第七章　STL

本章导读

STL是应用C++基本概念和功能的伟大作品，其出现相当程度地改变了C++程序设计面貌及其开发、应用模式。STL的基础是面向对象、模板、函数重载、运算符重载以及各种算法等综合而精巧的应用，更是泛型程序设计的典型应用。

学习STL，不仅要善于应用STL解决问题，同时解读STL相关代码，有助于深入理解及应用C++概念和功能，也是检验C++水平的重要标尺。

学习目标：

1. 了解STL体系；
2. 熟悉STL容器及其应用；
3. 熟悉仿函数及其应用；
4. 熟悉STL算法及其应用；

本章目录

第一节　认识STL

STL(Standard Template Library，简写为STL)是C++标准库的一部分，最初有HP公司的ALexander Stepanov和Meng Lee开发，是一个支持C++泛型编程的模板库。STL有多种不同的实现，但都遵守相同的接口标准，保证了C++程序的可移植性。

STL提供常用的数据结构和算法。STL中的T是template首字母简写，由此也可以看出，其设计的数据结构和算法支持多种数据类型。

为了让STL提供的算法和数据结构有更好通用性和灵活性，STL很好地应用了C++技术。学习STL，能更加深入地理解C++，认识C++语言之美。学习STL，不仅是学会应用STL，如能更好体味STL实现之道，将提升C++水平。

STL是常用且通用的数据结构和算法，针对具体问题，未必是最优解决方案，但一般说来是很好的解决方案，比如：排序，如果是针对特定数据的排序，也许需要专门的算法，但STL提供的排序算法通常都能满足应用，如果都是采用相同的算法思想，也许STL提供的更好，毕竟已经过千锤百炼。这有些像计算机里CPU和GPU，CPU因为通用，所以在图形方面不如GPU，GPU因为专用所以图形性能很强，但未必适合更多应用场景。

例程7-1是一个简单的STL应用，从第20行for_each(arrData,arrData+arrSize,PlusNum<int>(5))可以看出，for_each()函数由以下几个部分组成：数据源、数据访问、算法、数据处理。在C++中数据源可以是数组、vector等；数据访问可以是指针、也可以是迭代器；算法体现为不同的函数；数据处理体现为函数或者仿函数，以仿函数为主。

例程7-1

```
第1行  #include <iostream>
第2行  #include<algorithm>
第3行  using namespace std;
第4行  template<typename T>
第5行  class PlusNum{
第6行  public:
第7行          PlusNum(T Num):X(Num){}
第8行          void operator()(T Num){
第9行                  cout<<this->X+Num<<endl;;
第10行         }
第11行 private:
第12行         T X;
第13行 };
第14行 int main()
第15行 {
第16行         int arrData[]={12,43,23,54,67,87,11,34};
第17行         int arrSize=sizeof(arrData)/sizeof(arrData[0]);
第18行
第19行         //遍历指定范围内数据成员
第20行         for_each(arrData,arrData+arrSize,PlusNum<int>(5));
第21行
第22行         return 0;
第23行 }
```

从某种意义上将，数据源就是数据结构，数组是简单的数组结构，vector相对复杂一些。在C++中，提供了更多选择，如：vector、list、deque以及set(含multiset)、map(含multimap)等，不同数据结构有不同的特征，可以根据需要选择。这些数据结构能装容不同类型的数据，因此又被称为容器(container)，有关容器的更多内容，可参考本章第二节。

取出容器中的数据，交予算法处理，这取出之道为迭代器(Iterator)。迭代器分为多种，如：输入迭代器、输出迭代器、前向迭代器、双向迭代器、随机迭代器。选择何种迭代器即受制于容器特征，也根据需求而调整。有关迭代器的更多内容，可参考本章第三节。

程序设计中算法颇多，不同的算法能解决不同问题，在这些算法中，有不少属于通用算法。STL提供了很多通用算法。C++中STL常常经过千锤百炼，经过多种优化。如例程7-2所示的FILL函数就通过函数重载在处理字符时经过优化。另外，没有经验的开发人员也可能遗漏掉inline，这也将影响效率。例程中FILL()函数山区了迭代器范围检查，即算法的迭代器是否越界，在STL中已有处理。

例程7-2

```
第1行  #include <iostream>
第2行  #include<iterator>
第3行  using namespace std;
```

```
template<class _FwdIt,class _Ty>
inline void FILL(_FwdIt _First, _FwdIt _Last, const _Ty& _Val){
        _FILL(_First, _Last, _Val);
}
template<class _FwdIt,class _Ty>
inline void _FILL(_FwdIt _First, _FwdIt _Last, const _Ty& _Val){
        cout<<"_FILL _FwdIt"<<endl;
        for (;_First != _Last; ++_First)
                *_First =_Val;
}

inline void _FILLB(char *_First, char *_Last, int _Val){
        cout<<"_FILL char"<<endl;
        memset(_First, _Val, _Last - _First);
}

inline void _FILL(signed char *_First, signed char *_Last, int _Val){
        cout<<"_FILL char"<<endl;
        memset(_First, _Val, _Last - _First);
}

inline void _FILL(unsigned char *_First, unsigned char *_Last, int _Val){
        cout<<"_FILL char"<<endl;
        memset(_First, _Val, _Last - _First);
}
int main()
{
        int arrData[]={12,43,23,54,67,87,11,34};
        int arrSize=sizeof(arrData)/sizeof(arrData[0]);
        //本程序中FILL()对应STL算法中的fill()
        FILL(arrData,arrData+arrSize,11);

        return 0;
}
```

STL中的算法很多，可以分为：不变算法、可变算法、排序和搜索算法、数值算法、堆操作等。不变算法将不影响原容器中的数据，如for_each；可变算法将影响原容器中的数据，如fill算法执行后将用新数据填充覆盖源数据；排序和搜索算法专门用于对容器中的数据进行排序和搜索，提供多种类型的排序和搜索方式；数值算法专门用于数据处理；对操作用于对大量存储中堆中的数据进行处理。

数据处理是在算法上附加个性化的数据处理措施，如for_each遍历数据，而每个数据的处理交由数据处理部分完成。在C++中数据处理常用的函数对象，而C++提供了将普通函数、成员函数适配为函数对象的方法。

第二节　容器

容器(Container)是用来存储和组织C++基本数据类型和对象的对象。vector是容器，“数据结构与迭代器”中实现的数据结构，也是容器。在C++中，容器采用模板类实现，能装载各种类型的数据，包括C++的基本数据类型，如：int、double等，也包括自定义类实例化后的对象，如：UpDown等。

不同的数据结构，适用于不同场景。为方便开发，C++在STL中已经实现多种常用的数据结构，如：连续存储的vector、双向链表的

list等。根据容器特征，STL将容器分为：顺序容器(Sequence Container)、关联容器(Associative Container)和容器适配器(Container Adapter)三种。

- 顺序容器：顺序容器表示线性数据结构，包括：vector、list和deque三种；
- 关联容器：关联容器是非线性数据结构，可快速定位容器中的元素。这种容器可以保存集合或键/值对(key/value pair)。STL中有四种常用的关联容器set/multiset、map/multimap；
- 容器适配器：容器适配器是顺序容器的受限版本，用于处理特殊情况。STL中有三个容器适配器：stack(后进先出)、queue(先进先出)和priority_queue。

