C++ STL

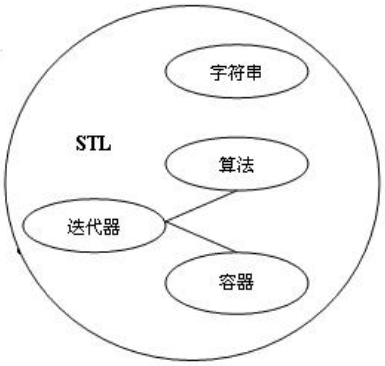
10、STL 实用技术专题

10.1 STL(标准模板库)理论基础

10.1.1 基本概念

STL（Standard Template Library，标准模板库)是惠普实验室开发的一系列软件的统称。现然主要出现在 C++中，但在被引入 C++之前该技术就已经存在了很长的一段时间。

STL 的从广义上讲分为三类：algorithm（算法）、container（容器）和 iterator（迭代器），容器和算法通过迭代器可以进行无缝地连接。几乎所有的代码都采用了模板类和模板函数的方式，这相比于传统的由函数和类组成的库来说提供了更好的代码重用机会。在 C++标准中，STL 被组织为下面的 13 个头文 件：<algorithm>、<deque>、<functional>、<iterator>、<vector>、<list>、<map>、<memory>、<numeric>、<queue>、<set>、<stack> 和<utility>。



STL 详细的说

- 容器（Container）　　//相当于链表
- 算法（Algorithm）
- 迭代器（Iterator）　　//相当于指针
- 仿函数（Function object）
- 适配器（Adaptor）
- 空间配制器（allocator）

使用 STL 的好处：

1. STL 是 C++的一部分，因此不用额外安装什么，它被内建在你的编译器之内。

2. STL 的一个重要特点是数据结构和算法的分离。尽管这是个简单的概念，但是这种分离确实使得 STL 变得非常通用。

例如，在 STL 的 vector 容器中，可以放入元素、基础数据类型变量、元素的地址；

STL 的 sort()函数可以用来操作 vector,list 等容器。

3. 程序员可以不用思考 STL 具体的实现过程，只要能够熟练使用 STL 就 OK 了。这样他们就可以把精力放在程序开发的别的方面。

4. STL 具有高可重用性，高性能，高移植性，跨平台的优点。

5.高可重用性：STL 中几乎所有的代码都采用了模板类和模版函数的方式实现，这相比于传统的由函数和类组成的库来说提供了更好的代码重用机会。关于模板的知识，已经给大家介绍了。

6.高性能:如 map 可以高效地从十万条记录里面查找出指定的记录,因为 map 是采用红黑树的变体实现的。(红黑树是平横二叉树的一种)

7.高移植性：如在项目 A 上用 STL 编写的模块，可以直接移植到项目 B 上。

8.跨平台:如用 windows 的 Visual Studio 编写的代码可以在 Mac OS 的 XCode 上直接编译。

9.程序员可以不用思考 STL 具体的实现过程，只要能够熟练使用 STL 就 OK 了。这样他们就可以把精力放在程序开发的别的方面。

总之：招聘工作中，经常遇到 C++程序员对 STL 不是非常了解。大多是有一个大致的映像，而对于在什么情况下应该使用哪个容器和算法都感到比较茫然。STL 是 C++程序员的一项不可或缺的基本技能，掌握它对提升 C++编程大有裨益。

10.1.2 容器

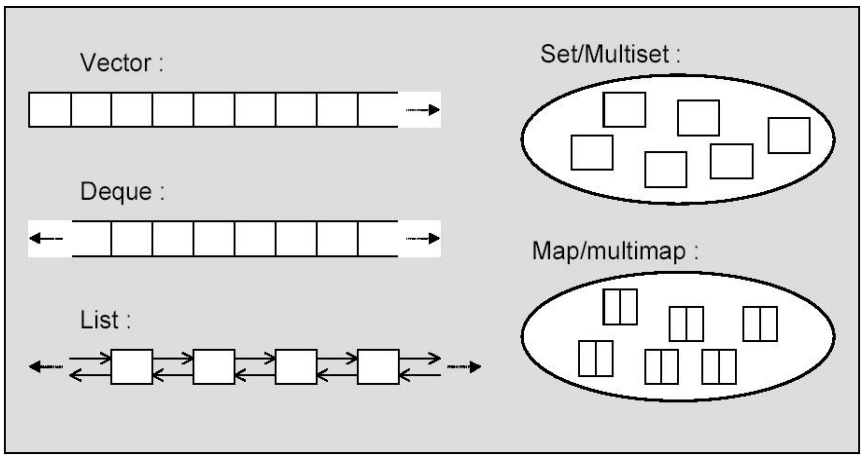
在实际的开发过程中，数据结构本身的重要性不会逊于操作于数据结构的算法的重要性，当程序中存在着对时间要求很高的部分时，数据结构的选择就显得更加重要。

经典的数据结构数量有限，但是我们常常重复着一些为了实现向量、链表等结构而编写的代码,这些代码都十分相似,只是为了适应不同数据的变化而在 细节上有所出入。STL 容器就为我们提供了这样的方便，它允许我们重复利用已有的实现构造自己的特定类型下的数据结构，通过设置一些模板，STL 容器对最常用的数据结构提供了支持，这些模板的参数允许我们指定容器中元素的数据类型，可以将我们许多重复而乏味的工作简化。

容器部分主要由头文 件<vector>,<list>,<deque>,<set>,<map>,<stack> 和<queue>组成。对于常用的一些容器和容器适配器（可以看作由其它容器实现的容器），可以通过下表总结一下它们和相应头文件的对应关系。

10.1.2.1 容器的概念

用来管理一组元素



10.1.2.2 容器的分类

序列式容器（Sequence containers）

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。

vector、*deque*、*list*

关联式容器（Associated containers）

元素位置取决于特定的排序准则，和插入顺序无关

set、*multiset*、*map*、*multimap*

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	<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>

10.1.3 迭代器

迭代器从作用上来说是最基本的部分，可是理解起来比前两者都要费力一些。软件设计有一个基本原则，所有的问题都可以通过引进一个间接层来简化，

这种简化在 STL 中就是用迭代器来完成的。概括来说，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。

10.1.4 算法

函数库对数据类型的选择对其可重用性起着至关重要的作用。举例来说，一个求方根的函数，在使用浮点数作为其参数类型的情况下的可重用性肯定比使用整型作为它的参数类性要高。而 C++通过模板的机制允许推迟对某些类型的选择，直到真正想使用模板或者说对模板进行特化的时候，STL 就利用了这一点提供了相当多的有用算法。它是在一个有效的框架中完成这些算法的——可以将所有的类型划分为少数的几类，然后就可以在模版的参数中使用一种类型替换掉同一种类中的其他类型。

STL 提供了大约 100 个实现算法的模版函数，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。

10.1.5 C++标准库

C++强大的功能来源于其丰富的类库及库函数资源。C++标准库的内容总共在 50 个标准头文件中定义。在 C++开发中，要尽可能地利用标准库完成。这样做的

直接好处包括：（1）成本：已经作为标准提供，何苦再花费时间、人力重新开发呢；（2）质量：标准库的都是经过严格测试的，正确性有保证；（3）效率：关于人的效率已经体现在成本中了，关于代码的执行效率要相信实现标准库的大牛们的水平；（4）良好的编程风格：采用行业中普遍的做法进行开发。

在 C++程序设计课程中，尤其是作为第一门程序设计课程，我们注重了语法、语言的机制等方面的内容。程序设计能力的培养有个过程，跨过基本的原理性知识直接进入到工程中的普遍做法，由于跨度决定了其难度。再者，在掌握了基本原理的基础上，在认识标准库的问题上完全可以凭借实践，逐步地掌握。标准库的学习不需要认认真真地读书，需要的是在了解概貌的情况下，在实践中深入。

这个任务就是要知道 C++程序设计课程中不讲的，但对程序设计又很重要的这部分内容。至少我们要能先回答出“有什么”的问题。

C++标准库的内容分为 10 类，分别是（建议在阅读中，将你已经用过或听说过的头文件划出来）：

C1. 标准库中与语言支持功能相关的头文件

头文件	描　　述
<cstddef>	定义宏 NULL 和 offsetof，以及其他标准类型 size_t 和 ptrdiff_t。与对应的标准 C 头文件的区别是，NULL 是 C++空指针常量的补充定义，宏 offsetof 接受结构或者联合类型参数，只要他们没有成员指针类型的非静态成员即可。
<limits>	提供与基本数据类型相关的定义。例如，对于每个数值数据类型，它定义了可以表示出来的最大值和最小值以及二进制数字的位数。
<climits>	提供与基本整数数据类型相关的 C 样式定义。这些信息的 C++样式定义在<limits>中
<cfloat>	提供与基本浮点型数据类型相关的 C 样式定义。这些信息的 C++样式定义在<limits>中
<cstdlib>	提供支持程序启动和终止的宏和函数。这个头文件还声明了许多其他杂项函数，例如搜索和排序函数,从字符串转换为数值等函数。它与对应的标准 C 头文件 stdlib.h 不同，定义了 abort(void)。abort()函数还有额外的功能，它不为静态或自动对象调用析构函数，也不调用传给 atexit()函数的函数。它还定义了 exit()函数的额外功能，可以释放静态对象，以注册的逆序调用 atexit()注册的函数。清除并关闭所有 打开的 C 流，把控制权返回给主机环境。
<new>	支持动态内存分配
<typeinfo>	支持变量在运行期间的类型标识
<exception>	支持异常处理，这是处理程序中可能发生的错误的一种方式
<cstdarg>	支持接受数量可变的参数的函数。即在调用函数时，可以给函数传送数量不等的数据项。它定义了宏 va_arg、va_end、va_start 以及 va_list 类型
<csetjmp>	为 C 样式的非本地跳跃提供函数。这些函数在 C++中不常用
<csignal>	为中断处理提供 C 样式支持

C2. 支持流输入/输出的头文件

头文件	描　　述
<iostream>	支持标准流 cin、cout、cerr 和 clog 的输入和输出，它还支持多字节字符标准流 wcin、wcout、wcerr 和 wclog。
<iomanip>	提供操纵程序，允许改变流的状态，从而改变输出的格式。
<ios>	定义 iostream 的基类
<istream>	为管理输出流缓存区的输入定义模板类
<ostream>	为管理输出流缓存区的输出定义模板类
<sstream>	支持字符串的流输入输出
<fstream>	支持文件的流输入输出
<iosfwd>	为输入输出对象提供向前的声明
<streambuf>	支持流输入和输出的缓存
<cstdio>	为标准流提供 C 样式的输入和输出
<cwchar>	支持多字节字符的 C 样式输入输出

C3. 与诊断功能相关的头文件

头文件	描　　述
<stdexcept>	定义标准异常。异常是处理错误的方式
<cassert>	定义断言宏，用于检查运行期间的情形
<cerrno>	支持 C 样式的错误信息

C4. 定义工具函数的头文件

头文件	描　　述
<utility>	定义重载的关系运算符，简化关系运算符的写入，它还定义了 pair 类型，该类型是一种模板类型，可以存储一对值。这些功能在库的其他地方使用
<functional>	定义了许多函数对象类型和支持函数对象的功能，函数对象是支持 operator()() 函数调用运算符的任意对象
<memory>	给容器、管理内存的函数和 auto_ptr 模板类定义标准内存分配器
<ctime>	支持系统时钟函数

C5. 支持字符串处理的头文件

头文件	描　　述
<string>	为字符串类型提供支持和定义，包括单字节字符串(由 char 组成)的 string 和多字节字符串(由 wchar_t 组成)
<cctype>	单字节字符类别
<cwctype>	多字节字符类别

头文件	描　　述
<cstring>	为处理非空字节序列和内存块提供函数。这不同于对应的标准 C 库头文件，几个 C 样式字符串的一般 C 库函数被返回值为 const 和非 const 的函数对替代了
<cwchar>	为处理、执行 I/O 和转换多字节字符序列提供函数，这不同于对应的标准 C 库头文件，几个多字节 C 样式字符串操作的一般 C 库函数被返回值为 const 和非 const 的函数对替代了。
<cstdlib>	为把单字节字符串转换为数值、在多字节字符和多字节字符串之间转换提供函数

C6. 定义容器类的模板的头文件

<vector>	定义 vector 序列模板，这是一个大小可以重新设置的数组类型，比普通数组更安全、更灵活
<list>	定义 list 序列模板，这是一个序列的链表，常常在任意位置插入和删除元素
<deque>	定义 deque 序列模板，支持在开始和结尾的高效插入和删除操作
<queue>	为队列(先进先出)数据结构定义序列适配器 queue 和 priority_queue
<stack>	为堆栈(后进先出)数据结构定义序列适配器 stack
<map>	map 是一个关联容器类型，允许根据键值是唯一的，且按照升序存储。multimap 类似于 map，但键不是唯一的。
<set>	set 是一个关联容器类型，用于以升序方式存储唯一值。multiset 类似于 set，但是值不必是唯一的。
<bitset>	为固定长度的位序列定义 bitset 模板，它可以看作固定长度的紧凑型 bool 数组

C7. 支持迭代器的头文件

头文件	描　　述
<iterator>	给迭代器提供定义和支持

C8. 有关算法的头文件

头文件	描　　述
<algorithm>	提供一组基于算法的函数，包括置换、排序、合并和搜索
<cstdlib>	声明 C 标准库函数 bsearch()和 qsort()，进行搜索和排序
<ciso646>	允许在代码中使用 and 代替&&

C9. 有关数值操作的头文件

头文件	描　　述
<complex>	支持复杂数值的定义和操作
<valarray>	支持数值矢量的操作
<numeric>	在数值序列上定义一组一般数学操作，例如 accumulate 和 inner_product
<cmath>	这是 C 数学库，其中还附加了重载函数，以支持 C++约定
<cstdlib>	提供的函数可以提取整数的绝对值，对整数进行取余数操作

C10. 有关本地化的头文件

头文件	描　　述
<locale>	提供的本地化包括字符类别、排序序列以及货币和日期表示。

<clocale>	对本地化提供 C 样式支持

C++标准库的所有头文件都没有扩展名。C++标准库以<cname>形式的标准头文件提供。在 <cname>形式标准的头文件中，与宏相关的名称在全局作用域中定义，其他名称在 std 命名空间中声明。在 C++中还可以使用 name.h 形式的标准 C 库头文件名

10.1.6 模板简要回顾

- ✧ 模板是实现代码重用机制的一种工具，实质就是实现类型参数化，即把类型定义为参数。
- ✧ C++提供两种模板：函数模板，类模板

函数模板的简介

- ✧ 函数模板就是建立一个通用的函数，其函数返回类型和形参类型不具体指定，而是用虚拟的类型来代表。
- ✧ 凡是函数体相同的函数都可以用函数模板来代替，不必定义多个函数，只需在模板中定义一次即可。
- ✧ 在调用函数时系统会根据实参的类型来取代模板中的虚拟类型，从而实现了不同函数的功能。

类模板的简介

- ✧ 我们先来看一下下面这个类，求最大值的类
- ✧ 和函数模板一样，类模板就是建立一个通用类，其数据成员的类型、成员函数的返回类型和参数类形都可以不具体指定，而用虚拟的类型来代表。
- ✧ 当使用类模板建立对象时，系统会根据实参的类型取代类模板中的虚拟类型，从而实现不同类的功能。

10.2 容器

10.2.1 STL 的 string →是一个 char*型的容器

1. String 概念

- ✧ string 是 STL 的字符串类型，通常用来表示字符串。而在使用 string 之前，字符串通常是用 char*表示的。string 与 char*都可以用来表示字符串，那么二者有什么区别呢。

string 和 char*的比较。

- ✧ string 是一个类, char*是一个指向字符的指针。

string 封装了 char*，管理这个字符串，是一个 char*型的容器。

✧ string 不用考虑内存释放和越界。

string 管理 char*所分配的内存。每一次 string 的复制，取值都由 string 类负责维护，不用担心复制越界和取值越界等。

✧ string 提供了一系列的字符串操作函数（这个等下会详讲）

查找 find，拷贝 copy，删除 erase，替换 replace，插入 insert

2.string 的构造函数

✧ 默认构造函数：

string s1(); //构造一个空的字符串 string s1。

✧ 拷贝构造函数：

string(const string &str); //构造一个与 str 一样的 string。

如 string s1(s2)。

✧ 带参数的构造函数

string(const char *s); //用字符串 s 初始化

//注意是*s ，不要只写 s

string(int n,char c); //用 n 个字符 c 初始化

2*.string 的遍历

```
//string 的 遍历
void main22()
{
    string s1 = "abcdefg";

    //1 数组方式
    for (int i=0; i<s1.length(); i++)
    {
        cout << s1[i] << " ";
    }
    cout << endl;

    //2 迭代器
    for (string::iterator it = s1.begin(); it != s1.end(); it++ )
    {
        cout << *it << " ";
```

```
    }
    cout << endl;

   //2* 迭代器
    try
    {
        for (int i=0; i<s1.length() + 3; i++)
        {
            cout << s1.at(i) << " ";  //抛出异常
        }
    }
    catch ( ... )
    {
        cout << "发生异常\n" ;
    }

    cout << "at 之后" << endl;

//若不采用 at 的方式，采用数组的方式来进行异常的处理时
    try
    {
        for (int i=0; i<s1.length() + 3; i++)
        {
            cout << s1[i] << " "; //出现错误 不向外面抛出异常 引起程序的中断
        }
    }
    catch ( ... )
    {
        cout << "发生异常\n" ;
    }
    */
}
```

3. string 的存取字符操作

- ✧ string 类的字符操作：
- ✧ const char &operator[] (int n) const;
- ✧ const char &at(int n) const;
- ✧ char &operator[] (int n); //如 str[3]
- ✧ char &at(int n);

✧ operator[]和 at()均返回当前字符串中第 n 个字符，但二者是有区别的。

主要区别在于 at()在越界时会抛出异常，[]在刚好越界时会返回(char)0，再继续越界时，编译器直接出错。如果你的程序希望可以通过 try,catch 捕获异常，建议采用 at()。

