

第十天课程笔记

一、回顾知识点

1.1 string容器

string容器，即为字符串，之前的表示 `const char *` 或 `const char []`。

```
// 构造函数
string(); //创建一个空的字符串 例如: string str;
string(const string& str); //使用一个 string 对象初始化另一个 string 对象
string(const char* s); //使用字符串 s 初始化
string(int n, char c); //使用 n 个字符 c 初始化

// 赋值
string& operator=(const char* s); //char*类型字符串 赋值给当前的字符串
string& operator=(const string &s); //把字符串 s 赋给当前的字符串
string& operator=(char c); //字符赋值给当前的字符串
string& assign(const char *s); //把字符串 s 赋给当前的字符串
string& assign(const char *s, int n); //把字符串 s 的前 n 个字符赋给当前的字符串
string& assign(const string &s); //把字符串 s 赋给当前字符串
string& assign(int n, char c); //用 n 个字符 c 赋给当前字符串
string& assign(const string &s, int start, int n); //将 s 从 start 开始 n 个字符赋值给字符串

// 取值
char& operator[](int n); // 通过[]方式取字符
char& at(int n); //通过 at 方法获取字符

// 字符串拼接
string& operator+=(const string& str); //重载+=操作符
string& operator+=(const char* str); //重载+=操作符
string& operator+=(const char c); //重载+=操作符
string& append(const char *s); //把字符串 s 连接到当前字符串结尾
string& append(const char *s, int n); //把字符串 s 的前 n 个字符连接到当前字符串结尾
string& append(const string &s); //同 operator+=( )
string& append(const string &s, int pos, int n); //把字符串 s 中从 pos 的 n 个字符连接到当前字符串结尾
string& append(int n, char c); //在当前字符串结尾添加 n 个字符 c

// 查找与替换
int find(const string& str, int pos = 0) const; //查找 str 第一次出现位置, 从 pos 开始查找, 如果未查找到返回 -1
int find(const char* s, int pos = 0) const; //查找 s 第一次出现位置, 从 pos 开始查找
int find(const char* s, int pos, int n) const; //从 pos 位置查找 s 的前 n 个字符第一次位置
int find(const char c, int pos = 0) const; //查找字符 c 第一次出现位置
int rfind(const string& str, int pos = npos) const; //查找 str 最后一次位置, 从 pos 开始查找
int rfind(const char* s, int pos = npos) const; //查找 s 最后一次出现位置, 从 pos 开始查找
```

```

int rfind(const char* s, int pos, int n) const; //从 pos 查找 s 的前 n 个字符最后一次位置
int rfind(const char c, int pos = 0) const; //查找字符 c 最后一次出现位置
string& replace(int pos, int n, const string& str); //替换从 pos 开始 n 个字符为字符串 str
string& replace(int pos, int n, const char* s); //替换从 pos 开始的 n 个字符为字符串 s

// 比较，返回值 0:相等， 1:大于s， -1: 小于s
int compare(const string &s) const; //与字符串 s 比较
int compare(const char *s) const; //与字符串 s

// 截取子字符串
string substr(int pos = 0, int n = npos) const; //返回由 pos 开始的 n 个字符组成的字符串

// 插入与删除
string& insert(int pos, const char* s); //插入字符串
string& insert(int pos, const string& str); //插入字符串
string& insert(int pos, int n, char c); //在指定位置插入 n 个字符 c
string& erase(int pos, int n = npos); //删除从pos 开始的 n 个

// 转成 char *
const char * c_str() 将当前字符串转成 char *

// 获取字符串的长度
int size();

```

string容器也支持迭代器： string::iterator

1.2 vector容器

vector 维护一个线性空间(线性连续空间), vector 迭代器所需要的操作行为 是(*,->, ++, --, +, -, +=, -=)运算符重载等。vector 支持随机存取, vector 提供的是随机访问迭代器(Random Access Iterator), 迭代器中的元素即为vector容器中的元素。

vector具有自动扩容的特性，具有容器的容量(capacity)和大小(size)的两个属性。

```

// 构造函数
vector<T> v;
vector(v.begin(), v.end());
vector(n, T elem);
vector(const vector &vec);

// 赋值
assign(beg, end); //将[beg, end)区间中的数据拷贝赋值给本身。
assign(n, elem); //将 n 个 elem 拷贝赋值给本身。
vector& operator=(const vector &vec); //重载等号操作符
swap(vec); // 将 vec 与本身的元素互换

// 大小
int size();
bool empty();
void resize(int num);
void resize(int num, elem);
int capacity();
void reserve(int len);

```

```

// 存取
T &at(int idx);
T &operator[](int idx);
T front();
T back(); // end() == &(back()) + 1

// 插入
void insert(iterator pos, int count, T ele);
void push_back(ele); //尾部插入元素 ele
void pop_back(); //删除最后一个元素
iterator erase(iterator start, iterator end); // [start, end)
iterator erase(iterator pos);
void clear(); //删除容器中所有元素

```

resize()+swap()实现vector收缩内存空间:

```

resize(n);
vector<T>(v).swap(v);

```

1.3 deque容器

概述:

vector 容器是单向开口的连续内存空间，**deque** 则是一种双向开口的连续线性空间。
deque 没有空间保留(**reserve**)功能
deque 容器是连续的空间，是由一段一段的定量的连续空间构成。

工作原理:

deque 最大的工作就是维护分段连续的内存空间的整体性的假象，并提供随机存取的接口。
deque 通过中央控制（**map**实现的），维持整体连续的假象，数据结构的设计及迭代器的前进、后退操作颇为繁琐。

中央控制：连续小空间，由**map**实现，存放是地址，地址指向的另一个连续空间为缓冲区。
 缓冲区：是 **deque** 的存储空间的主题。

API:

```

// 构造函数
deque<T> deqT;
deque(beg, end);
deque(n, T elem);
deque(const deque &deq);

// 赋值
assign(beg, end);
assign(n, T elem);
deque& operator=(const deque &deq);
swap(deq);

// 大小
size();

```

```

empty();
resize(num);
resize(num, elem);

// 双端插入和删除
void push_back(elem);
void push_front(elem);
void pop_back();
void pop_front();

// 读取
at(idx);
operator[];
front();
back();

// 插入
void insert(iterator pos,T elem);
void insert(iterator pos,int n,T elem);
void insert(iterator pos,iter_beg,iter_end);

// 删除
clear();//移除容器的所有数据
iterator erase(iterator beg,iterator end);
iterator erase(iterator pos);

```

1.4 stack容器

stack 是一种先进后出(First In Last Out,FIFO)的数据结构，只有一个出口，允许新增元素、移除元素、读取栈顶元素。Stack 不提供遍历功能，也不提供迭代器。

API:

```

stack<T> stkT;
stack(const stack &stk);

stack& operator=(const stack &stk);
void push(elem);
void pop();
T top();

empty();
size();

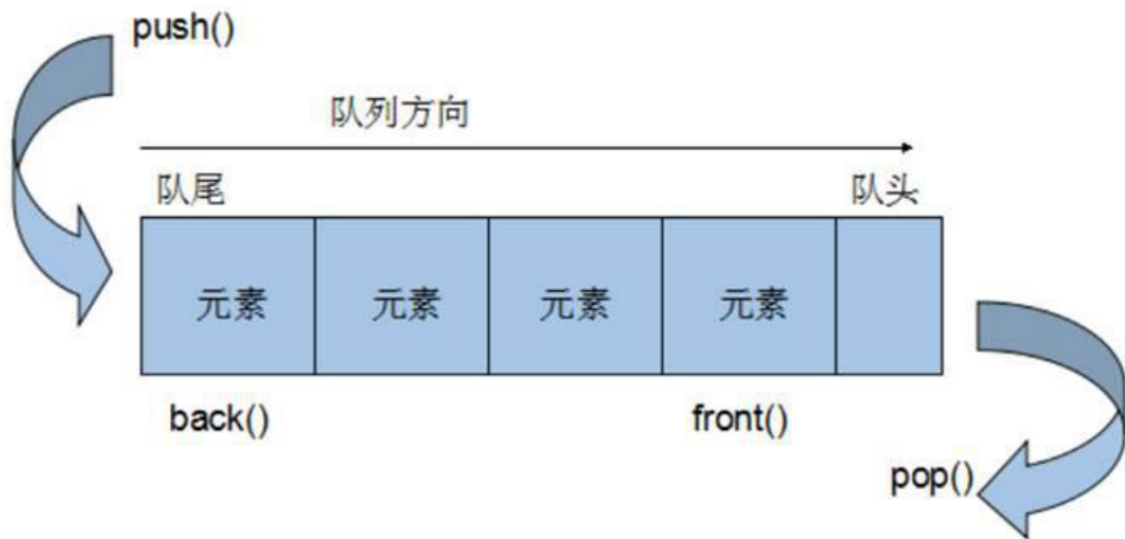
```

二、STL容器II

2.1 queue 容器

2.1.1 queue概念

Queue 是一种先进先出(First In First Out,FIFO)的数据结构, 它有两个出口, queue 容器允许从一端新增元素, 从另一端移除元素。



2.1.2 queue 没有迭代器

Queue 所有元素的进出都必须符合“先进先出”的条件, 只有 queue 的顶端元素, 才有机会被外界取用。Queue 不提供遍历功能, 也不提供迭代器。

2.1.3 常用API

2.1.3.1 构造函数

```
queue<T> queT; //queue 采用模板类实现, queue 对象的默认构造形式:  
queue(const queue &que); //拷贝构造函数
```

2.1.3.2 存取、插入和删除

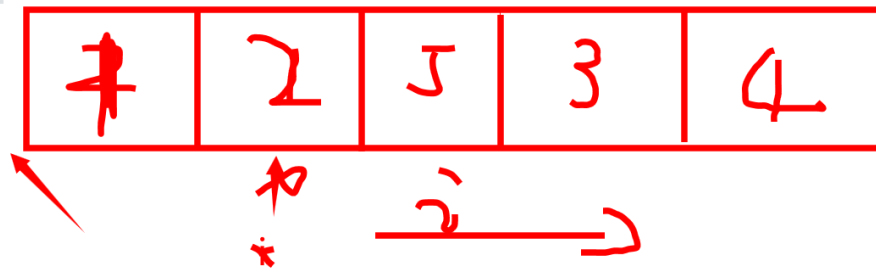
```
void push(elem); //往队尾添加元素  
void pop(); //从队头移除第一个元素  
T back(); //返回最后一个元素  
T front(); //返回第一个元素
```

2.1.3.3 赋值与大小

```
queue& operator=(const queue &que); //重载等号操作符  
  
empty(); //判断队列是否为空  
size(); //返回队列的大小
```

