[STL]

[C/C++学院]

1.基本概念

STL：Standard Template Library，标准模板库

定义：c++引入的一个标准类库

特点：

- 数据结构和算法的 c++实现（采用模板类和模板函数）
- 数据的存储和算法的分离
- 高复用性，高移植性

组件：

- 容器（Container）
- 算法（Algorithm）
- 迭代器（Iterator）
- 仿函数（Function object）
- 适配器（Adaptor）
- 空间配制器（allocator）

1.1.组件介绍

1.1.1.容器

定义：说白了就是数据的存储

1.1.1.1.序列型容器

定义：每个元素都有固定位置（取决于插入时机和地点，和元素值无关）

分类：

- 向量(vector) 将元素置于一个动态数组中加以管理，可以随机存取元素（用索引直接存取），数组尾部添加或移除元素非常快速。但是在中部或头部安插元素比较费时
- 列表(list) 双向链表，不提供随机存取（按顺序走到需存取的元素，O(n)），在任何位置上执行插入或删除动作都非常迅速，内部只需调整一下指针
- 双端队列(deque) 是“double-ended queue”的缩写，可以随机存取元素（用索引直接存取），数组头部和尾部添加或移除元素都非常快速。但是在中部或头部安插元素比较费时

1.1.1.2.容器适配器

定义:序列式容器的应用

分类：

- 栈(stack) 后进先出的值的排列
- 队列(queue) 先进先出的值的排列
- 优先队列(priority_queue) 元素的次序是由作用于所存储的值对上的某种谓词决定的的一种队列

1.1.1.3.关联式容器

定义：元素位置取决于特定的排序准则，和插入顺序无关。

分类：

- 集合(set/multiset)
  - 内部的元素依据其值自动排序，Set 内的相同数值的元素只能出现一次，Multisets 内可包含多个数值相同的元素，内部由二叉树实现，便于查找
- 映射(map/multimap)
  - 元素是成对的键值/实值，内部的元素依据其值自动排序，Map 内的相同数值的元素只能出现一次， Multimaps 内可包含多个数值相同的元素，内部由二叉树实现，便于查找（实际上是红黑树的二叉树的变种）

1.1.2.算法

定义:如果说容器是数据的存储，那么算法就是操作，只不过 stl 里面的算法都是模板函数，总共有 100 多个。比如算法 for_each 将为指定序列中的每一个元素调用指定的函数，stable_sort 以 你所指定的规则对序列进行稳定性排序等等。这样一来，只要熟悉了 STL 之后，许多代码可以被大大的化简，只需要通过调用一两个算法模板，就可以完成所需要 的功能并大大地提升效率。

算法部分主要由头文件<algorithm>，<numeric>和<functional>组 成。<algorithm>是所有 STL 头文件中最大的一个（尽管它很好理解），它是由一大堆模版函数组成的，可以认为每个函数在很大程度上 都是独立的，其中常用到的功能范围涉及到比较、交换、查找、遍历操作、复制、修改、移除、反转、排序、合并等等。<numeric>体积很 小，只包括几个在序列上面进行简单数学运算的模板函数，包括加法和乘法在序列上的一些操作。<functional>中则定义了一些模板类， 用以声明函数对象。

1.1.3.迭代器

定义：迭代器在 STL 中用来将算法和容器联系起来，起着一种黏和剂的作用。几乎 STL 提供的所有算法都是通 过迭代器存取元素序列进行工作的，每一个容器都定义了其本身所专有的迭代器，用以存取容器中的元素。

迭代器部分主要由头文件<utility>,<iterator>和<memory>组 成。<utility>是一个很小的头文件，它包括了贯穿使用在 STL 中的几个模板的声明，<iterator>中提供了迭代器 使用的许多方法，而对于<memory>的描述则十分的困难，它以不同寻常的方式为容器中的元素分配存储空间，同时也为某些算法执行期间产生 的临时对象提供机制,<memory>中的主要部分是模板类 allocator，它负责产生所有容器中的默认分配器。

1.1.4.仿函数

定义：仿函数本身不是函数， 而是一个类对象， 因为类中重载了函数运算符()， 即 operator().所以类对象具有了类似函数的功能。 可以使模板重加的灵活。 后来的 lambada 的它升级版本。

1.1.5.适配器

略

1.1.6.空间配制器

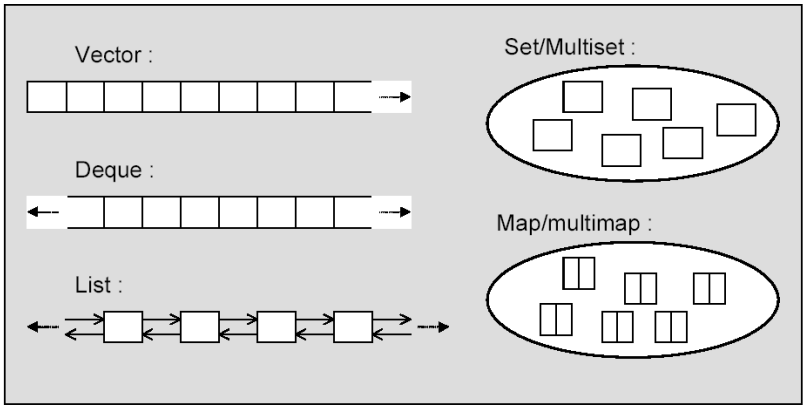
略

2.容器（Container）

总结式表格：

数据结构	描述	实现头文件
向量(vector)	连续存储的元素	<vector>
列表(list)	由节点组成的双向链表，每个结点包含着一个元素	<list>
双队列(deque)	连续存储的指向不同元素的指针所组成的数组	<deque>
集合(set)	由节点组成的红黑树，每个节点都包含着一个元素，节点之间以某种作用于元素对的谓词排列，没有两个不同的元素能够拥有相同的次序	<set>
多重集合(multiset)	允许存在两个次序相等的元素的集合	<set>
栈(stack)	后进先出的值的排列	<stack>
队列(queue)	先进先出的执的排列	<queue>
优先队列(priority_queue)	元素的次序是由作用于所存储的值对上的某种谓词决定的的一种队列	<queue>
映射(map)	由{键，值}对组成的集合，以某种作用于键对上的谓词排列	<map>
多重映射(multimap)	允许键对有相等的次序的映射	<map>

总结式图示：



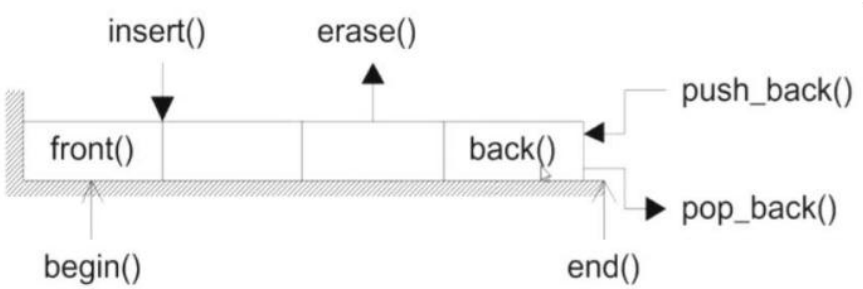
共同特点：

- 支持泛型
- 保存副本
- 内存管理

3.序列式容器

3.1.Vector 基本使用

3.1.1.结构图示



向量（vector）

3.1.2.构造初始化

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<int> vi;
    cout<<vi.size()<<endl;
    vector<int> vi2(3,4);
    cout<<vi2.size()<<endl;
    int arr[5] = {1,2,3,4,5};
    vector<int> vi3(arr,arr+5);
    for(int i=0; i<5; i++)
        cout<<vi3[i]<<endl;
    vector<int> vi4(vi3);
    for(int i=0; i<5; i++)
        cout<<vi4[i]<<endl;
    return 0;
}
```

3.1.3.使用迭代器

```
int main()
{
```

```
    int a[10] = {1,3,5,7,9,2,4,6,8,0};
    vector<int> vi(a,a+10);
    vector<int>::iterator itr = vi.begin();
    for(; itr != vi.end(); ++itr)
    {
        cout<<*itr<<endl;
    }
    vector<int>::const_iterator citr = vi.begin();
    for(; citr != vi.end(); ++citr)
    {
        cout<<*citr<<endl;
    }
    vector<int>::reverse_iterator ritr = vi.rbegin();
    for(; ritr != vi.rend(); ++ritr)
    {
        cout<<*ritr<<endl;
    }
    return 0;
}
```

3.1.4.元素的存取

```
int main()
{
    int a[10] = {1,3,5,7,9,2,4,6,8,0};
    vector<int> vi(a,a+10);
    cout<<vi[3]<<endl;
    cout<<vi.at(3)<<endl;
    cout<<vi.front()<<endl;
    cout<<vi.back()<<endl;
    return 0;
}
```

3.1.5.元素的修改

```
int main()
{
    vector<int> vi;
    vi.assign(10,5);
    for(int i=0; i<10; i++)
    {
        cout<<vi[i]<<endl;
    }
    vector<int> vi2;
```

```
    vi2.assign(vi.begin(),vi.end());
    for(int i=0; i<10; i++)
    {
        cout<<vi2[i]<<endl;
    }
    vi2.push_back(10);
    for(int i=0; i<vi2.size(); i++)
    {
        cout<<vi2[i]<<endl;
    }
    vi2.pop_back(); //无返回值， 可以先得到 back()的值， 然后 pop_back()
    for(int i=0; i<vi2.size(); i++)
    {
        cout<<vi2[i]<<endl;
    }
    return 0;
}
```

```
int main()
{
    vector<int> vi;
    vi.assign(3,4);
    // vi.insert(vi.end(),5);
    // for(int i=0; i<vi.size(); i++)
    // cout<<vi[i]<<endl;
    // vi.erase(vi.end()-1);
    // for(int i=0; i<vi.size(); i++)
    // cout<<vi[i]<<endl;
    vi.insert(vi.begin(),10);
    for(int i=0; i<vi.size(); i++)
        cout<<vi[i]<<endl;
    vi.insert(vi.end(),10);
    for(int i=0; i<vi.size(); i++)
        cout<<vi[i]<<endl;
    vi.erase(vi.end()-2);
    for(int i=0; i<vi.size(); i++)
        cout<<vi[i]<<endl;
    vi.clear();
    cout<<vi.size()<<endl;
    return 0;
}
```

```
int main()
```

```
{
    vector<int> a(3,4);
    vector<int> b(4,3);
    a.swap(b);
    for(int i=0; i<a.size(); i++)
        cout<<a[i]<<endl;
    for(int j=0; j<b.size(); j++)
        cout<<b[j]<<endl;
    return 0;
}
```

3.1.6.存储的空间

```
int main()
{
    vector<int> vi(3);
    cout<<vi.size()<<endl;
    cout<<vi.capacity()<<endl;
    vi.pop_back();
    cout<<vi.size()<<endl;
    cout<<vi.capacity()<<endl;
    vi.clear();
    cout<<boolalpha<<vi.empty()<<endl;
    cout<<vi.size()<<endl;
    cout<<vi.capacity()<<endl;
    vi.resize(5);
    cout<<vi.size()<<endl;
    cout<<vi.capacity()<<endl;
    vi.reserve(10);
    cout<<vi.size()<<endl;
    cout<<vi.capacity()<<endl;
    cout<<vi.max_size()<<endl;

    return 0;
}
```

3.1.7.提高篇

3.1.7.1.内存管理

```
int main()
{
    vector<int> vi;
    vi.reserve(16);
```

```
    for(int i=0; i<10; i++)
    {
        vi.push_back(i);
        cout<<"size:"<<vi.size()<<endl;
        cout<<"capacity:"<<vi.capacity()<<endl;
    }
    vi.resize(1);
    cout<<"size:"<<vi.size()<<endl;
    cout<<"capacity:"<<vi.capacity()<<endl;
    vi.reserve(10);
    cout<<"size:"<<vi.size()<<endl;
    cout<<"capacity:"<<vi.capacity()<<endl;

    return 0;
}
```

注意：

- 事先预估内存容量，避免内存的重复申请
- Resize 可以改变 size 的大小，变大等价于调用 push_back（），变小等价于 pop_back（），resize 不可以改变 capacity 的大小
- Reserve 只可变大 vector 的 capacity 而不可以使其减小

3.1.7.2.压入类对象