4.从 string 取得 const char*的操作

✧ const char *c_str() const;　　//返回一个以'\0'结尾的字符串的首地址

c_str()函数返回一个指向正规 C 字符串的指针

5.把 string 拷贝到 char*指向的内存空间的操作

✧ int copy(char *s, int n, int pos=0) const;

把当前串中以 pos 开始的 n 个字符拷贝到以 s 为起始位置的字符数组中，返回实际拷贝的数目。注意要保证 s 所指向的空间足够大以容纳当前字符串，不然会越界。

5*.字符指针和 string 的转换

✧ 1.将 string →字符指针

```
string s1 = "aaabbbb";
const char *c=s1.c_str() ;
```

✧ 2.将 字符指针→string

```
char* s="good boy";
string str=s;
```

✧ 3.将 string 拷贝到 buf1[ ]

```
string s1 = "aaabbbb";
char buf1[128] = {0};  //也可以在拷贝完以后自己加/0 否则会出错
s1.copy(buf1, 3, 0);  //注意只给你 copy3 个字符 不会变成 C 风格的字符串
```

或者：

```
string str = "good boy";
char buf1[20];
for(int i=0;i<str.length();i++)
   buf1[i] = str[i];
buf1[str.length()] = '\0';
```

6. string 的长度

- int length() const; //返回当前字符串的长度。长度不包括字符串结尾的'\0'。
- bool empty() const; //当前字符串是否为空

7. string 的赋值

- string &operator=(const string &s); //把字符串 s 赋给当前的字符串

  s1 = s2 将 s2 赋值给 s1

- string &assign(const char *s); //把字符串 s 赋给当前的字符串

```
tring str1, str2 = "War and Peace";
 str1.assign(str2);
cout << str1 << endl;    //结果为：War and Peace
```

  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 //注意是*s ，不要只写 s

- 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); //把字符串 s 中从 start 开始的 n 个字符赋给当前字符串

```
string str1, str2 = "War and Peace";
str1.assign(str2, 3, 4); //从 str2 第 3 个位置开始连续四个元素赋值给 str1
 cout << str1 << endl; //结果为：  and   //包含空格
```

8. string 字符串连接（将两个字符串连接起来）

- string &operator+=(const string &s); //把字符串 s 连接到当前字符串结尾
- string &operator+=(const char *s);//把字符串 s 连接到当前字符串结尾

  //注意是*s ，不要只写 s

- ✧ 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 开始的 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

例如：

法 1:

```
string s1 = "aaa";
string s2 = "bbb";
s1 = s1 + s2;
cout << "s1:" << s1 << endl;
```

法 2：

```
string s3 = "333";
string s4 = "444";
s3.append(s4);
cout << "s3:" << s3 << endl;
```

9. string 的比较

- ✧ int compare(const string &s) const; //与字符串 s 比较

  int compare(const char *s) const; //与字符串 s 比较

  //注意是*s ，不要只写 s
- ✧ compare 函数在>时返回 1，<时返回 -1，==时返回 0。比较区分大小写，比较时参考字典顺序，排越前面的越小。大写的 A 比小写的 a 小。

10. string 的子串

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

11. string 的查找 和 替换（重点）

查找

- int find(char c,int pos=0) const; //从 pos 开始查找字符 c (可以好几个字母)在当前字符串的位置 返回整数
- int find(const char *s, int pos=0) const; //从 pos 开始查找字符串 s 在当前字符串的位置 返回整数 看下面的程序，注意是*s
- int find(const string &s, int pos=0) const; //从 pos 开始查找字符串 s 在当前字符串中的位置 返回整数
- find 函数如果查找不到，就返回-1
- int rfind(char c, int pos=npos) const; //从 pos 开始从后向前查找字符 c 在当前字符串中的位置 返回整数 **看下面的程序帮助理解**
- int rfind(const char *s, int pos=npos) const;
- int rfind(const string &s, int pos=npos) const;
- rfind 是反向查找的意思，如果查找不到， 返回-1

程序 1:

```
string s1 = "wbm hello wbm 111 wbm 222 wbm 333";
size_t index = s1.find("a", 3);  //size_t 是标准 C 库中定义的，为 unsigned int
cout << "index: " << index;
```

运行结果： index: 4294967295

原因：未找到时 find 返回的是-1，但是 size_t 是无符号整型的数，所以显示的不是-1

程序 2:

```
string s1 = "wbm hello wbm 111 wbm 222 wbm 333";
int  index = s1.find("a", 3);  //size_t 是标准 C 库中定义的，应为 unsigned
cout << "index: " << index;
```

运行结果： index: -1　　// 就是-1

程序 3:rfind()的用法

（1）
```
string st = "abcdef";
cout << st.rfind("ef", 5) << endl;
```

运行结果：4

st.rfind(“ef”,5) 的含义：从 s[5]开始往前进行查找 ef ，得到的是从前面数 ef 的位置

（2）
```
string st = "abcdef";
string st2 = "ef";
string *st1 = &st2;
cout << st.rfind(*st1, 5) << endl;   //结果为  4
```

注意： 应该填入的是*str1，而不是 str1

替换

- ✧ string &replace(int pos, int n, const char *s);//删除从 pos 开始的 n 个字符，然后在 pos 处插入串 s
- ✧ string &replace(int pos, int n, const string &s);　//删除从 pos 开始的 n 个字符，然后在 pos 处插入串 s
- ✧ void swap(string &s2);　　//交换当前字符串与 s2 的值

```
//4 字符串的查找和替换
void main25()
{
    string s1 = "wbm hello wbm 111 wbm 222 wbm 333";
    size_t index = s1.find("wbm", 0);  //size_t 是标准 C 库中定义的，应为 unsigned int，在 64 位系统中为 long unsigned int。
    cout << "index: " << index;

    //求某个字符出现的位置和次数
    size_t offindex = s1.find("wbm", 0);
```

```
while (offindex != string::npos)
{
    i++;        // i 的初值为 0
    cout << "在下标 index: " << offindex << "找到 wbm\n";
    offindex = offindex + 1;     //必须要有，为了从下一个位置开始查找
    offindex = s1.find("wbm", offindex);
}
    cout << "一共出现 wbm 的次数为" << i ;

//替换
string s2 = "wbm hello wbm 111 wbm 222 wbm 333";
s2.replace(0, 3, "abcde") ;  //从 0 位置开始，删除三个元素，然后插入 abcde
cout << s2 << endl ;
```

12. String 的区间删除和插入

- ✧ string &insert(int pos, const char *s);
- ✧ string &insert(int pos, const string &s);

  //前两个函数在 pos 位置插入字符串 s 返回值为字符

- ✧ string &insert(int pos, int n, char c); //在 pos 位置 插入 n 个字符 c
- ✧ string &erase(int pos, int n=npos); //删除 pos 开始的 n 个字符，返回修改后的字符串

```
//截断（区间删除）和插入
void main26()
{
   string s1 = "hello1 hello2 hello1";
   string::iterator it = find(s1.begin(), s1.end(), 'l');  //单独的find 是一个算法，s1.find 才表示查找函数
                                                           //迭代器其实就是一个指针
   if (it != s1.end() )   //s1.end() 也表示的是一个迭代器
   {
```

```
        s1.erase(it);
    }
    cout << "s1 删除 1 以后的结果:" << s1 << endl;    //结果是 helo1 hello2
hello1

    s1.erase(s1.begin(), s1.end() );
    cout << "s1 全部删除:" << s1 << endl;
    cout << "s1 长度 " << s1.length() << endl;

    string s2 = "BBB";
    s2.insert(0, "AAA"); // 头插法
    s2.insert(s2.length(), "CCC");    //尾插法插入 CCC
    cout << s2 << endl;
}
```

13.string 算法相关

```
void main27()
{
    string s1 = "AAAbbb";
    //1 函数的入口地址 2 函数对象 3 预定义的函数对象
    transform(s1.begin(), s1.end(),s1.begin(), toupper);    //全部转化
为大写
    cout << "s1" << s1 << endl;

    string s2 = "AAAbbb";
    transform(s2.begin(), s2.end(), s2.begin(), tolower);    //全部转化
为小写
    cout << "s2:" << s2 << endl;
}
```

10.2.2 .vector 容器

1.vector 容器简介

- ✧ vector 是将元素置于一个动态数组中加以管理的容器。
- ✧ vector 可以随机存取元素（支持索引值直接存取， 用[]操作符或 at()方法，这个等下会详讲）。

vector 尾部添加或移除元素非常快速。但是在中部或头部插入元素或移除元素比较费时

2.vector 对象的默认构造

vector 采用模板类实现，vector 对象的默认构造形式

vector<T> vecT;

vector<int> vecInt;　　//一个存放 int 的 vector 容器。

vector<float> vecFloat;　　//一个存放 float 的 vector 容器。

vector<string> vecString;　　//一个存放 string 的 vector 容器。

…　　//尖括号内还可以设置指针类型或自定义类型。

Class CA{ };　　//类

vector<CA*> vecpCA;　　//用于存放 CA 对象的指针的 vector 容器。

vector<CA> vecCA;　　//用于存放 CA 对象的 vector 容器。**由于容器元素的存放是按值复制的方式**进行的，所以此时 CA 必须提供 CA 的拷贝构造函数，以保证 CA 对象间拷贝正常。

```
//数组元素的 添加和删除
void main31()
{
    vector<int> v1;

    cout << "length:" << v1.size() << endl;    //当没有元素插入时，此时 vector 大小为 0
    v1.push_back(1);
    v1.push_back(3);
    v1.push_back(5);
    cout << "length:" << v1.size() << endl;   //插入了 3 个数，此时 vector 大小为 3

    cout << "头部元素" << v1.front() << endl;

    //修改 头部元素
    //函数返回值当左值 应该返回一个引用
    v1.front() = 11.5;
    v1.back() = 55;

    while (v1.size() > 0)
    {
        cout << "尾部元素" << v1.back()<<endl; //获取尾部元素
```

```
        v1.pop_back(); //删除尾部元素
    }
}
```

3.vector 对象的带参数构造

理论知识

- ✧ vector(beg,end) ; //构造函数将[beg, end)区间中的元素拷贝给本身。注意该区间是左闭右开的区间。
- ✧ vector(n,elem) ; //构造函数将 n 个 elem 拷贝给本身。
- ✧ vector(const vector &vec); //拷贝构造函数

1.用数组来进行构造：

int iArray[ ] = {0,1,2,3,4};

vector<int> vecIntA(iArray, iArray+5);

2.用另一个容器来构造当前容器：

vector<int> vecIntB (vecIntA.begin() , vecIntA.end()); //用构造函数初始化容器 vecIntB

vector<int> vecIntB (vecIntA.begin() , vecIntA.begin()+3);

vector<int> vecIntB(3,9); //此代码运行后，容器 vecIntB 存放 3 个元素，每个元素的值是 9。

vector<int> vecIntD(vecIntA); //拷贝构造

```
//vector 的初始化
void main32()
{
    vector<int> v1;          //int 型的 vector 容器，默认构造
    v1.push_back(1);
    v1.push_back(3);
    v1.push_back(5);
    v1.push_back(7);

    vector<int> v2 = v1;     //用另一个容器来构造当前容器
    vector<int> v3(v1.begin(), v1.begin() + 2);   //同上
}
```

4.vector 的赋值

理论知识

- ✧ vector.assign(beg,end);　　//将[beg, end)区间中的数据拷贝赋值给本身。注意该区间是左闭右开的区间。
- ✧ vector.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
- ✧ vector.swap(vec);　// 将 vec 与本身的元素互换。

```
vector<int> vecIntA, vecIntB, vecIntC, vecIntD;
int   iArray[ ] = {0,1,2,3,4};
vecIntA.assign(iArray,iArray+5);    //注意是左闭右开区间的元素拷贝进去

vecIntB.assign( vecIntA.begin() , vecIntA.end() );   //用其它容器的迭代器作参数。

vecIntC.assign(3,9);

vector<int> vecIntD;
vecIntD = vecIntA;

vecIntA.swap(vecIntD);      //两个容器中的元素个数可以不一致
```

4*.vector 的遍历

```
//vector 的遍历 通过数组的方式
void main33()
{
    vector<int> v1(10);    //提前把内存准备好,注意与之前构造时的语句不同
                           //如果不提前把内存准备好，则无法通过数组方式遍历
    for (int i = 0; i<10; i++)   //vector 容器是动态数组，所以自然可以用数组的方式遍历，前提是必须把内存准备好
    {
        v1[i] = i + 1;
    }

printV(v1);  //自己创建的打印 vector 容器对象的语句，打印结果：1 2 3 4 5 6 7 8 9 10
}

void printV(vector<int> &v)  //自己构造的函数
```

```
{
    for (int i = 0; i<v.size(); i++)
    {
        cout << v[i] << " ";
    }
}

//push_back 的强化记忆  （注意！）
void main34()
{
    vector<int> v1(10);    //先准备 10 个内存内存准备好,但是并没有初始化
    v1.push_back(100);     //然后在插入 1 个数 为 100
    v1.push_back(200);     //然后在插入 1 个数 为 200

    cout << "size: " << v1.size() << endl;
    printV(v1);      //自己创建的打印函数
}

//push_back 会把之前准备的 10 个内存初始化为 0
```

运行结果为：size: 12

0 0 0 0 0 0 0 0 0 0 100 200

5.vector 的大小

理论知识

- ✧ vector.size(); //返回容器中元素的个数
- ✧ vector.empty(); //判断容器是否为空
- ✧ vector.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。
- ✧ vector.resize(num, elem); //重新指定容器的长度为 num，若容器变长，则以 elem 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。

例：

vecInt 是 vector<int> 声明的容器，现已包含 1,2,3 元素。

int iSize = vecInt.size(); //iSize == 3;

bool bEmpty = vecInt.empty(); // bEmpty == false;

执行 vecInt.resize(5); //此时里面包含 1,2,3,0,0 元素。

再执行 vecInt.resize(8,3); //此时里面包含 1,2,3,0,0,3,3,3 元素。

再执行 vecInt.resize(2); //此时里面包含 1,2 元素。

6.vector 末尾的添加和移除操作

```
vector<int> vecInt;
vecInt.push_back(1);  //在容器尾部加入一个元素
vecInt.push_back(3);
vecInt.push_back(5);
vecInt.push_back(7);
vecInt.push_back(9);
vecInt.pop_back();      //移除容器中最后一个元素
vecInt.pop_back();
// 结果为：1    3    5
```

7. vector 的数据存取

理论知识

- ✧ vec.at(idx); //返回索引 idx 所指的数据，如果 idx 越界，抛出 out_of_range 异常。
- ✧ vec[idx]; //返回索引 idx 所指的数据，越界时，运行直接报错

```
vector<int> vecInt;      //假设包含 1 ,3 ,5 ,7 ,9
vecInt.at(2) == vecInt[2]  ;          //5
vecInt.at(2) = 8;  或  vecInt[2] = 8;
vecInt 就包含 1, 3, 8, 7, 9 值

int iF = vector.front(); //iF==1
int iB = vector.back(); //iB==9
vector.front() = 11;    //vecInt 包含{11,3,8,7,9}
vector.back() = 19;    //vecInt 包含{11,3,8,7,19}
```

8.迭代器基本原理

- ✧ 迭代器是一个“可遍历 STL 容器内全部或部分元素”的对象。
- ✧ 迭代器指出容器中的一个特定位置。
- ✧ 迭代器就如同一个指针。
- ✧ 迭代器提供对一个容器中的对象的访问方法，并且可以定义了容器中对象的范围。
- ✧ 这里大概介绍一下迭代器的类别。

输入迭代器：也有叫法称之为“只读迭代器”，它从容器中读取元素，只能一次读入一个元素向前移动，只支持一遍算法，同一个输入迭代器不能两遍遍历一个序列。

输出迭代器：也有叫法称之为“只写迭代器”，它往容器中写入元素，只能一次写入一个元素向前移动，只支持一遍算法，同一个输出迭代器不能两遍遍历一个序列。

正向迭代器：组合输入迭代器和输出迭代器的功能，还可以多次解析一个迭代器指定的位置，可以对一个值进行多次读/写。

双向迭代器：组合正向迭代器的功能，还可以通过--操作符向后移动位置。

随机访问迭代器：组合双向迭代器的功能，还可以向前向后跳过任意个位置，可以直接访问容器中任何位置的元素。

- ✧ 目前本系列教程所用到的容器，都支持双向迭代器或随机访问迭代器，下面将会详细介绍这两个类别的迭代器。

9.双向迭代器与随机访问迭代器

双向迭代器支持的操作：

it++, ++it, it--, --it，*it， itA = itB，

itA == itB，itA != itB

其中 list,set,multiset,map,multimap 支持双向迭代器。

随机访问迭代器支持的操作：

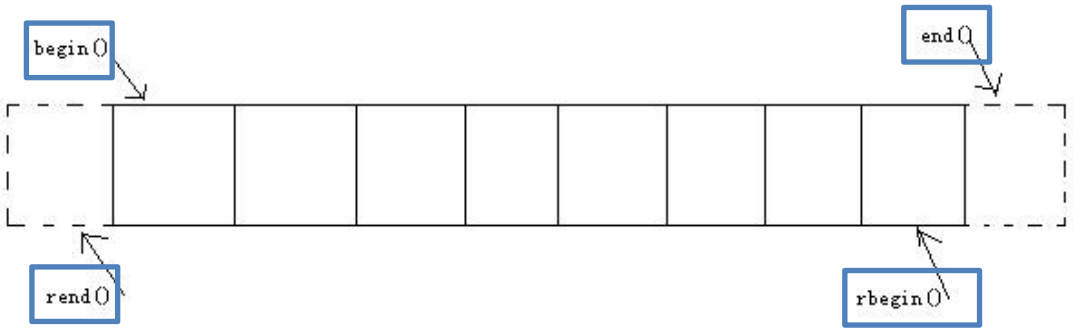
在双向迭代器的操作基础上添加

it+=i， it-=i， it+i(或 it=it+i)，it[i],

itA<itB, itA<=itB, itA>itB, itA>=itB 的功能。

其中 vector，deque 支持随机访问迭代器。

10.vector 与迭代器的配合使用



vector<int> vecInt; //定义一个容器，假设包含 1,3,5,7,9 元素

vector<int>::iterator it; //声明容器 vector<int>的迭代器。

it = vecInt.begin(); // *it == 1

++it; //或者 it++; *it == 3 ，前++的效率比后++的效率高，前++返回引用，后++返回值。

it += 2; //*it == 7

it = it+1; //*it == 9

++it; // it == vecInt.end(); 此时不能再执行*it,会出错!

```
//迭代器 end()的理解
//   1  3   5
//   ▲
//            ▲
//当 it == v1.end()的时候 说明这个容器已经遍历完毕了...
//end()的位置 应该是 5 的后面
```