如: 作业第七道

插入排序算法



i位置之前的数都是有序的

从i的前面找第一个比i大的数的位置，并插入到此位置上
删除当前i位置的数

```
void sort(deque<int> &d)
{
    vector<int> v(&d.front(), &d.back() + 1);
    d.clear();
    d.push_front(v.front()); // 插入第一个元素

    vector<int>::iterator it = v.begin() + 1;
    while (it != v.end()) // 待排序的数
    {
        deque<int>::iterator dit = d.begin(); // 有序队列的第一个元素位置
        int i = 0; // 从头开始查找第一个比当前位置大的数
        for (; i < d.size(); i++)
        {
            if (d.at(i) > *it)
            {
                break;
            }
        }
        d.insert(dit + i, *it);
        it++;
    }
}
```



deque



vector

如: queue的应用

```
#include <iostream>
#include <deque>
#include <queue>

using namespace std;

int main(int argc, char const *argv[])
{
    // 1. 创建queue
```

```

queue<int> q1;
q1.push(1);
q1.push(2);
q1.push(3);

// 显示并弹出所有的元素
while (!q1.empty())
{
    cout << q1.front() << " ";
    q1.pop();
}
cout << endl;
return 0;
}

```

```

disen@qfxxa:~/code2/day10$ g++ demo1.cpp
./a.out
disen@qfxxa:~/code2/day10$ ./a.out
1 2 3

```

2.2 list 容器

2.2.1 list 概念

list(链表)是一种物理存储单元上非连续、非顺序的存储结构，数据元素的逻辑顺序是通过链表中的指针链接次序实现的。

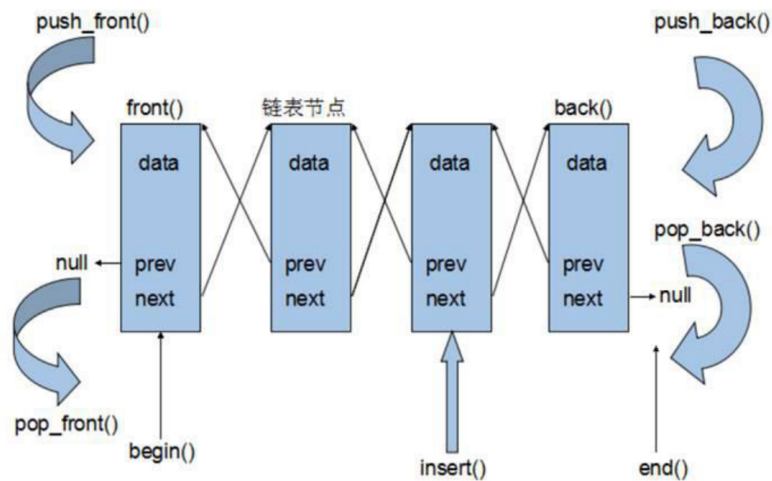
链表由一系列结点（链表中每一个元素称为结点）组成，结点可以在运行时动态生成。

每个结点包括两个部分：

- 1) 存储数据元素的数据域，
- 2) 存储下一个结点地址的指针域

list与vector的比较：

- 1) 相对于vector 的连续线性空间，list 就显得负责许多，每次插入或者删除一个元素，就是配置或者释放一个元素的空间。不浪费多余的空间，且插入与移除元素的操作是常数时间（稳定）。
- 2) list和vector 是两个最常被使用的容器，但list是由双向链表实现的。
- 3) list插入操作和删除操作都不会造成原有 list 迭代器的失效。 【重要特性】



list特点:

- 1) 采用动态存储分配, 不会造成内存浪费和溢出
- 2) 链表执行插入和删除操作十分方便, 修改指针即可, 不需要移动大量元素
- 3) 链表灵活, 但是空间和时间额外耗费较大

2.2.2 list 的迭代器

List 不能像 vector 一样以普通指针作为迭代器, 因为其节点不能保证在同一块连续的内存空间上。List 迭代器必须有能力指向 list 的节点, 并有能力进行正确的递增、递减、取值、成员存取操作。**递增**时指向下一个节点, **递减**时指向上一个节点, **取值**时取的是节点的数据值, **成员取用**时取的是节点的成员。

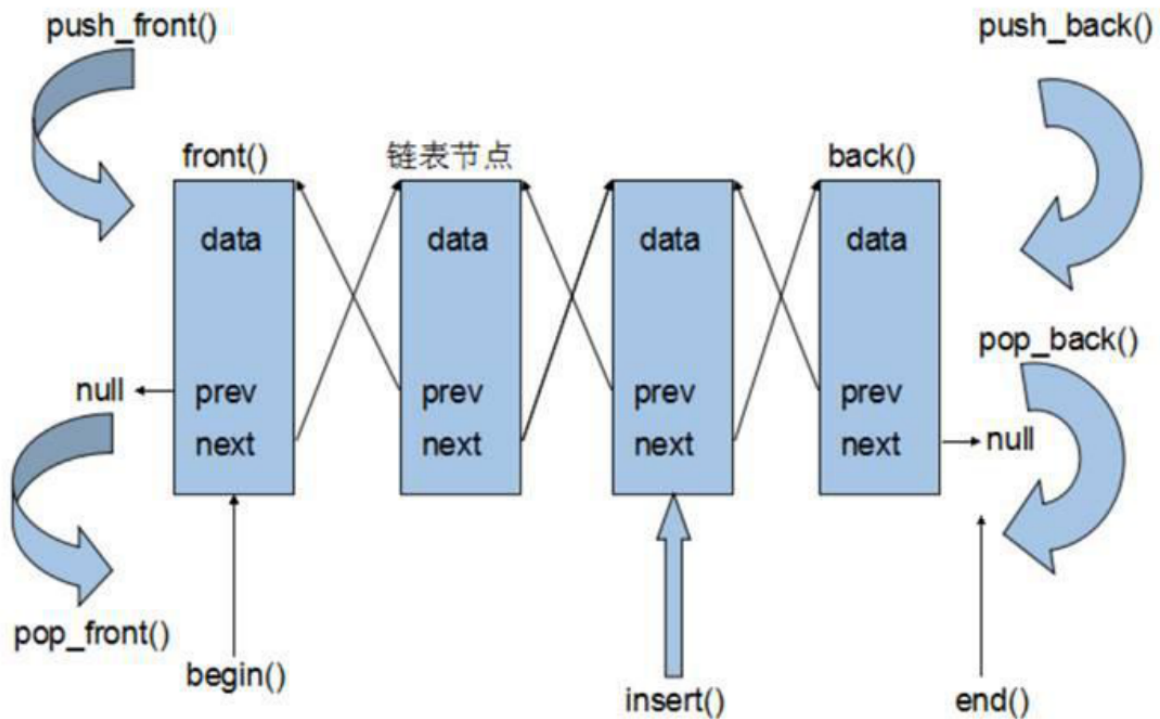
另外, list 是一个双向链表, 迭代器必须能够具备前移、后移的能力, 所以 list 容器提供的是 Bidirectional Iterators. (双向的迭代器)。

List 有一个重要的性质, 插入操作和删除操作都不会造成原有 list 迭代器的失效。

【注意】list的迭代器, 不支持 `+n` 操作。

2.2.3 list 数据结构

list 容器不仅是一个双向链表, 而且还是一个**循环的双向链表**。



2.2.4 常用API

2.2.4.1 构造函数

```
list<T> lst; //list 采用模板类实现, 对象的默认构造形式:
list(beg, end); //构造函数将 [beg, end) 区间中的元素拷贝给本身。
list(n, elem); //构造函数将 n 个 elem 拷贝给本身。
list(const list &lst); //拷贝构造函数
```

如:

```
#include <iostream>
#include <list>
#include <algorithm>

using namespace std;

template <typename T>
void print(list<T> &l)
{
    typename list<T>::iterator it = l.begin();
    for (; it != l.end(); it++)
    {
        cout << *it << " ";
    }
    cout << endl;
}

int main(int argc, char const *argv[])
{
    list<int> list1(6, 5);
    print(list1);

    string s1 = "abcdefg";
```

```

// 将string字符串转成list
list<char> list2(s1.begin(), s1.end());
print(list2);

return 0;
}

```

```

disen@qfxa:~/code2/day10$ ./a.out
5 5 5 5 5 5
a b c d e f g

```

2.2.4.2 插入和删除

```

push_back(elem); //在容器尾部加入一个元素
pop_back(); //删除容器中最后一个元素
push_front(elem); //在容器开头插入一个元素
pop_front(); //从容器开头移除第一个元素
iterator insert(pos, elem); //在 pos 位置插 elem 元素的拷贝，返回新数据的位置。
void insert(pos, n, elem); //在 pos 位置插入 n 个 elem 数据，无返回值。
void insert(pos, beg, end); //在 pos 位置插入 [beg, end) 区间的数据，无返回值。
clear(); //移除容器的所有数据

iterator erase(beg, end); //删除 [beg, end) 区间的数据，返回下一个数据的位置。
iterator erase(pos); //删除 pos 位置的数据，返回下一个数据的位置。
remove(elem); //删除容器中所有与 elem 值匹配的元素

```

如:

```

#include <iostream>
#include <list>
#include <algorithm>

using namespace std;

template <typename T>
void print(list<T> &l)
{
    typename list<T>::iterator it = l.begin();
    for (; it != l.end(); it++)
    {
        cout << *it << " ";
    }
    cout << endl;
}

int main(int argc, char const *argv[])
{
    string s = "abcdef";
    list<char> l(s.begin(), s.end());
    int size = l.size(); // 获取大小

    list<char>::iterator it = l.begin();
    for (int i = 0; i < size; i++)
    {

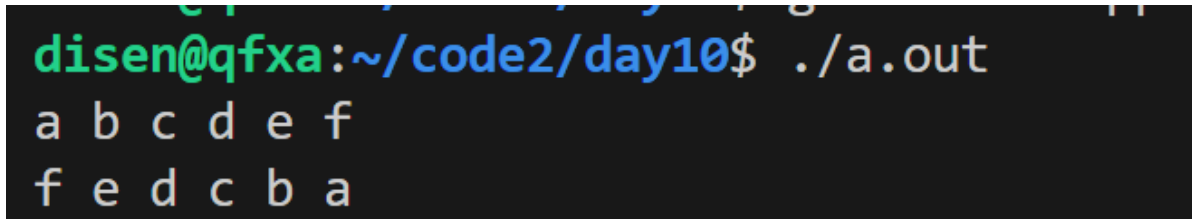
```

```

        it = l.insert(it, l.back());
        it++; // list迭代支持++, --, 不支持 +n
        l.pop_back();
    }

    print(l);
    return 0;
}