```
class
{
public:
    A()
    {
        cout<<"无参构造函数"<<this<<endl;
    }
    A(int i):_data(i){
        cout<<"有参构造函数"<<this<<endl;
    }
    A(const A & other)
    {
        cout<<"拷贝构造"<<&other<<"->to->"<<this<<endl;
    }
    A& operator=(const A & other)
    {
        cout<<"拷贝赋值"<<&other<<"->to->"<<this<<endl;
    }
    ~A()
    {
        cout<<"析构函数"<<this<<endl;
```

```
    }
private:
    int _data;
};

int main()
{
    // vector<A> v(3);
    vector<A> v(3,A(2));
    vector<A> v2(10);
    v2 = v;

    return 0;
}
```

注意：

- 当向容器中压的是类对象成员时，最好要保证无参构造器的存在，类似于数组中的类对象一样。
- 在必要的情况下，要自实现拷贝构造和拷贝赋值。
- 压入栈中的对象元素在堆中，所申请的内存已托管，无需再次打理。

3.1.7.3.压入类对象的指针

```
class A
{
public:
    A()
    {
        cout<<"无参构造函数"<<this<<endl;
    }
    A(int i):_data(i){
        cout<<"有参构造函数"<<this<<endl;
    }
    A(const A & other)
    {
        cout<<"拷贝构造"<<&other<<"->to->"<<this<<endl;
    }
    A& operator=(const A & other)
    {
        cout<<"拷贝赋值"<<&other<<"->to->"<<this<<endl;
    }
    ~A()
    {
```

```
        cout<<"析构函数"<<this<<endl;
    }
private:
    int _data;
};
int main()
{
    A *pa1 = new A;
    A *pa2 = new A;
    A *pa3 = new A;

    vector<A*> vap;
    vap.push_back(pa1);
    vap.push_back(pa2);
    vap.push_back(pa3);
    // delete pa1; delete pa2; delete pa3;
    vector<A*>::iterator itr = vap.begin();
    for(;itr != vap.end(); ++itr)
    {
        delete *itr;
    }

    return 0;
}
```

注意：

- 容器有内存托管的功能，但仅限于被压入的实体元素
- 若元素为指针类型，则指针被托管，而不是指针所指的空间被托管

3.1.7.4.讨论 reserve 和 resize

```
class A
{
public:
    A()
    {

        _data = 0;
        cout<<"无参构造函数"<<this<<endl;
    }
    A(int i):_data(i){
        cout<<"有参构造函数"<<this<<endl;
    }
    A(const A & other)
    {
```

```
        cout<<"拷贝构造"<<&other<<"->to->"<<this<<endl;
    }
    A& operator=(const A & other)
    {
        cout<<"拷贝赋值"<<&other<<"->to->"<<this<<endl;
    }
    ~A()
    {
        cout<<"析构函数"<<this<<endl;
    }
private:
    int _data;
};

int main()
{
    vector<A> va;
    // for(int i=0; i<5; i++)
    // {
    // va.push_back(A(i));
    // }
    va.reserve(100);
    // va.resize(5);
    for(int i=0; i<5; i++)
    {
        va.push_back(A(i));
    }

    return 0;
}
```

注意：

- Resize 会调用构造器，而 reserve 则不会调用构造器
- 事先 reserve 出空间来，可以大量的减少不必要的内存拷贝

3.1.7.5.讨论 insert 和 erase

```
class A
{
public:
    A()
    {

        _data = 0;
        cout<<"无参构造函数"<<this<<endl;
```

```
    }
    A(int i):_data(i){
        cout<<"有参构造函数"<<this<<endl;
    }
    A(const A & other)
    {
        this->_data = other._data;
        cout<<"拷贝构造"<<&other<<"->to->"<<this<<endl;
    }
    A& operator=(const A & other)
    {
        cout<<"拷贝赋值"<<&other<<"->to->"<<this<<endl;
    }
    void dis()
    {
        cout<<_data<<endl;
    }
    ~A()
    {
        cout<<"析构函数"<<this<<endl;
    }
private:
    int _data;
};
int main()
{
    vector<A> va;
    va.reserve(100);
    for(int i=0; i<5; i++)
    {
        va.push_back(A(i));
    }
    // va.back().dis();
    // va.pop_back();
    // va.push_back(100);
    // va.insert(va.end(),A(100));
    // va.erase(va.end()-1);
    // va.insert(va.begin(),A(10));
    va.erase(va.begin());

    return 0;
}
```

注意：

- 尾部的插入和删除是高效的 O(1),其他位置则是低效的 O(n)
- 故对于 vector 而言，提倡使用 pop_back(),尽量少用 insert 和 erase

3.1.7.6.迭代器的失效

原因：vector 是一个顺序容器，在内存中是一块连续的内存，当删除一个元素后，内存中的数据会发生移动，以保证数据的紧凑。所以删除一个数据后，其他数据的地址发生了变化，之前获取的迭代器根据原有的信息就访问不到正确的数据。

```
int main()
{
    vector<int> vi;
    int data[10] = {1,3,5,7,9,2,4,6,8,10};
    vi.assign(data,data+10);
    vector<int>::iterator itr = vi.begin();
    for(;itr != vi.end(); ++itr)
    {
        if(*itr%2==0)
            vi.erase(itr);
        else
        {
            cout<<*itr<<endl;
        }
    }
    return 0;
}
```

```
int main()
{
    vector<int> vi;
    int data[10] = {1,3,5,7,9,2,4,6,8,10};
    vi.assign(data,data+10);
    vector<int>::iterator itr = vi.begin();
    for(;itr != vi.end(); )
    {
        if(*itr%2==0)
            itr = vi.erase(itr);
        else
        {
            cout<<*itr<<endl;
            itr++;
        }
    }
    return 0;
```

```
}
```

注意：

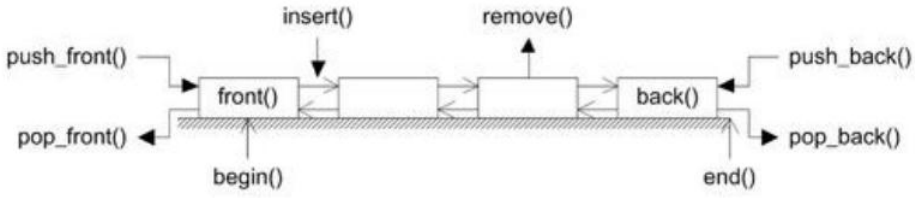
- 失效的迭代器，需要特殊处理

3.1.7.7.二维空间的生成

```
int main()
{
    vector<vector<int> > arr;
    // arr.resize(5);
    // for(int i=0; i<5; i++)
    // {
    // arr[i].resize(5);
    // }
    vector<int> vi(5);
    for(int i=0; i<5; i++)
    {
        arr.push_back(vi);
    }
    for(int i=0; i<5; i++)
    {
        for(int j=0; j<5; j++)
        {
            cout<<arr[i][j]<<" ";
        }
        cout<<endl;
    }

    return 0;
}
```

3.2.list 基本使用

3.2.1.结构图示



列表（list）

3.2.2.构造初始化

```
int main()
{
    list<int> one; cout<<"one size:"<<one.size()<<endl;
    list<int> two(5,10);cout<<"two size:"<<two.size()<<endl;

    list<int> three(two.begin(),two.end());
    cout<<"three size:"<<three.size()<<endl;

    list<int> four(three);
    cout<<"four size:"<<four.size()<<endl;

    return 0;
}
```

3.2.3.迭代器的使用

```
int main()
{
    list <int> li(5,10);
    list<int>::iterator itr = li.begin();
    for(; itr != li.end(); ++itr)
    {
        cout<<*itr<<endl; //list 的迭代器， 仅支持++
    }
    list<int>::const_reverse_iterator itr2 = li.crbegin();
    for(; itr2 != li.crend(); ++itr2)
    {
        cout<<*itr2<<endl;
    }

    return 0;
}
```

3.2.4.元素的存取

```
int main()
{
    list<int> li;
    for(int i=0; i<5; i++)
    {
        li.push_back(i);
    }
```

```
    cout<<li.front()<<endl;
    cout<<li.back()<<endl;

    return 0;
}
```

3.2.5.元素的空间

```
cout<<li.size()<<endl;
    cout<<boolalpha<<li.empty()<<endl;
    cout<<li.max_size()<<endl;
```

3.2.6.元素的修改

```
void print(list<int> &li)
{
    list<int>::iterator itr = li.begin();
    for(; itr != li.end(); ++itr)
    {
        cout<<*itr<<endl;
    }
    cout<<"++++++++++++++++++"<<endl;
}
int main()
{
    list<int> li;
    li.insert(li.begin(),2,3);
    list<int> li2;
    li2.assign(li.begin(),li.end());
    print(li);
    li.resize(0);
    print(li);
    li.resize(5);
    print(li);
    li.erase(li.begin());
    print(li);
    li.clear();
    print(li);
    li.swap(li2);

    return 0;
}

int main1()
```

```
{
    list<int> li;
    li.insert(li.begin(),2);
    print(li);
    li.push_back(3);
    print(li);
    li.pop_back(); //容器将弹和取数据分开
    print(li);
    li.push_front(10);
    print(li);
    li.pop_front();
    print(li);

    return 0;
}
```

3.2.7.list 提高

基本操作:

```
void print(list<int> &li)
{
    list<int>::iterator itr = li.begin();
    for(;itr !=li.end(); ++itr)
    {
        cout<<*itr<<endl;
    }
    cout<<"++++++++++"<<endl;
}

bool isSingleDigit(const int &value)
{
    return value<10;
}
class isOdd
{
public:
    bool operator()(const int &value)
    {
        return (value%2) == 1;
    }
};
int main()
{
    list<int> li;
```

```
}
int main3()
{
    list<int> li;
    int data[10] = {1,1,3,3,4,4,2,2,5,5};
    li.insert(li.begin(),data,data+10);
    li.unique();
    print(li);
    li.sort();
    print(li);
    li.reverse();
    print(li);
    return 0;
}
int main2()
{
    list<int> li;
    int data[5] = {1,20,3,40,5};
    li.insert(li.begin(),data,data+5);
    print(li);
    li.remove(5);
    print(li);
    // li.remove_if(isSingleDigit);
    // print(li);
    li.remove_if(isOdd());
    print(li);

    return 0;
}
int main1()
{
    list<int> la;
    la.assign(2,3);
    list<int> lb;
    lb.assign(2,4);
    list<int>::iterator itr = la.begin();
    ++itr;
    la.splice(itr,lb,lb.begin(),lb.end());
    print(la);

    return 0;
}
void print(list<int> &li)
```

```
{
    list<int>::iterator itr = li.begin();
    for(;itr !=li.end(); ++itr)
    {
        cout<<*itr<<endl;
    }
    cout<<"++++++++++++"<<endl;
}
int main()
{
    list<int> first;
    for(int i=5; i>0; i--)
        first.push_back(i);
    list<int> second;
    for(int i=10; i<50; i+=10)
        second.push_back(i);
    first.sort();
    second.sort();
    first.merge(second);
    print(first);
    cout<<"second size :"<<second.size()<<endl;

    return 0;
}
```

3.2.7.1.高效的插入和删除

```
class A
{
public:
    A()
    {
        cout<<"无参构造函数"<<this<<endl;
    }
    A(int i):_data(i){
        cout<<"有参构造函数"<<this<<endl;
    }
    A(const A & other)
    {
        cout<<"拷贝构造"<<&other<<"->to->"<<this<<endl;
    }
    A& operator=(const A & other)
    {
        cout<<"拷贝赋值"<<&other<<"->to->"<<this<<endl;
```