正向遍历：

```
for(vector<int>::iterator it=vecInt.begin(); it!=vecInt.end(); ++it)
{
        int iItem = *it;
        cout << iItem;      //或直接使用   cout << *it;
}
```

打印出： 1 3 5 7 9

逆向遍历：

```
for(vector<int>::reverse_iterator rit=vecInt.rbegin(); rit!=vecInt.rend(); ++rit)
                                        //注意，小括号内是++rit
{
```

```
        int iItem    = *rit;
        cout << iItem;//或直接使用 cout << *rit;
}
```

此时将打印出： 9 7 5 3 1

注意：这里迭代器的声明采用 vector<int>::reverse_iterator，而非 vector<int>::iterator。

迭代器还有其它两种声明方法：

vector<int>::const_iterator 与 vector<int>::const_reverse_iterator

以上两种分别是 vector<int>::iterator 与 vector<int>::reverse_iterator 的只读形式，使用这两种迭代器时，不会修改到容器中的值。

备注：不过容器中的 insert 和 erase 方法仅接受这四种类型中的 iterator，其它三种不支持。《Effective STL》建议我们尽量使用 iterator 取代 const_iterator、reverse_iterator 和 const_reverse_iterator。

11.vector 的插入

理论知识

- vector.insert(pos,elem); //在 pos 位置插入一个 elem 元素的拷贝，返回新数据的位置。
- vector.insert(pos,n,elem); //在 pos 位置插入 n 个 elem 数据，无返回值。
- vector.insert(pos,beg,end); //在 pos 位置插入[beg,end)区间的数据，无返回值

简单案例

```
vector<int> vecA;
vector<int> vecB;
vecA.push_back(1);
vecA.push_back(3);
vecA.push_back(5);
vecA.push_back(7);
vecA.push_back(9);

vecB.push_back(2);
vecB.push_back(4);
vecB.push_back(6);
vecB.push_back(8);      //{2,4,6,8}
```

```
vecA.insert(vecA.begin(), 11);              //{11, 1, 3, 5, 7, 9}
vecA.insert(vecA.begin()+1,2,33);       //{11,33,33,1,3,5,7,9}
vecA.insert(vecA.begin(),   vecB.begin(),   vecB.end() );
//     {2,4,6,8,11,33,33,1,3,5,7,9}
```

12. vector 的删除

理论知识：

- ✧ vector.clear();　　//移除容器的所有数据
- ✧ vec.erase(beg,end);　//删除[beg,end)区间的数据，返回下一个数据的位置。
- ✧ vec.erase(pos);　　//删除 pos 位置的数据，返回下一个数据的位置。

简单案例:

```
// 删除某一段区间内的元素
vecInt 是用 vector<int>声明的容器，现已包含按顺序的 1,3,5,6,9 元素。
vector<int>::iterator itBegin=vecInt.begin()+1;
vector<int>::iterator itEnd=vecInt.begin()+2;
vecInt.erase(itBegin,itEnd);
//此时容器 vecInt 包含按顺序的 1,6,9 三个元素。

// 删除容器中等于 3 的元素
//假设 vecInt 包含 1,3,2,3,3,3,4,3,5,3，

for(vector<int>::iterator it=vecInt.being(); it!=vecInt.end(); )      //小括号里不需写
++it
{
    if(*it == 3)
    {
        it   =   vecInt.erase(it);          //以迭代器为参数，删除元素 3，并把据删除后的下一个元素位置返回给迭代器。
      }                                     //此时，我们不用再写++it，

    else
    {
```

```
        ++it;
    }
}
```

```
//删除 vecInt 的所有元素
vecInt.clear();                //容器为空
```

13.vector 小结

这一讲，主要讲解如下要点：
容器的简介，容器的分类，各个容器的数据结构

Vector，deque，list，set，multiset，map，multimap

容器 vector 的具体用法（包括迭代器的具体用法）。
vertor 简介，vector 使用之前的准备，vector 对象的默认构造，vector 末尾的添加移除操作，vector 的数据存取，迭代器的简介，双向迭代器与随机访问迭代器 vector 与迭代器的配合使用，vector 对象的带参数构造，vector 的赋值，vector 的大小，vector 的插入，vector 的删除。

10.2.3 Deque 容器

1. Deque 简介

- deque 是“double-ended queue”的缩写，和 vector 一样都是 STL 的容器，deque 是双端数组，而 vector 是单端数组。
- deque 在接口上和 vector 非常相似，在许多操作的地方可以直接替换。
- deque 可以随机存取元素（支持索引值直接存取， 用[]操作符或 at()方法，这个等下会详讲）。
- deque 头部和尾部添加或移除元素都非常快速。但是在中部安插元素或移除元素比较费时。
- #include <deque>

2.deque 对象的默认构造

- deque 采用模板类实现，deque 对象的默认构造形式：deque<T> deqT;
- deque <int> deqInt;　　//一个存放 int 的 deque 容器。
- deque <float> deq Float;　　//一个存放 float 的 deque 容器。
- deque <string> deq String;　　//一个存放 string 的 deque 容器。
- ...

//尖括号内还可以设置指针类型或自定义类型。

3. deque 对象的带参数构造

理论知识

- deque(beg,end); //构造函数将[beg, end)区间中的元素拷贝给本身。注意该区间是左闭右开的区间。
- deque(n,elem); //构造函数将 n 个 elem 拷贝给本身。
- deque(const deque &deq); //拷贝构造函数。

```
deque<int> deqIntA;
deqIntA.push_back(1);
deqIntA.push_back(3);
deqIntA.push_back(5);
deqIntA.push_back(7);
deqIntA.push_back(9);

deque<int> deqIntB(deqIntA.begin(),deqIntA.end());          //1 3 5 7 9
deque<int> deqIntC(5,8);                                    //8 8 8 8 8
deque<int> deqIntD(deqIntA);                                //1 3 5 7 9
```

4. deque 的赋值

理论知识

- deque.assign(beg,end); //将[beg, end)区间中的数据拷贝赋值给本身。注意该区间是左闭右开的区间。
- deque.assign(n,elem); //将 n 个 elem 拷贝赋值给本身。
- deque& operator=(const deque &deq); //重载等号操作符
- deque.swap(deq); // 将 vec 与本身的元素互换

```
deque<int> deqIntA,deqIntB,deqIntC,deqIntD;
deqIntA.push_back(1);
deqIntA.push_back(3);
```

```
deqIntA.push_back(5);
deqIntA.push_back(7);
deqIntA.push_back(9);

deqIntB.assign(deqIntA.begin(),deqIntA.end()); // 1 3 5 7 9

deqIntC.assign(5,8);            //8 8 8 8 8

deqIntD = deqIntA;              //1 3 5 7 9

deqIntC.swap(deqIntD);          互换
```

5. deque 头部和尾部的添加移除操作

理论知识:

- ✧ deque.push_back(elem); //在容器尾部添加一个数据
- ✧ deque.push_front(elem); //在容器头部插入一个数据
- ✧ deque.pop_back(); //删除容器最后一个数据
- ✧ deque.pop_front(); //删除容器第一个数据

```
deque<int> deqInt;
deqInt.push_back(1);
deqInt.push_back(3);
deqInt.push_back(5);
deqInt.push_back(7);
deqInt.push_back(9);
deqInt.pop_front();
deqInt.pop_front();
deqInt.push_front(11);
deqInt.push_front(13);
deqInt.pop_back();
```

```
    deqInt.pop_back();
//deqInt   { 13,11,5}
```

6. deque 的数据存取

理论知识:

- deque.at(idx); //返回索引 idx 所指的数据，如果 idx 越界，抛出 out_of_range。
- deque[idx]; //返回索引 idx 所指的数据，如果 idx 越界，不抛出异常，直接出错。
- deque.front(); //返回第一个数据。
- deque.back(); //返回最后一个数据

```
deque<int> deqInt;
deqInt.push_back(1);
deqInt.push_back(3);
deqInt.push_back(5);
deqInt.push_back(7);
deqInt.push_back(9);

int iA = deqInt.at(0);        //1
int iB = deqInt[1];           //3
deqInt.at(0) = 99;           //99
deqInt[1] = 88;               //88

int iFront = deqInt.front();//99
int iBack = deqInt.back();  //9
deqInt.front() = 77;              //77
deqInt.back() = 66;               //66
```

7.deque 与迭代器

理论知识

- deque.begin(); //返回容器中第一个元素的迭代器。
- deque.end(); //返回容器中最后一个元素之后的迭代器。

- deque.rbegin(); //返回容器中倒数第一个元素的迭代器。
- deque.rend(); //返回容器中倒数最后一个元素之后的迭代器。

```
deque<int> deqInt;
deqInt.push_back(1);
deqInt.push_back(3);
deqInt.push_back(5);
deqInt.push_back(7);
deqInt.push_back(9);

for (deque<int>::iterator it=deqInt.begin(); it!=deqInt.end(); ++it)
{
    cout << *it;
    cout << "";
}
// 1 3 5 7 9

for (deque<int>::reverse_iterator rit=deqInt.rbegin(); rit!=deqInt.rend(); ++rit)
{
    cout << *rit;
    cout << "";
}
//9 7 5 3 1
```

8. deque 的大小

理论知识

- deque.size(); //返回容器中元素的个数 int 型变量
- deque.empty(); //判断容器是否为空 返回 ture 和 false
- deque.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。
- deque.resize(num, elem); //重新指定容器的长度为 num，若容器变长，则以 elem 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。

```
deque<int> deqIntA;
deqIntA.push_back(1);
deqIntA.push_back(3);
deqIntA.push_back(5);

int iSize = deqIntA.size();    //3

if (!deqIntA.empty())
{
    deqIntA.resize(5);          //1 3 5 0 0
    deqIntA.resize(7,1);   //1 3 5 0 0 1 1
    deqIntA.resize(2);          //1 3
}
```

9. deque 的插入

理论知识

- ✧ deque.insert(pos,elem); //在 pos 位置插入一个 elem 元素的拷贝，返回新数据的位置。
- ✧ deque.insert(pos,n,elem); //在 pos 位置插入 n 个 elem 数据，无返回值。
- ✧ deque.insert(pos,beg,end); //在 pos 位置插入[beg,end)区间的数据，无返回值。

```
deque<int> deqA;
deque<int> deqB;

deqA.push_back(1);
deqA.push_back(3);
deqA.push_back(5);
deqA.push_back(7);
deqA.push_back(9);

deqB.push_back(2);
deqB.push_back(4);
```

```
deqB.push_back(6);
deqB.push_back(8);

deqA.insert(deqA.begin(), 11);      // {11, 1, 3, 5, 7, 9}
deqA.insert(deqA.begin()+1,2,33);       // {11,33,33,1,3,5,7,9}
deqA.insert(deqA.begin(),deqB.begin(),deqB.end());
//{2,4,6,8,11,33,33,1,3,5,7,9}
```

10. deque 的删除

理论知识

- deque.clear(); //移除容器的所有数据
- deque.erase(beg,end); //删除[beg,end)区间的数据，返回下一个数据的位置。
- deque.erase(pos); //删除 pos 位置的数据，返回下一个数据的位置。

//删除区间内的元素

deqInt 是用 deque<int>声明的容器，现已包含按顺序的 1,3,5,6,9 元素。

```
deque<int>::iterator itBegin=deqInt.begin()+1;
deque<int>::iterator itEnd=deqInt.begin()+3;
deqInt.erase(itBegin,itEnd);
//此时容器 deqInt 包含按顺序的 1,6,9 三个元素。
```

//删除容器中的指定元素

假设 deqInt 包含 1,3,2,3,3,3,4,3,5,3，删除容器中等于 3 的元素

```
for(deque<int>::iterator it=deqInt.being(); it!=deqInt.end(); )     //小括号里不需写  ++it
{
   if(*it == 3)
   {
        it  =  deqInt.erase(it);        //以迭代器为参数，删除元素 3，并把数据删除                                后的下一个元素位置返回给迭代器。
                                        //此时，不用写++it;
   }
   else
```

```
    {
        ++it;
    }
}
```

```
//删除 deqInt 的所有元素
deqInt.clear();                //容器为空
```

11. deque 中查找某个数在数组下标的值

```
    deque<int>::iterator it = find(d1.begin(), d1.end(), -33);  //find 为一个
算法
    if (it != d1.end())
    {
    cout << "-33 数组下标是" << distance(d1.begin(), it) << endl;
                    //distance 为求偏移量的函数。    注意数组下标是从 0 开始的
    }
    else
    {
        cout << "没有找到值为-33 的元素" << endl;
    }
  //返回的是第一个值为-33 的元素的位置
```

10.2.4 stack 容器

stack 简介

- ✧ stack 是堆栈容器，是一种“先进后出”的容器。
- ✧ stack 是简单地装饰 deque 容器而成为另外的一种容器。
- ✧ #include <stack>

1. stack 对象的默认构造

stack 采用模板类实现， stack 对象的默认构造形式： stack <T> stkT;

```
stack <int> stkInt;              //一个存放 int 的 stack 容器。
stack <float> stkFloat;          //一个存放 float 的 stack 容器。
stack <string> stkString;        //一个存放 string 的 stack 容器。
...
//尖括号内还可以设置指针类型或自定义类型。
```

2. stack 对象的拷贝构造与赋值

stack(const stack &stk);　　//拷贝构造函数

stack& operator=(const stack &stk);　//重载等号操作符

```
stack<int> stkIntA;
stkIntA.push(1);
stkIntA.push(3);
stkIntA.push(5);
stkIntA.push(7);
stkIntA.push(9);

stack<int> stkIntB(stkIntA);    //拷贝构造
stack<int> stkIntC;
stkIntC = stkIntA;              //赋值
```

3. stack 的 push( )与 pop( )方法

stack.push(elem);　　//往栈头添加元素

stack.pop();　　//从栈头移除第一个元素

```
stack<int> stkInt;
stkInt.push(1);stkInt.push(3);stkInt.pop();
stkInt.push(5);stkInt.push(7);
stkInt.push(9);stkInt.pop();
stkInt.pop();
```

此时 stkInt 存放的元素是 1,5

4. stack 的数据存取

- stack.top();　//返回最后一个压入栈元素

```
stack<int> stkIntA;
stkIntA.push(1);
stkIntA.push(3);
stkIntA.push(5);
```

```
        stkIntA.push(7);
        stkIntA.push(9);

        int iTop = stkIntA.top();       //9
        stkIntA.top() = 19;          //19

//stack 容器中放入普通数据型元素
void main51()
{
    stack<int> s;
    //入栈
    for (int i = 0; i<10; i++)
    {
        s.push(i + 1);
    }
    cout << "栈的大小" << s.size() << endl;

    //出栈
    while (!s.empty())
    {
        int tmp = s.top();     //获取栈顶元素
        cout <<tmp << " ";
        s.pop();         //弹出栈顶元素
    }
}

//stack 容器中放入类的对象元素
void main52()
{
    Teacher t1, t2, t3;  //定义三个类的对象
    t1.age = 31;
    t2.age = 32;
    t3.age = 33;

    stack<Teacher> s;
    s.push(t1);    //往栈中放入 类对象元素
    s.push(t2);
    s.push(t3);
    while (!s.empty())
    {
        Teacher tmp = s.top();
        tmp.printT();    //打印栈中放入的对象的年龄
        s.pop();
```

```
    }
}

//stack 容器中放入类的对象的指针元素
void main53()
{
    Teacher t1, t2, t3;
    t1.age = 31;
    t2.age = 32;
    t3.age = 33;
    stack<Teacher *> s;     //栈中放入类的指针
    s.push(&t1);
    s.push(&t2);
    s.push(&t3);

    while (!s.empty())
    {
        Teacher *p = s.top();
        p->printT();
        s.pop();
    }
}
```

5. stack 的大小

- stack.empty();　　//判断堆栈是否为空
- stack.size();　　//返回堆栈的大小

```
    stack<int> stkIntA;
    stkIntA.push(1);
    stkIntA.push(3);
    stkIntA.push(5);
    stkIntA.push(7);
    stkIntA.push(9);

    if (!stkIntA.empty())
    {
        int iSize = stkIntA.size();     //5
    }
```

10.2.5 Queue 容器

1. Queue 简介

- ✧ queue 是队列容器，是一种“先进先出”的容器。
- ✧ queue 是简单地装饰 deque 容器而成为另外的一种容器。
- ✧ #include <queue>

2. queue 对象的默认构造

queue 采用模板类实现，queue 对象的默认构造形式：queue<T> queT; 如：

queue<int> queInt; //一个存放 int 的 queue 容器。

queue<float> queFloat; //一个存放 float 的 queue 容器。

queue<string> queString; //一个存放 string 的 queue 容器。

…

//尖括号内还可以设置指针类型或自定义类型。

3. queue 的 push()与 pop()方法

queue.push(elem); //往队尾添加元素

queue.pop(); //从队头移除第一个元素

```
queue<int> queInt;
queInt.push(1);queInt.push(3);
queInt.push(5);queInt.push(7);
queInt.push(9);queInt.pop();
queInt.pop();
```

此时 queInt 存放的元素是 5,7,9

4. queue 对象的拷贝构造与赋值

- ✧ queue(const queue &que); //拷贝构造函数
- ✧ queue& operator=(const queue &que); //重载等号操作符

```
queue<int> queIntA;
queIntA.push(1);
```

```
queIntA.push(3);
queIntA.push(5);
queIntA.push(7);
queIntA.push(9);

queue<int> queIntB(queIntA);        //拷贝构造
queue<int> queIntC;
queIntC = queIntA;                  //赋值
```