表7-1是STL中的常用容器及其主要特征，表7-2是所有容器都有的函数成员，表7-3是一级容器(即不包括容器适配器)都有的函数成员。

表7-1：STL容器类概要

容器名称	头文件	描述
vector	<vector>	直接访问任意元素，快速插入或删除尾部元素；
deque	<deque>	直接访问任意元素，快速插入、删除头部和尾部元素；
list	<list>	快速插入、删除任意位置元素；
set	<set>	快速查询元素，无重复关键字；
multiset	<set>	与set相同，但允许重复关键字；
map	<map>	键值对(key/value pair)映射，不允许重复关键字，使用关键字快速查询元素；
multimap	<map>	与map相同，但允许重复关键字；
stack	<stack>	后进先出容器
queue	<queue>	先进先出容器
priority_queue	<queue>	高优先级元素先删除；

表7-2：所有容器都有的函数成员

函数名称	描述
无参构造函数	构造一个空容器
带参数的构造函数	除无参构造函数外，每个容器都有多个带参数的构造函数
拷贝构造函数	从已有同类型容器中复制元素以创建一个新容器
析构函数	在容器销毁后执行清理工作，尤其是动态分配的内存
赋值运算符	将等号运算符右侧的容器复制到左侧容器中
empty()	判断容器中是否有元素，若没有元素则返回真
size()	返回容器中已有元素的数量
多个关系运算符	包括：<、<=、>、>=、==、!=等，顺序比较两个容器中对应元素，以确定大小

表7-3：一级容器都有的函数成员

函数名称	描述
C1.swap(C2)	交换C1与C2容器中的元素
C.max-size()	返回一个容器中可容纳的最大元素数目
C.clear()	清除容器中的所有元素
C.begin()	返回容器中第一个元素对应的迭代器
C.end()	返回容器中最后一个元素之后对应的迭代器
C.rbegin()	逆序迭代器，相当于最后一个元素对应的迭代器
C.rend()	逆序迭代器，相当于第一个元素之前对应的迭代器
C.erase(begin,end)	删除容器中begin和end之间的元素，不包括end对应的元素

1、顺序容器

STL定义了三种顺序容器：vector、list和deque(double-ended queue的简写，意为双端队列)，其不少操作较为相似，但内部实现机制大为不同，因此其类似操作，效率大为不同，如表7-4所示(注：不同机器耗时可能不同，如排除内存容量不够可能带来的特别差异，其相互关系应该不会太大变化，也能得到数据结构本身的解释)。

例程7-3

```
第1行  #include<iostream>
第2行  #include<vector>
第3行  #include<list>
第4行  #include<deque>
第5行  #include<ctime>
第6行  using namespace std;
第7行
第8行  int main(){
第9行          clock_t start,end;
第10行         int maxCount=50000;
第11行
第12行         //vector
第13行         vector<int> myVec;
第14行         start=clock();
第15行         for(int i=0;i<maxCount;++i)
第16行                 myVec.push_back(rand()%100);
第17行         end=clock();
第18行         cout<<"elapsed time of myVec:"<<end-start<<"\n\n";
第19行
第20行         //list
第21行         list<int> myList;
第22行         start=clock();
第23行         for(int i=0;i<maxCount;++i)
第24行                 myList.push_back(rand()%100);
第25行         end=clock();
第26行         cout<<"elapsed time of myList:"<<end-start<<"\n\n";
第27行
第28行         //deque
第29行         deque<int> myDeque;
第30行         start=clock();
第31行         for(int i=0;i<maxCount;++i)
第32行                 myDeque.push_back(rand()%100);
第33行         end=clock();
第34行         cout<<"elapsed time of myDeque:"<<end-start<<"\n\n";
第35行
第36行         return 0;
第37行 }
```

表7-4：顺序容器部分函数执行效率表

	尾部追加(毫秒)			头部插入(毫秒)			中间插入(毫秒)		
数量	vector	list	deque	vector	list	deque	vector	list	deque

5万	27	95	18	184723	97	18	……	185	817
50万	192	966	182	……	968	178	……	1794	8165
500万	1872	9487	1741	……	9627	1770	……	17929	81464

尾部追加方式形如：myVec.push_back(rand()%100);//myVec依次替换为myList和myDeque
头部插入方式形如：myList.push_front(rand()%100);//myList可替换为myDeque，vector没有push_front()成员函数
中间插入方式形如：pSeat_List=myList.insert(pSeat_List,rand()%100);//myList可替换为myVec和myDeque，但vector耗时太久									

从表中可以看出，vector、list和deque都支持push_back()函数成员，其效率都还不错，并且从5万到500万时，其耗时几乎都保持了线性增长；对于头部插入，vector不能原生支持，虽然可采用类似myVec.insert(myVec.begin(),rand()%100)代码模拟，但效率极低，无论是头部插入还是尾部插入，deque都保持了相当高的效率；对于中间位置插入，list表现优异。

容器的效率与容器内部的数据结构密切相关。表7-5是STL容器的数据结构及其优缺点描述。

表7-5：顺序容器的数据结构及其优缺点等

容器名称	描述
vector	具有类似数组特征，采用与数组相同的连续内存布局。任意位置元素的读取、修改具有常数时间，在序列尾部进行插入、删除效率极高，但在序列的头部和中间执行插入与删除操作耗时极高。 优点：内存和C完全兼容、高效随机访问。 缺点：头部和内部插入和删除元素代价巨大，需执行大量内存空间申请并执行内存拷贝等。 迭代器：随机迭代器 注：为提升效率，vector容量默认为实际容量2倍，但可压缩其空闲空间。
deque	deque是数组和链表的折衷，其内存连续，和vector相似。在序列头部执行插入和删除具有常数时间。 优点：高效随机访问(但远低于vector)，在内部插入和删除元素效率较高，在头尾插入和删除效率极高。 缺点：内存占比较高 迭代器：随机迭代器
list	list的数据结构是典型的链表，内存不连续，在头部、尾部和中间位置(几乎是头尾的两倍)具有近乎相同时间。 优点：在任意位置插入和删除元素具有几乎相同时间。 缺点：不支持随机访问，比vector占用更多的存储空间。 迭代器：双向迭代器

STL容器与迭代器关系密切。vector和deque支持随机迭代器，可以在记录中随意跳转，而list不支持随机迭代器，仅支持双向迭代器，其元素访问仅仅支持正向顺序或者反向顺序访问。vector和deque支持下标运算符[]，而list不支持。

2、关联容器

在顺序容器中，如果采用push_back()或push_front()添加元素，则元素的顺序按push_back()或push_front()的先后顺序排列；如果在中间插入，则元素顺序由插入位置决定。而对于关联容器，每个元素都有一个键(key)，容器中的元素顺序按键值升序排列，不能由程序员随意决定。关联容器的最大优点是能高效地根据键值查找元素。可以认为，顺序容器的经常行为是遍历(Traversal)，而关联容器则是查找(Search)。

在关联容器中，如是无序排列，则需要逐一比较，对于有N个元素的容器，则最坏情况需要N次比较；如果对容器中无序数据排序后则可采用各种查找算法，效率各有不同。而关联容器，则在添加元素时，即根据键值大小组织在一棵“平衡二叉树”中，最坏情况下，需要大约$\log_2 N$次比较就可以查找到指定元素。

关联容器分为set(集合)/map(映射)，set还可以分为不可重复的set和可重复的multiset(多重集合)，而map则分为不可重复的map和可重复的multimap(多重映射)。在set中，以元素本身作为键；而map则由键和对应的数据共同构成，如例程7-4所示。