```
    }
    ~A()
    {
        cout<<"析构函数"<<this<<endl;
    }
private:
    int _data;
};
int main()
{
    list<A> la(5,A());
    la.insert(la.begin(),A());
    la.erase(la.begin());
    la.push_back(A());
    la.push_front(A());
    la.pop_back();
    la.pop_front();

    return 0;
}
```

注意：

- List 优于 vector 的地方，就在高效的删除和插入
- 在可以用 list 取代 vector 的地方，尽量使用 list
- List 不能取代 vector 的地方在于下标的访问

3.2.7.2.压入对象和对象指针

```
int main()
{
    list<A> lap;
    lap.insert(lap.begin(),3, A());
    cout<<lap.size()<<endl;
    list<A>::iterator itr = lap.begin();
    for(; itr != lap.end(); ++itr)
    {

    }

    return 0;
}
```

```
int main()
{
    list<A*> lap;
```

```
    lap.insert(lap.begin(),3, new A());
    cout<<lap.size()<<endl;
    list<A*>::iterator itr = lap.begin();
    for(; itr != lap.end(); ++itr)
    {
        delete (*itr);
    }

    return 0;
}
```

注意：

- 压入对象和指针，本质是一样的，拷贝副本
- 但在压入指针时，内存处理上是不同的

3.2.7.3.迭代器失效

- 删除失效

```
int main()
{
    int data[] = {1,2,3,4,5,6,7,8,9,0};
    list<int> li;
    li.assign(data,data+sizeof(data)/sizeof(int));
    list<int>::iterator itr = li.begin();
    for(; itr !=li.end(); )
    {
        if(*itr%2==0)
        {
            itr = li.erase(itr);
        }
        else
        {
            cout<<*itr<<endl;
            ++itr;
        }
    }

    return 0;
}
```

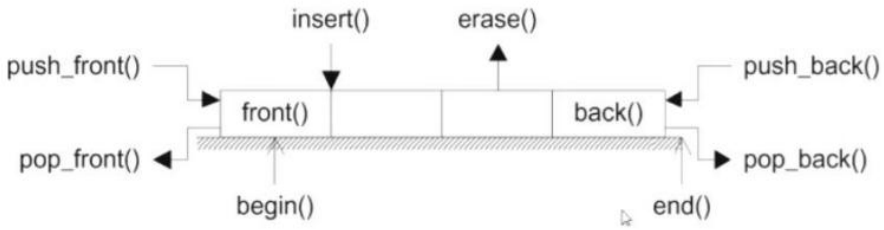
- 插入失效?

```
int main()
{
    int data[] = {1,2,3,4,5,6,7,8,9,0};
    list<int> li;
```

```
    li.assign(data,data+sizeof(data)/sizeof(int));
    list<int>::iterator itr = li.begin();
    for(; itr !=li.end(); ++itr)
    {
        if(*itr%2==0)
        {
            cout<<*itr<<endl;
            li.insert(itr,*itr);
        }
        cout<<*itr<<endl;
    }
    cout<<li.size()<<endl;

    return 0;
}
```

3.3.双端队列的基本使用

3.3.1.结构图示



双端队列（deque）

3.3.2.队列操作

```
#include <iostream>
#include <deque>
using namespace std;
int main()
{
    deque<int> di;
    cout<<di.size()<<endl;
    for(int i=0; i<10; i++)
    {
        di.push_back(i);
    }
    cout<<di.size()<<endl;
    while(!di.empty())
```

```
    {
        cout<<di.front()<<" ";
        di.pop_front();
    }
    cout<<endl<<di.size()<<endl;

    return 0;
}
```

3.3.3.栈操作

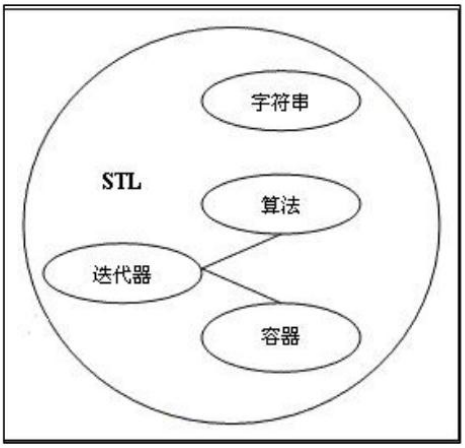
```
int main()
{
    deque<int> di;
    cout<<di.size()<<endl;
    for(int i=0; i<10; i++)
    {
        di.push_back(i);
    }
    cout<<di.size()<<endl;
    while(!di.empty())
    {
        cout<<di.back()<<" ";
        di.pop_back();
    }
    cout<<endl<<di.size()<<endl;

    return 0;
}
```

4.迭代器

定义：Iterator（迭代器）模式又称 Cursor（游标）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。

特点：

- 能够让迭代器与算法不干扰的相互发展，最后又能无间隙的粘合起来；
- 重载了 *，++，==，！=，= 运算符。用以操作复杂的数据结构；
- 容器提供迭代器，算法使用迭代器；



常用迭代器：

- iterator、const_iterator、reverse_iterator 和 const_reverse_iterator

应用：

- vector 中的应用

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    int a[10] = {1,3,5,7,9,2,4,6,8,0};
    vector<int> vi(a,a+10);
    vector<int>::iterator itr = vi.begin();
    for(;itr != vi.end(); ++itr)
    {
        cout<<*itr<<endl;
    }
    vector<int>::reverse_iterator ritr = vi.rbegin();
    for(;ritr != vi.rend(); ++ritr)
    {
        cout<<*ritr<<endl;
```

```
    }
    vector<int>::const_iterator citr = vi.begin();
    for(;citr != vi.end(); ++citr)
    {
        cout<<*citr<<endl;
        *citr = 1;
    }

    return 0;
}
```

- list 中的应用

```
#include <iostream>
#include <list>
using namespace std;

int main()
{
    int arr[] = {1,3,5,7,9,2,4,6,8,0};
    list<int> li(arr,arr+ sizeof(arr)/sizeof(arr[0]));
    auto itr = li.begin(); //c++11 支持
    for(; itr != li.end(); ++itr)
    {
        cout<<*itr<<endl;
    }

    return 0;
}
```

- map 中的应用

```
#include <iostream>
#include <map>
using namespace std;
int main()
{
    map<char,int> mci;
    mci['a'] = 100;
    mci['b'] = 200;
    mci['c'] = 300;
    auto itr = mci.begin();
    for(; itr != mci.end(); ++itr)
    {
        cout<<itr->first<<":"<<itr->second<<endl;
```

```
    }

    return 0;
}
```

迭代器失效问题：

根本原因 ：是对容器的某些操作修改了容器的内存状态（如容器重新加载到内存）或移动了容器内的某些元素。

- 使 vector 迭代器失效的操作
  - 向 vector 容器内添加元素(push_back,insert)
    1. 若向 vector 添加元素后，整个 vector 重新加载，即前后两次 vector 的 capacity()的返回值不同时，此时该容器 内的所有元素对应的迭代器都将失效
    2. 若该添加操作不会导致整个 vector 容器加载，则指向新插入元素后面的那些元素的迭代器都将失效。
  - 删除操作(erase,pop_back,clear)

    1. vector 执行删除操作后，被删除元素对应的迭代器以及其后面元素对应的迭代器都将失效。
  - resize 操作：调整当前容器的 size
    1. 若调整后 size>capacity，则会引起整个容器重新加载，整个容器的迭代器都将失效。
    2. 若调整后 size<capacity，则不会重新加载，具体情况如下：
       2.1. 若调整后容器的 size>调整前容器的 size，则原来 vector 的所有迭代器都不会失效；
       2.2. 若调整后容器的 size<调整前容器的 size，则容器中那些别切掉的元素对应的迭代器都将失效。
  - 赋值操作(v1=v2 /v1.assign(v2))

    1. 会导致左操作数 v1 的所有迭代器都失效， 显然右操作数 v2 的迭代器都不会失效。
  - 交换操作(v1.swap(v2))
    1. 由于在做交换操作时， v1,v2 均不会删除或插入元素， 所以容器内不会移动任何元素， 故 v1,v2 的所有迭代器都不会失效。
- 使 deque 迭代器失效的操作
  - 插入操作（push_front,push_back,insert)
    1. 在 deque 容器首部或尾部插入元素不会是任何迭代器失效；
    2. 在除去首尾的其他位置插入元素会使该容器的所有迭代器失效。
  - 删除操作
    1. 在 deque 首、尾删除元素只会使被删除元素对应的迭代器失效；
    2. 在其他任何位置的删除操作都会使得整个迭代器失效。
- 使 list/map/set 迭代器失效的操作

  由于 list/map/set 容器内的元素都是通过指针连接的， list 实现的数据结构是双向链表，而 map/set 实现的数据结构是红黑树，故这些容器的插入和删

除操作都只需更改指针的指向，不会移动容器内的元素，故在容器内增加元素时，不会使任何迭代器失效，而在删除元素时，仅仅会使得指向被删除的迭代器失效

5.仿函数

定义：重载了（）的类

使用：

```
bool isSingleDigit(const int &value)
{
    return value<10;
}
class isOdd
{
public:
    bool operator()(const int &value)
    {
        return (value%2) == 1;
    }
};
```

6.适配型容器

定义：STL 提供了三个容器适配器：queue、priority_queue、stack。这些适配器都是包装了 vector、list、deque 中某个顺序容器的包装器。注意：适配器没有提供迭代器，也不能同时插入或删除多个元素。

6.1.栈

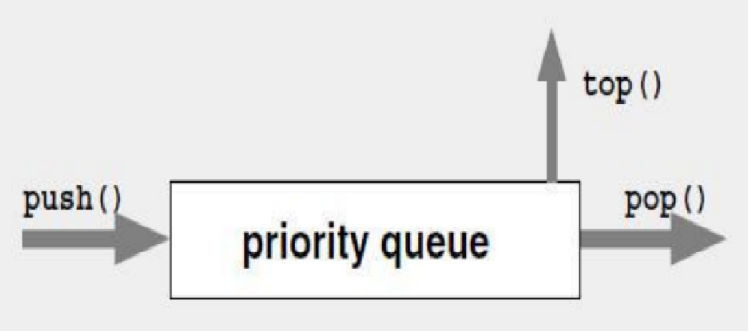
略

6.2.队列

略

6.3.优先队列

6.3.1.结构图示：



6.3.2.基本使用

```
#include <iostream>
#include <queue>
#include <functional>
using namespace std;

int main()
{
    int data[5] = {10,8,5,31,15};
    // queue<int> qi;
    priority_queue<int> qi;
    for(int i=0;i<5; i++)
    {
        qi.push(data[i]);
    }
    while(!qi.empty())
    {
```

```
        // cout<<qi.front()<<" ";
        cout<<qi.top()<<" ";
        qi.pop();
    }

    return 0;
}


int main()
{
    int data[5] = {10,8,5,31,15};
    // queue<int> qi;
    priority_queue<int,vector<int>,greater<int> > qi; //less<int>
    for(int i=0;i<5; i++)
    {
        qi.push(data[i]);
    }
    while(!qi.empty())
    {
        // cout<<qi.front()<<" ";
        cout<<qi.top()<<" ";
        qi.pop();
    }

    return 0;
}
```