5. queue 的数据存取

- ✧ queue.back();　　//返回最后一个元素
- ✧ queue.front();　　//返回第一个元素

```
queue<int> queIntA;
queIntA.push(1);
queIntA.push(3);
queIntA.push(5);
queIntA.push(7);
queIntA.push(9);

int iFront = queIntA.front();       //1
int iBack = queIntA.back();     //9

queIntA.front() = 11;               //11
queIntA.back() = 19;          //19
```

6. queue 的大小

- ✧ queue.empty();　　//判断队列是否为空
- ✧ queue.size();　　//返回队列的大小

```
queue<int> queIntA;
queIntA.push(1);
queIntA.push(3);
queIntA.push(5);
```

```
    queIntA.push(7);
    queIntA.push(9);

    if (!queIntA.empty())
    {
        int iSize = queIntA.size();     //5
}
```

10.2.6 List 容器

1. List 简介

- ✧ list 是一个双向链表容器，可高效地进行插入删除元素。
- ✧ list 不可以随机存取元素，所以不支持 at.(pos)函数与[]操作符。 It++(ok) it+5(err)
- ✧ #include <list>

2. list 对象的默认构造

list 采用采用模板类实现,对象的默认构造形式：list<T> lstT; 如：

```
list<int> lstInt;                    //定义一个存放 int 的 list 容器。
list<float> lstFloat;          //定义一个存放 float 的 list 容器。
list<string> lstString;          //定义一个存放 string 的 list 容器。
...
//尖括号内还可以设置指针类型或自定义类型。
```

3. list 对象的带参数构造

- ✧ list(beg,end); //构造函数将[beg, end)区间中的元素拷贝给本身。注意该区间是左闭右开的区间。
- ✧ 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。

```
list<int> lstIntA;
lstIntA.push_back(1);
lstIntA.push_back(3);
```

```
lstIntA.push_back(5);
lstIntA.push_back(7);
lstIntA.push_back(9);

list<int> lstIntB(lstIntA.begin(),lstIntA.end());     //1 3 5 7 9
list<int> lstIntC(5,8);                               //8 8 8 8 8
list<int> lstIntD(lstIntA);                           //1 3 5 7 9
```

4. list 的赋值

- ✧ list.assign(beg,end);　　//将[beg, end)区间中的数据拷贝赋值给本身。注意该区间是左闭右开的区间。
- ✧ list.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
- ✧ list.swap(lst);　　//将 lst 与本身的元素互换。

```
list<int> lstIntA,lstIntB,lstIntC,lstIntD;
lstIntA.push_back(1);
lstIntA.push_back(3);
lstIntA.push_back(5);
lstIntA.push_back(7);
lstIntA.push_back(9);

lstIntB.assign(lstIntA.begin(),lstIntA.end());        //1 3 5 7 9
lstIntC.assign(5,8);                                  //8 8 8 8 8
lstIntD = lstIntA;                                    //1 3 5 7 9
lstIntC.swap(lstIntD);                                //互换
```

5. list 头尾的添加移除操作

- ✧ list.push_back(elem);　//在容器尾部加入一个元素
- ✧ list.pop_back();　　//删除容器中最后一个元素
- ✧ list.push_front(elem);　　//在容器开头插入一个元素

- ✧ list.pop_front(); //从容器开头移除第一个元素

```
list<int> lstInt;
lstInt.push_back(1);
lstInt.push_back(3);
lstInt.push_back(5);
lstInt.push_back(7);
lstInt.push_back(9);
lstInt.pop_front();
lstInt.pop_front();
lstInt.push_front(11);
lstInt.push_front(13);
lstInt.pop_back();
lstInt.pop_back();
// lstInt    {13,11,5}
```

6. list 的数据存取

- ✧ list.front(); //返回第一个元素。
- ✧ list.back(); //返回最后一个元素。

```
list<int> lstInt;
lstInt.push_back(1);
lstInt.push_back(3);
lstInt.push_back(5);
lstInt.push_back(7);
lstInt.push_back(9);

int iFront = lstInt.front();        //1
int iBack = lstInt.back();      //9
lstInt.front() = 11;          //11
lstInt.back() = 19;           //19
```

7. list 与迭代器

- ✧ list.begin(); //返回容器中第一个元素的迭代器。

- list.end(); //返回容器中最后一个元素之后的迭代器。
- list.rbegin(); //返回容器中倒数第一个元素的迭代器。
- list.rend(); //返回容器中倒数最后一个元素的后面的迭代器。

```
list<int> lstInt;
lstInt.push_back(1);
lstInt.push_back(3);
lstInt.push_back(5);
lstInt.push_back(7);
lstInt.push_back(9);

for (list<int>::iterator it=lstInt.begin(); it!=lstInt.end(); ++it)
{
    cout << *it;
    cout << " ";
}
for(list<int>::reverse_iterator rit=lstInt.rbegin(); rit!=lstInt.rend();
++rit)
{
    cout << *rit;
    cout << " ";
}
```

8. list 的大小

- list.size(); //返回容器中元素的个数
- list.empty(); //判断容器是否为空
- list.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。
- list.resize(num, elem); //重新指定容器的长度为 num，若容器变长，则以 elem 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。

```
list<int> lstIntA;
lstIntA.push_back(1);
lstIntA.push_back(3);
lstIntA.push_back(5);

if (!lstIntA.empty())
{
    int iSize = lstIntA.size();     //3
    lstIntA.resize(5);              //1 3 5 0 0
    lstIntA.resize(7,1);            //1 3 5 0 0 1 1
    lstIntA.resize(2);              //1 3
}
```

9. list 的插入

- ✧ list.insert(pos,elem);　　//在 pos 位置插入一个 elem 元素的拷贝，返回新数据的位置。
- ✧ list.insert(pos,n,elem);　　//在 pos 位置插入 n 个 elem 数据，无返回值。
- ✧ list.insert(pos,beg,end);　　//在 pos 位置插入[beg,end)区间的数据，无返回值。

注意：

```
    //在 3 号位置插入元素，原来的 3 号位置变成 4 号位置  原来的 4 号位置变成 5 号位置

//list 不能随机访问
    //0  1  2   3   4   5
    //             ▲
    it = l.begin();
    it++;
    it++;              //list 只能逐个的进行访问
    it++;
    // it = it + 3;    // 不支持随机的访问容器
    l.insert(it, 100); //请问 100 插入在什么位置
    for (list<int>::iterator it = l.begin(); it != l.end(); it++)
    {
```

```
        cout << *it << " ";
    }
    list<int> lstA;
    list<int> lstB;

    lstA.push_back(1);
    lstA.push_back(3);
    lstA.push_back(5);
    lstA.push_back(7);
    lstA.push_back(9);

    lstB.push_back(2);
    lstB.push_back(4);
    lstB.push_back(6);
    lstB.push_back(8);

    lstA.insert(lstA.begin(), 11);            //{11, 1, 3, 5, 7, 9}
    lstA.insert(++lstA.begin(),2,33);       //{11,33,33,1,3,5,7,9}
    lstA.insert(lstA.begin() , lstB.begin() , lstB.end() );
    //{2,4,6,8,11,33,33,1,3,5,7,9}
```

10. list 的删除

- ✧ list.clear();　　//移除容器的所有数据
- ✧ list.erase(beg,end); //删除[beg,end)区间的数据，返回下一个数据的位置。
- ✧ list.erase(pos);　　//删除 pos 位置的数据，返回下一个数据的位置。
- ✧ lst.remove(elem);　//删除容器中所有与 elem 值匹配的元素。

//删除区间内的元素

lstInt 是用 list<int>声明的容器，现已包含按顺序的 1,3,5,6,9 元素。

```
list<int>::iterator itBegin=lstInt.begin();
++ itBegin;
list<int>::iterator itEnd=lstInt.begin();
++ itEnd;
```

```
++ itEnd;
++ itEnd;
lstInt.erase(itBegin,itEnd);
//此时容器 lstInt 包含按顺序的 1,6,9 三个元素。
```

// 删除指定的元素

假设 lstInt 包含 1,3,2,3,3,3,4,3,5,3，删除容器中等于 3 的元素的方法一

```
for(list<int>::iterator it=lstInt.being(); it!=lstInt.end(); )//小括号里不需写  ++it
{
   if(*it == 3)
   {
        it  =  lstInt.erase(it);  //以迭代器为参数，删除元素 3，并把数据删除后的下一个元素位置返回给迭代器。
                                  //此时，自动会++it;
   }
   else
   {
       ++it;
   }
}
```

删除容器中等于 3 的元素的方法二

```
lstInt.remove(3);
```

//删除 lstInt 的所有元素

```
lstInt.clear();     //容器为空
```

11. list 的反序排列

- ✧ lst.reverse();　　//反转链表，比如 lst 包含 1,3,5 元素，运行此方法后，lst 就包含 5,3,1 元素。

```
list<int> lstA;

lstA.push_back(1);
lstA.push_back(3);
lstA.push_back(5);
```

```
lstA.push_back(7);
lstA.push_back(9);

lstA.reverse();          //9 7 5 3 1
```

小结:

✧ 一、容器 deque 的使用方法
适合　在头尾添加移除元素。使用方法与 vector 类似。
✧ 二、容器 queue,stack 的使用方法
适合队列，堆栈的操作方式。
✧ 三、容器 list 的使用方法
适合在任意位置快速插入移除元素

10.2.7 优先级队列 priority_queue

- 最大值优先级队列、最小值优先级队列　//谁的值最大 或 最小 放在队头
- 优先级队列适配器 STL priority_queue
- 用来开发一些特殊的应用,请对 stl 的类库,多做扩展性学习

定义：

```
priority_queue<int, deque<int>>      pq;

priority_queue<int, vector<int>>      pq;
```

常用的函数：

- pq.empty()
- pq.size()
- pq.top()
- pq.pop()
- pq.push(item)

头文件：

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

void main81()
{
```

```
    priority_queue<int> p1;        //默认是最大值优先级队列
       //priority_queue<int, vector<int>, less<int> > p1; //相当于这样写

    priority_queue<int, vector<int>, greater<int>> p2; //最小值优先级队列
                                    //默认的是 vector 迭代器进行适配的

    p1.push(33);
    p1.push(11);
    p1.push(55);
    p1.push(22);
    cout <<"队列大小" << p1.size() << endl;
    cout <<"队头" << p1.top() << endl;

    while (p1.size() > 0)
    {
        cout << p1.top() << " ";
        p1.pop();
    }
    cout << endl;

    cout << "测试 最小值优先级队列" << endl;
    p2.push(33);
    p2.push(11);
    p2.push(55);
    p2.push(22);
    while (p2.size() > 0)
    {
        cout << p2.top() << " ";
        p2.pop();
    }
}
```

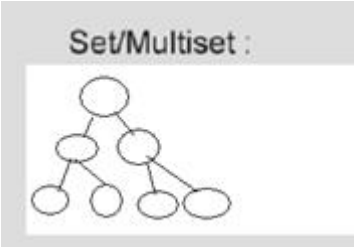
10.2.8 Set 和 multiset 容器

1. set/multiset 的简介

✧ set 是一个集合容器，其中所包含的元素是唯一的，集合中的元素按一定

的顺序排列。**元素插入过程是按排序规则插入**，所以不能指定插入位置。

- ✧ set 采用**红黑树**变体的数据结构实现，红黑树属于平衡二叉树。在插入操作和删除操作上比 vector 快。
- ✧ set 不可以直接存取元素。（不可以使用 at.(pos)与[]操作符）。
- ✧ multiset 与 set 的区别：set 支持唯一键值，每个元素值只能出现一次；而 multiset 中**同一值可以出现多次**。
- ✧ 不可以直接修改 set 或 multiset 容器中的元素值，因为该类容器是自动排序的。如果希望修改一个元素值，必须先删除原有的元素，再插入新的元素。
- ✧ #include <set>
- ✧




```
    set<int> set1;
    for (int i = 0; i<5; i++)
    {
        int tmp = rand();     //随机数
        set1.insert(tmp);
    }

    set1.insert(100);        //多次插入 100  在最后的结果中只会显示一个
    set1.insert(100);
    set1.insert(100);

for (set<int>::iterator it = set1.begin(); it != set1.end(); it++)
    {
        cout << *it << " ";
    }
    cout << endl;
```

```
选择C:\Users\CHENSHUAI\Desktop\vs2013 程序\de
41 100 6334 18467 19169 26500
hello...
请按任意键继续. . .
```

2. set/multiset 对象的默认构造

```
set<int> setInt;                    //一个存放 int 的 set 容器。
set<float> setFloat;                //一个存放 float 的 set 容器。
set<string> setString;              //一个存放 string 的 set 容器。
multiset<int> mulsetInt;            //一个存放 int 的 multi set 容器。
multi set<float> multisetFloat;     //一个存放 float 的 multi set 容器。
multi set<string> multisetString;   //一个存放 string 的 multi set 容器。
```

3. set 的插入与迭代器

- ✧ set.insert(elem); //在容器中插入元素。
- ✧ set.begin(); //返回容器中第一个数据的迭代器。
- ✧ set.end(); //返回容器中最后一个数据之后的迭代器。
- ✧ set.rbegin(); //返回容器中倒数第一个元素的迭代器。
- ✧ set.rend(); //返回容器中倒数最后一个元素的后面的迭代器。

```
set<int> setInt;  // set<int> 相当于 set<int,less<int>>。按照升序的方式排列
setInt.insert(3);
setInt.insert(1);
setInt.insert(5);
setInt.insert(2);
for(set<int>::iterator it=setInt.begin(); it!=setInt.end(); ++it)
{
      int iItem = *it;
      cout << iItem;       //或直接使用 cout << *it
}
//这样便顺序输出  1 2 3 5。
```

set.rbegin()与 set.rend()。略。

4. Set 集合的元素排序

- ✧ set<int,less<int> > setIntA; //该容器是按升序方式排列元素。

- ✧ set<int,greater<int>> setIntB;　　//该容器是按降序方式排列元素。
- ✧ set<int> 相当于 set<int,less<int>>。
- ✧ less<int>与 greater<int>中的 int 可以改成其它类型，该类型主要要跟 set 容纳的数据类型一致。
- ✧ 疑问 1：less<>与 greater<>是什么？
- ✧ 疑问 2：如果 set<>不包含 int 类型，而是包含自定义类型，set 容器如何排序？
- ✧ 要解决如上两个问题，需要了解容器的函数对象，也叫伪函数，英文名叫 functor。
- ✧ 下面将讲解什么是 functor，functor 的用法。

使用 stl 提供的函数对象

```
set<int,greater<int>> setIntB;
setIntB.insert(3);
setIntB.insert(1);
setIntB.insert(5);
setIntB.insert(2);
```

此时容器 setIntB 就包含了按顺序的 5,3,2,1 元素

5. 函数对象 functor 的用法　（自定义的数据的排序）

- ✧ 尽管函数指针被广泛用于实现函数回调，但 C++还提供了一个重要的实现回调函数的方法，那就是函数对象。
- ✧ functor，翻译成函数对象，伪函数，算符，是重载了“()”操作符的普通类对象。从语法上讲，它与普通函数行为类似。
- ✧ greater<>与 less<>就是 stl 内库自己提供函数对象。
- ✧ 下面举出 greater<int>的简易实现原理。

下面举出 greater<int>的简易实现原理。

```
struct greater
{
    bool operator() (const int& iLeft, const int& iRight)
    {
        return (iLeft>iRight);        //如果是实现 less<int>的话，这边是写 return
                                                    (iLeft<iRight);
    }
}
```

容器就是调用函数对象的 operator()方法去比较两个值的大小。

题目：学生包含学号，姓名属性，现要求任意插入几个学生对象到 set 容器中，使得容器中的学生按学号的升序排序。

解：

```
//学生类
class CStudent
{
    public:
        CStudent(int iID, string strName)
        {
            m_iID = iID;
            m_strName = strName;
        }
      int m_iID;                //学号
      string m_strName;         //姓名
}
```

//为保持主题鲜明，本类不写拷贝构造函数，不过也不需要写拷贝构造函数。但大家仍要有考虑拷贝构造函数的习惯。

```
//函数对象
struct StuFunctor    //自己规定的排序方式
{
        bool operator()    (const CStudent &stu1, const CStudent &stu2)
        {
            return (stu1.m_iID<stu2.m_iID);          //返回的是 真 和 假
        }
}
```

//set 容器按照比较的成员的唯一性进行插入数据的

```
//main 函数
void main()
{
    set<CStudent, StuFunctor> setStu;     //注意与普通数据类型构造是相同的。
  //当插入类对象时候，容器会按照我们已经自己规定的排序方式进行排序
        setStu.insert(CStudent(3,"小张"));
        setStu.insert(CStudent(1,"小李"));
        setStu.insert(CStudent(5,"小王"));
        setStu.insert(CStudent(2,"小刘"));
```

//此时容器 setStu 包含了四个学生对象，分别是按姓名顺序的“小李”，“小刘”，

“小张”，“小王”

//如果有两个人的学号的相同的，则第二个人的学号不能插入成功

```
//打印输出
    for (set<Student, StuFunctor >::iterator it = set1.begin(); it != set1.end(); it++)
    {
        cout << (*it).name<< " ";      //不能 it,    *it 是对象
    }
    system("pause");
    return 0;
}
```

另一个：

```
//typedef pair<iterator, bool> _Pairib;
//4 如何判断 set1.insert 函数的返回值
//Pair 的用法
void main94()
{
    Student s1("s1", 31);
    Student s2("s2", 22);
    Student s3("s3", 44);
    Student s4("s4", 11);
    Student s5("s5", 31);
    set<Student, FuncStudent> set1;

    pair<set<Student, FuncStudent>::iterator, bool> pair1 = set1.insert(s1);
    if (pair1.second == true)
    {
        cout << "插入 s1 成功" << endl;
    }
    else
    {
        cout << "插入 s1 失败" << endl;
    }

    set1.insert(s2);

//如何知道 插入 的结果
    pair<set<Student, FuncStudent>::iterator, bool> pair5 = set1.insert(s5);
//如果两个 31 岁 能插入成功
    if (pair5.second == true)
    {
```