例程7-4

```
第1行  /*
第2行          "ISBN978-7-302-22798-4,C++语言程序设计";
第3行          "ISBN978-7-111-24017-4,C++程序设计";
第4行          "ISBN978-7-115-14554-3,C++Primer中文版";
第5行          "ISBN978-7-508-31912-4,C++编程惯用法";
```

```
*/
#include<iostream>
#include<iterator>
#include<string>
#include<set>
#include<map>
using namespace std;

int main(){

        set<string> isbnList;
        isbnList.insert("ISBN978-7-302-22798-4");
        isbnList.insert("ISBN978-7-111-24017-4");
        isbnList.insert("ISBN978-7-115-14554-3");
        isbnList.insert("ISBN978-7-508-31912-4");
        copy(isbnList.begin(),isbnList.end(),ostream_iterator<string>(cout,"\n"));
        //输出顺序
        //ISBN978-7-111-24017-4、ISBN978-7-115-14554-3、
        //ISBN978-7-302-22798-4、ISBN978-7-508-31912-4

        map<string,string> bookList;
        bookList.insert(pair<string,string>("ISBN978-7-302-22798-4","C++语言程序设计"));
        bookList.insert(pair<string,string>("ISBN978-7-111-24017-4","C++程序设计"));
        bookList.insert(make_pair("ISBN978-7-115-14554-3","C++Primer中文版"));
        bookList.insert(make_pair("ISBN978-7-508-31912-4","C++编程惯用法"));
        for(map<string,string>::iterator p=bookList.begin();p!=bookList.end();++p)
                cout<<"书号:"<<p->first<<"\t书名:"<<p->second<<endl;

        return 0;
}
```

对于set，直接将ISBN(书号)插入即可，书号本身就是键值；而对map，则必须插入一对值，第一个值为为键值，第二个作为键对应的数据。map有些类似词典，键值如同检索的词条，而对应的数据如同该词条的解释。输入一对数据时，必须采用pair<dataType,dataType>(data1,data2)或make_pair(data1,data2)。pair是一个模板类，如例程7-5所示；make_pair()是一个模板函数，如例程7-6所示。

例程7-5

```
template<typename T1,typename T2>
struct pair{
        T1 firstArg;
        T2 secondArg;
        pair(){};
        pair(const T1&Arg1,const T2&Arg2):firstArg(Arg1),secondArg(Arg2){}
};
```

例程7-6

```
template<typename T1,typename T2>
inline Pair<T1,T2> make_Pair(const T1&Arg1, const T2&Arg2){
```

```
第3行          return Pair<T1,T2>(Arg1,Arg2);
第4行  }
```

关联容器的键值(key)必须能比较大小，即至少能进行小于运算符(<)。C++的基本数据类型，如int、double、char以及string等已必然支持小于运算符，而对于自定义class或struct，则必须重载小于运算符，否则将不能成为关联容器的键值，如例程7-7所示，其第7-14行定义了Book类，并在第11-13行定义了小于运算符。如果没有该运算符，Book类的实例化对象将不能插入到set中。对于本例中的Map，是否定义小于运算符，不影响其插入到bookMap中，原因是bookMap以bookISBN为键值，其数据类型为string，已经支持小于运算符。

例程7-7

```
第1行  #include<iostream>
第2行  #include<iterator>
第3行  #include<string>
第4行  #include<set>
第5行  #include<map>
第6行  using namespace std;
第7行  struct Book{
第8行          string bookISBN;
第9行          string bookName;
第10行         Book(string isbn,string name):bookISBN(isbn),bookName(name){};
第11行         bool operator<(const Book &rightSide)const{
第12行                 return this->bookISBN<rightSide.bookISBN;
第13行         }
第14行 };
第15行 int main(){
第16行         Book book1("ISBN978-7-302-22798-4","C++语言程序设计");
第17行         Book book2("ISBN978-7-111-24017-4","C++程序设计");
第18行         Book book3("ISBN978-7-115-14554-3","C++Primer中文版");
第19行         Book book4("ISBN978-7-508-31912-4","C++编程惯用法");
第20行         set<Book> bookSet;
第21行         bookSet.insert(book1);bookSet.insert(book2);
第22行         bookSet.insert(book3);bookSet.insert(book4);
第23行         for(set<Book>::iterator p=bookSet.begin();p!=bookSet.end();++p)
第24行                 cout<<(*p).bookISBN<<" "<<(*p).bookName<<"\t";
第25行
第26行         cout<<"\n\n";
第27行
第28行         map<string,Book> bookMap;
第29行         bookMap.insert(make_pair(book1.bookISBN,book1));bookMap.insert(make_pair(book2.bookISBN,book2));
第30行         bookMap.insert(make_pair(book3.bookISBN,book1));bookMap.insert(make_pair(book4.bookISBN,book2));
第31行         for(map<string,Book>::iterator p=bookMap.begin();p!=bookMap.end();++p)
第32行                 cout<<p->second.bookISBN<<" "<<p->second.bookName<<"\t";
第33行
第34行         return 0;
第35行 }
```

STL中的set和map采用“严格弱序关系”，在该关系中，仅需支持＜运算符，无需关心＜＝、＝＝、＞＝等。在bookSet中，如果执行book1＜book1，其结果为false等。在排序过程中，如果其bookISBN相同，而bookName不同，也视为相同对象，即set或map中，仅仅比较

关键值部分，不比较其余部分。

对于例程7-7，以Book整体为关键值不是好的选择，如bookSet。当需要查找时，需要输入整个Book类的对象，很不方面，而以C++基本类型作为关键值，则能很方便地输入，如bookMap，就可以ISBN字符串查找。

例程7-8用于统计输入英文语句中，各个字母出现次数，涉及到map的插入和查找功能。在执行插入时，返回pair对象，其第一个类型参数为map迭代器的迭代器，第二类型参数为bool。当成功插入时，返回被插入对象所在迭代器位置以及boolean值true；当插入失败时，其boolean值为false，如例程第10行所示。第11行的代码中的insertCharCount是一个pair对象，根据其second成员值，确定是否插入成功。如果插入失败，则表明该字母的统计结果已经存在，仅需在其出现值加1即可。而其位置为insertCharCount的first成员值(类型为迭代器)，通过->运算符找到其统计值并加1。

例程7-8

```
第1行  #include<iostream>
第2行  #include<iterator>
第3行  #include<map>
第4行  using namespace std;
第5行  int main(){
第6行          map<char,int> charCount;
第7行          istream_iterator<char> charInput(cin);
第8行          istream_iterator<char> charEnd;
第9行          for(istream_iterator<char> pChar=charInput;pChar!=charEnd;++pChar){
第10行                 pair<map<char,int>::iterator,bool>insertCharCount=charCount.insert(make_pair(*pChar,1));
第11行                 if(insertCharCount.second==false)(insertCharCount.first->second)++;
第12行         }
第13行         for(map<char,int>::iterator p=charCount.begin();p!=charCount.end();++p)
第14行                 cout<<p->first<<"出现次数:"<<p->second<<"\n";
第15行
第16行         //查找
第17行         map<char,int>::iterator pSeek=charCount.find('a');//查找字符'a'出现的次数
第18行         if(pSeek!=charCount.end()){
第19行                 cout<<"字符a出现次数："<<pSeek->second<<endl;
第20行         }else{
第21行                 cout<<"字符a没有出现"<<endl;
第22行         }
第23行
第24行         //列出所有大写字母出现次数
第25行         for(map<char,int>::iterator p=charCount.lower_bound('A');p!=charCount.upper_bound('Z');++p)
第26行                 cout<<p->first<<"出现次数:"<<p->second<<"\n";
第27行
第28行         return 0;
第29行 }
```

例程7-8第16-22行是关联容器的查找，其功能用find()成员函数实现，其返回值为相应对象的迭代器。当查找到指定关键字时，则迭代器指向该记录；当查找失败时，则迭代器指向迭代器结束。

关联容器的成员函数lower_bound()和upper_bound()分别返回查找关键值的迭代器开始和结束，如第25行所示。如果是multimap或multiset，关键值可以是同一个，相当于查找该第一个满足关键值的记录以及第一个不满足关键值的记录，也相当于该关键值的开始和结束，例程7-9第56、57行也是关于这两个成员函数的应用。

