标准模板库STL

一、容器(container)

（一）序列式容器(sequential container)

vector、deque、list三者的初始化、遍历、尾部插入与删除、头部插入与删除(deque、list)(这里要讲vector与deque的底层实现)、中间插入insert(vector、deque会有失效)、清空元素、list的特殊操作(sort、merge、splice、unique、reverse)。。

1.0、头文件

```
#include <vector>
#include <deque>
#include <list>
```

1.1、初始化

1.1.1、直接初始化为空

```
vector<int> numbers ;
deque<int>  numbers;
list<int>  numbers;
```

1.1.2、初始化为多个数据(默认初始化为0)

```
vector<int> numbers(10, 1);
deque<int> numbers(10, 1);
list<int> numbers(10, 1);
```

1.1.3、使用迭代器范围

```
int arr[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
vector<int> numbers(arr, arr + 10);//左闭右开区间
```

1.1.4、使用大括号

```
vector<int> numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};//deque与list直接将vector替换即可
```

1.2、遍历

```
//2.1、使用下标进行遍历(要求容器必须是支持下标访问的，list不支持下标，所以就不适用)
for(size_t idx = 0; idx != numbers.size(); ++idx)
{
    cout << numbers[idx] << " ";
}
cout << endl;

//2.2、使用迭代器进行遍历
vector<int>::iterator it;//或者使用常量迭代器vector<int>::const_iterator it
```

```
for(it = numbers.begin(); it != numbers.end(); ++it)
{
    cout << *it << " ";
}
cout << endl;

//2.3、使用auto加迭代器
auto it = numbers.begin();
for(; it != numbers.end(); ++it)
{
    cout << *it << " ";
}
cout << endl;

//2.4、使用for加上auto进行遍历
for(auto &elem : numbers)
{
    cout << elem << " ";
}
cout << endl;
```

1.3、尾部插入与删除

push_back与pop_back，这两个函数比较简单，三个都适用。push_back有个两倍扩容(gcc)

1.4、头部插入与删除

push_front与pop_front，这两个函数与尾部插入一样，比较简单，但是vector不适用，只有deque与list适用。

为什么vector不支持在头部进行插入与删除呢?

探讨vector的**底层实现**，三个指针：

_M_start：指向第一个元素的位置

_M_finish：指向最后一个元素的下一个位置

_M_end_of_storage：指向当前分配空间的最后一个位置的下一个位置

```
&numbers;//error,只是获取对象栈上的地址,也就是_M_start的地址
&numbers[0];//
&*numbers.begin();//ok
int *pdata = numbers.data();//ok
cout << "pdata = " << pdata << endl;//使用printf，思考一下printf与cout打印地址的区别
```

类型萃取帮助我们提取出自定义类型进行深拷贝，而内置类型统一进行浅拷贝，也就是所谓的值拷贝。

编译器此时并不知道MyIter::value_type代表的是一个型别或是一个member function或是一个data member。关键词typename的用意在于告诉编译器这是一个型别，才能顺利通过编译。

探索deque的**底层实现**：

物理上是不连续的，逻辑上是连续的。中控器数组、多个连续的小片段、迭代器是一个类。

中控器数组是一个二级指针，包括中控器的大小。

小片段内部是连续的，但是片段与片段之间是不连续的。

迭代器是一个类，deque有两个迭代器指针，一个指向第一个小片段，一个指向最后一个小片段。

1.5、中间插入

insert在中间进行插入，list使用起来很好，但是deque与vector使用起来就有问题，因为vector是物理上连续的，所以在中间插入元素会导致插入元素后面的所有元素向后移动，deque也有类似情况，可能因为插入而引起扩容导致迭代器失效(指向了新的空间)，即使没有扩容，插入之后的迭代器也失效了(不再指向之前的元素)

1.5.1、insert的几种插入方式

```
//5.1、直接在某个位置插入一个元素
iterator insert( iterator pos, const T& value );
iterator insert( const_iterator pos, const T& value );
 numbers.insert(it, 22);

//5.2、直接在某个位置插入count个元素
void insert(iterator pos, size_type count, const T& value)
iterator insert(const_iterator pos, size_type count, const T& value)
numbers.insert(it1, 4, 44);

//5.3、直接在某个位置插入迭代器范围的元素
template<class InputIt> void insert(iterator pos, InputIt first, InputIt last)
template<class InputIt> iterator insert(const_iterator pos, InputIt first, InputIt last)

vector<int> vec{51, 52, 53, 54, 55, 56, 57, 58, 59};
numbers.insert(it, vec.begin(), vec.end());

//5.4、插入一个大括号范围的元素
iterator insert(const_iterator pos, std::initializer_list<T> ilist)
numbers.insert(it, std::initialiser_list<int>{1, 2, 3});
```

1.5.2、insert导致迭代器失效

insert在插入的时候，可能导致扩容，所以会出现迭代器失效的问题。对于vector可以探讨一下：

```
//在中间插入的时候，迭代器可能会失效，如果在去操作可能发生段错误
//因为push_back每次只会插入一个，所以可以按照统一的形式2 * size()
//但是insert的时候，插入的元素的个数不定的，所以就不能一概而论
//capacity() = n, size() = t, insert的时候，插入的元素个数为m
//1、m < n - t,这个时候就没有扩容，所以直接插入
//2、n - t < m < t,就按照t的2倍去进行扩容，新的空间就是2 * t
//3、n - t < m < n 且m > t,就按照 t + m去进行扩容
//4、m > n时，就按照t + m去进行扩容
void test()
{
    vector<int> numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};//3、大括号形式
    display(numbers);
    cout << "numbers.size() = " << numbers.size() << endl;
    cout << "numbers.capacity() = " << numbers.capacity() << endl;

    cout << endl << "在容器尾部进行插入: " << endl;
```

```
    numbers.push_back(10);
    numbers.push_back(11);
    display(numbers);
    cout << "numbers.size() = " << numbers.size() << endl;
    cout << "numbers.capacity() = " << numbers.capacity() << endl;
    cout << endl << "在容器尾部进行删除: " << endl;
    numbers.pop_back();
    display(numbers);

    cout << endl << "在容器vector中间进行插入: " << endl;
    auto it = numbers.begin();
    ++it;
    ++it;
    numbers.insert(it, 22);
    display(numbers);
    cout << "*it = " << *it << endl;
    cout << "numbers.size() = " << numbers.size() << endl;
    cout << "numbers.capacity() = " << numbers.capacity() << endl;

    numbers.insert(it, 4, 44);//根据插入个数，进行分类扩容
    display(numbers);
    cout << "*it = " << *it << endl;
    cout << "numbers.size() = " << numbers.size() << endl;
    cout << "numbers.capacity() = " << numbers.capacity() << endl;

    //正确办法是重置迭代器的位置
    vector<int> vec{51, 52, 53};
    auto it1 = numbers.begin();
    ++it1;
    ++it1;
    numbers.insert(it1, vec.begin(), vec.end());
    display(numbers);
    cout << "*it1 = " << *it1 << endl;
    cout << "numbers.size() = " << numbers.size() << endl;
    cout << "numbers.capacity() = " << numbers.capacity() << endl;
}
```