```
        cout << "插入 s1 成功" << endl;
    }
    else
    {
        cout << "插入 s1 失败" << endl;
    }

//遍历
    for (set<Student, FuncStudent>::iterator it=set1.begin(); it!=set1.end();
it++ )
    {
        cout << it->age << "\t" <<  it->name << endl;
    }
}
```

6. set 对象的拷贝构造与赋值

- ✧ set(const set &st);　　//拷贝构造函数
- ✧ set& operator=(const set &st);　//重载等号操作符
- ✧ set.swap(st);　　//交换两个集合容器

```
set<int> setIntA;
setIntA.insert(3);
setIntA.insert(1);
setIntA.insert(7);
setIntA.insert(5);
setIntA.insert(9);

set<int> setIntB(setIntA);  //1 3 5 7 9

set<int> setIntC;
setIntC = setIntA;          //1 3 5 7 9

setIntC.insert(6);
setIntC.swap(setIntA);     //交换
```

7. set 的大小

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

```
set<int> setIntA;
setIntA.insert(3);
setIntA.insert(1);
setIntA.insert(7);
setIntA.insert(5);
setIntA.insert(9);

if (!setIntA.empty())
{
    int iSize = setIntA.size();   //5
}
```

8. set 的删除

- ✧ set.clear();　　//清除所有元素
- ✧ set.erase(pos);　//删除 pos 迭代器所指的元素，返回下一个元素的迭代器。
- ✧ set.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。
- ✧ set.erase(elem);　　//删除容器中值为 elem 的元素。

//删除区间内的元素

setInt 是用 set<int>声明的容器，现已包含按顺序的 1,3,5,6,9,11 元素。

```
set<int>::iterator itBegin=setInt.begin();
++ itBegin;
set<int>::iterator itEnd=setInt.begin();
++ itEnd;
++ itEnd;
++ itEnd;
setInt.erase(itBegin,itEnd);
```

```
//此时容器 setInt 包含按顺序的 1,6,9,11 四个元素。

//删除容器中第一个元素
setInt.erase(setInt.begin());   //6,9,11

//删除容器中值为 9 的元素
set.erase(9);

//删除 setInt 的所有元素
setInt.clear();             //容器为空
```

9. set 的查找

- ✧ set.find(elem); //查找 elem 元素，返回指向 elem 元素的迭代器。
- ✧ set.count(elem); //返回容器中值为 elem 的元素个数。对 set 来说，要么是 0，要么是 1。对 multiset 来说，值可能大于 1。
- ✧ set.lower_bound(elem); //返回第一个>=elem 元素的迭代器。
- ✧ set.upper_bound(elem); //返回第一个>elem 元素的迭代器。
- ✧ set.equal_range(elem); //返回容器中与 elem 相等的上下限的两个迭代器。上限是闭区间，下限是开区间，如[beg,end)。
  以上函数返回两个迭代器，而这两个迭代器被封装在 pair 中。
- ✧ 以下讲解 pair 的含义与使用方法。

```
set<int> setInt;
setInt.insert(3);
setInt.insert(1);
setInt.insert(7);
setInt.insert(5);
setInt.insert(9);

set<int>::iterator itA = setInt.find(5);
int iA = *itA;              //iA == 5
int iCount = setInt.count(5);   //iCount == 1

set<int>::iterator itB = setInt.lower_bound(5);
```

```
set<int>::iterator itC = setInt.upper_bound(5);
int iB = *itB;      //iB == 5
int iC = *itC;      //iC == 7
```

pair< set<int>::iterator, set<int>::iterator > pairIt = setInt.equal_range(5); //pair 是什么？

10. pair 的使用

- ✧ pair 译为对组，可以将两个值视为一个单元。
- ✧ pair<T1,T2>存放的两个值的类型，可以不一样，如 T1 为 int，T2 为 float。T1,T2 也可以是自定义类型。
- ✧ pair.first 是 pair 里面的第一个值，是 T1 类型。
- ✧ pair.second 是 pair 里面的第二个值，是 T2 类型。

```
set<int> setInt;
...  //往 setInt 容器插入元素 1,3,5,7,9
pair< set<int>::iterator , set<int>::iterator > pairIt = setInt.equal_range(5);
set<int>::iterator itBeg = pairIt.first;
set<int>::iterator itEnd = pairIt.second;
//此时 *itBeg==5  而  *itEnd == 7
```

小结

- ✧ 一、容器 set/multiset 的使用方法；
- ✧ 红黑树的变体，查找效率高，插入不能指定位置，插入时自动排序。
- ✧ 二、functor 的使用方法；
- ✧ 类似于函数的功能，可用来自定义一些规则，如元素比较规则。
- ✧ 三、pair 的使用方法。
- ✧ 对组，一个整体的单元，存放两个类型(T1,T2，T1 可与 T2 一样)的两个元素。

案例:

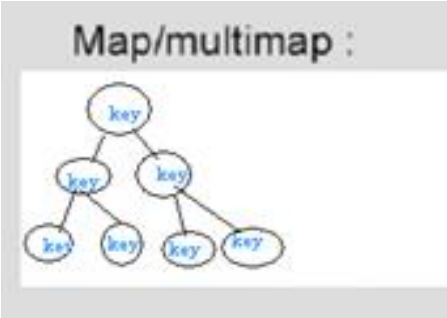
```
//遍历
第一种遍历的方法：使用迭代器
    for (multiset<int>::iterator it = set1.begin(); it != set1.end(); it++)
    {
        cout << *it << " ";
    }
```

```
    cout << endl;

第二种遍历的方法：使用函数 和 迭代器
    while (!set1.empty())
    {
        multiset<int>::iterator it = set1.begin();
        cout << *it << " ";
        set1.erase(it);
    }
}
```

10.2.9 Map 和 multimap 容器

1. map/multimap 的简介

- map 是标准的关联式容器，一个 map 是一个键值对序列，即(key,value)对。它提供基于 key 的快速检索能力。
- map 中 key 值是唯一的。集合中的元素按一定的顺序排列。元素插入过程是按排序规则插入，所以不能指定插入位置。
- map 的具体实现采用红黑树变体的平衡二叉树的数据结构。在插入操作和删除操作上比 vector 快。
- map 可以直接存取 key 所对应的 value，支持[]操作符，如 map[key]=value。
- multimap 与 map 的区别：map 支持唯一键值，每个键只能出现一次；而 multimap 中相同键可以出现多次。multimap 不支持[ ]操作符。
- #include <map>



2. map/multimap 对象的默认构造

map/multimap 采用模板类实现，对象的默认构造形式：

```
map<T1,T2> mapTT;
multimap<T1,T2>  multimapTT;
```

如：

```
map<int, char> mapA;
```

```
map<string,float> mapB;
//其中 T1,T2 还可以用各种指针类型或自定义类型
```

3. map 对象的拷贝构造与赋值

- ✧ map(const map &mp);　　//拷贝构造函数
- ✧ map& operator=(const map &mp);　//重载等号操作符
- ✧ map.swap(mp);　　//交换两个集合容器

例如:

```
map<int, string> mapA;
mapA.insert(pair<int,string>(3,"小张"));
mapA.insert(pair<int,string>(1,"小杨"));
mapA.insert(pair<int,string>(7,"小赵"));
mapA.insert(pair<int,string>(5,"小王"));

map<int ,string> mapB(mapA);　//拷贝构造

map<int, string> mapC;
mapC = mapA;　　//赋值

mapC[3] = "老张";
mapC.swap(mapA);　　//交换
```

4.map 的插入与迭代器

- ✧ map.insert(...);　　//往容器插入元素，返回 pair<iterator,bool>
- ✧ 在 map 中插入元素的三种方式：
- ✧ 假设　map<int, string> mapStu;
- ✧ 一、通过 pair(对组)的方式插入对象

```
mapStu.insert(　pair<int,string>(3,"小张") );
```

- ✧ 二、通过 pair 的方式插入对象

```
mapStu.inset(make_pair(-1, “校长-1”));
```

- ✧ 三、通过 value_type 的方式插入对象

```
mapStu.insert( map<int,string>::value_type(1,"小李") );
```

- 四、通过数组的方式插入值

  mapStu[3] = “小刘";

  mapStu[5] = “小王";

- 前三种方法，采用的是 insert()方法，该方法返回值为 pair<iterator,bool>
- //前三种方法 返回值为 pair<iterator,bool> 若 key 已经存在 则报错
- 方法四 若 key 已经存在,则修改
- 第四种方法非常直观，但存在一个性能的问题。插入 3 时，先在 mapStu 中查找主键为 3 的项，若没发现，则将一个键为 3，值为初始化值的对组插入到 mapStu 中，然后再将值修改成“小刘”。若发现已存在 3 这个键，则修改这个键对应的 value。
- string strName = mapStu[2]; //取操作或插入操作
- 只有当 mapStu 存在 2 这个键时才是正确的取操作，否则会自动插入一个实例，键为 2，值为初始化值。

假设 map<int, string> mapA;

//插入方式 1：利用插入函数通过对组方式插入

pair< map<int,string>::iterator, bool > pairResult = **mapA.insert(pair<int,string>(3,"小张"));**

int iFirstFirst = (pairResult.first)->first; //iFirst == 3;

string strFirstSecond = (pairResult.first)->second; //strFirstSecond 为"小张"

bool bSecond = pairResult.second; //bSecond == true;

另一个程序：

```
pair<map<int, string>::iterator, bool>mypair5 = map1.insert(map<int, string>::value_type(5, "teacher05"));
if (mypair5.second != true)
{   cout << "key 5  插入失败" << endl;   }
else
{   cout << mypair5.first->first << "\t" << mypair5.first->second << endl;   }
```

//前 3 种插入方法可以用来判断是否插入成功。

//插入方式 2：

```
mapA.insert(make_pair(3, "teacher04")) ;
mapA.insert(make_pair(4, "teacher05")) ;
```

//插入方式 3：

```
mapA.insert(map<int,string>::value_type(1,"小李"));
```

//插入方式 4：

```
mapA[3] = "小刘";                //修改 value
mapA[5] = "小王";
```

string str1 = mapA[2]; // 执行插入 string() 操作，返回的 str1 的字符串内容为空。

string str2 = mapA[3]; // 取得 value，str2 为"小刘"

//用迭代器遍历

```
for (map<int,string>::iterator it=mapA.begin(); it!=mapA.end(); ++it)
{
    pair<int, string> pr = *it;
    int iKey = pr.first;                可以直接用 it->first 和 it->second
    string strValue = pr.second;
}
```

map.rbegin()与 map.rend() 略。

- ✧ map<T1,T2,less<T1> > mapA; //该容器是按值的升序方式排列元素。未指定函数对象，默认采用 less<T1>函数对象。
- ✧ map<T1,T2,greater<T1>> mapB; //该容器是按键值的降序方式排列元素。
- ✧ less<T1>与 greater<T1> 可以替换成其它的函数对象 functor。
- ✧ 可编写自定义函数对象以进行自定义类型的比较，使用方法与 set 构造时所用的函数对象一样。
- ✧ map.begin(); //返回容器中第一个数据的迭代器。
- ✧ map.end(); //返回容器中最后一个数据之后的迭代器。
- ✧ map.rbegin(); //返回容器中倒数第一个元素的迭代器。
- ✧ map.rend(); //返回容器中倒数最后一个元素的后面的迭代器。

5. map 的大小

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

```
map<int, string> mapA;
mapA.insert(pair<int,string>(3,"小张"));
mapA.insert(pair<int,string>(1,"小杨"));
mapA.insert(pair<int,string>(7,"小赵"));
mapA.insert(pair<int,string>(5,"小王"));
if (mapA.empty())
{
    int iSize = mapA.size();        //iSize == 4
}
```

5. map 的删除

- map.clear();　　//删除所有元素
- map.erase(pos);　//删除 pos 迭代器所指的元素，返回下一个元素的迭代器。
- map.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。
- map.erase(keyElem);　　//删除容器中 key 为 keyElem 的对组。

```
map<int, string> mapA;
mapA.insert(pair<int,string>(3,"小张"));
mapA.insert(pair<int,string>(1,"小杨"));
mapA.insert(pair<int,string>(7,"小赵"));
mapA.insert(pair<int,string>(5,"小王"));

//删除区间内的元素
map<int,string>::iterator itBegin=mapA.begin();
++ itBegin;
++ itBegin;
map<int,string>::iterator itEnd=mapA.end();
mapA.erase(itBegin,itEnd);          //此时容器 mapA 包含按顺序的{1,"
```

小杨"}{3,"小张"}两个元素。

```
        mapA.insert(pair<int,string>(7,"小赵"));
        mapA.insert(pair<int,string>(5,"小王"));

  //删除容器中第一个元素
        mapA.erase(mapA.begin());       //此时容器 mapA 包含了按顺序的{3,"小张"}{5,"小王"}{7,"小赵"}三个元素

  //删除容器中 key 为 5 的元素
        mapA.erase(5);

  //删除 mapA 的所有元素
        mapA.clear();               //容器为空
```

6.map 的查找

- ✧ map.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();
- ✧ map.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。

```
map<int,string>::iterator it=mapStu.find(3);
if(it == mapStu.end())
{
        //没找到
}
else
{
         //找到了
          pair<int, string> pairStu = *it;
      int iID = pairStu.first;          //或    int   iID = it->first;
          string strName = pairStu.second;   //  或          string   strName   =
it->second;
}
```

- ✧ map.lower_bound(keyElem);　//返回第一个 key>=keyElem 元素的迭代器。
- ✧ map.upper_bound(keyElem);　//返回第一个 key>keyElem 元素的迭代器。

例如： mapStu 是用 map<int,string>声明的容器，已包含{1,"小李"}{3,"小张"}{5,"小王"}{7,"小赵"}{9,"小陈"}元素。map<int,string>::iterator it;

it = mapStu.lower_bound(5); //it->first==5 it->second=="小王"

it = mapStu.upper_bound(5); //it->first==7 it->second=="小赵"

it = mapStu.lower_bound(6); //it->first==7 it->second=="小赵"

it = mapStu.upper_bound(6); //it->first==7 it->second=="小赵"

✧ map.equal_range(keyElem); //返回容器中 key 与 keyElem 相等的上下限的两个迭代器。上限是闭区间，下限是开区间，如[beg,end)。

以上函数返回两个迭代器，而这两个迭代器被封装在 pair 中。

例如 map<int,string> mapStu;

... //往 mapStu 容器插入元素{1,"小李"}{3,"小张"}{5,"小王"}{7,"小赵"}{9,"小陈"}

pair<map<int,string>::iterator,map<int,string>::iterator>pairIt=mapStu.equal_range(5);

map<int, string>::iterator itBeg = pairIt.first;

map<int, string>::iterator itEnd = pairIt.second;

//此时 itBeg->first==5 , itEnd->first == 7,

itBeg->second=="小王", itEnd->second=="小赵"

Multimap 案例:

//1个key值可以对应多个valude =➔分组

//公司有销售部 sale （员工2名）、技术研发部 development （1人）、财务部 Financial （2人）

//人员信息有：姓名，年龄，电话、工资等组成

//通过 multimap进行 信息的插入、保存、显示

//分部门显示员工信息

```
#include <iostream>
using namespace std;
#include "map"
#include "string"
class Person
{
public:
```

```
    string    name;
    int       age;
    string    tel;
    double    saly;
};

void main1201()
{
    Person p1, p2, p3, p4, p5;

    p1.name = "王 1";
    p1.age = 31;

    p2.name = "王 2";
    p2.age = 32;

    p3.name = "张 3";
    p3.age = 33;

    p4.name = "张 4";
    p4.age = 34;

    p5.name = "赵 5";
    p5.age = 35;

    multimap<string, Person> map2;
    //sale 部门
    map2.insert(make_pair("sale", p1));
    map2.insert(make_pair("sale", p2));

    //development  部门
    map2.insert(make_pair("development", p3));
    map2.insert(make_pair("development", p4));

    //Financial  部门
    map2.insert(make_pair("Financial", p5));
```

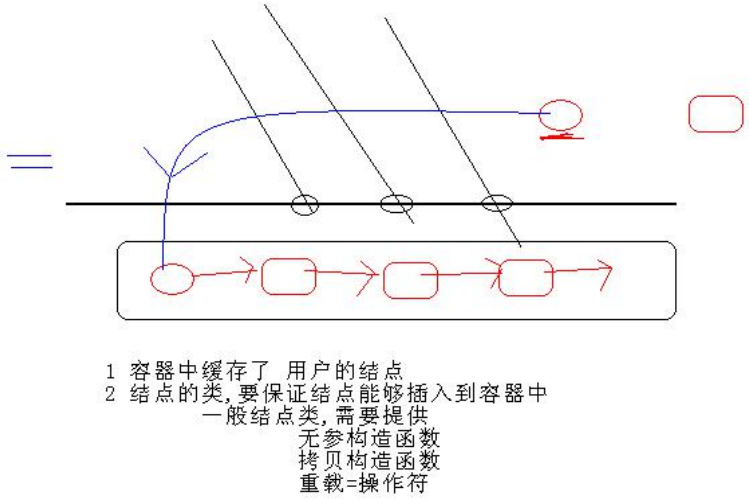
```
    for (multimap<string, Person>::iterator it = map2.begin(); it != map2.end(); it++)
    {
        cout << it->first << "\t" << it->second.name << endl;
    }
    cout << "遍历结束" << endl;

    //
    int num2 = map2.count("development");
    cout << "development 部门人数==>" << num2 << endl;

    cout << "development 部门员工信息" << endl;
    multimap<string, Person>::iterator it2 = map2.find("development");

    int tag = 0;
    while (it2 != map2.end() && tag < num2)
    {
        cout << it2->first << "\t" << it2->second.name << endl;
        it2++;
        tag++;
    }
}

// 修改容器里经常用的
//age = 32 修改成 name32
void main1202()
{
    Person p1, p2, p3, p4, p5;

    p1.name = "王 1";
    p1.age = 31;

    p2.name = "王 2";
    p2.age = 32;

    p3.name = "张 3";
```

```
    p3.age = 33;

    p4.name = "张 4";
    p4.age = 34;

    p5.name = "赵 5";
    p5.age = 35;

    multimap<string, Person> map2;
    //sale 部门
    map2.insert(make_pair("sale", p1));
    map2.insert(make_pair("sale", p2));