6.3.3. 自定义使用

```
enum Type
{
    NAME,SEX,SCORE
};
struct Student
{
    string name;
    char sex;
    float score;
};

class Compare
{
```

```
public:
    Compare(bool s = true,enum Type i = NAME):status(s),itemType(i)
    {}
    bool operator()(const struct Student & first,const struct Student
                    & second)
    {
        if(status)
        {
            switch (itemType)
            {
            case NAME:
                return first.name> second.name;
            case SEX:
                return first.sex> second.sex;
            case SCORE:
                return first.score> second.score;
            }
        }
        else
        {
            switch (itemType)
            {
            case NAME:
                return first.name < second.name;
            case SEX:
                return first.sex < second.sex;
            case SCORE:
                return first.score < second.score;
            }
        }
    }
private:
    bool status;
    enum Type itemType;
};

int main()
{
    struct Student s[5]
    {
        {"aa",'e',10},
        {"bb",'d',11},
        {"cc",'c',12},
```

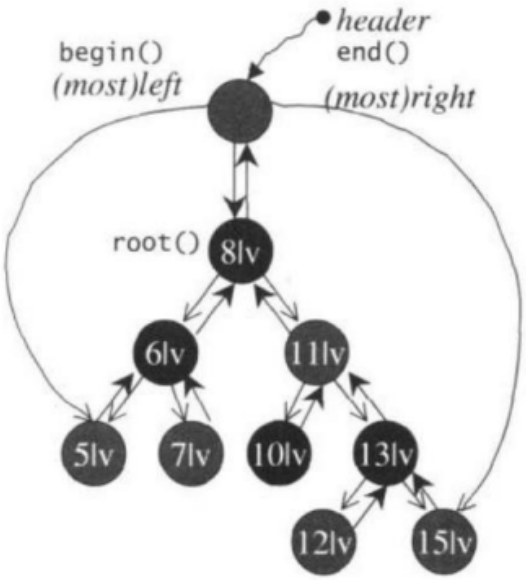
```
        {"dd",'b',13},
        {"ee",'a',14}
    };
    // priority_queue<struct Student> qs;
    typedef priority_queue<struct Student,vector<struct
Student>,Compare> QS;
    QS qs(Compare(true,SEX));
    for(int i=0; i<5; i++)
    {
        qs.push(s[i]);
    }
    while(!qs.empty())
    {
        struct Student t = qs.top();
        cout<<"name :"<<t.name<<endl;
        cout<<"sex :"<<t.sex<<endl;
        cout<<"score :"<<t.score<<endl;
        qs.pop();
    }

    return 0;
}
```

7.映射

7.1.map

特性：map 的特性是，所有元素会根据元素的键值自动被排序，map 的所有元素都是 pair,同时拥有实值(Value)和键值(Key).pair 的第一元素被视为键值，第二个元素被视为实值。map 不允许两个元素拥有相同的键值。

7.2.结构图示




```
template<class _T1, class _T2>
struct pair
{
    typedef _T1 first_type;
    typedef _T2 second_type;

    _T1 first;        // public
    _T2 second;       // public
    pair()
        : first(), second() { }
    pair(const _T1& __a, const _T2& __b)
        : first(__a), second(__b) { }
}
```

7.3.构造初始化

```
#include <iostream>
#include <vector>
#include <map>
using namespace std;

int main()
{
    vector<pair<string,int> > vp;

    vp.push_back(pair<string,int>("a",1));
    vp.push_back(pair<string,int>("b",2));
    vp.push_back(pair<string,int>("c",1));
    vp.push_back(pair<string,int>("a",1));
    vp.push_back(pair<string,int>("b",2));
    vp.push_back(pair<string,int>("c",1));

    map<string,int> msi(vp.begin(),vp.end());
    map<string,int>::iterator itr = msi.begin();

    for(; itr != msi.end(); ++itr)
    {
        cout<<"key:"<<itr->first<<"|value:"<<itr->second<<endl;
    }

    map<string,int> msi2(msi);
    itr = msi2.begin();
    for(; itr != msi2.end(); ++itr)
    {
        cout<<"key:"<<itr->first<<"| value:"<<itr->second<<endl;
    }

    return 0;
}
```

注意：

- Key 不可以重复，如果重复，则会忽略。但是 value 可以相同
- 以上为空构造器和范围构造器和复制构造器

7.4.元素的存取

```
int main()
{
    vector<pair<string,int> > vp;
```

```
    vp.push_back(pair<string,int>("a",1));
    vp.push_back(pair<string,int>("b",2));
    vp.push_back(pair<string,int>("c",1));

    map<string,int> msi(vp.begin(),vp.end());
    cout<<msi["a"]<<endl;
    cout<<msi["b"]<<endl;
    cout<<msi["c"]<<endl;

    msi["a"] = 100;
    msi["b"] = 200;
    msi["c"] = 100;
    msi["d"] = 100;

    cout<<msi["a"]<<endl;
    cout<<msi["b"]<<endl;
    cout<<msi["c"]<<endl;
    cout<<msi["d"]<<endl;

    cout<<(msi.at("a") = 500)<<endl;

    return 0;
}
```

注意：

- []操作符有插入和修改 value 的功效
- at 仅有修改和输出的功效

7.5.元素的修改

```
#include <iostream>
#include <map>
//#include <vector>
#include <list>
using namespace std;

void print(map<string,int> & msi)
{
    map<string,int>::iterator itr = msi.begin();
    for(; itr != msi.end(); ++itr)
    {
        cout<<"key:"<<itr->first<<"| value:"<<itr->second<<endl;
    }
    cout<<"==================="<<endl;
```

```
}

int main() //swap
{
    map<string,int> msi;

    msi.insert(pair<string,int>("a",1));
    msi.insert(pair<string,int>("b",2));
    msi.insert(pair<string,int>("c",3));
    print(msi);

    map<string,int> msi2;
    msi2.insert(pair<string,int>("x",1));
    msi2.insert(pair<string,int>("y",2));
    msi2.insert(pair<string,int>("z",3));

    msi.swap(msi2);
    print(msi);

    return 0;
}

int main2() //erase
{
    map<string,int> msi;

    msi.insert(pair<string,int>("a",1));
    msi.insert(pair<string,int>("b",2));
    msi.insert(pair<string,int>("c",3));
    print(msi);

    msi.erase("a");
    print(msi);

    msi.erase(msi.begin());
    print(msi);

    msi.erase(msi.begin(),msi.end());
    print(msi);

    msi.insert(pair<string,int>("a",1));
    msi.insert(pair<string,int>("b",2));
    msi.insert(pair<string,int>("c",3));
```

```
    msi.clear();
    print(msi);

    return 0;
}
int main1() //insert
{
    map<string,int> msi;

    // msi.insert(pair<string,int>("a",1));
    // msi.insert(pair<string,int>("b",2));
    // msi.insert(pair<string,int>("c",3));

    list<pair<string,int> > vp;

    vp.push_back(pair<string,int>("a",1));
    vp.push_back(pair<string,int>("b",2));
    vp.push_back(pair<string,int>("c",1));

    msi.insert(vp.begin(),vp.end());
    print(msi);

    return 0;
}
```

7.6.元素的操作

```
void print(map<char,int> & msi)
{
    map<char,int>::iterator itr = msi.begin();
    for(; itr != msi.end(); ++itr)
    {
        cout<<"key:"<<itr->first<<"| value:"<<itr->second<<endl;
    }
    cout<<"==================="<<endl;
}

int main() //lower_bound upper_bound range_bound
{
    map<char,int> msi;

    msi.insert(pair<char,int>('a',1));
    msi.insert(pair<char,int>('b',2));
```

```
    msi.insert(pair<char,int>('c',3));
    msi.insert(pair<char,int>('x',1));
    msi.insert(pair<char,int>('y',2));
    msi.insert(pair<char,int>('z',3));

    // map<char,int>::iterator itrlow = msi.lower_bound('b');
    // map<char,int>::iterator itrup = msi.upper_bound('y');
    // msi.erase(itrlow,itrup);
    print(msi);

    pair<map<char,int>::iterator,map<char,int>::iterator> ret;
    ret = msi.equal_range('c');

    cout<<ret.first->first<<":"<<ret.first->second<<endl;
    cout<<ret.second->first<<":"<<ret.second->second<<endl;

    return 0;
}

#if 1
int main2() //count
{
    map<char,int> msi;

    msi.insert(pair<char,int>('a',1));
    msi.insert(pair<char,int>('b',2));
    msi.insert(pair<char,int>('c',3));
    msi.insert(pair<char,int>('x',1));
    msi.insert(pair<char,int>('y',2));
    msi.insert(pair<char,int>('z',3));
    print(msi);

    for(char ch = 'a'; ch < 'z'; ch++)
    {
        if(msi.count(ch)>0)
            cout<<ch<<" is msi"<<endl;
        else

            cout<<ch<<" is not in msi"<<endl;
    }

    return 0;
}
```

```
int main1() //find
{
    map<string,int> msi;

    msi.insert(pair<string,int>("a",1));
    msi.insert(pair<string,int>("b",2));
    msi.insert(pair<string,int>("c",3));
    msi.insert(pair<string,int>("x",1));
    msi.insert(pair<string,int>("y",2));
    msi.insert(pair<string,int>("z",3));
    print(msi);

    cout<<msi.find("m")->second<<endl;
    cout<<msi.find("a")->second<<endl;

    map<string,int>::iterator itr;
    itr = msi.find("x");
    if(itr != msi.end())
        msi.erase(itr);
    print(msi);

    return 0;
}
#endif
```

7.7.提高

7.7.1.operator<

```
#include <iostream>
#include <map>
#include <algorithm>
using namespace std;

class KeyA
{
public:
    KeyA(string n, int s)
    {
        _name = n;
        _score = s;
    }
```

```
    friend bool operator<(const KeyA & first,const KeyA & second);
    string _name;
    int _score;
};

bool operator<(const KeyA & first,const KeyA & second)
{ /
    / return (first)._name > second._name;
    // return first._name <second._name;
    // return first._score >second._score;
    return first._score< second._score;
}

void print(const pair<class KeyA,int> & p)
{
    cout<<p.first._name<<endl;
    cout<<p.first._score<<endl;
    cout<<p.second<<endl;
    cout<<"++++++++++++++++++++++++++++"<<endl;
}

int main()
{
    map<class KeyA,int> mks;

    mks.insert(pair<class KeyA,int>(KeyA("aa",1),100));
    mks.insert(pair<class KeyA,int>(KeyA("bb",2),200));
    mks.insert(pair<class KeyA,int>(KeyA("cc",3),300));
    mks.insert(pair<class KeyA,int>(KeyA("dd",4),400));
    mks.insert(pair<class KeyA,int>(KeyA("ee",5),500));

    for_each(mks.begin(),mks.end(),print);

    return 0;
}
```