```



```

disen@qfxa:~/code2/day10$ ./a.out
a b c d e f
f e d c b a

```

如2：删除第3到5个字符

```

#include <iostream>
#include <list>
#include <algorithm>

using namespace std;

template <typename T>
void print(list<T> &l)
{
    typename list<T>::iterator it = l.begin();
    for (; it != l.end(); it++)
    {
        cout << *it << " ";
    }
    cout << endl;
}

int main(int argc, char const *argv[])
{
    string s = "abcdefh";
    list<char> l(s.begin(), s.end());
    print(l);
    // 删除cde
    list<char>::iterator it = l.begin(); // 删除节点的开始位置
    // l.erase(it + 3, it + 7); //error
    for (int i = 0; i < 2; i++)
        it++;
    list<char>::iterator it2 = it; // 删除节点的结束位置
    for (int i = 0; i < 3; i++)
        it2++;
    l.erase(it, it2); // [it, it2) 区间的所有节点

    print(l);
    return 0;
}

```

```
disen@qfxa:~/code2/day10$ ./a.out
a b c d e f h
a b f h
```

2.2.4.3 大小

```
size(); //返回容器中元素的个数
empty(); //判断容器是否为空
resize(num); //重新指定容器的长度为 num，若容器变长，则以默认值填充新位置。如果容器变短，则末尾超出容器长度的元素被删除
resize(num, elem);
```

2.2.4.4 赋值

```
assign(beg, end); //将[beg, end)区间中的数据拷贝赋值给本身。
assign(n, elem); //将 n 个 elem 拷贝赋值给本身。
list& operator=(const list &lst); //重载等号操作符
swap(lst); //将 lst 与本身的元素互换。
```

2.2.4.5 读取

```
front(); //返回第一个元素。
back(); //返回最后一个元素
```

2.2.4.6 反转和排序

```
reverse(); //反转链表
sort(); //list 排序
```

如:

```
#include <iostream>
#include <list>
#include <algorithm>

using namespace std;

template <typename T>
void print(list<T> &l)
{
    typename list<T>::iterator it = l.begin();
    for (; it != l.end(); it++)
    {
        cout << *it << " ";
    }
    cout << endl;
}

int main(int argc, char const *argv[])
{
    string s = "abcdefh";
    list<char> l(s.begin(), s.end());
```

```

print(l);
l.reverse();
print(l);

int m[] = {6, 4, 2, 8, 3, 1};
list<int> l2(m, m + 6);
print(l2);
l2.sort();    // 从小到大
l2.reverse(); // 从大到小
print(l2);
return 0;
}

```

```

disen@qfxa:~/code2/day10$ ./a.out
a b c d e f h
h f e d c b a
6 4 2 8 3 1
8 6 4 3 2 1

```

2.3 set/multiset 容器

2.3.1 set概念

set (集合)的特性是所有元素都会根据元素的键值自动被排序。

set 的元素即是键值 (key) 又是实值(value)，不允许两个元素有相同的键值。

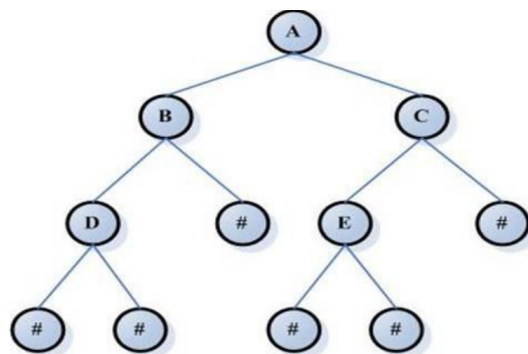
set 的 iterator 是一种 const_iterator，不允许修改set的键值。

set 拥有和 list 某些相同的性质，当对容器中的元素进行插入操作或者删除操作的时候，操作之前的迭代器，在操作完成之后依然有效，被删除的那个元素的迭代器必然是一个例外。

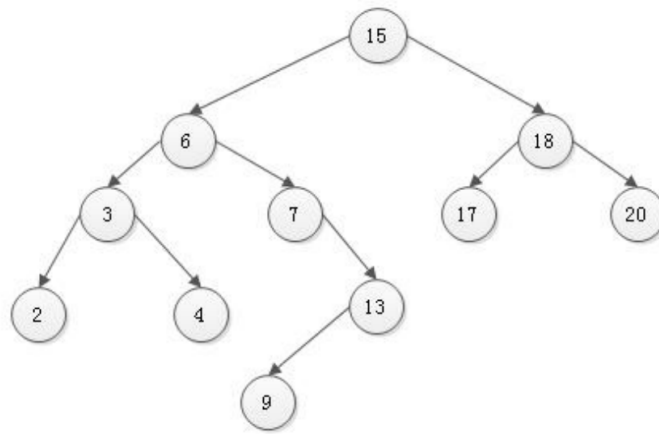
2.3.2 set数据结构

multiset 特性及用法和 set 完全相同，唯一的差别在于它允许键值重复。set 和multiset 的底层实现是红黑树，红黑树为平衡二叉树的一种。

二叉树就是任何节点最多只允许有两个子节点。分别是左子结点和右子节点：



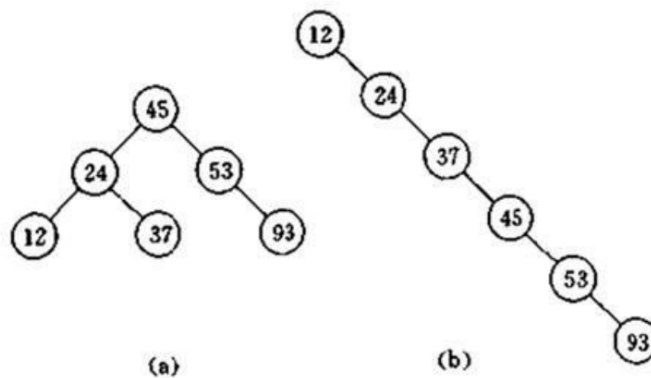
二叉搜索树，是指二叉树中的节点按照一定的规则进行排序，使得对二叉树中元素访问更加高效：



二叉搜索树的放置规则是：

任何节点的元素值一定大于其左子树中的每一个节点的元素值，并且小于其右子树的值。因此从根节点一直向左走，一直到无路可走，即得到最小值，一直向右走，直至无路可走，可得到最大值。那么在二叉搜索树中找到最大元素和最小元素是非常简单的事情。

如上图所示：那么当一个二叉搜索树的左子树和右子树不平衡的时候，那么搜索依据上图表示，搜索 9 所花费的时间要比搜索 17 所花费的时间要多，由于我们的输入或者经过我们插入或者删除操作，二叉树失去平衡，造成搜索效率降低。



2.3.3 常用API

2.3.3.1 构造函数

```

set<T> st; // set 默认构造函数:
multiset<T> mst; // multiset 默认构造函数:
set(const set &st); // 拷贝构造函数
set(begin, end); // 复制 [begin, end) 区间的数据到当前的集合中
  
```

2.3.3.2 赋值和大小

```

set& operator=(const set &st); // 重载等号操作符
swap(st); // 交换两个集合容器
size(); // 返回容器中元素的数目
empty(); // 判断容器是否为空
  
```

2.3.3.3 插入和删除

```
insert(elem);    //在容器中插入元素。
clear();         //清除所有元素
erase(pos);      //删除 pos 迭代器所指的元素，返回下一个元素的迭代器。
erase(beg, end); //删除区间[beg,end)的所有元素，返回下一个元素的迭代器。
erase(elem);     //删除容器中值为 elem 的元素。
```

如:

```
#include <iostream>
#include <set>

using namespace std;

void print(set<int> &s)
{
    set<int>::const_iterator it = s.begin();
    for (; it != s.end(); it++)
    {
        cout << *it << " ";
    }
    cout << endl;
}

void print(multiset<int> &s)
{
    multiset<int>::const_iterator it = s.begin();
    for (; it != s.end(); it++)
    {
        cout << *it << " ";
    }
    cout << endl;
}

int main(int argc, char const *argv[])
{
    int m[] = {1, 2, 3, 2, 3, 4};
    set<int> s1(m, m + 6);
    print(s1); // set的键值自动排序，且不重复

    multiset<int> s2(m, m + 6);
    print(s2); // multiset和set类同，但键值可以重复
    return 0;
}
```

```
disen@qfxa:~/code2/day10$ ./a.out
```

```
1 2 3 4
```

```
1 2 2 3 3 4
```

2.3.3.4 查找

```
find(key);    //查找键 key 是否存在,若存在,返回该键的元素的迭代器;若不存在,返回 set.end();
count(key);   //查找键 key 的元素个数, 针对multiset
lower_bound(keyElem); //返回第一个 key>=keyElem 元素的迭代器。
upper_bound(keyElem); //返回第一个 key>keyElem 元素的迭代器。
equal_range(keyElem); //返回容器中 key 与 keyElem 相等的上下限的两个迭代器。
```

如:

```
#include <iostream>
#include <set>

using namespace std;

void print(set<int> &s)
{
    set<int>::const_iterator it = s.begin();
    for (; it != s.end(); it++)
    {
        cout << *it << " ";
    }
    cout << endl;
}

void print(multiset<int> &s)
{
    multiset<int>::const_iterator it = s.begin();
    for (; it != s.end(); it++)
    {
        cout << *it << " ";
    }
    cout << endl;
}

int main(int argc, char const *argv[])
{
    int m[] = {5, 12, 16, 21, 22, 27, 29, 30};
    set<int> s1(m, m + 8);
    print(s1); // set的键值自动排序, 且不重复

    // 查看大于等于22值的所有元素
    set<int>::const_iterator it = s1.lower_bound(22);
    cout << "--大于等于22值的所有元素--" << endl;
    for (; it != s1.end(); it++)
        cout << *it << "\t";
    cout << endl;
    // 查看小于22值的所有元素
    it = s1.lower_bound(22);
    set<int>::const_iterator it0 = s1.begin();
    cout << "--小于22值的所有元素--" << endl;
    while (it0 != it)
    {
        cout << *it0 << "\t";
        it0++;
    }
}
```



```

    cout << endl;
    return 0;
}

```

```

disen@qfxa:~/code2/day10$ ./a.out
5 12 16 21 22 27 29 30
--大于等于22值的所有元素--
22      27      29      30
--小于22值的所有元素--
5       12      16      21

```

2.3.3.5 set 排序规则

set自动排序，但可以改变它的排序规则，默认从小到大。

自定义排序规则，可以使用struct或class, 声明 `bool operator()(v1, v2)` 仿函数重载。

在定义集合时，指定排序规则（类）：

```
set<数据元素的泛型, 排序规则类型> s;
```

如1：基本数据类型的排序规则

```

#include <iostream>
#include <set>
#include <algorithm>

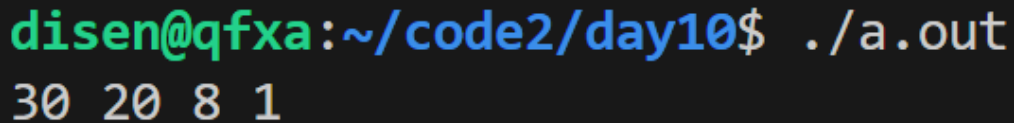
using namespace std;

class MyIntSort
{
public:
    bool operator()(const int &v1, const int &v2)
    {
        return v1 > v2; // 从大到小排序
    }
};

int main(int argc, char const *argv[])
{
    set<int, MyIntSort> s1;
    s1.insert(20);
    s1.insert(30);
    s1.insert(1);
    s1.insert(8);
    set<int, MyIntSort>::const_iterator it = s1.begin();
    for (; it != s1.end(); it++)
    {
        cout << *it << " ";
    }
    cout << endl;
}