1.6、元素的删除(这点上课没有讲例子，大家复试之前可以看看)

erase函数，删除容器中的元素，删除一个元素，删除迭代器范围。注意删除之后，对于vector而言，会导致删除之后的元素前移，从而导致删除之后的所有迭代器失效(迭代器的位置没有改变，但是因为元素的移动，导致迭代器指向的不是删除之前的元素，所以需要注意)(当然list的删除没有影响)。deque比vector复杂，要看pos前后的元素个数来决定(deque的erase函数可以看STL源码，需要看删除位置与size()的一半的大小，然后看是挪动前一半还是后一半，尽量减少挪动的次数)。

```
iterator erase(iterator position);
iterator erase(iterator first, iterator last);

//错误的用法：error
vector<int> vec;
for (int i = 0; i < 10; ++i)
{
    vec.push_back(i);
}
//删除不彻底，不能把连续的相同值删除
for (vector<int>::iterator it = vec.begin(); it != vec.end(); ++it)
{
```

```
    if(1 == *it) {
        vec.erase(it);
    }
}
//正确的用法：ok
vector<int> vec;
for (int i = 0; i < 10; ++i)
{
    vec.push_back(i);
}
vec.push_back(1);
vec.push_back(1);
vec.push_back(1);
vec.push_back(1);

//因为删除之后元素会移动，直接把迭代器的变化放在else语句中
for (auto it = vec.begin(); it != vec.end();)
{
    if (1 == *it) {
        it = vec.erase(it);
    }
    else {
        ++it;
    }
}
```

1.7、清空元素

clear(清空元素，元素个数为0)与shrink_to_fit(缩减到刚刚好)(vector与deque有这个函数，但是list没有，list清空元素时候，节点都销毁了)

1.8、获取容器中元素个数与空间大小

size()获取容器中元素个数，三者都有该函数

capacity()获取容器申请的空间大小，只有vector有这个函数。

1.9、list的特殊操作

1.9.1、排序函数sort

```
void sort();//默认以升序进行排序，其实也就是，使用operator<进行排序

template< class Compare > void sort(Compare comp);//其实也就是传入一个具有比较的类型，即函数对象
template <typename T1, typename T2>
struct Compare
{
    bool operator()(const T1 &a, const T2 &b) const
    {
        return a < b;
    }
};
```

1.9.2、移除重复元素unique

注意使用unique的时候，要保证元素list是已经排好顺序的，否则使用unique是没有用的。

1.9.3、逆置链表中的元素reverse

将链表逆置输出

1.9.4、合并链表的函数merge

合并的链表必须是有序的，如果没有顺序，合并没有效果。两个链表合并之后，另一个链表就为空了。

1.9.5、从一个链表转移元素到另一个链表splice

```
void splice(const_iterator pos, list& other)//移动一个链表到另一个链表的某个指定位置
void splice(const_iterator pos, list&& other)
//移动一个链表中的某个元素到另一个链表的某个指定位置
void splice(const_iterator pos, list& other, const_iterator it)
void splice(const_iterator pos, list&& other, const_iterator it);
//移动一对迭代器范围元素到另一个链表的某个指定位置
void splice(const_iterator pos, list& other, const_iterator first,
const_iterator last)
void splice(const_iterator pos, list&& other,const_iterator first,
const_iterator last)
```

整个list的特殊成员函数的使用

```
void test()
{
    list<int> numbers{8, 3, 4, 3, 6, 7, 6, 9, 1, 8, 9};
    numbers.unique();
    numbers.sort();//默认情况是以小于符号进行排序
    numbers.unique();//unique在去除重复元素的时候，链表必须为有序

    list<int> numbers2{11, 22, 33};
    numbers.merge(numbers2);

    numbers.reverse();

    list<int> numbers3{41, 42, 43, 44, 45, 46, 47};
    auto it = numbers.begin();
    ++it;
    ++it;

    auto it2 = numbers3.begin();
    ++it2;
    ++it2;
    numbers.splice(it, numbers3, it2);

    auto it3 = numbers.end();
    it = numbers.begin();
    --it3;
    numbers.splice(it, numbers, it3);
}
```

1.9.6、另外的一种插入方式emplace与emplace_back

emplace有放置的意思

```
void test()
{
    vector<Point> points;
    points.reserve(10);
    points.emplace_back(1, 2);
    points.emplace_back(3, 4);
    /* points.push_back(Point(1, 2));//减少临时变量的产生对内存的消耗 */
    points[0].print();
    points[1].print();
}
```

（二）关联式容器(associative container)

(set、multiset、map、multimap)：底层实现使用红黑树。初始化、遍历、查找(count,find)、插入(insert)(set与map需要判断插入是不是成功)，自定义类型需要去对Compare进行改写(std::less、std::greater、函数对象)。。

2.1、初始化

使用形式与序列式容器完全一致(可以参考序列式容器)。

set特征：

1、不能存放关键字key相同的元素，关键字必须唯一

2、默认以升序进行排列

3、底层实现是红黑树

红黑树的五大特征：

1、节点不是红色就是黑色

2、根节点是黑色的

3、叶子节点也是黑色的

4、如果一个节点是红色的，那么它的左右孩子节点必须是黑色的

5、从根节点到叶子节点上所有路径要保证黑色节点的个数相同

multiset特征：

1、可以存放关键字key相同的元素，关键字不唯一

2、默认以升序进行排列

3、底层实现是红黑树

map的特征：

1、存放的是键值对，也就是也个pair，即`pair<const Key, value>`,key值必须唯一，不能重复

2、默认按照关键字key进行升序排列

3、底层实现是红黑树

```
map<int, string> cities =
{
    {1, "北京"},//大括号方式进行初始化
    {2, "上海"}
    std::pair<int,string>(3, "广州"),//map中存的是pair类型，就直接使用pair
    std::pair<int,string>(4, "深圳"),
    std::make_pair(5, "武汉"),//std::make_pair()函数返回类型就是pair类型
    std::make_pair(6, "南京"),
    std::make_pair(2, "上海"),
    std::make_pair(3, "南京"),
};
display(cities);
```

multimap的特征：

1、存放的是键值对，也就是也个pair，即`pair<const Key, value>`,key值不唯一，可以重复

2、默认按照关键字key进行升序排列

3、底层实现是红黑树

2.2、遍历

使用形式与序列式容器完全一致(可以参考序列式容器)

```
template <typename Container>
void display(const Container &c)
{
    for(auto &elem : c)
    {
        cout << elem << " ";
    }
    cout << endl;
}
```

2.3、查找

两个函数count(返回元素的数目)与find函数(返回查找后的迭代器的位置)

```
size_t cnt = numbers.count(1);
size_t cnt2 = numbers.count(10);


set<int>::iterator it = numbers.find(10);
if(it == numbers.end())
{
    cout << "该元素不存在number中" << endl;
}
else
{
    cout << "该元素存在numbers中：  "  << *it << endl;
}
```