关联容器的成员函数equal_range()返回关键值类型为pair类型，其内容为迭代器的起点和终点，与lower_bound()和upper_bound()相似，如例程7-9第52行所示。如需查找指定关键值迭代器范围，使用equal_range()效率更高。不过，lower_bound()和upper_bound()不

但可以查找同一关键字，也可以查找不同关键字，如例程7-8第25行所示。另外，例程7-9还有成员count()的应用，用于计算关键值的出现次数。

例程7-9

```
第1行  #include<iostream>
第2行  #include<iterator>
第3行  #include<algorithm>
第4行  #include<string>
第5行  #include<map>
第6行  using namespace std;
第7行  struct JiaData{
第8行          string Level;//辈分
第9行          string Sex;//性别
第10行         string Name;//姓名
第11行         JiaData(string L,string S,string N):Level(L),Sex(S),Name(N){};
第12行         friend ostream& operator<<(ostream &out,const JiaData &JD);//友元函数
第13行 };
第14行 //定义JiaData的<<运算符
第15行 ostream &operator<<(ostream &out,const JiaData&JD){
第16行         out<<JD.Name<<"("<<JD.Sex<<","<<JD.Level<<")";
第17行         return out;
第18行 }
第19行 int main(){
第20行         JiaData JD[]={
第21行                 JiaData("水旁辈","男","贾演"),JiaData("水旁辈","男","贾源"),JiaData("代字辈","男","贾代化"),
第22行                 JiaData("代字辈","男","贾代善"),JiaData("代字辈","男","贾代儒"),JiaData("代字辈","男","贾代修"),
第23行                 JiaData("攴旁辈","男","贾敷"),JiaData("攴旁辈","男","贾敬"),JiaData("攴旁辈","男","贾赦"),
第24行                 JiaData("攴旁辈","男","贾政"),JiaData("攴旁辈","女","贾敏"),JiaData("攴旁辈","男","贾敕"),
第25行                 JiaData("攴旁辈","男","贾效"),JiaData("攴旁辈","男","贾敦"),JiaData("玉春辈","男","贾珍"),
第26行                 JiaData("玉春辈","男","贾琏"),JiaData("玉春辈","男","贾珠"),JiaData("玉春辈","男","贾宝玉"),
第27行                 JiaData("玉春辈","男","贾环"),JiaData("玉春辈","男","贾瑞"),JiaData("玉春辈","男","贾璜"),
第28行                 JiaData("玉春辈","男","贾琮"),JiaData("玉春辈","男","贾珩"),JiaData("玉春辈","男","贾珖"),
第29行                 JiaData("玉春辈","男","贾琛"),JiaData("玉春辈","男","贾琼"),JiaData("玉春辈","男","贾璘"),
第30行                 JiaData("玉春辈","女","贾元春"),JiaData("玉春辈","女","贾迎春"),JiaData("玉春辈","女","贾探春"),
第31行                 JiaData("玉春辈","女","贾惜春")
第32行 };
第33行
第34行         random_shuffle(JD,JD+31);//乱序数组，该函数来自STL的算法头文件algorithm。
第35行         //输出乱序后的名单
第36行         copy(JD,JD+31,ostream_iterator<JiaData>(cout,"\n"));
第37行
第38行         //注意：multimap是string和地址构成，此处地址来自数组
第39行         multimap<string,JiaData*> JiaMMap_Level,JiaMMap_Sex;
第40行         for(JiaData *p=JD;p!=JD+31;++p){
```

```
第41行                  JiaMMap_Level.insert(make_pair(p->Level,p));
第42行                  JiaMMap_Sex.insert(make_pair(p->Sex,p));
第43行          }
第44行
第45行          //count用于根据关键字统计数量
第46行          cout<<"玉春辈人数="<<JiaMMap_Level.count("玉春辈")<<endl;//输出：17
第47行          cout<<"男性人数="<<JiaMMap_Sex.count("男")<<endl;//输出：26
第48行
第49行          typedef multimap<string,JiaData*>::iterator JiaLevelIter;//定义类型别名
第50行
第51行          //equal_range返回一对迭代器，分别为开始和结束
第52行          pair<JiaLevelIter,JiaLevelIter> levelRange=JiaMMap_Level.equal_range("支旁辈");
第53行          for(JiaLevelIter p=levelRange.first;p!=levelRange.second;++p)
第54行                  cout<<*(p->second)<<endl;
第55行
第56行          JiaLevelIter pStart=JiaMMap_Level.lower_bound("支旁辈");
第57行          JiaLevelIter pStop=JiaMMap_Level.upper_bound("支旁辈");
第58行          for(JiaLevelIter p=pStart;p!=pStop;++p)
第59行                  cout<<*p->second<<endl;
第60行
第61行          system("pause");
第62行          return 0;
第63行  }
```

3、容器适配器

标准STL提供三种容器适配器，分别是stack、queue和priority_queue。之所以被称为适配器(adapter)，是因为它们由顺序容器变化而来，是顺序容器的定制版本，功能缩减版本，用于处理特定场景。作为容器适配器，STL允许程序员选择适合的顺序容器，如stack默认基于deque实现，但也可以选择vector或list。容器适配器没有迭代器，除支持表7-2所列所有容器的共同函数外，还支持push()和pop()函数，用于插入和删除元素。

容器适配器stack称之为栈，是一种后进先出容器。stack默认基于顺序容器deque实现，也可以选择vector或list。表7-6是stack的成员函数。另外，常用的size()和empty()函数属于容器的通用函数分别表示容器中元素数量以及判断容器是否为空。例程7-10用于数制转换，是stack的一个应用。第15、16、17行分别申明一个stack类型的变量，其中stackHex没有指定顺序容器，默认采用deque实现，而stackOct采用vector，stackBin则采用list。注意stack<char,list<char> >stackOct中char后面的两个＞符号之间必须有空格，否则编译器可能误解为右移运算符。

表7-6：栈的成员函数

函数名称	描述
push()	将元素添加到stack，新增元素位于栈顶
pop()	删除栈顶元素
top()	返回栈顶元素

例程7-10

```
第1行   #include<iostream>
第2行   #include<stack>
第3行   #include<list>
第4行   #include<vector>
第5行   using namespace std;
第6行
第7行   typedef unsigned int long uInt;
第8行
```

```
第9行   pair<uInt,char> Division(uInt dividend,uInt divisor);
第10行  template<typename T>void Exec(T& stackResult,uInt Dividend,uInt Divisor);
第11行  template<typename T>void Print(T &stackInput);
第12行
第13行  int main(){
第14行
第15行          stack<char> stackHex;
第16行          stack<char,vector<char> >stackBin;
第17行          stack<char,list<char> >stackOct;
第18行
第19行          uInt Dividend=123456789;
第20行          Exec(stackHex,Dividend,16);
第21行          Exec(stackOct,Dividend,8);
第22行          Exec(stackBin,Dividend,2);
第23行
第24行          cout<<Dividend<<"的十六进制=";Print(stackHex);cout<<"\n\n";
第25行          cout<<Dividend<<"的  八进制=";Print(stackOct);cout<<"\n\n";
第26行          cout<<Dividend<<"的  二进制=";Print(stackBin);cout<<"\n\n";
第27行          system("pause");
第28行          return 0;
第29行  }
第30行  pair<uInt,char> Division(uInt dividend,uInt divisor){
第31行          char Remainder=dividend%divisor;//余数
第32行          return pair<uInt,char>((dividend-Remainder)/divisor,Remainder<10?48+Remainder:55+Remainder);
第33行  }
第34行  template<typename T>
第35行  void Exec(T& stackResult,uInt Dividend,uInt Divisor){
第36行          uInt tmp=Dividend;
第37行          pair<uInt,char> rtnResult;
第38行          while(true){
第39行                  rtnResult=Division(tmp,Divisor);
第40行                  stackResult.push(rtnResult.second);
第41行                  tmp=rtnResult.first;
第42行                  if(rtnResult.first==0)break;
第43行          }
第44行  }
第45行  template<typename T>
第46行  void Print(T &stackInput){
第47行          while(!stackInput.empty()){
第48行                  cout<<stackInput.top();
第49行                  stackInput.pop();
第50行          }
第51行  }
```