    //development 部门
    map2.insert(make_pair("development", p3));
    map2.insert(make_pair("development", p4));

    //Financial 部门
    map2.insert(make_pair("Financial", p5));

    cout << "\n 按照条件 检索数据 进行修改 " << endl;
    for (multimap<string, Person>::iterator it = map2.begin(); it != map2.end();
it++)
    {
        //cout << it->first << "\t" << it->second.name << endl;
        if (it->second.age == 32)
        {
            it->second.name = "name32";
        }
    }
    for (multimap<string, Person>::iterator it = map2.begin(); it != map2.end();
it++)
    {
        cout << it->first << "\t" << it->second.name << endl;
    }
}
```

10.2.10 容器共性机制研究

10.2.9.1 容器的共通能力

C++模板是容器的概念。



理论提高：所有容器提供的都是值（value）语意，而非引用（reference）语意。**容器执行插入元素的操作时，内部实施拷贝动作。**所以 STL 容器内存储的元素必须**能够被拷贝**（必须提供拷贝构造函数）。//所以必须要有自己定义的拷贝构造函数。

- ✧ 除了 queue 与 stack 外，每个容器都提供可返回迭代器的函数，运用返回的迭代器就可以访问元素。
- ✧ 通常 STL 不会丢出异常。要求使用者确保传入正确的参数。
- ✧ 每个容器都提供了一个默认构造函数跟一个默认拷贝构造函数。
- ✧ 如已有容器 vecIntA。
- ✧ vector<int> vecIntB(vecIntA);　//调用拷贝构造函数，复制 vecIntA 到 vecIntB 中。
- ✧ 与大小相关的操作方法(c 代表容器)：
- ✧ c.size();　//返回容器中元素的个数
- ✧ c.empty();　//判断容器是否为空
- ✧ 比较操作(c1,c2 代表容器)：
- ✧ c1 == c2　　判断 c1 是否等于 c2
- ✧ c1 != c2　　判断 c1 是否不等于 c2
- ✧ c1 = c2　　把 c2 的所有元素指派给 c1

10.2.9.2 各个容器的使用时机

	A	B	C	D	E	F	G	H
1		vector	deque	list	set	multiset	map	multimap
2	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
3	可随机存取	是	是	否	否	否	对key而言是	否
4	元素搜寻速度	慢	慢	非常慢	快	快	对key而言快	对key而言快
5	快速安插移除	尾端	头尾两端	任何位置	-	-	-	-

- ✧ Vector 的使用场景：比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。
- ✧ 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。
- ✧ vector 与 deque 的比较：
- ✧ 一：vector.at()比 deque.at()效率高，比如 vector.at(0)是固定的，deque 的开始位置却是不固定的。
- ✧ 二：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。
- ✧ 三：deque 支持头部的快速插入与快速移除，这是 deque 的优点。
- ✧ 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，支持频繁的不确实位置元素的移除插入。
- ✧ 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的顺序排列。

- ✧ 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。

10.3 算法

10.3.1 算法基础

10.3.1.1 算法概述

- ✧ 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。
- ✧ 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。
- ✧ #include <algorithm>
- ✧ #include <numeric>
- ✧ #include <functional>

10.3.1.2 STL 中算法分类

- 操作对象
  - 直接改变容器的内容
  - 将原容器的内容复制一份, 修改其副本, 然后传回该副本
- 功能:
  - 非可变序列算法 指不直接修改其所操作的容器内容的算法
    - 计数算法　　count、count_if
    - 搜索算法　　search、find、find_if、find_first_of、…
    - 比较算法　　equal、mismatch、lexicographical_compare
  - 可变序列算法 指可以修改它们所操作的容器内容的算法
    - 删除算法　　remove、remove_if、remove_copy、…
    - 修改算法　　for_each、transform
    - 排序算法　　sort、stable_sort、partial_sort、

- 排序算法 包括对序列进行排序和合并的算法、搜索算法以及有序序列上的集合操作
- 数值算法 对容器内容进行数值计算

10.3.1.3 查找算法(13 个)：判断容器中是否包含某个值

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_search	<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_bound	<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);

10.3.1.4 堆算法(4 个)

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);

10.3.1.5 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);

10.3.1.6 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_difference	<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)
	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);

10.3.1.6 列组合算法(2 个)

提供计算给定集合按一定顺序的所有可能排列组合

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<classBidIt>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);

10.3.1.7 排序和通用算法(14 个)：提供元素排序策略

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<classRanIt> 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);

10.3.1.8 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& v

		old, 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);

10.3.1.9 生成和变异算法(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);

10.3.1.10 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);

10.3.1.11 常用算法汇总

- ✧ 常用的查找算法：
  adjacent_find()（ adjacent 是邻近的意思）,binary_search(),count(), count_if(),equal_range(),find(),find_if()。
- ✧ 常用的排序算法：
  merge(),sort(),random_shuffle()（shuffle 是洗牌的意思）,reverse()。
- ✧ 常用的拷贝和替换算法：
  copy() , replace(),
  replace_if() , swap()
- ✧ 常用的算术和生成算法：
  accumulate()（accumulate 是求和的意思）,fill(),。
- ✧ 常用的集合算法：
  set_union() , set_intersection(),
  set_difference()。
- ✧ 常用的遍历算法：
  for_each(), transform()（transform 是变换的意思）

10.3.2 STL 算法中函数对象和谓词

10.3.2.1 函数对象和谓词定义

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。**一个类对象，具有了某个函数的功能**，就是通过“对象名+(参数列表)”的方式使用一个类对象，如果没有上下文，完全可以把它看作一个函数对待。

使用方法：　通过重载类的 operator()来实现的。

“在标准库中，函数对象被广泛地使用以获得弹性”，标准库中的很多算法都可以使用函数对象或者函数来作为自定的回调行为；

回调函数：只有当某个函数（更确切的说是函数的指针）被作为参数，被另一个函数调用时，它才是回调函数。

谓词：

一元函数对象：函数参数 1 个；

二元函数对象：函数参数 2 个；

一元谓词 函数参数 1 个，函数返回值是 bool 类型，可以作为一个判断式

谓词可以是一个仿函数，也可以是一个回调函数。

二元谓词 函数参数 2 个，函数返回值是 bool 类型

一元谓词函数举例如下

1. 判断给出的 string 对象的长度是否小于 6

```
bool GT6(const string &s)
{
return s.size() >= 6;
}
```

2,判断给出的 int 是否在 3 到 8 之间

```
bool Compare( int i )
{
return ( i >= 3 && i <= 8 );
}
```

二元谓词举例如下

1，比较两个 string 对象，返回一个 bool 值，指出第一个 string 是否比第二个短

```
bool isShorter(const string &s1, const string &s2)
{
return    s1.size() < s2.size();
}
```

10.3.2.2 一元函数对象案例

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

```
#include <vector>
#include <list>
#include "set"
#include <algorithm>
#include "functional"

//函数对象  类重载了( )
template <typename T>      //参数化的类型
class ShowElemt
{   public:   ShowElemt( )   { n = 0; }      //默认构造函数

      void operator( )(T &t)           //重载运算符( )
      {     n++;
            cout << t << " ";
      }

      void printN()
      {     cout << "n:" << n << endl;    }
protected:
private:
      int n;
};

//函数模板   ==函数
template <typename T>
void FuncShowElemt(T &t)
{
      cout << t << endl;
}

//普通函数
void FuncShowElemt2(int &t)
{
      cout << t << " ";
}

//  函数对象和普通函数的异同   普通函数不能像函数对象那样记录状态
void main01()
{
      int a = 10;
      ShowElemt<int> showElemt;
      showElemt(a);                  //函数对象的()的执行  很像一个函数   //仿函数

      FuncShowElemt<int>(a);      //函数模板
```

```
    FuncShowElemt2(a);          //普通函数
}

// 函数对象是属于类对象,能突破函数的概念,能保持调用状态信息，可以进行扩充
```

for_each 函数的讲解

for_each 是进行遍历的函数，第三个参数是一个函数，即遍历时所进行的操作默认谁使用 for_each 谁去填写回调函数的入口地址，

```
    // for_each 算法中, 函数对象做函数参数
    // for_each 算法中, 函数对象当返回值

void main02()
{
    vector<int> v1;
    v1.push_back(1);
    v1.push_back(3);
    v1.push_back(5);

    //for_each(v1.begin(), v1.end(), ShowElemt<int>());   //匿名函数对象、匿名仿函数
    //for_each(v1.begin(), v1.end(), FuncShowElemt2);     //普通函数做第三个参数
            // 谁使用 for_each 谁去填写回调函数的入口地址

ShowElemt<int> show1;   //定义了一个函数对象，其实就是一个“特殊”的类对象
    //for_each(v1.begin(), v1.end(), show1);
    //重点： for_each 算法的 函数对象的传递 是元素值传递 ,不是引用传递 即 for_each 中的第三个参数的 show1 位置 其实是个形参，相当于只是 show1 的拷贝，不会影响原来的 show1

    cout << "通过 for_each 算法的返回值看调用的次数" << endl;
    show1 = for_each(v1.begin(), v1.end(), show1);        //for_each 返回的是函数对象 把运行结果拷贝出来 赋值给 show2 也可以
    show1.printN();
//结论 要点: 分清楚 stl 算法返回的值是迭代器 还是 谓词（函数对象） 是 stl 算法入门的重要点
}
```

10.3.2.3 一元谓词案例

```
template<typename T>
class IsDiv
{
public:
	IsDiv(const T &divisor)		// 使用的是按照引用传递的方式。
	{
		this->divisor = divisor;
	}
	bool operator()(T	&t)		// 一元谓词
	{
		return (t%divisor == 0);
	}
private:
	T divisor;
};

void main03( )
{
	vector<int> v2;
	for (int i = 1; i<10; i++)
	{
		v2.push_back(i);
	}
	int a = 4;
	IsDiv<int> myDiv(a);		//定义了一个类对象

	//find_if(v2.begin(), v2.end(), myDiv );

	//find_if 返回值是一个迭代器
	//要点: 分清楚 stl 算法返回的值是迭代器 还是 谓词（函数对象） 是 stl 算法入门的重要点

	vector<int>::iterator it;
	it = find_if(v2.begin(), v2.end(), IsDiv<int>(a)); //find_if 只能返回第一个查找的值

	if (it == v2.end())
```

```
    {
        cout << "容器中没有被 4 整除的元素" << endl;
    }
    else
    {
        cout << "第一个是被 4 整除的元素是:" << *it << endl;
    }
}
```

10.3.2.4 二元函数对象案例

```
template <typename T>
class    SumAdd
{
public:
    T operator( )(T t1, T t2)
    {
        return t1 + t2;
    }
};

void main04( )
{
//v1    v2 ==> v3  将容器 v1 和 v2 中的元素相加  到 v3 中
    vector<int> v1, v2;
    vector<int> v3;
    v1.push_back(1);
    v1.push_back(3);
    v1.push_back(5);

    v2.push_back(2);
    v2.push_back(4);
    v2.push_back(6);

    v3.resize(10);

    //transform  把运算结果的  迭代器的开始位置  返回出来
    transform(v1.begin(), v1.end(), v2.begin(), v3.begin(),    SumAdd<int>());
```

```
    for (vector<int>::iterator it = v3.begin(); it != v3.end(); it++)
    {
        cout << *it << " ";
    }
    cout << endl;
}

//transform 算法的用法
  outputIterator transform ( InputIterator1 first1, InputIterator1 last1,
                             InputIterator2 first2, OutputIterator result,
                             BinaryOperator binary_op );
```

first1, last1

指出要进行元素变换的第一个迭代器区间［first1,last1)。

first2

指出要进行元素变换的第二个迭代器区间的首个元素的迭代器位置，该区间的元素个数和第一个区间相等。

result

指出变换后的结果存放的迭代器区间的首个元素的迭代器位置

op

用一元函数对象 op 作为参数，执行其后返回一个结果值。它可以是一个函数或对象内的类重载 operator()。

binary_op

用二元函数对象 binary_op 作为参数，执行其后返回一个结果值。它可以是一个函数或对象内的类重载 operator()。

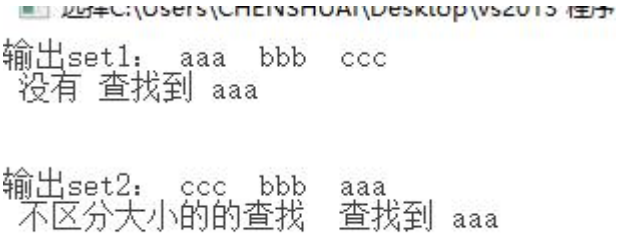
10.3.2.5 二元谓词案例

```
bool MyCompare(const int &a, const int &b)
{
    return a < b;        //从小到大
}
void main05()
{
    vector<int> v1(10);

    for (int i = 0; i<10; i++)
    {
```

```
        int tmp = rand() % 100;
        v1[i] = tmp;
    }

    cout << "普通遍历" << endl;
    for (vector<int>::iterator it = v1.begin(); it != v1.end(); it++)     //遍历
    {
        cout << *it << " ";
    }
    cout << endl;

    cout << "算法遍历" << endl;
    for_each(v1.begin(), v1.end(), FuncShowElemt2);       //遍历
    cout << endl;

    cout << "排序：" << endl;
     sort(v1.begin(), v1.end(), MyCompare);         //排序算法

    for_each(v1.begin(), v1.end(), FuncShowElemt2);
    cout << endl;
}

// 看不懂的一个程序
struct CompareNoCase    //set 需要指定函数对象   find 会进行调用（why？）
{
    bool operator()(const string &str1, const string &str2)
    {
        string str1_;
        str1_.resize(str1.size());
        transform(str1.begin(), str1.end(), str1_.begin(), tolower); //转化为小写

        string str2_;
        str2_.resize(str2.size());
        transform(str2.begin(), str2.end(), str2_.begin(), tolower); //转化为小写

        return (str1_ > str2_);         //从小到大进行排序
    }
```

```
};

void    main06()
{
    set<string> set1;
    set1.insert("bbb");
    set1.insert("aaa");
    set1.insert("ccc");

    cout << "输出 set1：    " ;
    for (set<string>::iterator it1 = set1.begin(); it1 != set1.end(); it1++)
    {
        cout << *it1<< "    ";
    }
    cout<< endl;

    set<string>::iterator it = set1.find("aAa"); //find 函数  默认  区分大小写
    if (it == set1.end())
    {
        cout << "  没有  查找到  aaa " << endl;
    }
    else
    {
        cout << "  查找到  aaa " << endl;
    }
    cout << endl<<endl;

    set<string, CompareNoCase> set2;
    set2.insert("bbb");
    set2.insert("ccc");
    set2.insert("aaa");

    cout << "输出 set2：    ";
    for (set<string>::iterator it1 = set2.begin(); it1 != set2.end(); it1++)
    {
        cout << *it1 << "    ";
    }
```

```
    cout << endl;

    set<string, CompareNoCase>::iterator it2 = set2.find("AAA");   //find 函数默认区分大小写
    if (it2 == set2.end())
    {
        cout << " 没有 查找到 aaa " << endl;
    }
    else
    {
        cout << " 不区分大小的的查找   查找到 aaa " << endl;
    }
}
```



10.3.2.6 预定义函数对象和函数适配器

1）预定义函数对象基本概念：标准模板库 STL 提前定义了很多预定义函数对象，#include <functional> 必须包含。

```
//1 使用预定义函数对象：
//类模板 plus<> 的实现了： 不同类型的数据进行加法运算，通过函数对象实现数据和算法的分离。
void main41()
{
    plus<int> intAdd;
    int x = 10;
    int y = 20;
    int z = intAdd(x, y); //等价于 x + y
    cout << z << endl;

    plus<string> stringAdd;
    string myc = stringAdd("aaa", "bbb");
    cout << myc << endl;

    vector<string> v1;
```

```
    v1.push_back("bbb");
    v1.push_back("aaa");
    v1.push_back("ccc");
    v1.push_back("zzzz");

    //缺省情况下，sort()用底层元素类型的小于操作符以升序排列容器的元素。
    //为了降序，可以传递预定义的类模板 greater,它调用底层元素类型的大于操作符：
    cout << "sort()函数排序" << endl;;
    sort(v1.begin(), v1.end(), greater<string>() ); //从大到小
    for (vector<string>::iterator it=v1.begin(); it!=v1.end(); it++ )
    {
        cout << *it << endl;
    }
}
```

2）算术函数对象

预定义的函数对象支持加、减、乘、除、求余和取反。调用的操作符是与 type 相关联的实例

加法：plus<Types>

```
    plus<string> stringAdd;
    sres = stringAdd(sva1,sva2);
```

减法：minus<Types>

乘法：multiplies<Types>

除法：divides<Tpye>

求余：modulus<Tpye>

取反：negate<Type>

```
    negate<int> intNegate;
    ires = intNegate(ires);
    Ires= UnaryFunc(negate<int>( ),Ival1);          //看不懂
```

3）关系函数对象

等于 equal_to<Tpye>

```
    equal_to<string> stringEqual;
    sres = stringEqual(sval1,sval2);
```

不等于　　not_equal_to<Type>

大于　　greater<Type>

大于等于　　greater_equal<Type>

小于　　less<Type>

小于等于　　less_equal<Type>

```
void main42()
{
	vector<string> v1;
	v1.push_back("bbb");
	v1.push_back("aaa");
	v1.push_back("ccc");
	v1.push_back("zzzz");
	v1.push_back("ccc");
	string s1 = "ccc";
	//int num = count_if(v1.begin(),v1.end(), equal_to<string>(),s1);  //编译是不通过的
//equal_to<string>( ) 有两个参数， left 参数来自容器，right 参数来自 sc
//bind2nd 函数适配器 ： 把 预定义函数对象 和 第二个参数进行绑定
	int num = count_if(v1.begin(),v1.end(),bind2nd(equal_to<string>(), s1));
	cout << num << endl;
}
```

4）逻辑函数对象

逻辑与 logical_and<Type>