还有三个用于查找的函数，这个函数对于multiset与multimap效果要好一点。

```
auto it = numbers.lower_bound(2);//不大于key的第一个位置
auto it2 = numbers.upper_bound(2);//大于key的第一个位置
```

```
3
4  while(it != it2)
5  {
6      cout << *it << " ";
7      ++it;
8  }
9  cout << endl;
10
11 cout << endl << endl;
12 std::pair<multiset<int>::iterator, multiset<int>::iterator> ret2 =
   numbers.equal_range(2);
13 /* auto ret2 = numbers.equal_range(2); */
14 while(ret2.first != ret2.second)
15 {
16     cout << *ret2.first << " ";
17     ++ret2.first;
18 }
19 cout << endl;
```

2.4、插入

insert相关的几个函数原型：

```
1  //value_type，若是set/multiset代表的是key，若是map/multimap代表的是pair<const
   key, value>
2  std::pair<iterator,bool> insert( const value_type& value );
3  std::pair<iterator,bool> insert( value_type&& value );
4  iterator insert( iterator hint, const value_type& value );
5  iterator insert( const_iterator hint, const value_type& value );
6  iterator insert( const_iterator hint, value_type&& value );
7  template< class InputIt > void insert( InputIt first, InputIt last );
8  void insert( std::initializer_list<value_type> ilist );
9  insert_return_type insert(node_type&& nh);
10 iterator insert(const_iterator hint, node_type&& nh);
```

具体代码示例如下：

```
1  auto  ret = numbers.insert(10);//如果是set插入，需要判断返回值，multiset就不需要
2  if(ret.second)
3  {
4      cout << "添加成功" << *ret.first << endl;
5  }
6  else
7  {
8      cout << "添加失败，这个元素已经存在set之中: " << endl;
9  }
10
11 //添加一对迭代器范围元素
12 vector<int> vec{10, 9, 8, 5, 30, 20, 11, 39};
13 numbers.insert(vec.begin(), vec.end());
14
15
16 numbers.insert(std::initializer_list<int>({100, 21, 500}));
17 display(numbers);
18
19 auto ret3 = points.insert(std::make_pair("999", Point(10, 40)));
20 if(ret3.second)
```

```
{
    cout << "添加元素成功 :  " << ret3.first->first << "---->"
         << ret3.first->second << endl;
}
else
{
    cout << "添加失败，该元素存在于map之中 : " << ret3.first->first << "--->"
         << ret3.first->second << endl;
}
```

2.5、删除

```
void erase(iterator pos);//删除某个位置元素
iterator erase(const_iterator pos);
iterator erase(iterator pos);
void erase(iterator first, iterator last);//删除某个返回的元素
iterator erase(const_iterator first, const_iterator last);
size_type erase(const key_type& key);
```

```
auto it2 = numbers.begin();
++it2;
++it2;
numbers.erase(it2);//删除某个位置元素
display(numbers);
```

2.6、修改

由于底层实现是红黑树，所以不支持修改。

2.7、下标访问

只有map支持下标访问，且兼具插入的功能，所以使用起来比较方便，但是时间复杂度是O(logN)

```
cout << "points[\"1\"] = " << points["1"] << endl;
cout << "points[\"0\"] = " << points["0"] << endl;
points["0"] = Point(10, 20);

```

2.8、针对自定义类型的操作

首先看看容器的模板类型

```
//这两个头文件都在#include <set>中
template< class Key,
          class Compare = std::less<Key>,
          class Allocator = std::allocator<Key>
        > class set;

template< class Key,
          class Compare = std::less<Key>,
          class Allocator = std::allocator<Key>
        > class multiset;

//这两个头文件都在#include <map>中
template< class Key,
          class T,
```

```
          class Compare = std::less<Key>,
          class Allocator = std::allocator<std::pair<const Key, T> >
        > class map;

template< class Key,
          class T,
          class Compare = std::less<Key>,
          class Allocator = std::allocator<std::pair<const Key, T> >
        > class multimap;
```

由上面的类模板里面的模板参数可以看出，对于自定义类型而言，std::less就不够用了，所以需要针对自定义类型进行改写，否则就会出bug。下面以自定义类型Point为例，以点到原点的距离为对象进行比较：

```
class Point
{
public:
    double getDistance() const
    {
        return hypot(_ix, _iy);
    }


    friend std::ostream &operator<<(std::ostream &os, const Point &rhs);//输出流运算符
    friend bool operator<(const Point &lhs, const Point &rhs);//std::less
    friend bool operator>(const Point &lhs, const Point &rhs);//std::greater
    friend class PointComparator;//函数对象

private:
    int _ix;
    int _iy;
};


//std::less其实也就是比较函数，小于符号
bool operator<(const Point &lhs, const Point &rhs)
{
    if(lhs.getDistance() < rhs.getDistance())
    {
        return true;
    }
    else if(lhs.getDistance() == rhs.getDistance())
    {
        return (lhs._ix < rhs._ix) || (lhs._iy < rhs._iy);
    }
    else
    {
        return false;
    }
}

//std::greater其实也就是比较函数，大于符号
bool operator>(const Point &lhs, const Point &rhs)
{
    //......
}
```

```
43
44  //std::less针对Point进行特化
45  namespace std
46  {
47  template <>
48  struct less<Point>
49  {
50      bool operator()(const Point &lhs, const Point &rhs)
51      {
52          //......
53      }
54  };
55  }//end of namespace std
56
57  //函数调用
58  struct PointComparator
59  {
60      bool operator()(const Point &lhs, const Point &rhs)
61      {
62        //.......
63      }
64  };
```

```
1   void test()
2   {
3       set<Point, PointComparator> points =
4       {
5           Point(1, 2),
6           Point(3, 4),
7           Point(-1, 2),
8           Point(3, 4),
9           Point(0, 6),
10      };
11      display(points);
12  }
13
14  //multiset而言
15  void test()
16  {
17      /* multiset<Point> points =  */
18      /* multiset<Point,std::greater<Point>> points =  */
19      multiset<Point, PointComparator> points
20      {
21          Point(1, 2),
22          Point(3, 4),
23          Point(-1, 2),
24          Point(1, -2),
25          Point(3, 4),
26          Point(0, 6),
27      };
28
29  }
30  //对于map与multimap而言，key值不是自定义类型，或者说不是我们自己定义的类型，所以无需这些
    函数或者特化
31  void test()
32  {
33      /* map<string, Point> points = */
```

```
    /* map<string, Point, std::greater<string>> points = */
    map<string, Point> points =
    {
        {"1", Point(1, 2)},
        pair<string, Point>("22", Point(-1, 2)),
        pair<string, Point>("333", Point(5, 6)),
        std::make_pair("4444", Point(0, 6)),
        std::make_pair("22", Point(0, 6)),
    };
    display(points);
}

void test()
{
    /* multimap<string, Point> points = */
    /* multimap<string, Point, std::greater<string>> points = */
    multimap<string, Point> points =
    {
        {"1", Point(1, 2)},
        pair<string, Point>("22", Point(-1, 2)),
        pair<string, Point>("333", Point(5, 6)),
        std::make_pair("4444", Point(0, 6)),
        std::make_pair("22", Point(-1, 2)),
    };
    display(points);
}
```

