 指向的位置

make_heap make_heap(RandomAccessIterator first, RandomAccessIterator last)
make_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp)
重新排列 [first, last) 的元素，使其成为一个堆

sort_heap sort_heap(RandomAccessIterator first, RandomAccessIterator last)
sort_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp)
排序 [first, last) 的元素

is_heap C++11 is_heap(RandomAccessIterator first, RandomAccessIterator last)
is_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp)
如果 [first, last) 是一个堆，返回 true，否则返回 false

is_heap_until C++11 is_heap_until(RandomAccessIterator first, RandomAccessIterator last)
is_heap_until(RandomAccessIterator first, RandomAccessIterator last, Compare comp)
返回第一个使 [first, last) 不是堆的元素的迭代器

3.1.8 其他

min min(const T& a, const T& b)
min(const T& a, const T& b, Compare comp)

min(initializer_list<T> il, Compare comp) C++11
返回两个元素或 il 中所有元素的最小值，前两个返回元素的引用

max max(const T& a, const T& b)
max(const T& a, const T& b, Compare comp)
max(initializer_list<T> il, Compare comp) C++11
返回两个元素或 il 中所有元素的最大值，前两个返回元素的引用

minmax C++11 minmax(const T& a, const T& b)
minmax(const T& a, const T& b, Compare comp)
minmax(initializer_list<T> il, Compare comp) C++11
返回两个元素或 il 中所有元素的最小和最大值的 pair，前两个返回元素的引用

min_element min_element (ForwardIterator first, ForwardIterator last, Compare comp)
返回区间内的最小值的迭代器，不指定 comp 则使用 operator <，区间为空则返回 last

max_element max_element (ForwardIterator first, ForwardIterator last, Compare comp)
返回区间内的最大值的迭代器，不指定 comp 则使用 operator <，区间为空则返回 last

minmax_element C++11 minmax_element (ForwardIterator first, ForwardIterator last, Compare comp)
返回区间内的最小和最大值的迭代器的 pair，不指定 comp 则使用 operator <，区间为空则返回 last

lexicographical_compare lexicographical_compare (InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, Compare comp)
如果第一个区间字典序小于第二个区间则返回 true，否则返回 false，不指定 comp 则使用 operator <

next_permutation next_permutation (BidirectionalIterator first, BidirectionalIterator last, Compare comp)
重新排列元素得到下一个置换，成功则返回 true

prev_permutation prev_permutation (BidirectionalIterator first, BidirectionalIterator last, Compare comp)
重新排列元素得到上一个置换

3.2 numeric

包含头文件：<numeric>

这个头文件定义了一系列应用于数字序列的操作，由于其灵活性，也可应用于其他序列。

3.2.1 accumulate

返回 [first, last) 范围内的数和 init 的和（或二元运算 op ）

- accumulate(InputIterator first, InputIterator last, T init)

- accumulate(InputIterator first, InputIterator last, T init, BinaryOperation binary_op)

3.2.2 adjacent_difference

计算 [first, last) 的相邻差（或二元运算 op）并保存到 result 开始的位置，返回目的区间的结束位置迭代器。若源区间为 x_i，目的区间为 y_i，则 $y_0 = x_0, y_1 = x_1 - x_0, y_2 = x_2 - x1, ...$。

- adjacent_difference(InputIterator first, InputIterator last, OutputIterator result)
- adjacent_difference(InputIterator first, InputIterator last,
  OutputIterator result, BinaryOperation binary_op)

3.2.3 inner_product

返回两个区间对应元素的积的和加到 init 上的结果（或 init = binary_op1(init, binary_op2(∗first1,∗first2))）。

- inner_product (InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, T init)
- inner_product (InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, T init,
  BinaryOperation1 binary_op1, BinaryOperation2 binary_op2)

3.2.4 partial_sum

计算区间元素的部分和（$y_i = sum(x_0, x_1, ..., x_i)$），并保存到 result 开始的位置，返回目的区间的结束位置的迭代器。

- partial_sum (InputIterator first, InputIterator last, OutputIterator result)
- partial_sum (InputIterator first, InputIterator last,
  OutputIterator result, BinaryOperation binary_op)

3.2.5 iota

用 (val, val+1, val+2, ...) 填充区间

- iota (ForwardIterator first, ForwardIterator last, T val)

3.3 utility

包含头文件：<utility>

这个头文件包含的是不相关的范围中共用的函数。

3.3.1 swap

C++11 之前，这个函数被包含在 <algorithm> 中。

元素 swap (T& a, T& b)

数组 C++11 swap(T (&a)[N], T (&b)[N])

3.3.2 make_pair

使用两个参数构造并返回一个数据对。

pair<T1, T2> make_pair (T1 x, T2 y)

3.3.3 forward

C++11

转发，如果参数不是一个左值引用则返回参数的右值引用，否则返回这个参数，而关于右值引用则是一个非常难以理解的话题，此处暂不多说。

- T&& forward (typename remove_reference<T>::type& arg)
- T&& forward (typename remove_reference<T>::type&& arg)