表7-7：队列的成员函数

容器适配器queue称之为队列，是一种先进先出容器。默认情况下，

queue采用deque实现，也可以用list实现。对于基础容器，要求必须提供front()、back()、push_back()和pop_front()函数成员，由于vector没有front()函数成员，因此不能采用vector容器。

队列是一种特殊的线性表。在队列queue中，保证只在队尾插入新元素，而且只在队首删除元素。表7-7是queue的成员函数，例程7-11是有关队列queue的示例。

函数名称	描述
push()	将元素添加到queue，新增元素位于队尾
pop()	删除队首元素
front()	返回队首元素
back()	返回队尾元素

例程7-11

```
第1行  #include<iostream>
第2行  #include<queue>
第3行  using namespace std;
第4行
第5行  int main(){
第6行          queue <int> myQ;
第7行          for(int i=0;i<50;++i)
第8行                  myQ.push(rand()%100);
第9行
第10行         while(!myQ.empty()){
第11行                 cout<<"首="<<myQ.front()<<"尾="<<myQ.back()<<endl;
第12行                 myQ.pop();//删除首元素
第13行         }
第14行
第15行         return 0;
第16行 }
```

容器适配器priority_queue称之为优先级队列，允许为队列中的元素设置优先级。这种队列不是直接将新元素放在队尾或者队首，而是放在优先级较低的元素前面。priority_queue默认使用 ＜操作符来确定元素之间的优先级关系，但用户可以根据需要制定优先级规则。priority_queue默认采用顺序容器vector作为基础容器，也可以指定为deque。在优先级队列中，优先级最高的元素手被访问或被删除，即在优先级队列最大者未必先进，但一定先出。在优先级队列中，默认按＜操作符确定优先等级，对于C++基本数据类型，已固然支持，对于自定义类，必须重载＜操作符，如例程7-12所示。另外，还可以指定优先级排列规则，如例程7-13所示。表7-8是priority_queue的成员成员函数。

表7-8：队列的成员函数

函数名称	描述
push()	将元素添加到priority_queue
pop()	删除队首元素
top()	返回队首元素

例程7-12

```
第1行  #include<iostream>
第2行  #include<vector>
第3行  #include<iterator>
第4行  #include<queue>
第5行  using namespace std;
第6行  struct Score{
第7行          int Math;
第8行          int Foreign;
第9行          int Chinese;
第10行         Score(int M=0,int F=0,int C=0):Math(M),Foreign(F),Chinese(C){}
第11行         bool operator<(const Score & rightSide)const{
```

```
第12行                  return (this->Math+this->Chinese+this->Foreign)
第13行                          <(rightSide.Math+rightSide.Chinese+rightSide.Foreign);
第14行          }
第15行          friend ostream & operator<<(ostream &out,Score &rightSide);
第16行  };
第17行  ostream & operator<<(ostream &out,Score &rightSide){
第18行          out<<"语文"<<rightSide.Chinese<<" 外语="<<rightSide.Foreign
第19行                  <<" 数学="<<rightSide.Math<<" 总分="<<(rightSide.Chinese+rightSide.Foreign+rightSide.Math);
第20行          return out;
第21行  }
第22行  int main(){
第23行          priority_queue<Score,vector<Score> >PQ;
第24行          for(int i=0;i<50;++i)
第25行                  PQ.push(Score(rand()%100,rand()%100,rand()%100));
第26行          while(!PQ.empty()){
第27行                  cout<<PQ.top()<<endl;
第28行                  PQ.pop();
第29行          }
第30行
第31行          return 0;
第32行  }
```

在例程7-13，优先级队列中载入的是数组中的数据地址，即一份实体数据，两种优先级排序方式，而不是两套实体数据，果如此，难免带来数据不一致性。

例程7-13

```
第1行   #include<iostream>
第2行   #include<vector>
第3行   #include<iterator>
第4行   #include<queue>
第5行   using namespace std;
第6行   struct Score{
第7行           int Math;
第8行           int Foreign;
第9行           int Chinese;
第10行          Score(int M=0,int F=0,int C=0):Math(M),Foreign(F),Chinese(C){}
第11行          bool operator<(const Score & rightSide)const{
第12行                  return (this->Math+this->Chinese+this->Foreign)
第13行                          <(rightSide.Math+rightSide.Chinese+rightSide.Foreign);
第14行          }
第15行          friend ostream & operator<<(ostream &out,Score &rightSide);
第16行  };
第17行  ostream & operator<<(ostream &out,Score &rightSide){
第18行          out<<"语文"<<rightSide.Chinese<<" 外语="<<rightSide.Foreign
第19行                  <<" 数学="<<rightSide.Math<<" 总分="<<(rightSide.Chinese+rightSide.Foreign+rightSide.Math);
第20行          return out;
```

```
第21行  }
第22行  struct cmpA{
第23行          bool operator() (Score *Left, Score*Right){
第24行                  return ((*Left).Math+(*Left).Chinese+(*Left).Foreign)
第25行                          <((*Right).Math+(*Right).Chinese+(*Right).Foreign);
第26行          }
第27行  };
第28行  struct cmpB{
第29行          bool operator() (Score *Left, Score*Right){
第30行                  return ((*Left).Math*1.5+(*Left).Chinese*1.8+(*Left).Foreign)
第31行                          <((*Right).Math*1.5+(*Right).Chinese*1.8+(*Right).Foreign);
第32行          }
第33行  };
第34行  int main(){
第35行          const int scoreSize=50;
第36行          Score arrScore[scoreSize];
第37行          for(int i=0;i<scoreSize;++i)
第38行                  arrScore[i]=Score(rand()%100,rand()%100,rand()%100);
第39行
第40行          cout<<"按cmpA方案排序";
第41行          priority_queue<Score*,vector<Score*>,cmpA>pqA;
第42行          for(int i=0;i<scoreSize;++i)
第43行                  pqA.push(arrScore+i);
第44行          while(!pqA.empty()){
第45行                  cout<<*(pqA.top())<<endl;
第46行                  pqA.pop();
第47行          }
第48行
第49行          cout<<"按cmpB方案排序";
第50行          priority_queue<Score*,deque<Score*>,cmpB>pqB;
第51行          for(int i=0;i<scoreSize;++i)
第52行                  pqB.push(arrScore+i);
第53行          while(!pqB.empty()){
第54行                  cout<<*(pqB.top())<<endl;
第55行                  pqB.pop();
第56行          }
第57行
第58行          return 0;
第59行  }
```

第三节　迭代器

第四节　函数对象

函数对象(function Object或Functor)是STL提供的一类主要组件，它使得STL的应用更加灵活方便，从而增加算法的灵活性。大多数STL算法都可以用一个函数对象作为参数。所谓函数对象，其实就是一个行为类似函数的对象，它可以不需要参数，也可以带有若干参数，其功能是获取一个值，或者改变操作的状态。在C++程序中，任何普通的函数和重载了括号运算符的类的对象都满足函数对象的特