（三）无序关联式容器(unordered associative container)

无序关联式容器的底层实现使用的是哈希表，关于哈希表有几个概念需要了解：哈希函数、哈希冲突、解决哈希冲突的方法、装载因子(装填因子、负载因子)

3.1、哈希函数

是一种根据关键码key去寻找值的数据映射的结构，即：根据key值找到key对应的存储位置。

```
size_t index = H(key)//由关键字获取所在位置
```

3.2、哈希函数的构造

1、直接定址法： H(key) = a * key + b

2、平方取中法： key^2 = 1234^2 = 1522756 ------>227

3、数字分析法：H(key) = key % 10000；

4、除留取余法：H(key) = key mod p (p <= m, m为表长)

3.3、哈希冲突

就是对于不一样的key值，可能得到相同的地址,即:H(key1) = H(key2)

3.4、解决哈希冲突的解决办法

1、开放定址法

2、链地址法 (推荐使用这种，这也是STL中使用的方法)

3、再散列法

4、建立公共溢出区

3.5、装载因子

装载因子 a = (实际装载数据的长度n)/(表长m)

a越大，哈希表填满时所容纳的元素越多，空闲位置越少，好处是提高了空间利用率，但是增加了哈希碰撞的风险，降低了哈希表的性能，所以平均查找长度也就越长；但是a越小，虽然冲突发生的概率急剧下降，但是因为很多都没有存数据，空间的浪费比较大，经过测试，装载因子的大小在[0.5~0.75]之间比较合理，特别是0.75

3.6、哈希表的设计思想

用空间换时间，注意数组本身就是一个完美的哈希，所有元素都有存储位置，没有冲突，空间利用率也达到极致。

3.7、四种无序关联式容器

(unordered_set、unordered_multiset、unordered_map、unordered_multimap)：底层实现使用哈希表。针对于自定义类型需要自己定义std::hash函数与std::equal_to函数，四种容器的类模板如下：

```
//unordered_set与unordered_multiset位于#include <unordered_set>中
template < class Key,
           class Hash = std::hash<Key>,
           class KeyEqual = std::equal_to<Key>,
           class Allocator = std::allocator<Key>
         > class unordered_set;

template < class Key,
           class Hash = std::hash<Key>,
           class KeyEqual = std::equal_to<Key>,
           class Allocator = std::allocator<Key>
         > class unordered_multiset;

//unordered_map与unordered_multimap位于#include <unordered_map>中
template< class Key,
          class T,
          class Hash = std::hash<Key>,
          class KeyEqual = std::equal_to<Key>,
          class Allocator = std::allocator< std::pair<const Key, T> >
        > class unordered_map;

template< class Key,
          class T,
          class Hash = std::hash<Key>,
          class KeyEqual = std::equal_to<Key>,
          class Allocator = std::allocator< std::pair<const Key, T> >
        > class unordered_multimap;
```

针对内置类型，初始化、遍历、查找、插入、删除、修改、下标访问这些与关联式容器类似，无序关联式容器中元素没有顺序，底层采用的是**哈希表**。特别是：对于自定义类型而言，没有针对key值对应的哈希函数以及比较函数，所以需要自己写。

```
namespace std
{
template <>
class hash<类名>
```

```
{
public:
size_t operator()(const 类名 &rhs) const//注意返回类型确定了就是std::size_t
{
    //hash函数的实现
    return (pt.getX() << 1) ^ (pt.getY() << 1);//哈希函数的设置
}//end of operator

};//end of class hash
}//end of namespace std
```

以及关于std::equal_to的实现。

```
bool operator()(const T &lhs, const T &rhs) const //返回类型是bool，参数是两个，返回时判断等号
{
    return lhs == rhs;
}

//或者直接重载等号运算符,设置为友元
bool operator==(const T &lhs, const T &rhs)
{
    return lhs == rhs;
}

//或者使用特化形式，类似哈希函数设置一样
namespace std
{
template <>//模板的特化
struct equal_to<Point>
{
    bool operator()(const Point &lhs, const Point &rhs) const
    {
        return (lhs.getX() == rhs.getX()) && (lhs.getY() == rhs.
    }
};
}//end of namespace std
```

可以使用具体的例子Point进行测试：

```
class Point
{
public:
    Point(int ix = 0, int iy = 0)
    : _ix(ix)
    , _iy(iy)
    {
        //cout << "Point(int = 0, int = 0)" << endl;
    }

    ~Point()
    {
        /* cout << "~Point()" << endl; */
    }

    double getDistance() const
```

```
    {
        return hypot(_ix, _iy);
    }

    int getX() const
    {
        return _ix;
    }

    int getY() const
    {
        return _iy;
    }

    friend class PointComparator;
    friend bool operator==(const Point &lhs, const Point
&rhs);//std::equal_to
    friend std::ostream &operator<<(std::ostream &os, const Point &rhs);

private:
    int _ix;
    int _iy;
};

std::ostream &operator<<(std::ostream &os, const Point &rhs)
{
    os << "("  << rhs._ix
       << ", " << rhs._iy
       << ")";

    return os;
}

bool operator==(const Point &lhs, const Point &rhs)
{
    /* return (lhs.getX() == rhs.getX()) && (lhs.getY() == rhs.getY()); */
    return (lhs._ix == rhs._ix) && (lhs._iy == rhs._iy);
}

namespace std
{
template <>//模板的特化
struct hash<Point>
{
    size_t operator()(const Point &pt) const
    {
        return (pt.getX() << 1) ^ (pt.getY() << 1);//哈希函数的设置
    }
};
}//end of namespace std

//针对std::equal_to的特化
namespace std
{
template <>//模板的特化
struct equal_to<Point>
{
    bool operator()(const Point &lhs, const Point &rhs) const
```

```
    {
        return (lhs.getX() == rhs.getX()) && (lhs.getY() == rhs.getY());
    }

};
}//end of namespace std

struct PointHasher
{
    size_t operator()(const Point & pt) const
    {
        cout << "PointHasher::opeator()(const Point &)" << endl;
        return (pt.getX() << 1) ^ (pt.getY() << 1);
    }
};
```

（四）容器适配器

priority_queue的使用，这里讲堆排序，大小堆的建立(八大排序算法，这个大家要熟练，这是基本功)

优先级队列默认使用std::less,但是体现出来是一个大根堆。(可以直接看源码，也就是堆排序的建堆、堆调整)

首先看看其类模板形式

```
template < class T,
           class Container = std::vector<T>,
           class Compare = std::less<typename Container::value_type>
         > class priority_queue;
```