```

```
    return 0;
}
```



```
disen@qfxa:~/code2/day10$ ./a.out
30 20 8 1
```

如2: set的键值为类对象, 类对象必须提供排序规则

```
#include <iostream>
#include <set>
#include <algorithm>

using namespace std;

class Student
{
public:
    string name;
    int age;
    float score;
    Student(const string &name, int age, float score)
    {
        this->name = name;
        this->age = age;
        this->score = score;
    }
};

class MyStudentByAgeSort
{
public:
    bool operator()(const Student &s1, const Student &s2)
    {
        return s1.age > s2.age;
    }
};

class MyStudentByScoreSort
{
public:
    bool operator()(const Student &s1, const Student &s2)
    {
        return s1.score > s2.score;
    }
};

int main(int argc, char const *argv[])
{
    Student *a = new Student[5]{
        Student("disen", 21, 720),
        Student("lucy", 20, 680),
        Student("mack", 19, 718),
        Student("judy", 22, 595),
        Student("rose", 20, 490)};
    set<Student, MyStudentByScoreSort> s1(a, a + 5);
}
```

```

set<Student, MyStudentByScoreSort>::const_iterator it = s1.begin();
cout << "name\tage\tscore" << endl;
for (; it != s1.end(); it++)
{
    cout << (*it).name << "\t" << (*it).age << "\t" << (*it).score << endl;
}
return 0;
}

```

```

disen@qfxa:~/code2/day10$ g++ demo6.cpp -std=c++11
./disen@qfxa:~/code2/day10$ ./a.out
name    age    score
disen   21     720
mack    19     718
lucy    20     680
judy    22     595
rose    20     490

```

2.3.4 对组(pair)

对组(pair)将一对值组合成一个值，这一对值可以具有不同的数据类型，两个值可以分别用 pair 的两个公有属性 first 和 second 访问。

类模板: `template <class T1, class T2> class pair`

用法一:

```

pair<string, int> pair1(string("name"), 20);
cout << pair1.first << endl;
cout << pair1.second << endl;

```

用法二:

```

pair<string, int> pair2 = make_pair("name", 30);
cout << pair2.first << endl;
cout << pair2.second << endl;

```

用法三:

```

pair<string, int> pair3 = pair2; // 拷贝构造函数
cout << pair3.first << endl;
cout << pair3.second << endl;

```

如:

```

#include <iostream>
#include <set>
#include <algorithm>

using namespace std;

```

```
int main(int argc, char const *argv[])
{
    pair<int, string> p1(1, "disen");
    pair<int, string> p2 = make_pair(2, "lucy");
    cout << "id=" << p1.first << ",name=" << p1.second << endl;
    cout << "id=" << p2.first << ",name=" << p2.second << endl;
    return 0;
}
```

```
disen@qfxa:~/code2/day10$ ./a.out
id=1,name=disen
id=2,name=lucy
```

2.4 map/multimap 容器

2.4.1 map概念

map 的特性是所有元素都会根据元素的键值自动排序。

map 所有的元素都是pair,同时拥有实值和键值, pair 的第一元素被视为键值, 第二元素被视为实值, map 不允许两个元素有相同的键值。【键值是唯一的】

multimap 和 map 的操作类似, 唯一区别 multimap 键值可重复。

map 和 multimap 都是以红黑树为底层实现机制。

【注意】map迭代器的元素是pair对组, 通过pair对象获取键值(first)和实值(second)。

2.4.2 常用API

2.4.2.1 构造函数

```
map<T1, T2> mapTT; //map 默认构造函数:
map(const map &mp); //拷贝构造函数
map(begin, end); // 复制[begin, end)区间的pair对到当前map中。
```

2.4.2.2 赋值和大小

```
map& operator=(const map &mp); //重载等号操作符
swap(mp); //交换两个集合

size(); //返回容器中元素的数目
empty(); //判断容器是否为空
```

2.4.2.3 插入

```
insert(pair<...>(...)); //往容器插入元素, 返回 pair<iterator,bool>
insert(make_pair(...))
insert(map<T1, T2>::value_type(...));

T2& operator[](T1 &key);
```

如1: 创建一个map对象, 键为int, 值为string, 存储学生的学号和姓名

```

#include <iostream>
#include <map>

using namespace std;

int main(int argc, char const *argv[])
{
    map<int, string> ms;
    ms.insert(pair<int, string>(1, "小李子1"));
    ms.insert(map<int, string>::value_type(4, "小崔"));
    ms.insert(make_pair(3, "小汪同学"));
    ms.insert(pair<int, string>(2, "小李子2"));
    // 8是map的键值
    ms[8] = "小郑同学";

    map<int, string>::iterator it = ms.begin();
    for (; it != ms.end(); it++)
    {
        cout << "sid=" << (*it).first << ",name=" << (*it).second << endl;
    }
    return 0;
}

```

如2：昨天作业第8题

输入括号，验证括号的有效性

```

#include <iostream>
#include <map>
#include <stack>

using namespace std;

string lefts = "{[(";
map<char, char> map1;
bool valid(const char *p)
{
    stack<char> st;
    int maxdepth = 0;
    while (*p)
    {
        const char &cp = *p;
        if (lefts.find(cp) != -1)
        {
            // 入栈
            st.push(*p);
        }
        else
        {
            // 弹栈
            if (st.empty())
                return false;
            const char &top = st.top();
            if (map1[cp] != top)
                return false;

            if (maxdepth < st.size())

```

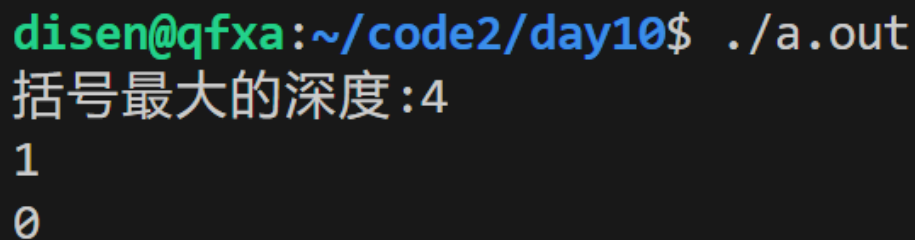
```

        maxdepth = st.size();
        st.pop();
    }
    p++;
}
cout << "括号最大的深度:" << maxdepth << endl;
return true;
}

int main(int argc, char const *argv[])
{
    map1.insert(make_pair('}', '{'));
    map1.insert(make_pair(')', '('));
    map1.insert(make_pair(']', '['));

    cout << valid("{([])}O[(((O))]") << endl;
    cout << valid("{([])}") << endl;
    return 0;
}

```



```

disen@qfxa:~/code2/day10$ ./a.out
括号最大的深度:4
1
0

```

2.4.2.4 删除

```

clear();    //删除所有元素
erase(pos); //删除 pos 迭代器所指的元素，返回下一个元素的迭代器。
erase(beg,end); //删除区间[beg,end)的所有元素，返回下一个元素的迭代器。
erase(keyElem); //删除容器中 key 为 keyElem 的对

```

如:

```

#include <iostream>
#include <map>
#include <algorithm>

using namespace std;

int main(int argc, char const *argv[])
{
    map<int, string> m;
    m.insert(make_pair(1, "disen"));
    m.insert(make_pair(2, "lucy"));
    cout << m.size() << endl;
    m.erase(1); // 1是键值
    cout << m.size() << endl;

    return 0;
}

```

```
disen@qfxa:~/code2/day10$ ./a.out
2
1
```

2.4.2.5 查找

`find(key)`; //查找键 `key` 是否存在,若存在,返回该键的元素的迭代器;若不存在,返回 `map.end()`;
`count(keyElem)`; //返回容器中 `key` 为 `keyElem` 的对组个数。对 `map` 来说,要么是 0,要么是 1。
对 `multimap` 来说,值可能大于 1。
`lower_bound(keyElem)`; //返回第一个 `key>=keyElem` 元素的迭代器。
`upper_bound(keyElem)`; //返回第一个 `key>keyElem` 元素的迭代器。
`equal_range(keyElem)`; //返回容器中 `key` 与 `keyElem` 相等的上下限的两个迭代

如:

```
#include <iostream>
#include <map>
#include <algorithm>

using namespace std;

int main(int argc, char const *argv[])
{
    map<int, string> m;
    m.insert(make_pair(1, "disen"));
    m.insert(make_pair(2, "lucy"));
    m.insert(make_pair(3, "jack"));

    map<int, string>::const_iterator it = m.find(1);
    if (it == m.end())
    {
        cout << "未查找到" << endl;
    }
    else
    {
        const pair<int, string> &p = *it;
        cout << p.first << "," << p.second << endl;
    }

    return 0;
}
```

```
disen@qfxa:~/code2/day10$ ./a.out
1,disen
```

2.5 STL 容器使用时机

2.5.1 结构与操作比较

	vector	deque	list	set	multiset	map	multimap
典型内存结构	单端数组	双端数组	双向链表	二叉树	二叉树	二叉树	二叉树
可随机存取	是	是	否	否	否	对 key 而言：不是	否

	vector	deque	list	set	multiset	map	multimap
元素搜寻速度	慢	慢	非常慢	快	快	对 key 而言：快	对 key 而言：快
元素安插移除	尾端	头尾两端	任何位置	-	-	-	-

2.5.2 vector 的使用场景

如软件历史操作记录的存储，我们经常要查看历史记录，比如上一次的记录，上上次的记录，但却不会去删除记录，因为记录是事实的描述。

2.5.3 deque 的使用场景

比如排队购票系统，对排队者的存储可以采用 **deque**，支持头端的快速移除，尾端的快速添加。如果采用 **vector**，则头端移除时，会移动大量的数据，速度慢。

2.5.4 vector与deque的比较

- 1) **vector.at()** 比 **deque.at()** 效率高，
如 **vector.at(0)** 是固定的，**deque** 的开始位置却是不固定的
- 2) 如果有大量释放操作的话，**vector** 花的时间更少，这跟二者的内部实现有关
- 3) **deque** 支持头部的快速插入与快速移除，这是 **deque** 的优点