7.7.2. 带有函数比较器的 map 构造器

```
class KeyA
{
public:
    KeyA(string n, int s)
    {
        _name = n;
```

```
        _score = s;
    }
    string _name;
    int _score;
};

typedef bool (*funcComp)(const KeyA &first,const KeyA &second);

bool comp(const KeyA &first,const KeyA &second)
{
    return first._name <second._name;
}

void print(const pair<class KeyA,int> & p)
{
    cout<<p.first._name<<endl;
    cout<<p.first._score<<endl;
    cout<<p.second<<endl;
    cout<<"++++++++++++++++++++++++++++"<<endl;
}

int main()
{
    map<KeyA,int,funcComp> mks(comp);

    mks.insert(pair<KeyA,int>(KeyA("aa",1),100));
    mks.insert(pair<KeyA,int>(KeyA("bb",2),200));
    mks.insert(pair<KeyA,int>(KeyA("cc",3),300));
    mks.insert(pair<KeyA,int>(KeyA("dd",4),400));
    mks.insert(pair<KeyA,int>(KeyA("ee",5),500));

    for_each(mks.begin(),mks.end(),print);

    return 0;
}
```

7.7.3.带有类比较器的 map 构造器

```
class KeyA
{
public:
    KeyA(string n, int s)
    {
        _name = n;
```

```
        _score = s;
    }
    string _name;
    int _score;
};

class Compare
{
public:
    bool operator()(const KeyA &first,const KeyA &second)
    {
        return first._name < second._name;
    }

};

void print(const pair<class KeyA,int> & p)
{
    cout<<p.first._name<<endl;
    cout<<p.first._score<<endl;
    cout<<p.second<<endl;
    cout<<"++++++++++++++++++++++++++++"<<endl;
}

int main()
{
    map<KeyA,int,Compare> mks;// map<KeyA,int,Compare>
mks(Compare(1,2));

    mks.insert(pair<KeyA,int>(KeyA("aa",1),100));
    mks.insert(pair<KeyA,int>(KeyA("bb",2),200));
    mks.insert(pair<KeyA,int>(KeyA("cc",3),300));
    mks.insert(pair<KeyA,int>(KeyA("dd",4),400));
    mks.insert(pair<KeyA,int>(KeyA("ee",5),500));

    for_each(mks.begin(),mks.end(),print);

    return 0;
}
```

7.8.multimap

特性：特性与 map 完全相同，只是它不允许键值重复

7.8.1.使用

```
#include <iostream>
#include <map>
#include <algorithm>
using namespace std;

void print(const pair<string,int> & p)
{
    cout<<p.first<<endl;
    cout<<p.second<<endl;
}

int main()
{
    multimap<string,int> msi;

    msi.insert(pair<string,int>("a",1));
    msi.insert(pair<string,int>("b",2));
    msi.insert(pair<string,int>("c",3));
    msi.insert(pair<string,int>("d",4));

    msi.insert(pair<string,int>("a",1));
    msi.insert(pair<string,int>("b",2));
    msi.insert(pair<string,int>("c",3));
    msi.insert(pair<string,int>("d",4));

    for_each(msi.begin(),msi.end(),print);

    return 0;
}
```

8.集合

8.1.set

特性：所有元素都会根据元素的键值自动被排序，set 的元素不像 map 那样可以同时拥有实值(Value)和键值(Key),set 的键值就是实值，实值就是键值。set 不允许两个元素有相同的键值。

8.2.Set 使用

```
#include <iostream>
#include <set>
using namespace std;

int main()
{
    int data[10] = {1,2,3,4,5,1,2,3,4,5};
    set<int> si(data,data+10);

    set<int>::iterator itr = si.begin();

    for(;itr != si.end(); ++itr)
    {
        cout<<*itr<<endl;
    }

    return 0;
}
```

8.3.multiset

特性：和 set 完全一样，只是它不允许键值重复

8.4.multiset 使用

```
#include <iostream>
#include <set>
using namespace std;

int main()
{
    int data[10] = {1,2,3,4,5,1,2,3,4,5};

    multiset<int> si(data,data+10);
    multiset<int>::iterator itr = si.begin();

    for(;itr != si.end(); ++itr)
```

```
    {
        cout<<*itr<<endl;
    }

    return 0;
}
```

9.常用算法

特点：算法部分主要由头文件<algorithm>，<numeric>和<functional>组成。

- <algorithm>是所有 STL 头文件中最大的一个，其中常用到的功能范围涉及到比较、交换、查找、遍历操作、复制、修改、反转、排序、合并等等。
- <numeric>体积很小，只包括几个在序列上面进行简单数学运算的模板函数，包括加法和乘法在序列上的一些操作。
- <functional>中则定义了一些模板类，用以声明函数对象。

说明：

STL 提供了大量实现算法的模版函数，只要我们熟悉了 STL 之后，许多代码可以被大大的化简，只需要通过调用一两个算法模板，就可以完成所需要的功能，从而大大地提升效率。

分类：

- 查找算法（13 个）

函数名	头文件	函数功能
adjacent_find	<algorithm>	在iterator对标识元素范围内,查找一对相邻重复元素,找到则返回指向这对元素的第一个元素的ForwardIterator .否则返回last.重载版本使用输入的二元操作符代替相等的判断
	函数原形	template<class FwdIt> FwdIt adjacent_find(FwdIt first, FwdIt last);
		template<class FwdIt, class Pred> FwdIt adjacent_find(FwdIt first, FwdIt last, Pred pr);
binary_searh	<algorithm>	在有序序列中查找 value,找到返回 true.重载的版本实用指定的比较函数对象或函数指针来判断相等
	函数原形	template<class FwdIt, class T> bool binary_search(FwdIt first, FwdIt last, const T& val);
		template<class FwdIt, class T, class Pred> bool binary_search(FwdIt first, FwdIt last, const T& val,Pred pr);
count	<algorithm>	利用等于操作符,把标志范围内的元素与输入值比较,返回相等元素个数
	函数原形	template<class InIt, class Dist> size_t count(InIt first, InIt last,const T& val, Dist& n);
count_if	<algorithm>	利用输入的操作符,对标志范围内的元素进行操作,返回结果为 true 的个数
	函数原形	template<class InIt, class Pred, class Dist> size_t count_if(InIt first, InIt last, Pred pr);
equal_range	<algorithm>	功能类似 equal，返回一对 iterator，第一个表示 lower_bound，第二个表示 upper_bound
	函数原形	template<class FwdIt, class T> pair<FwdIt, FwdIt> equal_range(FwdIt first, FwdIt last,const T& val);
		template<class FwdIt, class T, class Pred> pair<FwdIt, FwdIt> equal_range(FwdIt first, FwdIt last,const T& val, Pred pr);
find	<algorithm>	利用底层元素的等于操作符,对指定范围内的元素与输入值进行比较.当匹配时,结束搜索,返回该元素的一个 InputIterator
	函数原形	template<class InIt, class T> InIt find(InIt first, InIt last, const T& val);
find_end	<algorithm>	在指定范围内查找"由输入的另外一对 iterator 标志的第二个序列"的最后一次出现.找到则返回最后一对的第一个 ForwardIterator,否则返回输入的"

		另外一对"的第一个 ForwardIterator.重载版本使用用户输入的操作符代替等于操作
	函数原形	template<class FwdIt1, class FwdIt2> FwdIt1 find_end(FwdIt1 first1, FwdIt1 last1,FwdIt2 first2, FwdIt2 last2);
		template<class FwdIt1, class FwdIt2, class Pred> FwdIt1 find_end(FwdIt1 first1, FwdIt1 last1,FwdIt2 first2, FwdIt2 last2, Pred pr);
find_first_of	<algorithm>	在指定范围内查找"由输入的另外一对 iterator 标志的第二个序列"中任意一个元素的第一次出现。重载版本中使用了用户自定义操作符
	函数原形	template<class FwdIt1, class FwdIt2> FwdIt1 find_first_of(FwdIt1 first1, FwdIt1 last1,FwdIt2 first2, FwdIt2 last2);
		template<class FwdIt1, class FwdIt2, class Pred> FwdIt1 find_first_of(FwdIt1 first1, FwdIt1 last1,FwdIt2 first2, FwdIt2 last2, Pred pr);
find_if	<algorithm>	使用输入的函数代替等于操作符执行 find
		template<class InIt, class Pred> InIt find_if(InIt first, InIt last, Pred pr);
lower_bound	<algorithm>	返回一个 ForwardIterator，指向在有序序列范围内的可以插入指定值而不破坏容器顺序的第一个位置.重载函数使用自定义比较操作
	函数原形	template<class FwdIt, class T> FwdIt lower_bound(FwdIt first, FwdIt last, const T& val);
		template<class FwdIt, class T, class Pred> FwdIt lower_bound(FwdIt first, FwdIt last, const T& val, Pred pr);
upper_boud	<algorithm>	返回一个 ForwardIterator,指向在有序序列范围内插入 value 而不破坏容器顺序的最后一个位置，该位置标志一个大于 value 的值.重载函数使用自定义比较操作
	函数原形	template<class FwdIt, class T> FwdIt upper_bound(FwdIt first, FwdIt last, const T& val);
		template<class FwdIt, class T, class Pred> FwdIt upper_bound(FwdIt first, FwdIt last, const T& val, Pred pr);
search	<algorithm>	给出两个范围，返回一个 ForwardIterator,查找成功指向第一个范围内第一次出现子序列(第二个范围)的位置，查找失败指向 last1,重载版本使用自定义的比较操作
	函数原形	template<class FwdIt1, class FwdIt2> FwdIt1 search(FwdIt1 first1, FwdIt1 last1,FwdIt2 first2, FwdIt2 last2);
		template<class FwdIt1, class FwdIt2, class Pred> FwdIt1 search(FwdIt1 first1, FwdIt1 last1, FwdIt2 first2, FwdIt2 last2, Pred pr);
search_n	<algorithm>	在指定范围内查找 val 出现 n 次的子序列。重载版本使用自定义的比较操作
	函数原形	template<class FwdIt, class Dist, class T> FwdIt search_n(FwdIt first, FwdIt last,Dist n, const T& val);
		template<class FwdIt, class Dist, class T, class Pred> FwdIt search_n(FwdIt first, FwdIt last,Dist n, const T& val, Pred pr);

- 堆算法

函数名	头文件	函数功能
make_heap	<algorithm>	把指定范围内的元素生成一个堆。重载版本使用自定义比较操作

	函数原形	template<class RanIt> void make_heap(RanIt first, RanIt last);
		template<class RanIt, class Pred> void make_heap(RanIt first, RanIt last, Pred pr);
pop_heap	<algorithm>	并不真正把最大元素从堆中弹出，而是重新排序堆。它把 first 和 last-1 交换，然后重新生成一个堆。可使用容器的 back 来访问被"弹出"的元素或者使用 pop_back 进行真正的删除。重载版本使用自定义的比较操作
	函数原形	template<class RanIt> void pop_heap(RanIt first, RanIt last);
		template<class RanIt, class Pred> void pop_heap(RanIt first, RanIt last, Pred pr);
push_heap	<algorithm>	假设 first 到 last-1 是一个有效堆，要被加入到堆的元素存放在位置 last-1，重新生成堆。在指向该函数前，必须先把元素插入容器后。重载版本使用指定的比较操作
	函数原形	template<class RanIt>void push_heap(RanIt first, RanIt last);
		template<class RanIt, class Pred> void push_heap(RanIt first, RanIt last, Pred pr);
sort_heap	<algorithm>	对指定范围内的序列重新排序，它假设该序列是个有序堆。重载版本使用自定义比较操作
	函数原形	template<class RanIt> void sort_heap(RanIt first, RanIt last);
		template<class RanIt, class Pred> void sort_heap(RanIt first, RanIt last, Pred pr);