具体的使用例子如下：

```
void test()
{
    vector<int> numbers{2, 3, 6, 8, 9, 1, 4, 7, 5};
    priority_queue<int, vector<int>, std::greater<int>> pque;
    for(size_t idx = 0; idx != numbers.size(); ++idx)
    {
        pque.push(numbers[idx]);
        cout << "当前优先级队列中，优先级最高的元素是: " << pque.top() << endl;
    }

    while(!pque.empty())
    {
        cout << pque.top() << " ";
        pque.pop();
    }
    cout << endl;
}

class Point
{
    //......
};

bool operator<(const Point &lhs, const Point &rhs)
{
```

```
    //......
}

bool operator>(const Point &lhs, const Point &rhs)
{
    //......
}

//函数对象
struct PointComparator
{
    bool operator()(const Point &lhs, const Point &rhs)
    {
        //......
    }
};

void test2()
{
    vector<Point> numbers{
        Point(1, 2),
        Point(3, 4),
        Point(-1, 2),
        Point(5, 6),
        Point(0, 2),
        Point(-3, 2),
        Point(7, 8),
    };

    priority_queue<Point, vector<Point>, std::less<Point>> pque;

    for(size_t idx = 0; idx != numbers.size(); ++idx)
    {
        pque.push(numbers[idx]);
        cout << "当前优先级队列中，优先级最高的元素 : " << pque.top() << endl;
    }

    while(!pque.empty())
    {
        cout << pque.top() << " ";
        pque.pop();
    }
    cout << endl;
}
```

二、迭代器(iterator)

迭代器(iterator)模式又称为游标(Cursor)模式，用于提供一种方法顺序访问一个聚合对象中各个元素, 而又不需暴露该对象的内部表示。或者这样说可能更容易理解：Iterator模式是运用于聚合对象的一种模式，通过运用该模式，使得我们可以在不知道对象内部表示的情况下，按照一定顺序（由iterator提供的方法）访问聚合对象中的各个元素。

2.1、迭代器产生原因(或者本质)

Iterator类的访问方式就是把不同集合类的访问逻辑抽象出来，使得不用暴露集合内部的结构而达到循环遍历集合的效果。

2.2、迭代器的类型

随机访问迭代器(RandomAccessIterator)

双向迭代器(BidirectionalIterator)

前向迭代器(ForwardIterator)

输出迭代器(OutputIterator)

输入迭代器(InputIterator)

以及其头文件#include

2.3、为什么定义这么多迭代器?

物尽其用，使得具体的操作使用具体类型的迭代器，避免迭代器的功能太大或者太小，导致使用起来不方便。。

每个容器及其对应的迭代器的类型图表如下:

容器	类内迭代器类别
vector	随机访问迭代器
deque	随机访问迭代器
list	双向迭代器
set	双向迭代器
multiset	双向迭代器
map	双向迭代器
multimap	双向迭代器
unordered_set	前向迭代器
unordered_multiset	前向迭代器
unordered_map	前向迭代器
unordered_multimap	前向迭代器

每个迭代器的类型与其对应的操作。。。

类别 简写	输出 output	输入 input	前向 Forward	双向 Bidirection	随机 Random
读		=*p	=*p	=*p	=*p
访问		->	->	->	-> []
写	*p=		*p=	*p=	*p=
迭代	++	++	++	++/--	++/--/+/-/+=/-=
比较		==/!=	==/!=	==/!=	==/!=/</>/<=/>=

2.4、流迭代器

流迭代器是特殊的迭代器，可以将输入/输出流作为**容器**看待(因为输入输出都有**缓冲区**的概念)

关于流迭代器的模板形式：

```
template< class T,
          class CharT = char,
          class Traits = std::char_traits<CharT>,
          class Distance = std::ptrdiff_t
        > class istream_iterator
     : public std::iterator<std::input_iterator_tag, T, Distance, const T*,
const T&>
template< class T,
          class CharT = char,
          class Traits = std::char_traits<CharT>,
          class Distance = std::ptrdiff_t
        > class istream_iterator;

template< class T,
          class CharT = char,
          class Traits = std::char_traits<CharT>
        > class ostream_iterator
          : public std::iterator<std::output_iterator_tag,  void, void,
void, void>
template< class T,
          class CharT = char,
          class Traits = std::char_traits<CharT>
        > class ostream_iterator;
```

具体的代码示例：

```
void test()
{
    vector<int> numbers{1, 2, 3, 4, 5};

    ostream_iterator<int> osi(cout, "\n");
    copy(numbers.begin(), numbers.end(), osi);
}


void test()
{
    vector<int> numbers;
```

```
    cout << "111" << endl;
    istream_iterator<int> isi(cin);
    cout << "222" << endl;
    /* copy(isi, istream_iterator<int>(), numbers.begin());//对于vector插入元素应该调用push_back */
    copy(isi, istream_iterator<int>(), std::back_inserter(numbers));//插入迭代器
    cout << "333" << endl;

    copy(numbers.begin(), numbers.end(), ostream_iterator<int>(cout , "\n"));
}
```

2.5、迭代器适配器

back_inserter函数模板，返回类型是back_insert_iterator

back_insert_iterator是类模板，底层调用了push_back函数

front_inserter函数模板，返回类型是front_insert_iterator

front_insert_iterator是类模板，底层调用了push_front函数

inserter函数模板，返回类型是insert_iterator

insert_iterator是类模板，底层调用了insert函数

```
template< class Container >
std::back_insert_iterator<Container> back_inserter( Container& c );

template< class Container >
std::front_insert_iterator<Container> front_inserter( Container& c );

template< class Container >
std::insert_iterator<Container> inserter( Container& c, typename Container::iterator i );
```

三者对应的可能实现如下：

```
template< class Container >
std::back_insert_iterator<Container> back_inserter( Container& c )
{
    return std::back_insert_iterator<Container>(c);
}

template< class Container >
std::front_insert_iterator<Container> front_inserter( Container& c )
{
    return std::front_insert_iterator<Container>(c);
}

template< class Container >
std::insert_iterator<Container> inserter( Container& c, typename Container::iterator i )
{
    return std::insert_iterator<Container>(c, i);
```

```
}
```

具体的使用示例：

```
//insert_iterator.cc
void test()
{
    vector<int> numbers1{1, 3, 5};
    list<int> numbers2{20, 30, 40};

    //back_insert_iterator底层是调用了push_back
    copy(numbers2.begin(), numbers2.end(), back_insert_iterator<vector<int>>
(numbers1));
    copy(numbers1.begin(), numbers1.end(), ostream_iterator<int>(cout, "
"));
    cout  << endl;

    //front_insert_iterator底层是调用了push_front
    copy(numbers1.begin(), numbers1.end(), front_insert_iterator<list<int>>
(numbers2));
    copy(numbers2.begin(), numbers2.end(), ostream_iterator<int>(cout, "
"));
    cout << endl;

    set<int> numbers3{12, 13, 16, 13};
    auto sit = numbers3.begin();
    ++sit;

    //insert_iterator底层是调用了insert
    copy(numbers1.begin(), numbers1.end(), insert_iterator<set<int>>
(numbers3, sit));
    copy(numbers3.begin(), numbers3.end(), ostream_iterator<int>(cout, "
"));
    cout << endl;
}
```

2.6、逆向迭代器(反向迭代器)

```
void test()
{
    vector<int> numbers{1, 4, 6, 90, 34};
    vector<int>::reverse_iterator it = numbers.rbegin();
    /* auto it = numbers.rbegin(); */
    for(; it != numbers.rend(); ++it)
    {
        cout << *it << " ";
    }
    cout << endl;
}
```