2.5.5 list 的使用场景

比如公交车乘客的存储，随时可能有乘客下车，支持频繁的不确定位置元素的移除插入

2.5.6 set 的使用场景

比如对手机游戏的个人得分记录的存储，存储要求从高分到低分顺序排列

2.5.7 map 的使用场景

比如按 **ID** 号存储十万个用户，想要快速要通过 **ID** 查找对应的用户。二叉树的查找效率，这时就体现出来了。如果是 **vector** 容器，最坏的情况下可能要遍历完整个容器才能找到该用户。

三、STL算法【了解】

3.1 函数对象

重载函数调用操作符的类，其对象常称为函数对象（function object），即它们是行为类似函数的对象，也叫仿函数(functor)，其实就是重载“()”操作符，使得类对象可以像函数那样调用。

【注意】

1. 函数对象(仿函数)是一个类，不是一个函数。
2. 函数对象(仿函数)重载了“()”操作符使得它可以像函数一样调用。

函数对象分类：

一元仿函数（unary_functor）：重载的 operator() 要求获取一个参数
二元仿函数（binary_functor）：重载的 operator() 要求获取两个参数

如：

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

class A{ // A 是一元仿函数
public:
    A(int n): n(n){}
    int n;
    A &operator()(int i){
        this->n += i;
        return *this;
    }
};

int main(){
    A a(10);
    a(20);
    a(30);
    cout << a.n << endl;
}
```

结果：60

函数对象的作用：

STL 提供的算法往往都有两个版本，其中一个版本表现出最常用的某种运算，另一版本则允许用户通过 template 参数的形式来指定所要采取的策略。

对于 template 参数的形式 意思，如 指定set的排序规则

```
set<Student, MyStudentSort> s;
```

如：

```

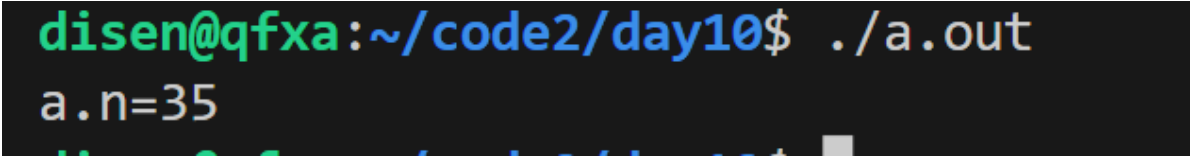
#include <iostream>
using namespace std;

class A
{ // A 是一元仿函数
public:
    A(int n) : n(n) {}
    int n;
    A &operator()(int i)
    {
        this->n += i;
        return *this;
    }
};

// 函数对象作为函数的参数使用
void print(A a, int n) // A a = A(30)
{
    a(n);
    cout << "a.n=" << a.n << endl;
}

int main()
{
    // 打印a对象的值加额外的值的结果
    print(A(30), 5);
    return 0;
}

```



```

disen@qfxa:~/code2/day10$ ./a.out
a.n=35

```

【总结】

- 1、函数对象通常不定义构造函数和析构函数，所以在构造和析构时不会发生任何问题，避免了函数调用的运行时间问题。
- 2、函数对象超出普通函数的概念，函数对象可以有自己的状态
- 3、函数对象可内联编译、性能好，用函数指针几乎不可能
- 4、模版函数对象使函数对象具有通用性，这也是它的优势之一

3.2 谓词

谓词是指普通函数或重载的 `operator()` 返回值是 `bool` 类型的函数对象(即仿函数)，依据函数接收的参数又分为 `一元谓词` 和 `二元谓词` 等，谓词可作为一个判断式。

如：

gt 大于，ge 大于等于，lt 小于，le 小于等于，eq 等于，ne 不等于

```

#include <iostream>
using namespace std;

```

```

class GtFive{ // GtFive 一元谓词
public:
    bool operator()(int n){
        return n > 5;
    }
};

class Gt{ // Gt 二元谓词
public:
    bool operator()(const int &n1, const int& n2){
        return n1 > n2;
    }
};

void print1(GtFive gt, int *p, int size){
    for(int i=0;i<size;i++){
        if(gt(p[i]))
            cout << p[i] << " ";

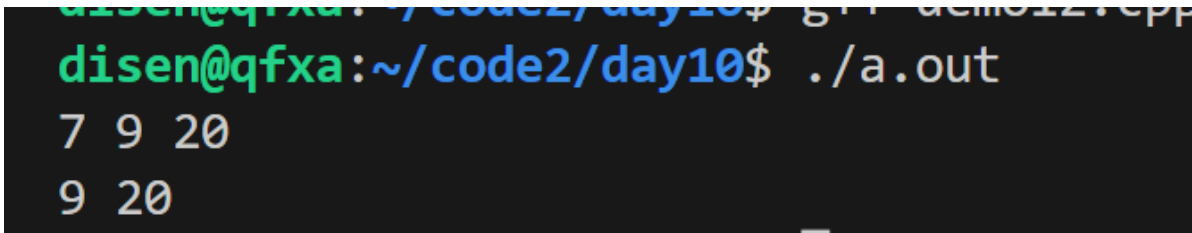
    }
    cout << endl;
}

void print2(int *p, int size, Gt gt, int n){
    for(int i=0;i<size;i++){
        if(gt(p[i], n))
            cout << p[i] << " ";

    }
    cout << endl;
}

int main(){
    int ms[]={1,2, 4, 7, 9, 20};
    print1(GtFive(), ms, 6);
    print2(ms, 6, Gt(), 7);
    return 0;
}

```



```

disen@qfxa:~/code2/day10$ ./a.out
7 9 20
9 20

```

3.3 内建函数对象

引入头文件 `<algorithm>`

STL 内建了一些函数对象，分为算数类函数对象、关系运算类函数对象、逻辑运算类仿函数等。这些仿函数所产生的对象，用法和一般函数完全相同，当然我们还可以产生无名的临时对象来履行函数功能。

3.3.1 算数类函数对象

6 个算数类函数对象,除了 `negate` 是一元运算, 其他都是二元运算。

```
template<class T> T plus<T> //加法仿函数
template<class T> T minus<T> //减法仿函数
template<class T> T multiplies<T> //乘法仿函数
template<class T> T divides<T> //除法仿函数
template<class T> T modulus<T> //取模仿函数
template<class T> T negate<T> //取反仿函数
```

如: 打印set中的元素时, 将元素和指定的数值进行相加并输出

```
#include <iostream>
#include <set>
#include <algorithm>

using namespace std;

void print(set<int>::const_iterator start,
          set<int>::const_iterator end, plus<int> p1, int m)
{
    for (; start != end; start++)
    {
        cout << p1(*start, m) << " ";
    }
    cout << endl;
}

void print(set<int>::const_iterator start,
          set<int>::const_iterator end, multiplies<int> p1, int m)
{
    for (; start != end; start++)
    {
        cout << p1(*start, m) << " ";
    }
    cout << endl;
}

int main(int argc, char const *argv[])
{
    int ms[] = {3, 5, 2, 1, 9, 10, 8, 7};
    set<int> s(ms, ms + 8);
    // for_each(begin, end, callback);
    print(s.begin(), s.end(), plus<int>(), 10);
    // 以2倍的打印集合中的元素值
    print(s.begin(), s.end(), multiplies<int>(), 2);
    return 0;
}
```

```
disen@qfxa:~/code2/day10$ ./a.out
11 12 13 15 17 18 19 20
2 4 6 10 14 16 18 20
```

3.3.2 关系运算类函数对象

6 个关系运算类函数对象,每一种都是二元运算

```
template<class T> bool equal_to<T>//等于
template<class T> bool not_equal_to<T>//不等于
template<class T> bool greater<T>//大于
template<class T> bool greater_equal<T>//大于等于
template<class T> bool less<T>//小于
template<class T> bool less_equal<T>//小于等于
```

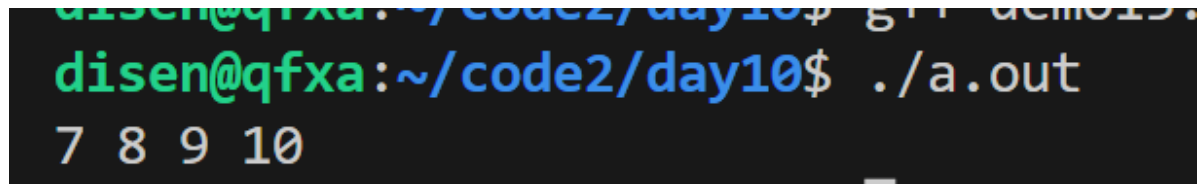
如:

```
#include <iostream>
#include <set>
#include <algorithm>

using namespace std;

void print(set<int>::const_iterator start,
          set<int>::const_iterator end, greater<int> gt, int m)
{
    for (; start != end; start++)
    {
        if (gt(*start, m))
            cout << *start << " ";
    }
    cout << endl;
}

int main(int argc, char const *argv[])
{
    int ms[] = {3, 5, 2, 1, 9, 10, 8, 7};
    set<int> s(ms, ms + 8);
    // 打印大于 5的所有集合元素
    print(s.begin(), s.end(), greater<int>(), 5);
    return 0;
}
```



The image shows a terminal window with the command prompt 'disen@qfxa:~/code2/day10\$./a.out'. The output of the program is '7 8 9 10'.

3.3.3 逻辑运算类运算函数

逻辑运算类运算函数,not 为一元运算, 其余为二元运算

```
template<class T> bool logical_and<T>//逻辑与
template<class T> bool logical_or<T>//逻辑或
template<class T> bool logical_not<T>//逻辑非
```

3.3.4 函数对象适配器

函数适配器 bind1st, bind2nd (参数绑定)

现在我有这个需求 在遍历容器的时候, 我希望将容器中的值全部加上 100 之后显示出来, 怎么做?

尝试: `for_each(v.begin(), v.end(), bind2nd(myprint(),100));`

需要我们自己的函数对象继承 `binary_function` 或者 `unary_function`

3.3.4.1 函数适配器

1) 创建函数对象类, 继承 `binary_function<T1,T2,T3>`, 声明 `()` 重载, 其中T1、T2是重载函数的2个参数, T3为重载函数的返回值类型。

【注意】定义public的 `()` 重载时, 必须由const修饰。

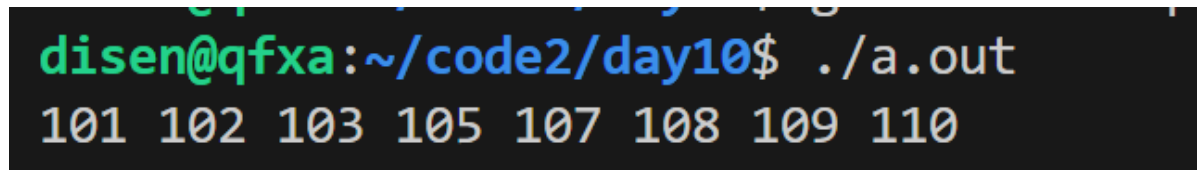
```
// 定义打印容器元素的二元仿函数
class Print3Plus : public binary_function<int, int, void>
{
public:
    void operator()(const int &n1, const int &n2) const
    {
        cout << n1 + n2 << " ";
    }
};
```

2) 应用函数适配器