征，因此都可以作为函数对象传递给算法作为参数使用。

函数对象与函数指针相比较有两个方面的优点：首先如果被重载的调用操作符是inline函数则编译器能够执行内联编译，提供可能的性能好处；其次函数对象可以拥有任意数目的额外数据，用这些数据可以缓冲结果，也可以缓冲有助于当前操作的数据。

函数对象是一个类，它重载了函数调用操作符operator() ,该操作符封装了一个函数的功能。典型情况下函数对象被作为实参传递给泛型算法，当然我们也可以定义独立的函数对象实例。

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。函数对象（也称“算符”）是重载了“()”操作符的普通类对象。因此从语法上讲，函数对象与普通的函数行为类似。

用函数对象代替函数指针有几个优点，首先，因为对象可以在内部修改而不用改动外部接口，因此设计更灵活，更富有弹性。函数对象也具备有存储先前调用结果的数据成员。在使用普通函数时需要将先前调用的结果存储在全程或者本地静态变量中，但是全程或者本地静态变量有某些我们不愿意看到的缺陷。

其次，在函数对象中编译器能实现内联调用，从而更进一步增强了性能。这在函数指针中几乎是不可能实现的。

STL提供了标准的函数对象。为了更好地理解STL函数对象，看看例程7-14。在第49行能看到main()函数中应用了countIF()函数(与STL的count_if相似)，而查看countIF()的申明与定义，尤其是第9行的 if (Pred(*First))++count;能看出Pred()函数或仿函数，仅能处理一个参数，而第49行countIF()函数的功能是统计小于30的数值，LESS是一个对象，从其重载的括号运算符代码 bool operator()(const T&Left,const T&Right)const{return (Left<Right);}能看出需要两个参数。这就出现了矛盾！在主函数中的countIF()函数使用bind2ND(与STL的bind2nd相似)函数将LESS的括号操作符中的第二个参数固化。

例程7-14

```
第1行  #include <iostream>
第2行  #include<vector>
第3行  using namespace std;
第4行  //根据STL的algorithm编写，有所调整。
第5行  template<typename InputIterator,typename Predicator>
第6行  inline int countIF(InputIterator First,InputIterator Last,Predicator Pred){
第7行          int count=0;
第8行          for (;First!=Last;++First)
第9行                  if (Pred(*First))++count;
第10行
第11行         return (count);
第12行 }
第13行 template<typename Arg1,typename Result>
第14行 struct UnaryBase{//一元函数的基类，参考STL的unary_function
第15行         typedef Arg1 ubFirstArgument;//第一个参数
第16行         typedef Result ubResultData;//结果参数
第17行 };
第18行 template<typename Arg1,typename Arg2,typename Result>
第19行 struct BinaryBase{//二元函数的基类，参考STL的binary_function基类
第20行         typedef Arg1 bbFirstArgument;//定义第一个参数类型
第21行         typedef Arg2 bbSecondArgument;//定义第二个参数类型
第22行         typedef Result bbResultData;//返回结果类型
第23行 };
第24行 template<typename T>
第25行 struct LESS:public BinaryBase<T, T, bool>{//小于，参考STL的less派生类
第26行         bool operator()(const T&Left,const T&Right)const{return (Left<Right);}
第27行 };
第28行 //参考STL编写
```

```
第29行  template<typename BinOP>
第30行  class binder2ND:public UnaryBase<typename BinOP::bbFirstArgument,typename BinOP::bbResultData>{
第31行  protected:
第32行          BinOP OP;
第33行          typename BinOP::bbSecondArgument Arg2;
第34行  public:
第35行          binder2ND(const BinOP&Oper,const typename BinOP::bbSecondArgument & valRight):OP(Oper),Arg2(valRigh
        t){}
第36行          ubResultData operator()(const ubFirstArgument &ubArg1)const{return OP(ubArg1,Arg2);}
第37行  };
第38行  //参考STL编写
第39行  template<typename BinOP,typename rightVal>
第40行  binder2ND<BinOP> bind2ND(const BinOP&OP,const rightVal&vRight){
第41行          return binder2ND<BinOP>(OP,vRight);
第42行  };
第43行
第44行  int main()
第45行  {
第46行          vector<int> myVec;
第47行          for(int i=0;i<50;++i)myVec.push_back(rand()%100);
第48行
第49行          int countNum=countIF(myVec.begin(),myVec.end(),bind2ND(LESS<int>(),30));
第50行          cout<<"数量="<<countNum<<endl;
第51行
第52行          return 0;
第53行  }
```

bind2ND函数被执行，将返回binder2ND()对象，在其binder2ND构造函数中，输入LESS对象和两个参数中的第二个参数。在binder2ND对象中的圆括号重载中仅需要输入一个参数，但需要比较的第二个参数已经通过其构造函数传入，因此实现了两个参数的比较。从这个意义上讲，经过bind2ND()函数后，调用binder2ND对象的单参数圆括号运算符，在函数内，该单参数函数传入一个参数，执行OP(ubArg1,Arg2)，OP为LESS，调用LESS的圆括号操作符，比较ubArg1和Arg2，并返回bool类型值。

例程7-14有助于理解STL的标准函数对象，以及这些函数对象的适配器。在STL中，将具有0个参数的函数对象称之为产生器(Generator)、1个参数称之为一元函数(unary function)，2个参数称之为二元函数(binary function)。另外，将判断真假比较大小返回bool型的函数对象称之为谓词(predicate)，1个参数的称之为一元谓词，2个参数称之为二元谓词，例程中LESS就是一个二元谓词。

表7-9是STL标准库中的二元函数对象，例程7-15其应用示例。

表7-9：算术运算函数对象

函数对象	类型	描述
plus()	Binary，算术	相当于加法，输入两个参数类型为T的操作数X、Y，返回X+Y
minus()	Binary，算术	相当于减法，输入两个参数类型为T的操作数X、Y，返回X-Y
multiplies()	Binary，算术	相当于乘法，输入两个参数类型为T的操作数X、Y，返回X*Y
divides()	Binary，算术	相当于除法，输入两个参数类型为T的操作数X、Y，返回X/Y
modulus()	Binary，算术	相当于求模(求余数)，输入两个参数类型为T的操作数X、Y，返回X%Y
negate()	Unary，算术	相当于求反，输入一个参数类型为T的操作数X，返回-X

例程7-15

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

int main()
{
        vector<int> myVec(100);
        for(int i=0;i<100;++i)myVec.push_back(rand()%100);
        cout<<accumulate(myVec.begin(),myVec.end(),0,plus<int>());

        return 0;
}
```

表7-10是STL标准库中的谓词函数对象，例程7-16其应用示例。

表7-10：STL标准库中的谓词函数对象

函数对象	类型	描述
equal_to()	Binary，关系	相等。输入两个类型为T的操作数X、Y，返回X==Y
not_equal_to()	Binary，关系	不等。输入两个类型为T的操作数X、Y，返回X!=Y
less()	Binary，关系	小于。输入两个类型为T的操作数X、Y，返回X<Y
greater()	Binary，关系	大于。输入两个类型为T的操作数X、Y，返回X>Y
less_equal()	Binary，关系	小于等于。输入两个类型为T的操作数X、Y，返回X<=Y
greater_equal()	Binary，关系	大于等于。输入两个类型为T的操作数X、Y，返回X>=Y
logical_not()	Unary，逻辑	逻辑取反。输入类型为T的一个操作数X，返回!X
logical_and()	Binary，逻辑	逻辑与。输入两个类型为T的操作数X、Y，返回X&&Y
logical_or()	Binary，逻辑	逻辑或。输入两个类型为T的操作数X、Y，返回X||Y

例程7-16

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

int main()
{
        vector<int> myVec;
        for(int i=0;i<100;++i)myVec.push_back(rand()%100);
        cout<<"\nBefore Sorting\n";
        copy(myVec.begin(),myVec.end(),ostream_iterator<int>(cout," "));