三、适配器(adapter)

3.1、**定义**：适配器就是Interface(接口)，对容器、迭代器和算法进行包装，但其实质还是容器、迭代器和算法，只是不依赖于具体的标准容器、迭代器和算法类型，容器适配器可以理解为容器的模板，迭代器适配器可理解为迭代器的模板，算法适配器可理解为算法的模板。

3.2、**本质**：适配器是使一事物的行为类似于另一事物的行为的一种机制。

1、容器适配器

stack、queue、priority_queue，上面已经讲过，直接使用即可

2、迭代器适配器

back_insert_iterator、front_insert_iterator、insert_iterator，迭代器中已经讲过

3、函数适配器

bind1st、bind2nd、bind(后面算法阶段会讲解使用)

not1、not2 否定器

mem_fn 成员函数绑定器，后面会讲解

四、算法(algorithm)

算法中包含很多对容器进行处理的算法，使用迭代器来标识要处理的数据或数据段、以及结果的存放位置，有的函数还作为对象参数传递给另一个函数，实现数据的处理。

这些算法可以操作在多种容器类型上,所以称为"泛型"，泛型算法不是针对容器编写，而只是单独依赖迭代器和迭代器操作实现。

4.1、头文件

```
#include <algorithm>  //泛型算法
#include <numeric>    //泛化的算术算法
```

4.2、分类

1、非修改式序列操作：不改变容器的内容，如find()、for_each()等。
2、修改式序列操作：可以修改容器中的内容，如transform()、random_shuffle()、copy等。
3、排序和相关操作：包括各种排序函数等，如sort()等。
4、通用数字运算：计算两个容器的内部乘积等。

4.3、一元函数以及一元断言/一元谓词：

一元函数：函数的参数只有一个；一元断言/一元谓词：函数的参数只有一个，并且返回类型是bool类型。

二元函数：函数的参数有两个；二元断言/二元谓词：函数的参数两个，并且返回类型是bool类型。

```
//一元断言/一元谓词                                    //一元函数
bool func(int number)                                  void func(int number)
{                                                      {
            return number > 5;                                 cout << number
<< " ";
}                                                      }
```

4.4、非修改式算法(for_each)

模板形式

```
template< class InputIt, class UnaryFunction >
UnaryFunction for_each( InputIt first, InputIt last, UnaryFunction f );
//UnaryFunction一元函数
```

使用for_each的具体例子，进而衍生出lambda的概念。

```
void display(int &numbers)
{
    ++numbers;
    cout << numbers << " ";
}

void test()
{
    vector<int> numbers{2, 4, 19, 34, 1, 2, 1, 3, 6, 9, 6, 9, 4, 2};
    std::sort(numbers.begin(), numbers.end());//针对二分查找添加的排序函数
    /* for_each(numbers.begin(), numbers.end(), display); */

    //lambda表达式c++11
    for_each(numbers.begin(), numbers.end(), [] (int &number){
             ++number;
             cout << number << " ";

             });

    int ret = std::count(numbers.begin(), numbers.end(), 3);

    auto it2 = std::find(numbers.begin(), numbers.end(), 3);

    //时间复杂度O(log(N))，查找的时候，元素必须有序，所以需要配合sort算法使用。
    bool flag = std::binary_search(numbers.begin(), numbers.end(), 3);
}

int main(int argc, char **argv)
{
    test();
    return 0;
}
```

4.5、修改式算法(copy/remove_if)

介绍一下修改式算法中的算法copy，前面已经讲过，这个比较简单，源码上面可见：

```
vector<int>  numbers;
vector<int>  numbers2;
copy(numbers.begin(),  numbers.end(), back_inserter(number2));
copy(numbers.begin(),  numbers.end(), back_insert_iterator(number2));
```

看看具体的使用方法，从具体的方法中了解remove_if的原理：

```

bool func(int number)
{
    return number > 5;
}

void test()
{
    vector<int> numbers{1, 3, 5, 4, 9, 7, 6, 2, 10};
    copy(numbers.begin(), numbers.end(), ostream_iterator<int>(cout, " "));
    cout << endl;

    auto it = remove_if(numbers.begin(), numbers.end(), [] (int number){
                        return number > 5;
                        });

    numbers.erase(it, numbers.end());
    copy(numbers.begin(), numbers.end(), ostream_iterator<int>(cout, " "));
    cout << endl;
}

void test3()
{
    int func1(int a, int b = 5);
    int func2(int a);


    vector<int> numbers{1, 3, 5, 4, 9, 7, 6, 2, 10};
    copy(numbers.begin(), numbers.end(), ostream_iterator<int>(cout, " "));

    std::greater<int> gt;

    auto it = remove_if(numbers.begin(), numbers.end(), std::bind2nd(gt,
5));
    numbers.erase(it, numbers.end());
    copy(numbers.begin(), numbers.end(), ostream_iterator<int>(cout, " "));
}
```

4.6、函数绑定器的详解：

4.6.1、头文件

#include

4.6.2、模板形式

```
template< class F, class T >  std::binder1st<F> bind1st( const F& f, const T& x );
template< class F, class T >  std::binder2nd<F> bind2nd( const F& f, const T& x );
```

模板形式中，两个函数绑定器的第一个参数就是一个函数，第二个参数就是一个数字，如果F是一个二元函数(普通二元函数或者二元谓词)，我们可以绑定F的第一个参数(bind1st)或者第二个参数(bind2nd)，达到我们想要的效果(使用二元谓词的效果)

4.6.3、使用形式

上面remove_if.cc代码中有使用(bind1st与bind2nd)

4.6.4、C++11中bind函数的使用

```
template< class F, class... Args >
/*unspecified*/ bind( F&& f, Args&&... args );//返回类型没有确定

template< class R, class F, class... Args >
/*unspecified*/ bind( F&& f, Args&&... args );//返回类型没有确定
```

bind函数的使用相比于bind1st以及bind2nd更加的具有通用性，因为后者只能绑定一个参数，而bind可以绑定任意个参数。

bind可以绑定到普通函数、成员函数。具体例子如下：

```
//bind.cc
int add(int x, int y)   //普通函数形式
{
    cout << "int add(int, int)" << endl;
    return x + y;
}

class Example
{
public:
    int add(int x, int y)  //成员函数形式
    {
        cout << "int Example::add(int, int)" << endl;
        return x + y;
    }
};

void test()
{
    //int() 函数类型==>函数标签
    //bind可以改变函数的形态
    auto f = bind(add, 1, 2);
    cout << "f() = " << f() << endl;//使用形式与C语言中的函数指针类型
```

```
24
25      Example example;
26      //int()
27      auto f2 = bind(&Example::add, &example, 10, 20);//非静态的成员函数的第一个隐
    含参数是this指针
28      cout << "f() = " << f() << endl;//使用形式与C语言中的函数指针类型
29
30      //占位符
31      /* int(int) */
32      using namespace std::placeholders;
33      auto f3 = bind(add, 1, _1);//函数参数的绑定并不固定，并不一定要全部给出
34      cout << "f3(10) = " << f3(10) << endl;
35  }
```