```
for_each(v.begin(), v.end(), bind1st(函数对象类(), x));
for_each(v.begin(), v.end(), bind2nd(函数对象类(), x));
```

如:

```
int main(int argc, char const *argv[])
{
    int ms[] = {3, 5, 2, 1, 9, 10, 8, 7};
    set<int> s(ms, ms + 8);
    for_each(s.begin(), s.end(), bind1st(Print3Plus(), 100));
    cout << endl;
    return 0;
}
```



```
disen@qfxa:~/code2/day10$ ./a.out
101 102 103 105 107 108 109 110
```

bind1st 和 bind2nd 区别?

bind1st : 将参数绑定为函数对象的第一个参数
bind2nd : 将参数绑定为函数对象的第二个参数
bind1st和bind2nd 将二元函数对象转为一元函数对象

如:

```

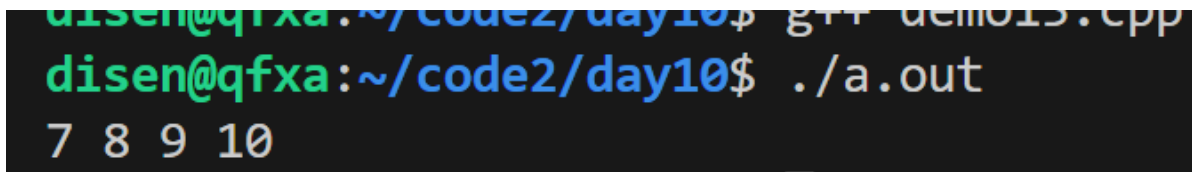
#include <iostream>
#include <set>
#include <algorithm>

using namespace std;

class PrintGt5Adapter : public binary_function<int, int, void>
{
public:
    void operator()(const int &n1, const int &n2) const
    {
        if (n1 > n2)
            cout << n1 << " ";
    }
};

int main(int argc, char const *argv[])
{
    int ms[] = {3, 5, 2, 1, 9, 10, 8, 7};
    set<int> s(ms, ms + 8);
    // 打印大于 5的所有集合元素
    for_each(s.begin(), s.end(), bind2nd(PrintGt5Adapter(), 5));
    cout << endl;
    return 0;
}

```



```

disen@qfxa:~/code2/day10$ g++ demo13.cpp
disen@qfxa:~/code2/day10$ ./a.out
7 8 9 10

```

3.3.4.2 取反适配器

- 1) 定义 `unary_function<int, bool>` 的函数对象类
- 2) 应用

```

find_if(v.begin(), v.end(), 函数对象类());
find_if(v.begin(), v.end(), not1(函数对象类()));
// 动态给定条件
find_if(v.begin(), v.end(), not1(bind2nd(greater<int>(), 5)));
sort(v.begin(), v.end(), not2(less<int>()));
// 匿名函数
for_each(v.begin(), v.end(), [](int val){cout << val << " "; });

```

其中的 `not1` 对一元函数对象取反, `not2` 对二元函数对象取反。

如: 查找第一个大于5的元素

```

#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;
// 自定义一个适配器, 只需要容器中元素即可。
class Gt5 : public unary_function<int, bool>

```

```

{
public:
    bool operator()(const int &n) const
    {
        return n > 5;
    }
};

int main(int argc, char const *argv[])
{
    int m[] = {1, 2, 2, 3, 5, 10};
    vector<int> v(m, m + 6);
    // find_if(start, end, callback) 返回查找到的第一个元素的地址;
    // vector<int>::iterator it = find_if(v.begin(), v.end(), not1(Gt5()));
    vector<int>::iterator it = find_if(v.begin(), v.end(), Gt5());
    cout << *it << endl;
    return 0;
}

```

```

disen@qfxa:~/code2/day10$ ./a.out
10

```

如：自定义二元适配器仿函数，实现查找大于n的第一个容器元素, 尝试反适配

```

#include <iostream>
#include <vector>
#include <algorithm>

class GtN : public binary_function<int, int, bool>
{
public:
    bool operator()(const int &n1, const int &n2) const
    {
        return n1 > n2;
    }
};

int main(int argc, char const *argv[])
{
    int m[] = {1, 2, 2, 3, 5, 10};
    vector<int> v(m, m + 6);
    // find_if(start, end, callback) 返回查找到的第一个元素的地址;
    vector<int>::iterator it = find_if(v.begin(), v.end(), not1(bind2nd(GtN(),
1)));
    cout << *it << endl;
    return 0;
}

```

```

disen@qfxa:~/code2/day10$ ./a.out
1

```


3.3.4.3 函数指针适配器

应用 ptr_fun()

【注意】仿函数的参数不能使用引用

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;
// 自定义一个适配器，只需要容器中元素即可。
class Gt5 : public unary_function<int, bool>
{
public:
    bool operator()(const int &n) const
    {
        return n > 5;
    }
};

class GtN : public binary_function<int, int, bool>
{
public:
    bool operator()(const int &n1, const int &n2) const
    {
        return n1 > n2;
    }
};

void print(int n1, int n2)
{
    cout << n1 << "-" << n2 << " " << endl;
}

bool gtn(int n1, int n2)
{
    return n1 > n2;
}

int main(int argc, char const *argv[])
{
    int m[] = {1, 2, 2, 3, 5, 10};
    vector<int> v(m, m + 6);
    // for_each(v.begin(), v.end(), bind2nd(ptr_fun(print), 2));
    vector<int>::iterator it = find_if(v.begin(), v.end(), bind2nd(ptr_fun(gtn),
4));
    cout << *it << endl;
    return 0;
}
```

```
disen@qfxa:~/code2/day10$ ./a.out
```

```
5
```

3.3.4.4 成员函数适配器

应用 mem_fun_ref

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

class Stu
{
private:
    string name;
    int age;

public:
    Stu(const string &name, int age)
    {
        this->name = name;
        this->age = age;
    }
    void show()
    {
        cout << "name is " << name << ", age is " << age << endl;
    }
};

int main(int argc, char const *argv[])
{
    vector<Stu> vs;
    vs.push_back(Stu("disen", 18));
    vs.push_back(Stu("lucy", 20));
    vs.push_back(Stu("jack", 15));
    vs.push_back(Stu("mack", 19));
    // 遍历容器中所有成员， 成员函数作为仿函数时，则通过容器成员调用它的仿函数。
    for_each(vs.begin(), vs.end(), mem_fun_ref(&Stu::show));
    return 0;
}
```

```
disen@qfxa:~/code2/day10$ ./a.out
name is disen, age is 18
name is lucy, age is 20
name is jack, age is 15
name is mack, age is 19
```

3.4 算法应用

算法中常用的功能涉及到比较、交换、查找、遍历、复制、修改、反转、排序、合并等。

3.4.1 常用遍历算法

3.4.1.1 for_each

```
/*
遍历算法 遍历容器元素
@param beg 开始迭代器
@param end 结束迭代器
@param _callback 函数回调或者函数对象
@return 函数对象
*/
for_each(iterator beg, iterator end, _callback);
```

3.4.1.2 transform

将指定容器区间元素搬运到另一容器中，

【注意】不会给目标容器分配内存，所以需要我们提前分配好内存

```
/*
@param beg1 源容器开始迭代器
@param end1 源容器结束迭代器
@param beg2 目标容器开始迭代器
@param _callback 回调函数或者函数对象
@return 返回目标容器迭代器
*/
transform(iterator beg1, iterator end1, iterator beg2, _callback);
```

如1:

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

int myfuncTest02(int val)
{
    return val;
}

int main(int argc, char const *argv[])
{
    vector<int> v1;
    v1.push_back(10);
    v1.push_back(20);
    v1.push_back(30);
    v1.push_back(40);
    v1.push_back(50);
```

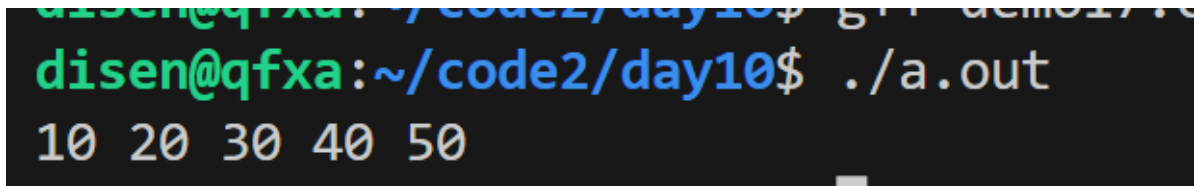
```

vector<int> v2;
v2.resize(v1.size());

transform(v1.begin(), v1.end(), v2.begin(), myfuncTest02);

copy(v2.begin(), v2.end(), ostream_iterator<int>(cout, " "));
cout << endl;
return 0;
}

```



```

disen@qfxa:~/code2/day10$ ./a.out
10 20 30 40 50

```

3.4.2 常用查找算法

```

/*
find 算法 查找元素
@param beg 容器开始迭代器
@param end 容器结束迭代器
@param value 查找的元素
@return 返回查找元素的位置
*/
find(iterator beg, iterator end, value)
/*
find_if 算法 条件查找
@param beg 容器开始迭代器
@param end 容器结束迭代器
@param callback 回调函数或者谓词(返回 bool 类型的函数对象)
@return 返回查找元素的位置
*/
find_if(iterator beg, iterator end, _callback);
/*
adjacent_find 算法 查找相邻重复元素
@param beg 容器开始迭代器
@param end 容器结束迭代器
@param _callback 回调函数或者谓词(返回 bool 类型的函数对象)
@return 返回相邻元素的第一个位置的迭代器
*/
adjacent_find(iterator beg, iterator end, _callback);
/*
binary_search 算法 二分查找法
注意: 在无序序列中不可用
@param beg 容器开始迭代器
@param end 容器结束迭代器
@param value 查找的元素
@return bool 查找返回 true 否则 false
*/
bool binary_search(iterator beg, iterator end, value);
/*
count 算法 统计元素出现次数
@param beg 容器开始迭代器
@param end 容器结束迭代器
@param value 回调函数或者谓词(返回 bool 类型的函数对象)
*/

```

```

@return int 返回元素个数
*/
count(iterator beg, iterator end, value);
/*
count_if 算法 统计元素出现次数
@param beg 容器开始迭代器
@param end 容器结束迭代器
@param callback 回调函数或者谓词(返回 bool 类型的函数对象)
@return int 返回元素个数
*/
count_if(iterator beg, iterator end, _callback);