- 关系算法（8 个）

函数名	头文件	函数功能
equal	<algorithm>	如果两个序列在标志范围内元素都相等，返回 true。重载版本使用输入的操作符代替默认的等于操作符
	函数原形	template<class InIt1, class InIt2> bool equal(InIt1 first, InIt1 last, InIt2 x);
		template<class InIt1, class InIt2, class Pred> bool equal(InIt1 first, InIt1 last, InIt2 x, Pred pr);
includes	<algorithm>	判断第一个指定范围内的所有元素是否都被第二个范围包含，使用底层元素的<操作符，成功返回 true。重载版本使用用户输入的函数
	函数原形	template<class InIt1, class InIt2> bool includes(InIt1 first1, InIt1 last1,InIt2 first2, InIt2 last2);
		template<class InIt1, class InIt2, class Pred> bool includes(InIt1 first1, InIt1 last1,InIt2 first2, InIt2 last2, Pred pr);
lexicographical_compare	<algorithm>	比较两个序列。重载版本使用用户自定义比较操作
	函数原形	template<class InIt1, class InIt2> bool lexicographical_compare(InIt1 first1, InIt1 last1,InIt2 first2, InIt2 last2);
		template<class InIt1, class InIt2, class Pred> bool lexicographical_compare(InIt1 first1, InIt1 last1,InIt2 first2, InIt2 last2, Pred pr);
max	<algorithm>	返回两个元素中较大一个。重载版本使用自定义比较操作
	函数原形	template<class T> const T& max(const T& x, const T& y);
		template<class T, class Pred> const T& max(const T& x, const T& y, Pred pr);

max_element	<algorithm>	返回一个 ForwardIterator，指出序列中最大的元素。重载版本使用自定义比较操作
	函数原形	template<class FwdIt> FwdIt max_element(FwdIt first, FwdIt last);
		template<class FwdIt, class Pred> FwdIt max_element(FwdIt first, FwdIt last, Pred pr);
min	<algorithm>	返回两个元素中较小一个。重载版本使用自定义比较操作
	函数原形	template<class T> const T& min(const T& x, const T& y);
		template<class T, class Pred> const T& min(const T& x, const T& y, Pred pr);
min_element	<algorithm>	返回一个 ForwardIterator，指出序列中最小的元素。重载版本使用自定义比较操作
	函数原形	template<class FwdIt> FwdIt min_element(FwdIt first, FwdIt last);
		template<class FwdIt, class Pred> FwdIt min_element(FwdIt first, FwdIt last, Pred pr);
mismatch	<algorithm>	并行比较两个序列，指出第一个不匹配的位置，返回一对 iterator，标志第一个不匹配元素位置。如果都匹配，返回每个容器的 last。重载版本使用自定义的比较操作
	函数原形	template<class InIt1, class InIt2> pair<InIt1, InIt2> mismatch(InIt1 first, InIt1 last, InIt2 x);
		template<class InIt1, class InIt2, class Pred> pair<InIt1, InIt2> mismatch(InIt1 first, InIt1 last, InIt2 x, Pred pr);

- 集合算法（4 个）

函数名	头文件	函数功能
set_union	<algorithm>	构造一个有序序列，包含两个序列中所有的不重复元素。重载版本使用自定义的比较操作
	函数原形	template<class InIt1, class InIt2, class OutIt> OutIt set_union(InIt1 first1, InIt1 last1, InIt2 first2, InIt2 last2, OutIt x);
		template<class InIt1, class InIt2, class OutIt, class Pred> OutIt set_union(InIt1 first1, InIt1 last1,InIt2 first2, InIt2 last2,OutIt x, Pred pr);
set_intersection	<algorithm>	构造一个有序序列，其中元素在两个序列中都存在。重载版本使用自定义的比较操作
	函数原形	template<class InIt1, class InIt2, class OutIt> OutIt set_intersection(InIt1 first1, InIt1 last1,InIt2 first2, InIt2 last2, OutIt x);
		template<class InIt1, class InIt2, class OutIt, class Pred> OutIt set_intersection(InIt1 first1, InIt1 last1,InIt2 first2,InIt2 last2, OutIt x, Pred pr);
set_difference	<algorithm>	构造一个有序序列，该序列仅保留第一个序列中存在的而第二个中不存在的元素。重载版本使用自定义的比较操作
	函数原形	template<class InIt1, class InIt2, class OutIt> OutIt set_difference(InIt1 first1, InIt1 last1,InIt2 first2, InIt2 last2, OutIt x);
		template<class InIt1, class InIt2, class OutIt, class Pred> OutIt set_difference(InIt1 first1, InIt1 last1, InIt2 first2, InIt2 last2, OutIt x, Pred pr);
set_symmetric_d	<algorithm>	构造一个有序序列，该序列取两个序列的对称差集(并集-交集)

ifference	函数原形	template<class InIt1, class InIt2, class OutIt> OutIt set_symmetric_difference(InIt1 first1, InIt1 last1, InIt2 first2, InIt2 last2, OutIt x);
		template<class InIt1, class InIt2, class OutIt, class Pred> OutIt set_symmetric_difference(InIt1 first1, InIt1 last1, InIt2 first2, InIt2 last2, OutIt x, Pred pr);

- 列组合算法（2 个）

函数名	头文件	函数功能
next_permutation	<algorithm>	取出当前范围内的排列，并重新排序为下一个排列。重载版本使用自定义的比较操作
	函数原形	template<class BidIt> bool next_permutation(BidIt first, BidIt last);
		template<class BidIt, class Pred> bool next_permutation(BidIt first, BidIt last, Pred pr);
prev_permutation	<algorithm>	取出指定范围内的序列并将它重新排序为上一个序列。如果不存在上一个序列则返回 false。重载版本使用自定义的比较操作
	函数原形	template<class BidIt> bool prev_permutation(BidIt first, BidIt last);
		template<class BidIt, class Pred> bool prev_permutation(BidIt first, BidIt last, Pred pr);

- 排序和通用算法（14 个）

函数名	头文件	函数功能
inplace_merge	<algorithm>	合并两个有序序列，结果序列覆盖两端范围。重载版本使用输入的操作进行排序
	函数原形	template<class BidIt> void inplace_merge(BidIt first, BidIt middle, BidIt last);
		template<class BidIt, class Pred> void inplace_merge(BidIt first, BidIt middle, BidIt last, Pred pr);
merge	<algorithm>	合并两个有序序列，存放到另一个序列。重载版本使用自定义的比较
	函数原形	template<class InIt1, class InIt2, class OutIt> OutIt merge(InIt1 first1, InIt1 last1,InIt2 first2, InIt2 last2, OutIt x);
		template<class InIt1, class InIt2, class OutIt, class Pred> OutIt merge(InIt1 first1, InIt1 last1,InIt2 first2, InIt2 last2, OutIt x, Pred pr);
nth_element	<algorithm>	将范围内的序列重新排序，使所有小于第 n 个元素的元素都出现在它前面，而大于它的都出现在后面。重载版本使用自定义的比较操作
	函数原形	template<class RanIt> void nth_element(RanIt first, RanIt nth, RanIt last);
		template<class RanIt, class Pred> void nth_element(RanIt first, RanIt nth, RanIt last, Pred pr);
partial_sort	<algorithm>	对序列做部分排序，被排序元素个数正好可以被放到范围内。重载版本使用自定义的比较操作
	函数原形	template<class RanIt> void partial_sort(RanIt first, RanIt middle, RanIt last);

		template<class RanIt, class Pred> void partial_sort(RanIt first, RanIt middle, RanIt last, Pred pr);
partial_sort_copy	<algorithm>	与 partial_sort 类似，不过将经过排序的序列复制到另一个容器
	函数原形	template<class InIt, class RanIt> RanIt partial_sort_copy(InIt first1, InIt last1,RanIt first2, RanIt last2);
		template<class InIt, class RanIt, class Pred> RanIt partial_sort_copy(InIt first1, InIt last1,RanIt first2, RanIt last2, Pred pr);
partition	<algorithm>	对指定范围内元素重新排序，使用输入的函数，把结果为 true 的元素放在结果为 false 的元素之前
	函数原形	template<class BidIt, class Pred> BidIt partition(BidIt first, BidIt last, Pred pr);
random_shuffle	<algorithm>	对指定范围内的元素随机调整次序。重载版本输入一个随机数产生操作
	函数原形	template<class RanIt> void random_shuffle(RanIt first, RanIt last);
		template<class RanIt, class Fun> void random_shuffle(RanIt first, RanIt last, Fun& f);
reverse	<algorithm>	将指定范围内元素重新反序排序
	函数原形	template<class BidIt> void reverse(BidIt first, BidIt last);
reverse_copy	<algorithm>	与 reverse 类似，不过将结果写入另一个容器
	函数原形	template<class BidIt, class OutIt> OutIt reverse_copy(BidIt first, BidIt last, OutIt x);
rotate	<algorithm>	将指定范围内元素移到容器末尾，由 middle 指向的元素成为容器第一个元素
	函数原形	template<class FwdIt> void rotate(FwdIt first, FwdIt middle, FwdIt last);
rotate_copy	<algorithm>	与 rotate 类似，不过将结果写入另一个容器
	函数原形	template<class FwdIt, class OutIt> OutIt rotate_copy(FwdIt first, FwdIt middle, FwdIt last, OutIt x);
sort	<algorithm>	以升序重新排列指定范围内的元素。重载版本使用自定义的比较操作
	函数原形	template<class RanIt> void sort(RanIt first, RanIt last);
		template<class RanIt, class Pred> void sort(RanIt first, RanIt last, Pred pr);
stable_sort	<algorithm>	与 sort 类似，不过保留相等元素之间的顺序关系
	函数原形	template<class BidIt> void stable_sort(BidIt first, BidIt last);
		template<class BidIt, class Pred> void stable_sort(BidIt first, BidIt last, Pred pr);
stable_partition	<algorithm>	与 partition 类似，不过不保证保留容器中的相对顺序
	函数原形	template<class FwdIt, class Pred> FwdIt stable_partition(FwdIt first, FwdIt last, Pred pr);

- 删除和替换算法（15 个）

函数名	头文件	函数功能
copy	<algorithm>	复制序列
	函数原形	template<class InIt, class OutIt> OutIt copy(InIt first, InIt last, OutIt x);
copy_backward	<algorithm>	与 copy 相同，不过元素是以相反顺序被拷贝
	函数原形	template<class BidIt1, class BidIt2> BidIt2 copy_backward(BidIt1 first, BidIt1 last, BidIt2 x);
iter_swap	<algorithm>	交换两个 ForwardIterator 的值
	函数原形	template<class FwdIt1, class FwdIt2> void iter_swap(FwdIt1 x, FwdIt2 y);
remove	<algorithm>	删除指定范围内所有等于指定元素的元素。注意，该函数不是真正删除函数。内置函数不适合使用 remove 和 remove_if 函数
	函数原形	template<class FwdIt, class T> FwdIt remove(FwdIt first, FwdIt last, const T& val);
remove_copy	<algorithm>	将所有不匹配元素复制到一个制定容器，返回 OutputIterator 指向被拷贝的末元素的下一个位置
	函数原形	template<class InIt, class OutIt, class T> OutIt remove_copy(InIt first, InIt last, OutIt x, const T& val);
remove_if	<algorithm>	删除指定范围内输入操作结果为 true 的所有元素
	函数原形	template<class FwdIt, class Pred> FwdIt remove_if(FwdIt first, FwdIt last, Pred pr);
remove_copy_if	<algorithm>	将所有不匹配元素拷贝到一个指定容器
	函数原形	template<class InIt, class OutIt, class Pred> OutIt remove_copy_if(InIt first, InIt last, OutIt x, Pred pr);
replace	<algorithm>	将指定范围内所有等于 vold 的元素都用 vnew 代替
	函数原形	template<class FwdIt, class T> void replace(FwdIt first, FwdIt last,const T& vold, const T& vnew);
replace_copy	<algorithm>	与 replace 类似，不过将结果写入另一个容器
	函数原形	template<class InIt, class OutIt, class T> OutIt replace_copy(InIt first, InIt last, OutIt x,const T& vold, const T& vnew);
replace_if	<algorithm>	将指定范围内所有操作结果为 true 的元素用新值代替
	函数原形	template<class FwdIt, class Pred, class T> void replace_if(FwdIt first, FwdIt last,Pred pr, const T& val);
replace_copy_if	<algorithm>	与 replace_if，不过将结果写入另一个容器
	函数原形	template<class InIt, class OutIt, class Pred, class T> OutIt replace_copy_if(InIt first, InIt last, OutIt x, Pred pr, const T& val);
swap	<algorithm>	交换存储在两个对象中的值
	函数原形	template<class T> void swap(T& x, T& y);
swap_range	<algorithm>	将指定范围内的元素与另一个序列元素值进行交换
	函数原形	template<class FwdIt1, class FwdIt2> FwdIt2 swap_ranges(FwdIt1 first, FwdIt1 last, FwdIt2 x);
unique	<algorithm>	清除序列中重复元素，和 remove 类似，它也不能真正删除元素。重载版本使用自定义比较操作