```

```
第16行          cout<<"\n\nAfter Sorting.From Greater\n\n";
第17行          sort(myVec.begin(),myVec.end(),greater<int>());
第18行          copy(myVec.begin(),myVec.end(),ostream_iterator<int>(cout," "));
第19行
第20行          cout<<"\n\nAfter Sorting.From Less.\n\n";
第21行          sort(myVec.begin(),myVec.end(),less<int>());
第22行          copy(myVec.begin(),myVec.end(),ostream_iterator<int>(cout," "));
第23行          cout<<"\n\n";
第24行
第25行          int countGreater75=count_if(myVec.begin(),myVec.end(),bind2nd(greater<int>(),75));
第26行          cout<<"大于75的个数="<<countGreater75<<endl<<endl;
第27行
第28行          int countNotGreater75=count_if(myVec.begin(),myVec.end(),bind2nd(less_equal<int>(),75));
第29行          cout<<"不大于75的个数="<<countNotGreater75<<endl<<endl;
第30行
第31行          return 0;
第32行  }
```

表7-11是STL标准库中的函数适配器，例程7-17、例程7-18其应用示例。

表7-11：STL标准库中的函数适配器函数

函数对象	描述
bindlst	绑定固定值到二元函数的第一个参数位置
bind2nd	绑定固定值到二元函数的第二个参数位置
not1	生成一元函数的逻辑反
not2	生成二元函数的逻辑反
ptr_fun	将非成员函数适配成函数对象，返回一元或者二元函数对象
mem_fun	将成员函数适配成函数对象，返回一元或者二元函数对象，适用于对象指针时；
mem_fun_ref	将成员函数适配成函数对象，返回一元或者二元函数对象，适用于对象本体时；

例程7-17

```
第1行   #include <iostream>
第2行   #include<algorithm>
第3行   #include<functional>
第4行   #include<numeric>
第5行   #include<vector>
第6行   using namespace std;
第7行
第8行   bool lessThan(int A,int B){
第9行           return A<B;
第10行  }
第11行  int main()
第12行  {
第13行          vector<int> myVec;
第14行          for(int i=0;i<100;++i)myVec.push_back(rand()%100);
第15行
```

```
第16行          //ptr_fun与bind2nd的使用
第17行          int countLess75=count_if(myVec.begin(),myVec.end(),bind2nd(ptr_fun(lessThan),75));
第18行          cout<<"小于75的数量="<<countLess75<<endl;
第19行
第20行          //ptr_fun与bind1st的使用
第21行          int countGreaterEqual75=count_if(myVec.begin(),myVec.end(),bind1st(ptr_fun(lessThan),75));
第22行          cout<<"大于等于75的数量="<<countGreaterEqual75<<endl;
第23行
第24行          //ptr_fun、bind2nd及not1的使用
第25行          countGreaterEqual75=count_if(myVec.begin(),myVec.end(),not1(bind2nd(ptr_fun(lessThan),75)));
第26行          cout<<"大于等于75的数量="<<countGreaterEqual75<<endl;
第27行
第28行          //ptr_fun、bind2nd及not2的使用
第29行          countGreaterEqual75=count_if(myVec.begin(),myVec.end(),bind2nd(not2(ptr_fun(lessThan)),75));
第30行          cout<<"大于等于75的数量="<<countGreaterEqual75<<endl;
第31行
第32行          return 0;
第33行  }
```

例程7-18

```
第1行   #include <iostream>
第2行   #include<algorithm>
第3行   #include<functional>
第4行   #include<numeric>
第5行   #include<vector>
第6行   using namespace std;
第7行
第8行   class Rectangle{
第9行   public:
第10行          Rectangle(int W,int H):Width(W),Height(H){}
第11行          void displayArea(){
第12行                  cout<<"Width="<<Width<<" Height="<<Height<<" Area="<<Width*Height<<endl;
第13行          }
第14行  private:
第15行          int Width,Height;
第16行  };
第17行  int main()
第18行  {
第19行          vector<Rectangle> myRectangle;
第20行          for(int i=0;i<50;++i)
第21行                  myRectangle.push_back(Rectangle((rand()%10+1),(rand()%10+1)));
第22行
第23行          for_each(myRectangle.begin(),myRectangle.end(),mem_fun_ref(&Rectangle::displayArea));
第24行
第25行          cout<<"\n\n指针形态\n\n";
```

```
第26行          vector<Rectangle *> pRectangle;
第27行          for(int i=0;i<50;++i)
第28行                  pRectangle.push_back(new Rectangle((rand()%10+1),(rand()%10+1)));
第29行
第30行          for_each(pRectangle.begin(),pRectangle.end(),mem_fun(&Rectangle::displayArea));
第31行
第32行          //回复new开辟的内存空间
第33行          for(int i=0;i<50;++i)delete pRectangle[i];
第34行
第35行          return 0;
第36行  }
```

第五节　函数对象体系演进

例程7-19

```
第1行   #include<iostream>
第2行   #include<vector>
第3行   #include<functional>
第4行   using namespace std;
第5行
第6行   template<typename InputIterator, typename ToDo>
第7行   int countIF(InputIterator First, InputIterator Last, ToDo DoSome){
第8行           int count = 0;
第9行           for (; First != Last; ++First){
第10行                  if (DoSome(*First))++count;
第11行          }
第12行          return (count);
第13行  }
第14行  template<typename T, typename ToDo>
第15行  void forEach(T beg, T end, ToDo doSome){
第16行          while (beg != end)doSome(*beg++);
第17行  }
第18行
第19行  template<typename T>
第20行  bool lessThan(T a, T b){
第21行          return a < b;
第22行  }
第23行  template<typename T>
第24行  struct LT{
第25行          bool operator()(T a, T b){return a < b;}
第26行  };
第27行  template<typename T>
第28行  struct LTnum{
第29行          LTnum(T n) :num(n){ ; }
第30行          bool operator()(T a){ return a < num; }
第31行          T num;
```

```
第32行  };
第33行  template<typename T>
第34行  bool greaterThan(T a, T b){
第35行          return a > b;
第36行  }
第37行  template<typename T>
第38行  struct GT{
第39行          bool operator()(T a, T b){ return a> b; }
第40行  };
第41行  template<typename T>
第42行  void bubbleSort(T beg, T end){
第43行          T p, last = end;
第44行          while (true){
第45行                  p = beg;
第46行                  while (true){
第47行                          if ((p + 1) == last)break;
第48行                          if ((*p) < *(p + 1)){        auto tmp = *p;*p = *(p + 1);*(p + 1) = tmp;}
第49行                          ++p;
第50行                  }
第51行                  --last; if (last == beg)break;
第52行          }
第53行  }
第54行
第55行  template<typename T, typename ToDo>
第56行  void bubbleSort(T beg, T end, ToDo Compare){
第57行          T p, last = end;
第58行          while (true){
第59行                  p = beg;
第60行                  while (true){
第61行                          if ((p + 1) == last)break;
第62行                          if (Compare(*p, *(p + 1))){        auto tmp = *p;        *p = *(p + 1);*(p + 1) = tm
p;}
第63行                          ++p;
第64行                  }
第65行                  --last; if (last == beg)break;
第66行          }
第67行  }
第68行
第69行  int main(){
第70行          int dataA[] = { 121, 102, 56, 34, 67, 11, 340 };
第71行          vector<int> myVec = { 12, 43, 45, 12, 45, 67, 78, 2, 234 };
第72行
第73行          unsigned arrSize = sizeof(dataA) / sizeof(dataA[0]);
第74行
```

```
第75行          forEach(dataA, dataA + arrSize, [](int x){cout << x << "\t"; });
第76行          cout << endl << "forEach(dataA, dataA + arrSize, bind2nd(less<int>(),100));" << endl;
第77行          cout << endl << countIF(dataA, dataA + arrSize, bind2nd(less<int>(), 100)) << endl;
第78行          cout << endl;
第79行
第80行          bubbleSort(dataA, dataA + arrSize);
第81行          forEach(dataA, dataA + arrSize, [](int x){cout << x << "\t"; });
第82行          cout << endl;
第83行
第84行          //按3的余数排序
第85行          cout << "%3" << endl;
第86行          bubbleSort(dataA, dataA + arrSize, [](int x, int y){return (x % 3)>(y % 3); });
第87行          forEach(dataA, dataA + arrSize, [](int x){cout << x << "\t"; });
第88行          cout << endl;
第89行          cout << lessThan(5, 10) << endl;
第90行          //降序
第91行          bubbleSort(dataA, dataA + arrSize, lessThan<int>);
第92行          forEach(dataA, dataA + arrSize, [](int x){cout << x << "\t"; });
第93行          cout << endl;
第94行
第95行          //升序
第96行          bubbleSort(dataA, dataA + arrSize, greaterThan<int>);
第97行          forEach(dataA, dataA + arrSize, [](int x){cout << x << "\t"; });
第98行          cout << endl;
第99行
第100行         cout << endl << "bubbleSort(myVec.begin(), myVec.end(), LT<int>());" << endl;
第101行         bubbleSort(myVec.begin(), myVec.end(), LT<int>());
第102行         forEach(myVec.begin(), myVec.end(), [](int x){cout << x << "\t"; });
第103行
第104行         cout << endl << "bubbleSort(myVec.begin(), myVec.end(), GT<int>());" << endl;
第105行         bubbleSort(myVec.begin(), myVec.end(), GT<int>());
第106行         forEach(myVec.begin(), myVec.end(), [](int x){cout << x << "\t"; });
第107行
第108行         cout << endl<<countIF(myVec.begin(), myVec.end(), [](int x){return x < 50; }) << endl;
第109行         cout << endl << countIF(myVec.begin(), myVec.end(), not1(bind2nd(less<int>(), 50))) << endl;
第110行         cout << endl << countIF(myVec.begin(), myVec.end(), LTnum<int>(50) )<< endl;
第111行
第112行         system("pause");
第113行         return 0;
第114行 }
```