```

如1: 打印第一个相邻的数

```

#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main(int argc, char const *argv[])
{
    int m[] = {1, 2, 3, 3, 5, 10};
    vector<int> v(m, m + 6);
    // 将当前容器的相邻的两个数传入仿函数中进行比较
    vector<int>::iterator it = adjacent_find(v.begin(), v.end(), equal_to<int>
());
    std::cout << *it << std::endl;
    return 0;
}

```

如2: 二分查找

```

#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

void print(int n)
{
    cout << n << " ";
}

int main(int argc, char const *argv[])
{
    int m[] = {1, 9, 6, 3, 5, 2, 10};
    vector<int> v(m, m + 7);
    // 排序
    sort(v.begin(), v.end(), less<int>());
    for_each(v.begin(), v.end(), print);
    cout << endl;
    // 二分查找算法: 要求容器的元素是有序的
    bool ret = binary_search(v.begin(), v.end(), 3);
    cout << ret << endl;
    return 0;
}

```

```
}
```

```
disen@qfxa:~/code2/day10$ g++ demo16.cpp  
./a.out disen@qfxa:~/code2/day10$ ./a.out  
1 2 3 5 6 9 10  
1
```

如3：统计容器中20的个数

```
#include <iostream>  
#include <vector>  
#include <algorithm>  
using namespace std;  
  
int myfuncTest02(int val)  
{  
    return val;  
}  
  
int main(int argc, char const *argv[])  
{  
    vector<int> v1;  
    v1.push_back(10);  
    v1.push_back(20);  
    v1.push_back(20);  
    v1.push_back(40);  
    v1.push_back(50);  
  
    cout << count(v1.begin(), v1.end(), 20) << endl;  
  
    return 0;  
}
```

```
disen@qfxa:~/code2/day10$ ./a.out  
2
```

如4：统计大于20的元素个数

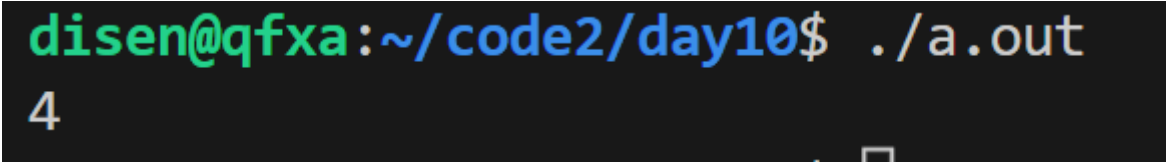
```
#include <iostream>  
#include <vector>  
#include <algorithm>  
using namespace std;  
  
int main(int argc, char const *argv[])  
{  
    vector<int> v1;  
    v1.push_back(10);  
    v1.push_back(20);  
    v1.push_back(20);  
    v1.push_back(40);  
    v1.push_back(50);
```

```

        cout << count_if(v1.begin(), v1.end(), bind2nd(greater<int>(), 10)) << endl;

        return 0;
    }

```



```

disen@qfxa:~/code2/day10$ ./a.out
4

```

3.4.3 常用排序算法

```

/*
merge 算法 容器元素合并，并存储到另一容器中
注意：两个容器必须是有序的
@param beg1 容器 1 开始迭代器
@param end1 容器 1 结束迭代器
@param beg2 容器 2 开始迭代器
@param end2 容器 2 结束迭代器
@param dest 目标容器开始迭代器
*/
merge(iterator beg1, iterator end1, iterator beg2, iterator end2, itera
tor dest)
/*
sort 算法 容器元素排序
@param beg 容器 1 开始迭代器
@param end 容器 1 结束迭代器
@param _callback 回调函数或者谓词(返回 bool 类型的函数对象)
*/
sort(iterator beg, iterator end, _callback)
/*
random_shuffle 算法 对指定范围内的元素随机调整次序
@param beg 容器开始迭代器
@param end 容器结束迭代器
*/
random_shuffle(iterator beg, iterator end)
/*
reverse 算法 反转指定范围的元素
@param beg 容器开始迭代器
@param end 容器结束迭代器
*/
reverse(iterator beg, iterator end)

```

如1: 随机打乱

```

#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

void print(int n)
{
    cout << n << " ";
}

```

```

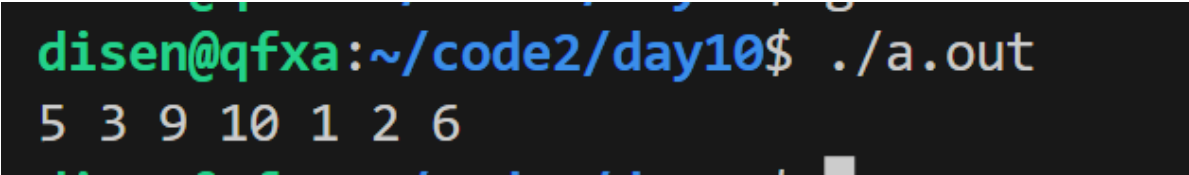
}

int main(int argc, char const *argv[])
{
    int m[] = {1, 9, 6, 3, 5, 2, 10};
    vector<int> v(m, m + 7);

    random_shuffle(v.begin(), v.end());
    for_each(v.begin(), v.end(), print);
    cout << endl;

    return 0;
}

```



```

disen@qfxa:~/code2/day10$ ./a.out
5 3 9 10 1 2 6

```

3.4.4 常用拷贝和替换算法

```

/*
copy 算法 将容器内指定范围的元素拷贝到另一容器中
@param beg 容器开始迭代器
@param end 容器结束迭代器
@param dest 目标起始迭代器
*/
copy(iterator beg, iterator end, iterator dest)

/*
replace 算法 将容器内指定范围的旧元素修改为新元素
@param beg 容器开始迭代器
@param end 容器结束迭代器
@param oldvalue 旧元素
@param newvalue 新元素
*/
replace(iterator beg, iterator end, oldvalue, newvalue)

/*
replace_if 算法 将容器内指定范围满足条件的元素替换为新元素
@param beg 容器开始迭代器
@param end 容器结束迭代器
@param callback 函数回调或者谓词(返回 Bool 类型的函数对象)
@param oldvalue 新元素
*/
replace_if(iterator beg, iterator end, _callback, newvalue)

/*
swap 算法 互换两个容器的元素
@param c1 容器 1
@param c2 容器 2
*/
swap(container c1, container c2)

```

如1: 复制

```

#include <iostream>
#include <vector>
#include <algorithm>

```



```

using namespace std;

void print(int n)
{
    cout << n << " ";
}

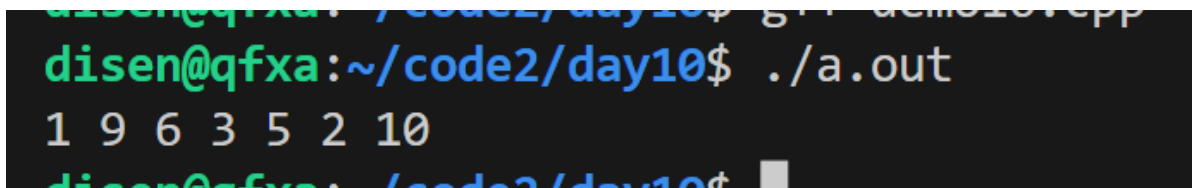
int main(int argc, char const *argv[])
{
    int m[] = {1, 9, 6, 3, 5, 2, 10};
    vector<int> v(m, m + 7);

    vector<int> v2;
    v2.resize(v.size());
    // 复制之前，需要自己先创建新的容器
    copy(v.begin(), v.end(), v2.begin());

    for_each(v2.begin(), v2.end(), print);
    cout << endl;

    return 0;
}

```



```

disen@qfxa:~/code2/day10$ ./a.out
1 9 6 3 5 2 10

```

如2：复制的元素直接输出

【注意】引入 `<iterator>` 头文件

```

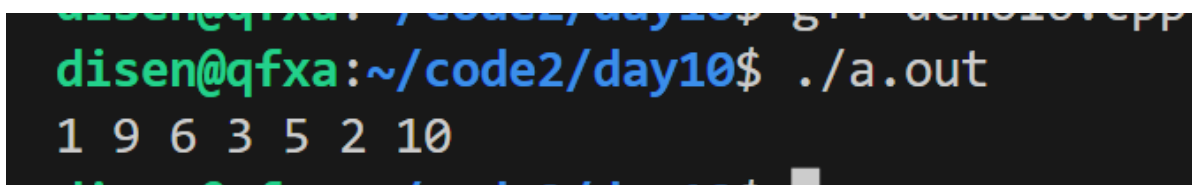
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

using namespace std;

int main(int argc, char const *argv[])
{
    int m[] = {1, 9, 6, 3, 5, 2, 10};
    vector<int> v(m, m + 7);
    // 复制的元素直接输出
    copy(v.begin(), v.end(), ostream_iterator<int>(cout, " "));
    cout << endl;

    return 0;
}

```



```

disen@qfxa:~/code2/day10$ ./a.out
1 9 6 3 5 2 10

```

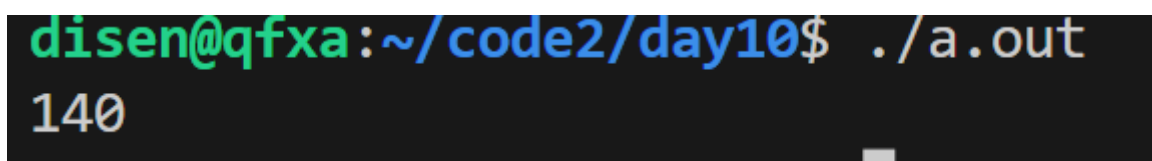
3.4.5 常用算术生成算法

引入 `<numeric>` 头文件

```
/*  
accumulate 算法 计算容器元素累计总和  
@param beg 容器开始迭代器  
@param end 容器结束迭代器  
@param value 累加值，额外加的值，可以为0  
@return 累加后的数值  
*/  
accumulate(iterator beg, iterator end, value)  
/*  
fill 算法 向容器中添加元素  
@param beg 容器开始迭代器  
@param end 容器结束迭代器  
@param value t 填充元素  
*/  
fill(iterator beg, iterator end, value)
```

如1：累加容器中所有元素

```
#include <iostream>  
#include <vector>  
#include <algorithm>  
#include <numeric>  
using namespace std;  
  
int main(int argc, char const *argv[])  
{  
    vector<int> v1;  
    v1.push_back(10);  
    v1.push_back(20);  
    v1.push_back(20);  
    v1.push_back(40);  
    v1.push_back(50);  
  
    int total;  
    total = accumulate(v1.begin(), v1.end(), 0);  
    cout << total << endl;  
    return 0;  
}
```



```
disen@qfxa:~/code2/day10$ ./a.out  
140
```