```
logical_and<int> indAnd;
 ires = intAnd(ival1,ival2);
  dres=BinaryFunc( logical_and<double>(),dval1,dval2);
```

逻辑或 logical_or<Type>

逻辑非 logical_not<Type>

```
logical_not<int> IntNot;
Ires = IntNot(ival1);
Dres=UnaryFunc( logical_not<double>,dval1);
```

10.3.2.7 函数适配器

1）函数适配器的理论知识

STL 中已经定义了大量的函数对象，但是有时候需要对函数返回值进行进一步的简单计算，或者填上多余的参数，不能直接代入算法。函数适配器实现了这一功能，将一种函数对象转化为另一种符合要求的函数对象。函数适配器可以分为 4 大类：绑定适配器（bind adaptor）、组合适配器（composite adaptor）、指针函数适配器（pointer function adaptor）和成员函数适配器（member function adaptor）。

STL 中所有的函数适配器由表 10-6 列出。

表 10-6 STL 标准库中的函数适配器

STL 函数适配器	类 型	功 能 说 明
binder1st	绑定适配器	将数值绑定到二元函数的第一个参数，适配成一元函数
binder2nd	绑定适配器	将数值绑定到二元函数的第二个参数，适配成一元函数
unary_negate	组合适配器	将一元谓词的返回值适配成其逻辑反
binary_negate	组合适配器	将二元谓词的返回值适配成其逻辑反
pointer _ to _ unary _function	指针函数适配器	将普通一元函数指针适配成 unary_function
pointer _ to _ binary _function	指针函数适配器	将普通二元函数指针适配成 binary_function
mem_fun_t	成员函数适配器	将无参数类成员函数适配成为一元函数对象，第 1 个参数为该类的指针类型
mem_fun_ref_t	成员函数适配器	将无参数类成员函数适配成为一元函数对象，第 1 个参数为该类的引用类型
mem_fun1_t	成员函数适配器	将单参数类成员函数适配成为二元函数对象，第 1 个参数为该类的指针类型
mem_fun1_ref_t	成员函数适配器	将单参数类成员函数适配成为二元函数对象，第 1 个参数为该类的引用类型
const_mem_fun_t	成员函数适配器	将无参数类常量成员函数适配成为一元函数对象，第 1 个参数为该类的常量指针类型
const_mem_fun_ref_t	成员函数适配器	将无参数类常量成员函数适配成为一元函数对象，第 1 个参数为该类的常量引用类型
const_mem_fun1_t	成员函数适配器	将单参数类常量成员函数适配成为二元函数对象，第 1 个参数为该类的常量指针类型
const_mem_fun1_ref_t	成员函数适配器	将单参数类常量成员函数适配成为二元函数对象，第 1 个参数为该类的常量引用类型

直接构造 STL 中的函数适配器通常会导致冗长的类型声明。为简化函数适配器的构造，STL 还提供了函数适配器辅助函数(如表 10-7 所示)，借助于泛型自动推断技术，无须显式的类型声明便可实现函数适配器的构造。

表 10-7 STL 标准库中的函数适配器辅助函数

适配器辅助函数	功 能 说 明
bind1st	辅助构造 binder1st 适配器实例，绑定固定值到二元函数的第一个参数位置
bind2nd	辅助构造 binder2nd 适配器实例，绑定固定值到二元函数的第二个参数位置
not1	辅助构造 unary_negate 适配器实例，生成一元函数的逻辑反函数
not2	辅助构造 binary_negate 适配器实例，生成二元函数的逻辑反函数
ptr_fun	辅助构造一般函数指针的 pointer_to_unary_function 或 pointer_to_binary_function 适配器实例
mem_fun	辅助构造 mem_fun_t 等成员函数适配器实例，返回一元或二元函数对象
mem_fun_ref	辅助构造 mem_fun_ref_t 等成员函数适配器实例，返回一元或二元函数对象

2）常用函数函数适配器

标准库提供一组函数适配器，用来特殊化或者扩展一元和二元函数对象。

常用适配器是：

1. 绑定器（binder）: binder 通过把二元函数对象的一个实参绑定到一个特殊的值上，将其转换成一元函数对象。C＋＋标准库提供两种预定义的 binder 适配器：bind1st 和 bind2nd，前者把值绑定到二元函数对象的第一个实参上，后者绑定在第二个实参上。

2. 取反器(negator) : negator 是一个将函数对象的值翻转的函数适配器。标准库提供两个预定义的 ngeator 适配器：not1 翻转一元预定义函数对象的真值,而 not2 翻转二元谓词函数的真值。

常用函数适配器列表如下：

bind1st(op, value)

bind2nd(op, value)

not1(op)

not2(op)

mem_fun_ref(op)

mem_fun(op)

ptr_fun(op)

3）常用函数适配器案例

```
//////////////////////////////////////////////////////////////////////////////
class IsGreat                    //创建一个一元谓词
{
public:
    IsGreat(int i)
    {
        m_num = i;
    }
    bool operator( )(int &num)
    {
        if (num > m_num)
        {
            return true;
        }
        return false;
    }
private:
    int m_num;
};

void main22( )
```

```
{
    vector<int> v1;
    for (int i = 0; i<10; i++)
    {
        v1.push_back(i + 1);
    }

    for (vector<int>::iterator it = v1.begin( ); it != v1.end( ); it++)
    {
        cout << *it << " ";
    }
    cout << endl;

    int num1 = count(v1.begin( ), v1.end( ), 3);
    cout << "num1:" << num1 << endl;

    //通过 谓词 求大于 2 的个数
    int num2 = count_if(v1.begin(), v1.end(), IsGreat(2));
    cout << "num2:" << num2 << endl;

//通过 预定义的函数对象 求大于 2 的个数
    //greater<int>() 有两个参数 左参数来自容器的元素 ， 右参数固定成 2 （通过函数适配器辅助函数 bind2nd 做的）
    int num3 = count_if(v1.begin(), v1.end(), bind2nd(greater<int>(), 2));
    cout << "num3:" << num3 << endl;
```

10.3.2.8 STL 的容器算法迭代器的设计理念

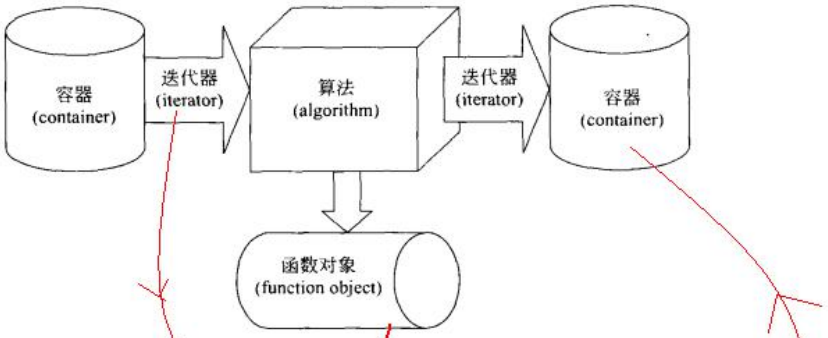


图 10-1　STL 组件之间的关系

```
template<class InputIterator, class OutputIterator, class UnaryFunction>
OutputIterator transform(InputIterator first, InputIterator last, OutputIterator
result, UnaryFunction op) {
    for (;first!=last; ++first, ++result)
        *result=op(*first);
    return result;
}
```

1 算法的重要特性，就是统一性；可以实现自定义数据类型和标准数据类型的运算。

1） STL 的容器通过类模板技术，实现数据类型和容器模型的分离

2） STL 的迭代器技术实现了遍历容器的统一方法；也为 STL 的算法提供了统一性奠定了基础

3） STL 的算法，通过函数对象实现了自定义数据类型的算法运算；所以说：STL 的算法也提供了统一性。

核心思想：其实函数对象本质就是回调函数，回调函数的思想：就是任务的编写者和任务的调用者有效解耦合。函数指针做函数参数。

4） 具体例子：transform 算法的输入，通过迭代器 first 和 last 指向的元算作为输入；通过 result 作为输出；通过函数对象来做自定义数据类型的运算。

10.3.3 常用的遍历算法

for_each()

- ✧ for_each:　用指定函数依次对指定范围内所有元素进行迭代访问。该函数不得修改序列中的元素。
- ✧ 函数定义。For_each(begin, end, func);

```
template<class _InIt,
 class _Fn1> inline
 _Fn1 for_each(_InIt _First, _InIt _Last, _Fn1 _Func)
```

```
    {    // perform function for each element
    _DEBUG_RANGE(_First, _Last);
    _DEBUG_POINTER(_Func);
    return (_For_each(_Unchecked(_First), _Unchecked(_Last), _Func));
    }
```

✧ 注意 for_each 的第三个参数 **函数对象做函数参数，函数对象做返回值**
第三个参数是形参，不会改变原来的实参。

```
}

class   CMyShow                 //一元函数对象
{
public:
    CMyShow( )
    {
        num = 0;
    }
    void operator( )(int &n)
    {
        num++;
        cout << n << " ";
    }
    void printNum( )
    {
        cout << "num:" << num << endl;
    }
protected:
private:
    int num;
};

void main41_foreach( )
{
    vector<int> v1;
    v1.push_back(1);
    v1.push_back(3);
    v1.push_back(5);
    CMyShow mya;        //定义一个函数对象
    CMyShow my1 = for_each(v1.begin(), v1.end(), mya);   //注意 find 使用的只是 mya 的拷贝
    mya.printNum();          //ma1 和 my1  是两个不同的对象
    my1.printNum();
```

```
}
运行结果：1 3 5   num:   0        //可以看出，丝毫没有影响原来的 mya
    num:3
```

transform( )

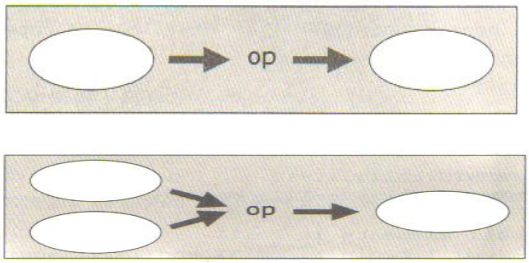
- ✧ transform:与 for_each 类似，遍历所有元素，但可对容器的元素进行修改
- ✧ transform( )算法有两种形式:
  - ✧ transform(b1, e1, b2, op)
  - ✧ transform(b1, e1, b2, b3, op)

```
template<class _InIt,class _OutIt,class _Fn1> inline
_OutIt transform(_InIt _First, _InIt _Last, _OutIt _Dest, _Fn1 _Func)
```

- ✧ transform( )的作用
  - ✧ 例如：可以一个容器的元素，通过 op，变换到另一个容器中（同一个容器中）

    transform(b1, e1, b2, op)
  - ✧ 也可以把两个容器的元素，通过 op，变换到另一个容器中

    transform(b1, e1, b2, b3, op)



注意： 1.如果目标与源相同，transform()就和 for_each()一样。
2.如果想以某值替换符合规则的元素，应使用 replace()算法

```
template<typename T>          //函数模板，既可以输出 vector 类中数据，也可输出 list 类中数据
void   printV( T &v)
{
    for ( T::iterator it = v.begin(); it != v.end(); it++)
    {
        cout << *it << " ";
    }
}
void main42_transform()
{
```

```
    vector<int> v1;
    v1.push_back(1);
    v1.push_back(3);
    v1.push_back(5);

    printV(v1);
    cout << endl;

    //transform      使用回调函数
    transform(v1.begin(), v1.end(), v1.begin(), increase);
    printV(v1);
    cout << endl;
    //transform 使用 预定义的函数对象
    transform(v1.begin( ), v1.end( ), v1.begin( ), negate<int>( ));
    printV(v1);
    cout << endl;

    //transform 使用 函数适配器辅助函数 和函数对象
    list<int> mylist;
    mylist.resize(v1.size());
                // 把 vector 中的数据运算后放到 list 容器中
    transform(v1.begin(), v1.end(), mylist.begin(), bind2nd(multiplies<int>(),
10));
    printList(mylist);
    cout << endl;

    //transform 也可以把运算结果 直接输出到屏幕  必须有头文件#include
"iterator"  输出流迭代器的头文件
    transform(v1.begin(), v1.end(), ostream_iterator<int>(cout, " "),
negate<int>());
    cout << endl;
}
```

for_each ( )和 transform ( )算法比较

1）STL 算法 – 修改性算法

- for_each()
- copy()

- copy_backward()
- transform()
- merge()
- swap_ranges()
- fill()
- fill_n()
- generate()
- generate_n()
- replace
- replace_if()
- replace_copy()
- replace_copy_if()

2）

- for_each( )　　速度快　　不灵活
- transform( ) 速度慢　　非常灵活

```
//一般情况下：for_each 所使用的函数对象，参数是引用，没有返回值
void mysquare(int &num)
{
    num = num * num ;
}

//transform 所使用的函数对象，参数一般不使用引用，而是还有返回值
int mysquare2(int num)    //结果的传出，必须是通过返回值
{
    return num = num * num;
}

void main_foreach_pk_tranform()
{
    vector<int>   v1;
    v1.push_back(1);
    v1.push_back(3);
    v1.push_back(5);

    vector<int>v2 = v1;
    for_each(v1.begin(), v1.end(), mysquare);
    printAA(v1);
    cout << endl;

    transform(v2.begin(), v2.end(), v2.begin(), mysquare2);
    printAA(v2);
    cout << endl;
}
```

10.3.4 常用的查找算法

adjacent_find ( )

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迭代器。 否则返回 past-the-end。

```
void main44_adjacent_find()
{
	vector<int> v1;
	v1.push_back(1);
	v1.push_back(2);
	v1.push_back(2);
	v1.push_back(3);
	v1.push_back(5);

	vector<int>::iterator it = adjacent_find(v1.begin(), v1.end());
	if (it == v1.end())
	{
		cout << "没有找到  重复的元素" << endl;
	}
	else
	{
		cout << *it << endl;
	}
	int index = distance(v1.begin(), it);		//查找到的具体位置
	cout << index << endl;
	}
```

binary_search （二分法查找）

在**有序序列中**查找 value,找到则返回 true。注意：在无序序列中，不可使用。

```
// 0    1    2    3 .....    n-1
//二分法  1K = 1024    10 次    速度快
void main45_binary_search()
{
	vector<int> v1;
	v1.push_back(1);
	v1.push_back(3);
	v1.push_back(5);
	v1.push_back(7);
	v1.push_back(9);
```

```
bool b = binary_search(v1.begin(), v1.end(), 7);
if (b == true)
{
    cout << "找到了" << endl;
}
else
{
    cout << "没到了" << endl;
}
```

count ( )

利用等于操作符，把标志范围内的元素与输入值比较，返回相等的个数。

```
vector<int> vecInt ;
vecInt.push_back(1);
vecInt.push_back(2) ;
vecInt.push_back(2) ;
vecInt.push_back(4) ;
vecInt.push_back(2) ;
vecInt.push_back(5) ;
int iCount = count(vecInt.begin(),vecInt.end(),2);    // iCount==3
```

count_if ( )

假设 vector<int> vecIntA，vecIntA 包含 1,3,5,7,9 元素

```
//先定义比较函数 必须是 bool 型
bool GreaterThree(int iNum)
{
    if(iNum>=3)
    {
        return true;
    }
    else
    {
        return false;
    }
}

int   iCount = count_if(vecIntA.begin( ), vecIntA.end(), GreaterThree);
```

```
//此时 iCount == 4
```

find ( )

- ✧ find: 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，返回该元素的迭代器。
- ✧ equal_range: 返回一对 iterator， 第一个表示 lower_bound,第二个表示 upper_bound。

```
void main47_find_findif()
{
	vector<int> v1;
	v1.push_back(1);
	v1.push_back(3);
	v1.push_back(5);
	v1.push_back(7);
	v1.push_back(7);
	v1.push_back(9);
	v1.push_back(7);

	vector<int>::iterator it = find(v1.begin(), v1.end(), 5);
	cout << "*it:" << *it << endl;

	//第一个大于 3 的位置
	vector<int>::iterator it2 = find_if(v1.begin(), v1.end(), GreatThree);
	cout << "*it2:" << *it2 << endl;
}
```

find_if ( )

find_if: 使用输入的函数代替等于操作符执行 find。返回被找到的元素的迭代器。

假设 vector<int> vecIntA，vecIntA 包含 1,3,5,3,9 元素

vector<int>::it = find_if(vecInt.begin(),vecInt.end(),GreaterThree);

此时 *it==3, *(it+1)==5, *(it+2)==3, *(it+3)==9

10.3.5 常用的排序算法

merge ( )

- ✧ 以下是排序和通用算法：提供元素排序策略
- ✧ merge: 合并两个有序序列，存放到另一个序列。

```
void main_merge()
```

```
{
    vector<int> v1;
    v1.push_back(1);
    v1.push_back(3);
    v1.push_back(5);

    vector<int> v2;
    v2.push_back(2);
    v2.push_back(4);
    v2.push_back(6);

    vector<int> v3;   //v3 要存放六个元素，指定空间必须大于 6,否则编译错误
                      //v3.resize(v1.size() + v2.size());
    v3.resize(7);

    merge(v1.begin(), v1.end(), v2.begin(), v2.end(), v3.begin());
    printV(v3);
}
```

运行结果： 1 2 3 4 5 6 0

sort ( )

✧ sort: 以默认升序的方式重新排列指定范围内的元素。若要改排序规则，可以输入比较函数。

```
class Student
{
public:
    Student(string name, int id)       //构造函数
    {
        m_name = name;
        m_id = id;
    }
    void printT()
    {
        cout << "name: " << m_name << " id " << m_id << endl;
    }
public:
    string    m_name;
    int       m_id;
```

```
};

bool CompareS(Student &s1, Student &s2)     //自定义的排序函数，必须是布尔型
{
    return (s1.m_id < s2.m_id);
}

void main_sort()
{
    Student s1("老大", 1);
    Student s2("老二", 2);
    Student s3("老三", 3);
    Student s4("老四", 4);
    vector<Student> v1;
    v1.push_back(s4);
    v1.push_back(s1);
    v1.push_back(s3);
    v1.push_back(s2);

    for (vector<Student>::iterator it = v1.begin(); it != v1.end(); it++)
    {
        it->printT();
    }