	函数原形	template<class FwdIt> FwdIt unique(FwdIt first, FwdIt last);
		template<class FwdIt, class Pred> FwdIt unique(FwdIt first, FwdIt last, Pred pr);
unique_copy	<algorithm>	与 unique 类似，不过把结果输出到另一个容器
	函数原形	template<class InIt, class OutIt> OutIt unique_copy(InIt first, InIt last, OutIt x);
		template<class InIt, class OutIt, class Pred> OutIt unique_copy(InIt first, InIt last, OutIt x, Pred pr);

- 生成和编译算法（6 个）

函数名	头文件	函数功能
fill	<algorithm>	将输入值赋给标志范围内的所有元素
	函数原形	template<class FwdIt, class T> void fill(FwdIt first, FwdIt last, const T& x);
fill_n	<algorithm>	将输入值赋给 first 到 first+n 范围内的所有元素
	函数原形	template<class OutIt, class Size, class T> void fill_n(OutIt first, Size n, const T& x);
for_each	<algorithm>	用指定函数依次对指定范围内所有元素进行迭代访问，返回所指定的函数类型。该函数不得修改序列中的元素
	函数原形	template<class InIt, class Fun> Fun for_each(InIt first, InIt last, Fun f);
generate	<algorithm>	连续调用输入的函数来填充指定的范围
	函数原形	template<class FwdIt, class Gen> void generate(FwdIt first, FwdIt last, Gen g);
generate_n	<algorithm>	与 generate 函数类似，填充从指定 iterator 开始的 n 个元素
	函数原形	template<class OutIt, class Pred, class Gen> void generate_n(OutIt first, Dist n, Gen g);
transform	<algorithm>	将输入的操作作用与指定范围内的每个元素，并产生一个新的序列。重载版本将操作作用在一对元素上，另外一个元素来自输入的另外一个序列。结果输出到指定容器
	函数原形	template<class InIt, class OutIt, class Unop> OutIt transform(InIt first, InIt last, OutIt x, Unop uop);
		template<class InIt1, class InIt2, class OutIt, class Binop> OutIt transform(InIt1 first1, InIt1 last1, InIt2 first2,OutIt x, Binop bop);

- 算数算法（4 个）

函数名	头文件	函数功能
accumulate	<numeric>	iterator 对标识的序列段元素之和，加到一个由 val 指定的初始值上。重载版本不再做加法，而是传进来的二元操作符被应用到元素上
	函数原形	template<class InIt, class T> T accumulate(InIt first, InIt last, T val);
		template<class InIt, class T, class Pred> T accumulate(InIt first, InIt last, T val, Pred pr);
partial_sum	<numeric>	创建一个新序列，其中每个元素值代表指定范围内该位置前所有元素之和。重载版本使用自定义操作代替加法

	函数原形	template<class InIt, class OutIt> OutIt partial_sum(InIt first, InIt last,OutIt result);
		template<class InIt, class OutIt, class Pred> OutIt partial_sum(InIt first, InIt last,OutIt result, Pred pr);
product	<numeric>	对两个序列做内积(对应元素相乘，再求和)并将内积加到一个输入的初始值上。重载版本使用用户定义的操作
	函数原形	template<class InIt1, class InIt2, class T> T product(InIt1 first1, InIt1 last1,Init2 first2, T val);
		template<class InIt1, class InIt2, class T,class Pred1, class Pred2> T product(InIt1 first1, InIt1 last1,Init2 first2, T val, Pred1 pr1, Pred2 pr2);
adjacent_difference	<numeric>	创建一个新序列，新序列中每个新值代表当前元素与上一个元素的差。重载版本用指定二元操作计算相邻元素的差
	函数原形	template<class InIt, class OutIt> OutIt adjacent_difference(InIt first, InIt last,OutIt result);
		template<class InIt, class OutIt, class Pred> OutIt adjacent_difference(InIt first, InIt last,OutIt result, Pred pr);

9.1.常用算法举例

9.1.1.遍历算法

- for_each

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

void print(int & v)
{
    cout<<v<<endl;
    v = 200;
}

class Print
{
public:
    Print()
    {
        count = 0;
    }
    void operator()(int &v)
    {
        cout<<v<<endl;
```

```
        count ++;
    }
    int getCount()
    {
        return count;
    }
private:
    int count;
};

int main()
{
    vector<int> vi(4,5);
    Print retObj;

    for_each(vi.begin(),vi.end(),print);

    retObj = for_each(vi.begin(),vi.end(),Print());
    cout<<retObj.getCount()<<endl;

    return 0;
}
```

9.2.排序算法

sort	对给定区间所有元素进行排序
stable_sort	对给定区间所有元素进行稳定排序
partial_sort	对给定区间所有元素部分排序
partial_sort_copy	对给定区间复制并排序
partition	使得符合某个条件的元素放在前面
stable_partition	相对稳定的使得符合某个条件的元素放在前面

注意：系统自己为 sort 提供了 less 仿函数，在 STL 中还提供了其它仿函数

```
sort 中的其他比较函数
equal_to 相等
not_equal_to 不相等
less 小于
greater 大于
less_equal 小于等于
greater_equal 大于等于
```

使用仿函数注意：

- 不能直接写入仿函数的名字，而是要写其重载的（）函数：less<int>();
- 当容器中元素是标准类型（int，float，char,string 等），可以直接使用
- 如果是自定义类型或者按照自定义方式排序，需要：
  - 自己写比较函数
  - 重载<>操作符

```
#include <vector>
using namespace std;
//全排序
void print(int &v) // 输出一次调用一次
{
cout<<v<<" ";
}

int main()
{
vector<int> vi;
for(int i=0; i<5; i++) vi.push_back(i);

sort(vi.begin(),vi.end(),less<int>());

for_each(vi.begin(),vi.end(),print);

return 0;
}
```

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
//稳定排序
void print(int &v)
{
    cout<<v<<" ";
}

int main()
{
    vector<int> vi;
    for(int i=0; i<5; i++) vi.push_back(i);

    stable_sort(vi.begin(),vi.end(),greater<int>());
```

```
    for_each(vi.begin(),vi.end(),print);

    return 0;
}
```

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
//部分排序
void print(vector<int> & vi)
{
    vector<int>::iterator itr = vi.begin();
    for(;itr != vi.end(); ++itr)
    {
        cout<<*itr<<" ";
    }
    cout<<"+++++++++++++++"<<endl;
}

int main()
{
    int a[10]={3,6,9,2,5,8,1,4,7,0};
    vector<int> vi(a,a+10);

    // partial_sort(vi.begin(),vi.begin()+5,vi.end());
    partial_sort(vi.begin(),vi.begin()+5,vi.end(),greater<int>());
    print(vi);

    return 0;
}
```

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
//部分赋值并排序
void print(vector<int> & vi)
{
    vector<int>::iterator itr = vi.begin();
    for(;itr != vi.end(); ++itr)
    {
        cout<<*itr<<" ";
```

```
    }

    cout<<"+++++++++++++++"<<endl;
}
int main()
{
    int data[10]={3,6,9,2,5,8,1,4,7,0};
    vector<int> vi(5);

    // partial_sort(vi.begin(),vi.begin()+5,vi.end());
    partial_sort_copy(data,data+10,vi.begin(),vi.end(),less<int>());
    print(vi);

    return 0;
}
```

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
//分区操作
void print(vector<int> & vi)
{
    vector<int>::iterator itr = vi.begin();
    for(;itr != vi.end(); ++itr)
    {
        cout<<*itr<<" ";
    }

    cout<<"+++++++++++++++"<<endl;
}

bool myfunc(int i )
{
    return (i%2) == 1;
}

int main()
{
    int data[10]={3,6,9,2,5,8,1,4,7,0};
    vector<int> vi(data,data+10);

    // partial_sort(vi.begin(),vi.begin()+5,vi.end());
```

```
    partition(vi.begin(),vi.end(),myfunc);
    print(vi);

    return 0;
}
```

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
using namespace std;
//排序 n 个元素
int main()
{
    vector<int> vi;
    for(int i=0; i<10; i++) vi.push_back(i);
    random_shuffle(vi.begin(),vi.end()); //重新洗牌的意思

    nth_element(vi.begin(),vi.begin()+3,vi.end());

    copy(vi.begin(),vi.end(),ostream_iterator<int>(cout," "));

    return 0;
}
```

- 总结：（effective stl 建议）
  - 若需对 vector, string, deque, 或 array 容器进行全排序，你可选择 sort 或 stable_sort;
  - 若只需对 vector, string, deque, 或 array 容器中取得 top n 的元素，部分排序 partial_sort 是首选.
  - 若对于 vector, string, deque, 或 array 容器,你需要找到第 n 个位置的元素或者你需要得到 top n 且不关系 top n 中的内部 顺序, nth_element 是最理想的;
  - 若你需要从标准序列容器或者 array 中把满足某个条件 或者不满足某个条件的元素分开，你最好使用 partition 或 stable_partition;
  - 若使用的 list 容器，你可以直接使用 partition 和 stable_partition 算法，你可以使用 list::sort 代替 sort 和 stable_sort 排 序。

9.3.查找算法

find() 比较条件为相等的查找
find_if() 自定义比较函数版
search_n() 查询单个元素重复出现的位置
adjacent_find() 查询区间中重复元素出现的位置
lower_bound() 有序区间中查询元素边界
binary_search() 有序区间的二分查找，返回真假
min_element() 查找最小元素