如2：将容器中所有元素替换为100

```
#include <iostream>  
#include <vector>  
#include <algorithm>  
#include <iterator>
```

```

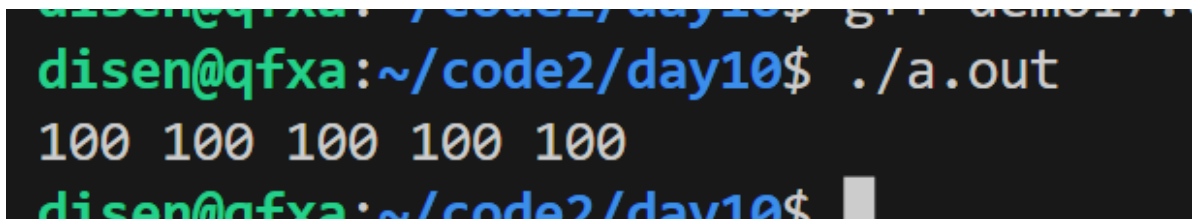
#include <numeric>
using namespace std;

int main(int argc, char const *argv[])
{
    vector<int> v1;
    v1.push_back(10);
    v1.push_back(20);
    v1.push_back(20);
    v1.push_back(40);
    v1.push_back(50);

    fill(v1.begin(), v1.end(), 100);
    copy(v1.begin(), v1.end(), ostream_iterator<int>(cout, " "));
    cout << endl;

    return 0;
}

```



```

disen@qfxa:~/code2/day10$ ./a.out
100 100 100 100 100
disen@qfxa:~/code2/day10$

```

3.4.6 常用集合算法

```

/*
set_intersection 算法 求两个 set 集合的交集
注意:两个集合必须是有序序列
@param beg1 容器 1 开始迭代器
@param end1 容器 1 结束迭代器
@param beg2 容器 2 开始迭代器
@param end2 容器 2 结束迭代器
@param dest 目标容器开始迭代器
@return 目标容器的最后一个元素的迭代器地址
*/
set_intersection(iterator beg1, iterator end1, iterator beg2, iterator
end2, iterator dest)
/*
set_union 算法 求两个 set 集合的并集
注意:两个集合必须是有序序列
@param beg1 容器 1 开始迭代器
@param end1 容器 1 结束迭代器
@param beg2 容器 2 开始迭代器
@param end2 容器 2 结束迭代器
@param dest 目标容器开始迭代器
@return 目标容器的最后一个元素的迭代器地址
*/
set_union(iterator beg1, iterator end1, iterator beg2, iterator end2, i
terator dest)
/*
set_difference 算法 求两个 set 集合的差集

```

注意:两个集合必须是有序序列

@param beg1 容器 1 开始迭代器

@param end1 容器 1 结束迭代器

@param beg2 容器 2 开始迭代器

@param end2 容器 2 结束迭代器

@param dest 目标容器开始迭代器

@return 目标容器的最后一个元素的迭代器地址

*/

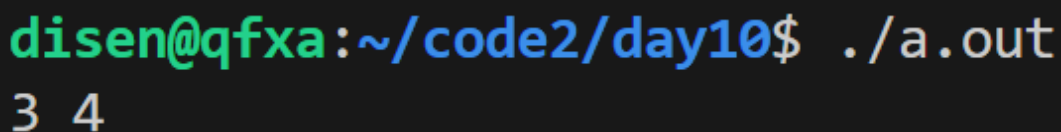
```
set_difference(iterator beg1, iterator end1, iterator beg2, iterator end2, iterator dest)
```

如1: 交集

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

int main(int argc, char const *argv[])
{
    int arr1[] = {1, 2, 3, 4, 5};
    int arr2[] = {10, 20, 3, 4, 50};
    set<int> s1(arr1, arr1 + 5);
    set<int> s2(arr2, arr2 + 5);
    vector<int> v3;
    v3.resize(5);

    set_intersection(s1.begin(), s1.end(), s2.begin(), s2.end(), v3.begin());
    // 查找第一个0的位置
    vector<int>::iterator it = find_if(v3.begin(), v3.end(),
bind2nd(equal_to<int>(), 0));
    v3.erase(it, v3.end()); // 删除所有的0
    // 打印
    copy(v3.begin(), v3.end(), ostream_iterator<int>(cout, " "));
    cout << endl;
    return 0;
}
```



```
disen@qfxa:~/code2/day10$ ./a.out
3 4
```

如2: 求并集

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

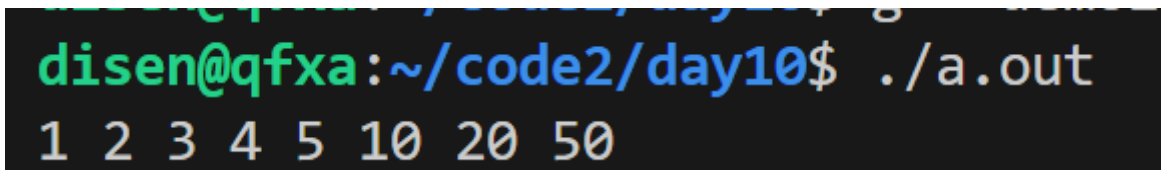
int main(int argc, char const *argv[])
{
```

```

int arr1[] = {1, 2, 3, 4, 5};
int arr2[] = {10, 20, 3, 4, 50};
set<int> s1(arr1, arr1 + 5);
set<int> s2(arr2, arr2 + 5);
vector<int> v3;
v3.resize(s1.size() + s2.size());

set_union(s1.begin(), s1.end(), s2.begin(), s2.end(), v3.begin());
// 查找第一个0的位置
vector<int>::iterator it = find_if(v3.begin(), v3.end(),
bind2nd(equal_to<int>(), 0));
v3.erase(it, v3.end()); // 删除所有的0
// 打印
copy(v3.begin(), v3.end(), ostream_iterator<int>(cout, " "));
cout << endl;
return 0;
}

```



```

disen@qfxa:~/code2/day10$ ./a.out
1 2 3 4 5 10 20 50

```

如3: 左差集

```

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

int main(int argc, char const *argv[])
{
    int arr1[] = {1, 2, 3, 4, 5};
    int arr2[] = {10, 20, 3, 4, 50};
    set<int> s1(arr1, arr1 + 5);
    set<int> s2(arr2, arr2 + 5);
    vector<int> v3;
    v3.resize(s1.size());

    set_difference(s1.begin(), s1.end(), s2.begin(), s2.end(), v3.begin());
    // 查找第一个0的位置
    vector<int>::iterator it = find_if(v3.begin(), v3.end(),
bind2nd(equal_to<int>(), 0));
    v3.erase(it, v3.end()); // 删除所有的0
    // 打印
    copy(v3.begin(), v3.end(), ostream_iterator<int>(cout, " "));
    cout << endl;
    return 0;
}

```

```
disen@qfxa:~/code2/day10$ ./a.out
1 2 5
```

如4: 右差集

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

int main(int argc, char const *argv[])
{
    int arr1[] = {1, 2, 3, 4, 5};
    int arr2[] = {10, 20, 3, 4, 50};
    set<int> s1(arr1, arr1 + 5);
    set<int> s2(arr2, arr2 + 5);
    vector<int> v3;
    v3.resize(s1.size());

    set_difference(s2.begin(), s2.end(), s1.begin(), s1.end(), v3.begin());
    // 查找第一个0的位置
    vector<int>::iterator it = find_if(v3.begin(), v3.end(),
bind2nd(equal_to<int>(), 0));
    v3.erase(it, v3.end()); // 删除所有的0
    // 打印
    copy(v3.begin(), v3.end(), ostream_iterator<int>(cout, " "));
    cout << endl;
    return 0;
}
```

```
disen@qfxa:~/code2/day10$ g++ demo17.cpp
disen@qfxa:~/code2/day10$ ./a.out
10 20 50
```

3.4 综合应用

参考代码:

```
#include <iostream>
#include <string>
#include <vector>
#include <map>
#include <time.h>
#include <algorithm>
#include <deque>
using namespace std;
class Person
{
    friend void playGame(int index, vector<int> &v, map<int, Person> &m,
vector<int> &v1);
```

```

private:
    int num;
    string name;
    float score[3];
public:
    Person() {}
    Person(int num, string name)
    {
        this->num = num;
        this->name = name;
    }
};

void createPerson(vector<int> &v, map<int, Person> &m)
{
    int i=0;
    string tmpName="ABCDEFGHJKLMNOPQRSTUVWXYZ";
    for(i=100;i<124;i++)
    {
        //存放选手编号
        v.push_back(i);

        //关联编号-选手
        string name="选手";
        name+=tmpName[i-100];
        m.insert(make_pair(i, Person(i, name)));
    }
}

void playGame(int index, vector<int> &v, map<int, Person> &m, vector<int> &v1)
{
    //统计分组人数
    int count = 0;
    //定义multimap容器存放<分数, 编号>
    multimap<float, int, greater<float>> mul;

    //设置随机数种子
    srand(time(NULL));
    random_shuffle(v.begin(), v.end());

    cout<<"-----第"<<index<<"轮比赛-----"<<endl;
    //逐个选手参加比赛
    vector<int>::iterator it=v.begin();
    for(; it!=v.end(); it++)
    {
        count++;

        //定deque容器存放评委分数
        deque<float> d;

        //10个评委打分
        int i=0;
        for(i=0;i<10;i++)
        {
            d.push_back((float)(rand()%41+60));
        }

        //对deque容器排序
        sort(d.begin(), d.end());
        //去掉最高分, 最低分
    }
}

```

```

        d.pop_back();
        d.pop_front();

        float avg=accumulate(d.begin(),d.end(),0)/d.size();
        //更新map容器中选手成绩
        m[*it].score[index-1] = avg;

        mul.insert(make_pair(avg, *it));

        if(count%6 == 0)//刚好一组 6人
        {
            multimap<float,int,greater<float>>>::iterator mit=mul.begin();

            //取出前三的编号
            int i=0;
            for(i=0;i<3;i++,mit++)
            {
                v1.push_back((*mit).second);
            }

            cout<<"\t-----第"<<count/6<<"组的成绩-----"<<endl;
            //遍历当前组所有人员成绩
            for(i=0,mit=mul.begin();i<6;i++,mit++)
            {
                cout<<"\t\t"<<m[(*mit).second].name<<" 编号:"<<(*mit).second\
                    <<" 成绩"<<(*mit).first;

                if(i<3)
                    cout<<" (恭喜晋级)"<<endl;
                cout<<endl;
            }

            //清空mul容器
            mul.clear();
        }

    }

}

int main(int argc, char *argv[])
{
    //定义vector容器存放编号   map容器存放<编号,选手>
    vector<int> v;
    map<int, Person> m;

    //创建编号和选手
    createPerson(v, m);

    //参加第一轮比赛
    vector<int> v1;//存放晋级编号
    playGame(1, v, m, v1);

    vector<int> v2;
    playGame(2, v1, m, v2);

    vector<int> v3;
    playGame(3, v2, m, v3);

    return 0;
}

```