函数指针的特点：

```
1   typedef int(*pFunc)();//函数指针
2
3   int func1()
4   {
5       return 5;
6   }
7
8   int func2()
9   {
10      return 10;
11  }
12
13  void test2()
14  {
15      //int a = 10;
16      pFunc f = func1;//对于f注册回调函数，C语言是可以实现多态的(通过函数指针实现多态)
17      cout << "f() = " << f() << endl;//执行回调函数
18
19      f = func2;
20      cout << "f() = " << f() << endl;
21  }
```

bind函数的另外一个问题，就是占位符的概念，占位符本身是什么，占位符的数字代表什么？？？

占位符本身所在的位置是形参的位置，占位符的数字代表实参传递时候的位置。。。具体例子可以看看：

```
1   void func3(int x1, int x2, int x3, const int &x4, int x5)
2   {
3       cout << "(" << x1
4            <<", "  << x2
5            << ", " << x3
6            << ", " << x4
7            << ", " << x5
8            << ")" << endl;
9   }
10
11  void test3()
12  {
13      using namespace std::placeholders;//占位符
```

```
    int number = 100;


    auto f = bind(func3, 2, _1, _2, std::cref(number), number);

    number = 300;
    f(20, 30, 400, 50000, 5678);//对于没有作用的实参是无效的参数
}
```

bind函数的返回类型到底是什么呢？其实就是一种函数类型，就是一种可以装函数类型的容器，即：function。有了function之后，bind函数还可以绑定到数据成员上面来，其实就是int()这中类型上来。。。。

```
template< class >   class function;
template< class R, class... Args >   class function<R(Args...)>//R就是返回类型，Args是参数
```

```
void test3()
{

    std::function<int()> f = bind(add, 1, 2);
    cout << "f() = " << f() << endl;

    Test test;
    f = bind(&Test::add, &test, 10, 20);
    cout << "f() = " << f() << endl;

    f = bind(&Test::data, &test);//可以绑定到数据成员上面来
    cout << "f() = " << f() << endl;

    using namespace std::placeholders;
    std::function<int(int)> f3 = bind(add, 1, _1);
    cout << "f3(10) = " << f3(10) << endl;
}
```

全部详细的代码就是这样，可以看看：

```
int add(int x, int y)
{
    cout << "int add(int, int)" << endl;
    return x + y;
}

class Example
{
public:
    int add(int x, int y)
    {
        cout << "int Example::add(int, int)" << endl;
        return x + y;
    }

    int data = 1234;//C++11新特性对于数据成员的初始化
};

```

```cpp
void test()
{
    std::function<int()> f = bind(add, 1, 2);
    cout << "f() = " << f() << endl;

    Example example;
    f = bind(&Example::add, &example, 10, 20);
    cout << "f() = " << f() << endl;

    f = bind(&Example::data, &example);//可以绑定到数据成员上面来
    cout << "f() = " << f() << endl;

    using namespace std::placeholders;
    std::function<int(int)> f3 = bind(add, 1, _1);
    cout << "f3(10) = " << f3(10) << endl;
}

typedef int(*pFunc)();//函数指针

int func1()
{
    return 5;
}

int func2()
{
    return 15;
}

void test2()
{
    pFunc f = func1;
    cout << "f() = " << f() << endl;

    f = func2;
    cout << "f() = " << f() << endl;
}

void func3(int x1, int x2, int x3, const int &x4, int x5)
{
    cout << "(" << x1
         <<", "  << x2
         << ", " << x3
         << ", " << x4
         << ", " << x5
         << ")" << endl;
}

void test3()
{
    using namespace std::placeholders;
    int number = 100;

    auto f = bind(func3, 2, _1, _2, std::cref(number), number);

    number = 300;
    f(20, 30, 400, 50000, 5678);//对于没有作用的实参是无效的参数
}
```

再来看看std::function与std::bind结合使用体现出多态性的例子，也就是注册回调函数与执行回调函数。

```
class Figure
{
public:
    using DisplayCallback = function<void()>;
    using AreaCallback = function<double()>;

    DisplayCallback  _displayCallback;
    AreaCallback  _areaCallback;

    //注册回调函数
    void setDisplayCallback(DisplayCallback &&cb)
    {
        _displayCallback = std::move(cb);
    }

    void setAreaCallback(AreaCallback &&cb)
    {
        _areaCallback = std::move(cb);
    }

    //执行回调函数
    void handleDispalyCallback() const
    {
        if(_displayCallback)
        {
            _displayCallback();
        }
    }

    double handleAreaCallback() const
    {
        if(_areaCallback)
        {
            return _areaCallback();
        }
        else
        {
            return 0;
        }
    }
};

//执行回调函数
void test(const Figure &fig)//这里使用const对象，所以处理函数必须定义为const版本
{
    fig.handleDispalyCallback();
    cout << "'s area is : " << fig.handleAreaCallback() << endl;
}

class Rectangle
{
```

```
public:
    Rectangle(double length, double width)
    : _length(length)
    , _width(width)
    {
    }

    void display(int ix) const //相比之前，这里display函数可以传参
    {
        cout << "Rectangle ";
    }

    double area() const
    {
        return _length * _width;
    }
private:
    double _length;
    double _width;
};

class Circle
{
public:
    Circle(double radis)
    : _radis(radis)
    {
    }

    void show() const
    {
        cout << "Circle ";
    }

    double getArea() const
    {
        return _radis * _radis * 3.1415;
    }
private:
    double _radis;
};

class Traingle
{
public:
    Traingle(double a, double b, double c)
    : _a(a)
    , _b(b)
    , _c(c)
    {
    }

    void print() const
    {
        cout << "Traingle ";
    }

    //海伦公式
```

```
    double calcArea() const
    {
        double tmp = (_a + _b + _c)/2;

        return sqrt(tmp * (tmp - _a) * (tmp - _b) * (tmp - _c));
    }
private:
    double _a;
    double _b;
    double _c;
};

void test()
{
    Rectangle rectangle(10, 20);
    Circle circle(10);
    Traingle traingle(3, 4, 5);

    Figure figure;
    figure.setDisplayCallback(bind(&Rectangle::display, &rectangle, 10));
    figure.setAreaCallback(bind(&Rectangle::area, &rectangle));
    test(figure);

    figure.setDisplayCallback(bind(&Circle::show, &circle));
    figure.setAreaCallback(bind(&Circle::getArea, &circle));
    test(figure);

    figure.setDisplayCallback(bind(&Traingle::print, &traingle));
    figure.setAreaCallback(bind(&Traingle::calcArea, &traingle));
    test(figure);
}
```

4.6.5、成员函数绑定器

mem_fun()　　//容器参数为类指针

mem_fun_ref()　//容器参数为类对象

mem_fn() 两者都能用(C++11)，其使用更具有广泛性，直接使用代码就ok

```
class Number
{
public:
    Number(size_t data = 0)
    : _data(data)
    {
    }

    void print() const
    {
        cout << _data << " ";
    }

    bool isEven() const
    {
        return (0 == _data % 2);
    }