样例：

```
#include <utility>
#include <iostream>
// function with lvalue and rvalue reference overloads:
void overloaded (const int& x) {std::cout << "[lvalue]";}
void overloaded (int&& x) {std::cout << "[rvalue]";}
// function template taking rvalue reference to deduced type:
template <class T> void fn (T&& x) {
    overloaded (x);              // always an lvalue
    overloaded (std::forward<T>(x)); // rvalue if argument is rvalue
}

int main () {
    int a;
    std::cout << "calling fn with lvalue: ";
    fn (a);
    std::cout << '\n';

    std::cout << "calling fn with rvalue: ";
    fn (0);
    std::cout << '\n';
    return 0;
}
/*
output:

calling fn with lvalue: [lvalue][lvalue]
calling fn with rvalue: [lvalue][rvalue]
*/
```

3.3.4 move

C++11

返回参数的右值引用。

remove_reference<T>::type&& move (T&& arg)

样例：

```
#include <iostream>
#include <string>
int main()
{
    std::string s="string";
    std::cout<<"s.length: "<<s.length()<<'\n';
    std::string t=std::string(std::move(s));
    std::cout<<"s.length: "<<s.length()<<'\n';
    std::cout<<"t.length: "<<t.length()<<'\n';
    return 0;
}
/*
output:

s.length: 6
s.length: 0
t.length: 6
*/
```

3.3.5 move_if_noexcept

C++11

如果参数带有不产生异常的移动构造函数或不带有拷贝构造函数则返回参数的右值引用，否则返回左值引用。

```
typename conditional < is_nothrow_move_constructible<T>::value ||
    !is_copy_constructible<T>::value,
    T&&, const T& >::type move_if_noexcept(T& arg)
```

样例：

```
#include <utility>
#include <iostream>
// function with lvalue and rvalue reference overloads:
template <class T>
void overloaded (T& x)
```

```
{std::cout << "[lvalue]\n";}
template <class T>
void overloaded (T&& x)
{std::cout << "[rvalue]\n";}

struct A { // copyable + moveable (noexcept)
    A() noexcept {}
    A (const A&) noexcept {}
    A (A&&) noexcept {}
};
struct B { // copyable + moveable (no noexcept)
    B() {}
    B (const B&) {}
    B (B&&) {}
};
struct C { // moveable only (no noexcept)
    C() {}
    C (C&&) {}
};

int main () {
    std::cout << "A: "; overloaded (std::move_if_noexcept(A()));
    std::cout << "B: "; overloaded (std::move_if_noexcept(B()));
    std::cout << "C: "; overloaded (std::move_if_noexcept(C()));
    return 0;
}
/*
output:

A: [rvalue]
B: [lvalue]
C: [rvalue]
*/
```

3.3.6 declval

C++11

返回这个类的右值引用但不指向任何对象，这个函数只能应用于不带计算的操作（比如 sizeof 和 decltype）。这个辅助函数用来引用一个类的成员，尤其是当不知道类的构造函数或无法创建这个类的对象（如抽象基类）的时候。

typename add_rvalue_reference<T>::type declval()

样例：

```cpp
#include <iostream>
#include<utility>
struct A {
  virtual int value() = 0;
};
int main ()
{
    std::cout<<sizeof(std::declval<A>().value())<<'\n';
    decltype(std::declval<A>().value()) x=0; //int
    return 0;
}
/*
output:

4
*/
```

3.3.7 pair

C++11

将两个可能不同类型的元素组成一对，是 tuple 的特例。

构造函数

1. pair()
2. pair(const pair<U, V>& pr)
3. pair(pair<U, V>&& pr)
4. pair(const first_type& a, const second_type& b)
5. pair (U&& a, V&& b)
6. pair(piecewise_construct_t pwc, tuple<Args1...> first_args, tuple<Args2...> second_args)

operator = 返回自身

1. pair& operator= (const pair& pr)
2. pair& operator= (const pair<U, V>& pr)
3. pair& operator= (pair&& pr)
4. pair& operator= (pair<U, V>&& pr)

swap swap(pair& pr) 交换两个对象

3.3.8 piecewise_construct_t

C++11

这是一个空的类，用来重载 pair 的一个构造器。
类的一个常量 piecewise_construct。
样例：

```
#include <utility>
#include <iostream>
#include <tuple>
#include <vector>
#include <string>

int main () {
   std::pair < std::string, std::vector<int> >
      foo (
            std::piecewise_construct,
            std::forward_as_tuple("test"),
            std::forward_as_tuple(3,10)
         );
   std::cout << "foo.first: " << foo.first << '\n';
   std::cout << "foo.second:";
   for (int& x: foo.second) std::cout << ' ' << x;
   std::cout << '\n';

   return 0;
}
/*
output:

foo.first: test
foo.second: 10 10 10
*/
```

3.3.9 rel_ops

这是一个命名空间，用来声明关系操作的模板函数，如果类重载了 ==, < 两个关系运算，包含了这个命名空间就可以使用另外四个关系运算符