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
    int data[5] = {1,2,3,4,5};
    int *p = find(data,data+5,4);
    if(p != data+5)
        printf("%d\n",*p);
    else
        printf("find none\n");

    vector<int> vi(data,data+5);

    vector<int>::iterator itr;
    itr = find(vi.begin(),vi.end(),10);
    if(itr != vi.end())
        cout<<*itr<<endl;
    else
        cout<<"find none"<<endl;

    return 0;
}
```

```
bool isSingle(int &v)
{
    return v%5==0;
}

class isOdd
{
public:
```

```
    bool operator()(int &v)
    {
        return v%2 == 1;
    }
};

int main()
{
    int data[5] = {1,2,3,4,5};
    vector<int> vi(data,data+5);

    vector<int>::iterator itr;
    itr = find_if(vi.begin(),vi.end(),isSingle);
    if(itr != vi.end())
        cout<<"in the vector"<<endl;
    else
        cout<<"find none"<<endl;

    itr = find_if(vi.begin(),vi.end(),isOdd());
    if(itr != vi.end())
        cout<<"in the vector"<<endl;
    else
        cout<<"find none"<<endl;

    return 0;
}
```

9.4.区间查找

search() 查找子区间首次出现的位置
find_end() 查找子区间最后一次出现的位置
equal() 判断两个区间是否相等，返回 bool 值
mismatch() 查询两个区间首次出现不同的位置；

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int myfunc(int i,int j)
{
    return (i-j) == 1;
```

```
}

int main()
{
    int data[10] = {1,2,3,4,5,6,7,8,9,0};
    vector<int> sea(data,data+10);

    int needle[2] = {4,5};

    vector<int>::iterator itr;
    // itr = search(sea.begin(),sea.end(),needle,needle+2);
    itr = search(sea.begin(),sea.end(),needle,needle+2,myfunc);

    if(itr != sea.end())
        cout<<"in the vector "<<*itr<<endl;
    else
        cout<<"find none"<<endl;

    return 0;
}
```

```
int main()
{
    int data[5] = {10,30,50,70,90};
    vector<int> vi;

    for(int i=1; i<6; i++)
        vi.push_back(i*10);

    pair<vector<int>::iterator, int *> ret;
    ret = mismatch(vi.begin(),vi.end(),data);

    cout<<"first mismatch:"<<*(ret.first)<<endl;
    cout<<"second:"<<*(ret.second)<<endl;

    return 0;
}
```

9.5.删除算法

remove(iterator first,iterator last,const T& val)
移除所有值到等于 val 的元素，函数返回删除后的序列的末尾位置，序列的大小不变，用最后一个元素进行补齐。
remove_copy(iterator1 first,iterator1 last,iterator2 dest,const T& value)
在复制的过程中移除所有值等于 value 的元素。
unique(iterator first,iterator last) 移除区间中的连续重复元素
unique_copy(iterator1 first,iterator1 last,iterator2 dest)
复制时移除区间中的连续重复元素

```
#include <iostream>
#include <iterator>
#include <algorithm>
#include <vector>
using namespace std;

int main()
{
    int source[]={1,4,4,6,1,2,2,3,1,6,6,6,5,7,5,4,4};
    int n = sizeof(source)/sizeof(int);

    vector<int> vi(source,source+n);
    copy(vi.begin(),vi.end(),ostream_iterator<int>(cout," "));
    cout<<endl;

    vector<int>::iterator itr;
    itr = remove(vi.begin(),vi.end(),4);
    vi.erase(itr,vi.end());

    copy(vi.begin(),vi.end(),ostream_iterator<int>(cout," "));
    cout<<endl;

    return 0;
}
```

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
using namespace std;

int main()
{
```

```
    int data[10] = {1,2,3,4,5,8,9,9,9,8};
    vector<int> vi(10);

    remove_copy(data,data+10,vi.begin(),9);
    copy(vi.begin(),vi.end(),ostream_iterator<int>(cout," "));

    return 0;
}
```

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
using namespace std;

int main()
{
    int data[10] = {1,2,3,4,5,8,9,9,9,8};
    vector<int> vi(data,data+10);

    sort(vi.begin(),vi.end());

    vector<int>::iterator itr = unique(vi.begin(),vi.end());

    vi.erase(itr,vi.end());

    copy(vi.begin(),vi.end(),ostream_iterator<int>(cout," "));

    return 0;
}
```

```
int main()
{
    int data[10] = {1,2,3,4,5,8,9,9,9,8};
    vector<int> vi(10);

    sort(data,data+10);

    unique_copy(data,data+10,vi.begin());

    copy(vi.begin(),vi.end(),ostream_iterator<int>(cout," "));

    return 0;
```

```
}
```

9.6.集合算法

set_union:构造一个有序序列，包含两个序列中所有的不重复元素。
set_intersection:构造一个有序序列，其中元素在两个序列中都存在。
set_difference:构造一个有序序列，该序列仅保留第一个序列中存在的而第二个序列中不存在的元素。

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
using namespace std;

int main()
{
    int first[5] = {1,2,5,7,9};
    int second[5] = {1,2,6,8,10};

    vector<int> vi(20);
    vector<int>::iterator itr;

    sort(first,first+5);
    sort(second,second+5);

    itr = set_union(first,first+5,second,second+5,vi.begin());

    vi.resize(itr -vi.begin());
    copy(vi.begin(),vi.end(),ostream_iterator<int>(cout," "));

    return 0;
}
```

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

int main()
{
    int first[5] = {1,2,5,7,9};
    int second[5] = {1,2,6,8,10};

    vector<int> vi(20);
```

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

    sort(first,first+5);
    sort(second,second+5);

    itr = set_intersection(first,first+5,second,second+5,vi.begin());

    vi.resize(itr -vi.begin());
    copy(vi.begin(),vi.end(),ostream_iterator<int>(cout," "));

    return 0;
}
```

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
using namespace std;

int main()
{
    int first[5] = {1,2,5,7,9};
    int second[5] = {1,2,6,8,10};

    vector<int> vi(20);
    vector<int>::iterator itr;

    sort(first,first+5);
    sort(second,second+5);

    itr = set_difference(first,first+5,second,second+5,vi.begin());

    vi.resize(itr -vi.begin());
    copy(vi.begin(),vi.end(),ostream_iterator<int>(cout," "));

    return 0;
}
```

10.项目

10.1.大数据检索

现有数据，格式如下：

C/C++163.com,小码哥，（单位：广州），电话：110120130140

这种文件打开，检索和修改效率很低，现在要求设计一款软件，以邮箱为检索关键字，对文件进行检索和修改。

10.1.1.代码实现

```
#include <iostream>
#include <fstream>
#include <list>
#include <map>
#include <string.h>
using namespace std;


class Infor
{
public:
    Infor(string mail, string other)
        :_mail(mail),_other(other){
    }
    string getMail()
    {
        return _mail;
    }
    string getOther()
    {
        return _other;
    }
private:
    string _mail;
    string _other;
};

void loadInfor(list<Infor*>& lif,map<string,Infor*> &msi)
{
    Infor * pi;
    FILE *fp = fopen("ddw.txt","r+");
    char buf[1024] = {0};
    char *mail,*other;
```

```
    while(fgets(buf,1024,fp) != NULL)
    {
        mail = buf;
        other = strchr(buf,',');
        if(other == NULL)
            continue;
        *(other) = '\0';
        pi = new Infor(mail,other+1);
        lif.push_back(pi);
        msi.insert(pair<string,Infor*>(pi->getMail(),pi));
    }

    map<string,Infor*>::iterator itr = msi.begin();


}


void searchInfor(list<Infor*>& lif,map<string,Infor*> &msi)
{
    string find;
    int choice;
    map<string,Infor*>::iterator itr;

    while(1)
    {
        system("cls");
        printf("1->list search 2->map search 3 return release\n");
        cin>>choice;
        switch (choice)
        {
        case 1:
            cin>>find;
            itr = msi.find(find);
            if(itr != msi.end())
                cout<<"find"<<endl;
            else
                cout<<"find none"<<endl;
            getchar();
            getchar();
            getchar();
            getchar();
```

```
            break;
        case 2:
            break;
        case 3:
            break;
        default:
            break;
        }
    }
}

int main()
{
    list<Infor*> lif;
    map<string,Infor*> msi;
    loadInfor(lif,msi);
    searchInfor(lif,msi);
    return 0;
}
```

10.2.演讲比赛

某市举行一场演讲比赛（ speech_contest ），共有 24 个人参加。比赛共三轮，前两轮为淘汰赛，第三轮为决赛。

比赛方式：分组比赛，每组 6 个人；选手每次要随机分组，进行比赛；

第一轮分为 4 个小组，每组 6 个人。比如 100-105 为一组，106-111 为第二组，依次类推，每人分别按照抽签（draw）顺序演讲。当小组演讲完后，淘汰组内排名最后的三个选手，然后继续下一个小组的比赛。

第二轮分为 2 个小组，每组 6 人。比赛完毕，淘汰组内排名最后的三个选手，然后继续下一个小组的比赛。

第三轮只剩下 6 个人，本轮为决赛，选出前三名。

比赛评分：10 个评委打分，去除最低、最高分，求平均分

每个选手演讲完由 10 个评委分别打分。该选手的最终得分是去掉一个最高分和一个最低分，求得剩下的 8 个成绩的平均分。

选手的名次按得分降序排列，若得分一样，按参赛号升序排名。

用 STL 编程，求解这个问题

- 请打印出所有选手的名字与参赛号，并以参赛号的升序排列。
- 打印每一轮比赛后，小组比赛成绩和小组晋级名单
- 打印决赛前三名，选手名称、成绩。

10.2.1.需求分析

总体思想：

- 产生选手（ABCDEFGHIJKLMNOPQRSTUVWXYZ）姓名，得分，选手编号
- 第一轮：选手抽签 选手比赛 查看比赛结果
- 第二轮：选手抽签 选手比赛 查看比赛结果
- 第三轮：选手抽签 选手比赛 查看比赛结果

实现步骤：

1. 把选手信息，选手得分信息，选手比赛抽签信息，选手晋级信息保存到容器中，这时候需要确定各个容器的选型
2. 选手设计一个类 Speaker（姓名和得分）
3. 所有选手的编号和选手信息，可放在容器 map<int,Speak> mes;
4. 所有选手编号名单，可以放在容器 vector<int> v;
5. 第一轮晋级名单，可以放在容器 vector<int> v1;
6. 第二轮晋级名单，可以放在容器 vector<int> v2;
7. 第三轮晋级名单，可以放在容器 vector<int> v3;
8. 每个小组的比赛得分信息，按照从小到大的顺序放在 multimap<score,编号，greater<int> >中；
9. 每个选手的得分，可以放在容器 deque<int> dscore；方便去除最低最高分

10.2.2.实现代码

```
#include <iostream>
#include <algorithm>
#include <map>
#include <string>
#include <vector>
#include <deque>
#include <numeric>
#include <functional>
#include <ctime>

using namespace std;

//定义选手
class Speaker
{
public:

private:
    string name;
    int score[3];//分别存放 3 轮比赛的成绩
};

//生成所有演讲选手，并初始化，map<int,Speaker>(<编号，选手>)
```

```
void createSpaker(map<int,Speaker> &mes,vector<int> &v)
{

}

//选手抽签
void speechContestDraw(vector<int> &v)
{

}

//进行比赛
//int seq:第几轮比赛
//vector<int> &src;当前比赛编号名单
//vector<int> &dst;晋级比赛编号名单
//map<int,Speaker>&mes;选手信息
int speechContest(int seq,vector<int> &src,vector<int>
&dst,map<int,Speaker>&mes)
{

}

//打印比赛成绩
//int seq:第几轮比赛
//vector<int> &v;当前比赛胜利者编号名单
//map<int,Speaker>&mes;选手信息
int speechContestPrint(int seq,vector<int> &v,map<int,Speaker>&mes)
{

}

int main()
{
    srand(time(0));

    //设计容器
    map<int,Speaker> mes;//选手信息
    vector<int> v;//所有选手编号名单
    vector<int> v1;//第一轮比赛晋级编号名单
    vector<int> v2;//第二轮比赛晋级编号名单
    vector<int> v3;//第三轮比赛晋级编号名单

    //产生选手
    createSpaker(mes,v);
```

```
    //第一轮
    speechContestDraw(v);
    speechContest(1,v,v1,mapSpeaker);
    speechContestPrint(1,v1,mapSpeaker);

    //第二轮
    speechContestDraw(v1);
    speechContest(2,v1,v2,mapSpeaker);
    speechContestPrint(2,v2,mapSpeaker);

    //第三轮
    speechContestDraw(v2);
    speechContest(3,v2,v3,mapSpeaker);
    speechContestPrint(3,v3,mapSpeaker);

    return 0;

}
```