```

```
    bool isPrime() const
    {
        if(1 == _data)
        {
            return false;
        }
        //质数/素数
        for(size_t idx = 2; idx <= _data/2; ++idx)
        {
            if(0 == _data % idx)
            {
                return false;
            }
        }
        return true;
    }
private:
    size_t _data;
};

void test()
{
    vector<Number> numbers;
    for(size_t idx = 1; idx != 10; ++idx)
    {
        numbers.push_back(Number(idx));
    }

    std::for_each(numbers.begin(), numbers.end(), mem_fn(&Number::print));

    //erase函数参数之前说过，两个参数，
    numbers.erase(remove_if(numbers.begin(), numbers.end(),
mem_fn(&Number::isEven)), numbers.end());
    std::for_each(numbers.begin(), numbers.end(), mem_fn(&Number::print));

    numbers.erase(remove_if(numbers.begin(), numbers.end(),
mem_fn(&Number::isPrime)), numbers.end());
    std::for_each(numbers.begin(), numbers.end(), mem_fn(&Number::print));
}
```

五、函数对象(functor)

函数对象是可以以函数方式与()结合使用的任意对象，包括：(functor-仿函数)
1、函数名；
2、指向函数的指针；
3、重载了()操作符的类对象(即定义了函数operator()()的类)。

以上就是对函数对象做了一个扩充，具体的使用形式已经用过，此处不再赘述。

六、空间配置器(allocator)

在C++中所有STL容器的空间分配其实都是使用的std::allocator,它是可以感知类型的空间分配器，并将空间的申请与对象的构建分离开来。

单个对象使用new的时候，空间的申请与对象的构造好像是没有分开的，连在一起了，但是对于容器vector而言，其有个函数叫做reserve函数，先申请空间，然后在在该空间上构建对象。

为什么要将空间的申请与对象的构建分开呢，因为对于批量元素而言，肯定不会是将申请的空间全部的占满，肯定不是申请一个空间，然后在该空间构建对象，然后再申请一个空间，接着再构建对象。

6.1、头文件

#include

```
template< class T >  struct allocator;
template<>  struct allocator<void>;
```

6.2、空间申请与释放以及对象的构建与销毁的四个函数如下

```
//空间的申请,申请的是原始的，未初始化的空间
pointer allocate( size_type n, const void * hint = 0 );
T* allocate( std::size_t n, const void * hint);
T* allocate( std::size_t n );

//空间的释放
void deallocate( T* p, std::size_t n );

//对象的构建，在指定的未初始化的空间上构建对象，使用的是定位new表达式
void construct( pointer p, const_reference val );

//对象的销毁
void destroy( pointer p );
```

将内存的申请与对象的构建分开的例子，比如：自己实现一个自定义Vector

6.3、定位new表达式

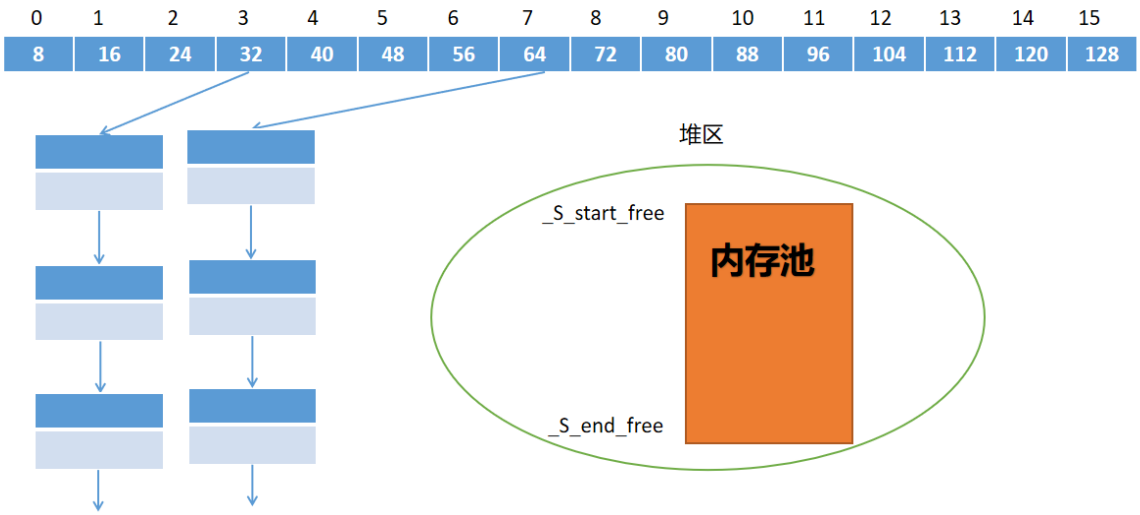
定位new表达式接受指向未构造内存的指针，并在该空间中初始化一个对象或者数组。不分配内存，它只在特定的，预分配的内存地址构造一个对象。

```
template <class _T1, class _T2>
inline void construct(_T1* __p, const _T2& __value) {
  _Construct(__p, __value);
}

template <class _T1, class _T2>
inline void _Construct(_T1* __p, const _T2& __value) {
  new ((void*) __p) _T1(__value);//定位new表达式，在执行空间__p上构建对象
}

template <class _Tp>
inline void destroy(_Tp* __pointer) {
  _Destroy(__pointer);
}

```

```cpp
template <class _Tp>
inline void _Destroy(_Tp* __pointer) {
  __pointer->~_Tp();
}
```

6.4、真正具备空间分配功能的std::alloc

两级空间配置器，第一级空间配置器使用类模板**malloc_alloc_template，其底层使用的是malloc/free进行空间的申请与释放，二级空间配置器使用类模板，**default_alloc_template，其底层根据申请空间大小有分为两个分支进行，第一分支是当申请的空间大于128字节的时候，还是走__malloc_alloc_template，当申请的空间小于128字节的使用，使用**内存池+16个自由链表**的结构进行。



6.4.1、为什么要分成两部分呢?

1、向系统堆要求空间

2、考虑多续状态(multi-threads)

3、考虑内存不足时的应变措施

4、考虑过多小块内存造成的内存碎片

6.4.1.0、内存碎片

内部碎片：页式管理、段式管理、段页式管理(局部性原理)，无法避免，但是通过算法可以优化。

外部碎片：申请堆内存之间的片段空隙，这个是可以合理使用的。

6.4.2、分成两部分的解决办法?

一级空间配置器：使用malloc/free系统调用，缺点：频繁的用户态到内核态的切换，开销大(brk,mmap)

二级空间配置器：内存池+16个自由链表，优点：以空间换时间，缺点：内存占用比较大，如果内存有限，内存不可控，这也是早期STL提出时候不被重用的原因，那是内存较小。。

6.4.3、源码解析

以《STL源码剖析》这本书的例子进行研究，先申请32字节空间，然后申请64字节空间，接着申请96字节空间，最后申请72字节(**假设此时内存池耗尽、堆空间没有大于72字节的连续空间**)

6.4.3.1、释放内存的deallocate

对应一级空间配置器，直接使用free将内存回收到堆空间。

对应二级空间配置器，直接将用完后的空间链回到相应的链表下面，使用头插法进行连接。。

(The end)