    //sort  根据自定义函数对象  进行自定义数据类型的排序
    //替换  算法的统一性  (实现的算法和数据类型的分离) ===>技术手段函数对象
    sort(v1.begin(), v1.end(), CompareS);

    for (vector<Student>::iterator it = v1.begin(); it != v1.end(); it++)
    {
        it->printT();
    }
}
```

random_shuffle ( )

- ✧ random_shuffle:　　对指定范围内的元素随机调整次序。

```
srand(time(0));          //设置随机种子

void main_random_shuffle()
```

```
{
    vector<int> v1;
    v1.push_back(1);
    v1.push_back(3);
    v1.push_back(5);
    v1.push_back(7);

    random_shuffle(v1.begin(), v1.end());
    printV(v1);

    string str = "abcdefg";
    random_shuffle(str.begin(), str.end());
    cout << "str: " << str << endl;     }
```

reverse ( )

```
{   vector<int> vecInt;
    vecInt.push_back(1);
    vecInt.push_back(3);
    vecInt.push_back(5);
    vecInt.push_back(7);
    vecInt.push_back(9);

    reverse(vecInt.begin(), vecInt.end());          //{9,7,5,3,1}
}
```

10.3.6 常用的拷贝和替换算法

copy ( )

```
void main52_copy()
{
    vector<int> v1;
    v1.push_back(1);
    v1.push_back(3);
    v1.push_back(5);
    v1.push_back(7);

    vector<int> v2;
    v2.resize(v1.size());   //该语句必须要有，并且大小要比原来的空间要大，否则出错
```

```
    copy(v1.begin(), v1.end(), v2.begin());
    printV(v2);
}
```

replace()

- replace(beg,end,oldValue,newValue): 将指定范围内的所有等于 oldValue 的元素替换成 newValue。

```
void main53_replace_replace_if()
{
    vector<int> v1;
    v1.push_back(1);
    v1.push_back(3);
    v1.push_back(5);
    v1.push_back(7);
    v1.push_back(3);
    replace(v1.begin(), v1.end(), 3, 8);
    printV(v1);
    cout << endl;

    // >=5
    replace_if(v1.begin(), v1.end(), great_equal_5, 1);
    printV(v1);
    cout << endl;
}
```

运行结果：1 8 5 7 8

1 1 1 1 1

replace_if()

- replace_if : 将指定范围内所有操作结果为 true 的元素用新值替换。

用法举例：

replace_if(vecIntA.begin(),vecIntA.end(),GreaterThree,newVal)

见上面。

swap( )

- swap:　交换两个容器的元素

```
void main54_swap()
{
	vector<int> v1;
	v1.push_back(1);
	v1.push_back(3);
	v1.push_back(5);

	vector<int> v2;
	v2.push_back(2);
	v2.push_back(4);
	//v2.push_back(6);

	swap(v1, v2);		//两个容器中的元素个数不同时仍然可以赋值成功。
	printV(v1);
	cout << endl;
	printV(v2);
}
运行结果：	2	4			//并没有出现 0 元素
1 3 5
```

10.3.7 常用的算术和生成算法

accumulate ( )

- ✧ accumulate: 对指定范围内的元素求和，然后结果再加上一个由 val 指定的初始值。

```
#include<numeric>
void main55_accumulate()
{
	vector<int> v1;
	v1.push_back(1);
	v1.push_back(3);
	v1.push_back(5);
	int tmp = accumulate(v1.begin(), v1.end(), 100);	//指定的初始值必须要有，自己指定
	cout << tmp << endl;
}
```

运行结果： 109

fill ( )

- ✧ fill: 将输入值赋给标志范围内的所有元素。

```
void main56_fill()
{
    vector<int> v1;
    v1.push_back(1);
    v1.push_back(3);
    v1.push_back(5);

    fill(v1.begin(), v1.end(), 8);
    printV(v1);
}
```

运行结果：8 8 8

10.3.8 常用的集合算法

set_union( ), set_intersection( ), set_difference( )

- ✧ set_union: 构造一个**有序序列**，包含两个**有序序列**的**并集**。
- ✧ set_intersection: 构造一个**有序序列**，包含**两个有序序列**的**交集**。
- ✧ set_difference: 构造一个有序序列，该序列保留第一个有序序列中存在而第二个有序序列中不存在的元素。

```
void main57_union()
{
    vector<int> v1;
    v1.push_back(1);
    v1.push_back(3);
    v1.push_back(5);

    vector<int> v2;
    v2.push_back(2);
    v2.push_back(4);
    v2.push_back(6);

    vector<int> v3;
```

```
    v3.resize(v1.size() + v2.size());

     set_union(v1.begin(), v1.end(), v2.begin(), v2.end(), v3.begin());
    cout <<"并集为：" <<"    ";
    printV(v3);
    cout << endl;

    vector<int> v4;
    v4.resize(v1.size() + v2.size());
    set_intersection(v1.begin(), v1.end(), v2.begin(), v2.end(), v4.begin());
    cout << "交集为：" << "    ";
    printV(v4);
    cout << endl;

    vector<int> v5;
    v5.resize(v1.size() + v2.size());
    set_difference(v1.begin(), v1.end(), v2.begin(), v2.end(), v5.begin());
    cout << "第一个序列中存在而第二个序列不存在的：" << "    ";
    printV(v5);
    cout << endl;
}
```

运行结果：　并集为：　1 2 3 4 5 6

交集为：　0 0 0 0 0 0　　//开辟的空间过大，

第一个序列中存在而第二个序列不存在的：　1 3 5 0 0 0　//最后几个未用到

10.4 STL 综合案例

10.4.1 案例学校演讲比赛

10.4.1.1 学校演讲比赛介绍

1）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。

2）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。

4）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。

10.4.1.2 需求分析

```
//产生选手（ABCDEFGHIJKLMNOPQRSTUVWXYZ ）姓名、得分；选手编号
//第 1 轮    选手抽签  选手比赛  查看比赛结果
//第 2 轮    选手抽签  选手比赛  查看比赛结果
//第 3 轮    选手抽签  选手比赛  查看比赛结果
```

10.4.1.3 实现思路

用到的容器

1. 编号和信息： map<int, speaker> mapSpeaker
按照个人编号 100、101、…… 123 的方式存入

2. 选手的编号信息： vector<int>

第一轮： V1： 所有选手编号信息
第二轮： V2： 第1轮晋级的选手编号信息
第三轮： V3： 第2轮晋级的选手编号信息
最终： V4： 第3轮晋级的前三名选手编号信息

3. 选手的成绩和编号： multimap<int, int, greater<int>> multimap Group
(一个小组内)
成绩 编号 按成绩降序排列
该容器只在评分时用，且每个小组的都使用，一个小组用完后必须清空数据

4. 计算每位选手得分时： deque<int> dScore
双端数组为了方便去掉最高分和最低分

10.4.1.4 实现细节

1） 搭建框架

2） 完善业务函数
random_shuffle

3） 测试

```
void main()
{
    //定义数据结构 所有选手放到容器中
    map<int, Speaker>   mapSpeaker;
    vector<int>         v1; //第 1 轮演讲比赛  名单
```

```
    vector<int>          v2; //第 2 轮演讲比赛  名单
    vector<int>          v3; //第 3 轮演讲比赛  名单
    vector<int>          v4; //最后  演讲比赛    名单

    //产生选手
    GenSpeaker(mapSpeaker, v1);

    //第 1 轮        选手抽签  选手比赛  查看比赛结果(晋级名单  得分情况)
    cout << "\n\n\n 任意键,开始第一轮比赛" << endl;
    cin.get();
    speech_contest_draw(v1);            // 产生第 1 轮参赛顺序
    speech_contest(0, v1, mapSpeaker, v2); //选手比赛，将第 1 轮晋级选手编号放入 V2 中
    speech_contest_print(0, v2, mapSpeaker);   // 打印第 1 轮晋级选手的编号姓名 分数

    //第 2 轮      选手抽签  选手比赛  查看比赛结果
    cout << "\n\n\n 任意键,开始第二轮比赛" << endl;
    cin.get();
    speech_contest_draw(v2);      // 产生第 2 轮参赛顺序
    speech_contest(1,    v2, mapSpeaker, v3); //比赛，将第 2 轮晋级选手编号放入 V3 中
    speech_contest_print(1, v3, mapSpeaker);   //打印第 2 轮晋级选手的编号  姓名  分数

    //第 3 轮 选手抽签  选手比赛  查看比赛结果
    cout << "\n\n\n 任意键,开始第三轮比赛" << endl;
    cin.get( );
    speech_contest_draw(v3);   // 产生第 3 轮参赛顺序
    speech_contest(2,    v3, mapSpeaker, v4); //选手比赛，将第 2 轮晋级的选手编号放入
V4 中
    speech_contest_print(2, v4, mapSpeaker);   //打印第 2 轮晋级选手的 编号  姓名  分
数
    system("pause");
}

//产生选手
int GenSpeaker(map<int, Speaker> &mapSpeaker, vector<int> &v1)
  //为什么采取引用的方式？ 因为我们输入对原 mapSpeaker 和 v1 写入数据，而不是对它们的副本

//选手抽签
int speech_contest_draw(vector<int>&v)
```

//上一轮的晋级者即为此轮的参赛者。 V 代表的是存放上一轮的晋级者编号的容器

//选手比赛

int speech_contest(int index, vector<int> &v1, map<int, Speaker> &mapSpeaker, vector<int> &v2)

// 用 index 来代表第几轮比赛，为 0 时表示第 1 轮，为 1 时表示第 2 轮。 当然不这样设置也可以。

因为选手比赛此函数需要输出小组内的选手的成绩以及编号等元素，所以设置的函数的输入参数比较多。

//打印选手比赛晋级名单

int speech_contest_print(int index, vector<int> v, map<int, Speaker> & mapSpeaker)

//函数的输入参数比较多也是为了防止可能会用到

源码：

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

#include "string"
#include <vector>
#include <list>
#include "set"
#include <algorithm>
#include "functional"
#include "iterator"          //输出流迭代器的头文件
#include<numeric>
#include "map"
#include "deque"             //双端数组

class Speaker
{
public:
    string   m_name;
    int      m_score[3];      //用来存放 3 轮比赛的成绩，有些人会用不到
};

//产生选手
int GenSpeaker(map<int, Speaker> &mapSpeaker, vector<int> &v)
{
```

```
    string str = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";        //所有选手的名字
    random_shuffle(str.begin(), str.end());     //random_shuffle 对指定范围内的元素随机调整次序。

    for (int i = 0; i<24; i++)
    {
        Speaker tmp;                    //只需要定义一个类对象，循环使用即可
        tmp.m_name = "选手";
        tmp.m_name = tmp.m_name + str[i];   //使用的是字符串的连接
        mapSpeaker.insert(pair<int, Speaker>(100 + i, tmp)); //将编号和个人信息插入到 map 容器中，
    }        //编号是按照顺序从 100 到 123 开始的，题目要求 编号相邻的 6 个人为一组

    for (int i = 0; i<24; i++)
    {
        v.push_back(100 + i);        //将参赛人员的编号放进 vector 容器中。
    }
    return 0;
}


//选手抽签    相当于随机排序
int speech_contest_draw(vector<int> &v)
{
    random_shuffle(v.begin(), v.end());
    return 0;
}

//选手比赛
int speech_contest(int index, vector<int> &v1, map<int, Speaker> &mapSpeaker, vector<int> &v2)
//小组的比赛得分 记录下来 ； 求出前三名 后 3 名
{
    multimap<int, int, greater<int>> multmapGroup;     //用来存放小组的成绩和编号 按照成绩的降序排列
    int    tmpCount = 0;     //采用相邻的 6 个为一组的方式

    for (vector<int>::iterator it = v1.begin(); it != v1.end(); it++)       //为晋级的
```

```
成员进行逐个的打分
    {
        tmpCount ++;          //为了进行小组内的处理

      //打分
        {
            deque<int> dscore;
            for (int j = 0; j<10; j++) //10 个评委打分
            {
                int score = 50 + rand() % 50;       //用随机数来表示评委的打分
                dscore.push_back(score);              //把分数放进 deque 容器中
            }
            sort(dscore.begin(), dscore.end());     //排序
            dscore.pop_back();
            dscore.pop_front();       //去除最低分  最高分

            //求平均分
            int scoresum = accumulate(dscore.begin(), dscore.end(), 0);       //
利用累加算法
            int scoreavg = scoresum / dscore.size();              //求平均值
            mapSpeaker[*it].m_score[index] = scoreavg;       //map 容器的 value
的插入方法选手得分  存入 map 容器中
            multmapGroup.insert(pair<int, int>(scoreavg, *it));       //用来存储
成绩和编号
        }

    //处理分组 ，随机排序以后连续的六个人为一组
        if (tmpCount % 6 == 0)
        {
            cout << "小组的比赛成绩" << endl;
            for (multimap<int, int, greater<int>>::iterator mit =
multmapGroup.begin(); mit != multmapGroup.end(); mit++)
            {
                //编号   姓名   得分
                cout << mit->second << "\t" <<
mapSpeaker[mit->second].m_name << "\t" << mit->first << endl;
            }       // mapSpeaker[mit->second].m_name   对应的编号的人员的
姓名

//前三名晋级
```

```
while (multmapGroup.size() > 3)
                {
multimap<int, int, greater<int>>::iterator it1 = multmapGroup.begin();
        v2.push_back(it1->second); //  把前 3 名的编号    放到 v2  晋级名单  中
                    multmapGroup.erase(it1);          //把第一个位置抹去
                }
                multmapGroup.clear();          //每一个小组完事有都要  清空本小组
比赛成绩
            }
        }
        return 0;
};
```

//查看比赛结果

```
int speech_contest_print(int index, vector<int> &v, map<int, Speaker>
&mapSpeaker)
{
    printf("第%d 轮  晋级名单\n", index + 1);
    for (vector<int>::iterator it = v.begin(); it != v.end(); it++)
    {
        cout << "参赛编号: " << *it << "\t" << mapSpeaker[*it].m_name << "\t"
<< mapSpeaker[*it].m_score[index] << endl;
    }
    return 0;
};

void main( )
{
    //容器的设计     基本的数据结构模型
    map<int, Speaker>   mapSpeaker;    //参加比赛的选手的编号  个人信息  以
及  成绩
    vector<int>          v1; //第 1 轮  演讲比赛  名单      只是采用的编号
    vector<int>          v2; //第 2 轮  演讲比赛  名单
    vector<int>          v3; //第 3 轮  演讲比赛  名单
    vector<int>          v4; //最后前三名  演讲比赛  名单

    //产生选手    得到第一轮选手的比赛名单          //编号是怎么产生的？？
    GenSpeaker(mapSpeaker, v1);
```

```
    //第 1 轮  选手抽签  选手比赛  查看比赛结果
    cout << "\n\n\n 任意键,开始第 1 轮比赛" << endl;
    cin.get();
    speech_contest_draw(v1);            //产生参赛顺序
    speech_contest(0, v1, mapSpeaker, v2);     //第一轮   选手参加比赛  选手的得分结果放到 V2 里面
    speech_contest_print(0, v2, mapSpeaker);

    //第 2 轮  选手抽签  选手比赛  查看比赛结果
    cout << "\n\n\n 任意键,开始第 2 轮比赛" << endl;
    cin.get();
    speech_contest_draw(v2);
    speech_contest(1, v2, mapSpeaker, v3);       //第二轮   选手参加比赛  选手的得分结果放到 V3 里面
    speech_contest_print(1, v3, mapSpeaker);

    //第 3 轮  选手抽签  选手比赛  查看比赛结果
    cout << "\n\n\n 任意键,开始第 3 轮比赛" << endl;
    cin.get();
    speech_contest_draw(v3);
    speech_contest(2, v3, mapSpeaker, v4);
    speech_contest_print(2, v4, mapSpeaker);


    system("pause");
    return;
}
```

运行结果：

```
任意键,开始第 1 轮比赛

小组的比赛成绩
106        选手 Z     83
111        选手 F     80
104        选手 A     77
121        选手 T     73
112        选手 O     73
108        选手 E     70
小组的比赛成绩
102        选手 J     76
115        选手 Q     75
```

118 选手 U 73
123 选手 K 72
109 选手 P 68
101 选手 B 68
小组的比赛成绩
120 选手 I 79
119 选手 R 76
113 选手 N 75
110 选手 S 74
107 选手 V 73
117 选手 D 64
小组的比赛成绩
114 选手 X 80
100 选手 M 80
103 选手 Y 79
116 选手 G 72
105 选手 L 69
122 选手 W 67
第 1 轮 晋级名单
参赛编号: 106 选手 Z 83
参赛编号: 111 选手 F 80
参赛编号: 104 选手 A 77
参赛编号: 102 选手 J 76
参赛编号: 115 选手 Q 75
参赛编号: 118 选手 U 73
参赛编号: 120 选手 I 79
参赛编号: 119 选手 R 76
参赛编号: 113 选手 N 75
参赛编号: 114 选手 X 80
参赛编号: 100 选手 M 80
参赛编号: 103 选手 Y 79

任意键,开始第 2 轮比赛

小组的比赛成绩
106 选手 Z 87
119 选手 R 81
103 选手 Y 79
115 选手 Q 76
118 选手 U 70
104 选手 A 66
小组的比赛成绩
100 选手 M 76
111 选手 F 75

102 选手 J 74
120 选手 I 72
114 选手 X 72
113 选手 N 68
第 2 轮 晋级名单
参赛编号: 106 选手 Z 87
参赛编号: 119 选手 R 81
参赛编号: 103 选手 Y 79
参赛编号: 100 选手 M 76
参赛编号: 111 选手 F 75
参赛编号: 102 选手 J 74

任意键,开始第 3 轮比赛

小组的比赛成绩
119 选手 R 82
111 选手 F 77
102 选手 J 74
103 选手 Y 71
106 选手 Z 67
100 选手 M 63
第 3 轮 晋级名单
参赛编号: 119 选手 R 82
参赛编号: 111 选手 F 77
参赛编号: 102 选手 J 74

10.4.2 案例：足球比赛

作业: 市区中学,足球比赛