如何实现countIF(myVec.begin(), myVec.end(),bind2nd(less<int>(), 50))中bind2nd(less<int>(),50)类似的功能呢？首先countIF(beg,end,ToDo)函数中，ToDo必须是一元函数、lambda函数或函数对象。因此，bind2nd()函数的返回值在此处必须是一元函数对象。虽然可以自我编写小于50的函数对象或者函数，甚至更方便的lambda函数，但已经定义了lessThan对象，应该对其复用。

从bind2nd()调用场景看应该是函数，这个函数返回一个一元函数对象。假定定义如例程7-20所示的ltNum类以及rtnUnary()函数，很

明显能满足countIF()函数的需求。

例程7-20

```
第1行  class ltNum{
第2行  public:
第3行          ltNum(int N) :Num(N){ ; }
第4行          bool operator()(int argLeft){return argLeft < Num; }
第5行  private:
第6行          int Num;
第7行  };
第8行
第9行  ltNum rtnUnary(int N){
第10行         return ltNum(N);
第11行 }
```

但例程7-20仅仅支持int类型，明显通用性不好，修改如例程7-21所示。

例程7-21

```
第1行  template<typename T>
第2行  class ltNum{
第3行  public:
第4行          ltNum(T N) :Num(N){ ; }
第5行          bool operator()(T argLeft){return argLeft < Num; }
第6行  private:
第7行          T Num;
第8行  };
第9行  template<typename T>
第10行 ltNum<T> rtnUnary(T N){
第11行         return ltNum<T>(N);
第12行 }
```

假定除了ltNum之外，还有gtNum该怎么办呢？如例程7-22所示，其中gtNum与ltNum非常相似。此时该如何调整rtnUnary()函数呢？

例程7-22

```
第1行  template<typename T>
第2行  class ltNum{
第3行  public:
第4行          ltNum(T N) :Num(N){ ; }
第5行          bool operator()(T argLeft){return argLeft < Num; }
第6行  private:
第7行          T Num;
第8行  };
第9行  template<typename T>
第10行 class gtNum{
第11行 public:
第12行         gtNum(T N) :Num(N){ ; }
第13行         bool operator()(T argLeft){ return argLeft > Num; }
第14行 private:
```

```
第15行          T Num;
第16行  };
```

能根据需要选择gtNum或者ltNum，只有将之参数化，如例程7-23所示，第4行是其调用方式。

例程7-23

```
第1行  template<typename DoSome>
第2行  DoSome rtnUnary(DoSome ToDo){
第3行          return ToDo;
第4行  }
第5行  cout << endl << countIF(myVec.begin(), myVec.end(), rtnUnary(ltNum<int>(50))) << endl;
```

通过上述演变，可以发现只要能返回一个一元函数对象，就可以满足countIF的需求，并且类的对象可以作为参数。一个类的数据成员，可以是各种类型，当然也包括int、UpDown以及ltNum或者gtNum等。可以考虑构造一个类，以类的对象作为数据成员，如例程7-24所示。在这个类中，可以将一个函数对象作为参数，并且重载了圆括号运算符，只有一个参数，因此满足countIF()的要求，其调用格式如第11行所示。不过这种方式太繁琐了，可以修改为如例程7-25所示。

例程7-24

```
第1行  template<typename DoSome,typename T>
第2行  class DoDo{
第3行  public:
第4行          DoDo(DoSome todo) :ToDo(todo){ ; }
第5行          bool operator()(T x){
第6行                  return ToDo(x);
第7行          }
第8行  private:
第9行          DoSome ToDo;
第10行 };
第11行 cout << endl << countIF(myVec.begin(), myVec.end(), DoDo<ltNum<int>,int>(ltNum<int>(50))) << endl;
```

上例还不能满足需求，因为不能传入二元函数对象，可以修改如例程7-25所示，调用方式如第12行所示。

例程7-25

```
第1行  template<typename DoSome, typename T>
第2行  class DoDo2{
第3行  public:
第4行          DoDo2(DoSome todo, T N) :ToDo(todo), Num(N){ ; }
第5行          bool operator()(T x){
第6行                  return ToDo(x, Num);
第7行          }
第8行  private:
第9行          DoSome ToDo;
第10行         T Num;
第11行 };
第12行 cout << endl << countIF(myVec.begin(), myVec.end(), DoDo2<LT<int>,int>(LT<int>(),50)) << endl;
```

这个调用方式太复杂， DoDo2<LT<int>,int>明显存在重复，第二个int能不能省去呢？这个int明显是其中LT<int>的一部分。为了能取得其数据类型，需要在LT和GT类中简单增加代码如：typedef T value_type，然后在DoDo2中通过value_type获得其对应的数据类型。例程7-26是其优化。注意与例程7-25进行比较。

例程7-26

```
template<typename T>
struct LT{
        typedef T value_type;
        bool operator()(T a, T b){return a < b;}
};

struct GT{
        typedef T value_type;
        bool operator()(T a, T b){ return a> b; }
};

template<typename DoSome>
class DoDo2{
public:
        DoDo2(DoSome todo, typename DoSome::value_type N) :ToDo(todo), Num(N){ ; }
        bool operator()(typename DoSome::value_type x){
                return ToDo(x, Num);
        }
private:
        DoSome ToDo;
        typename DoSome::value_type Num;
};
cout << endl << countIF(myVec.begin(), myVec.end(), DoDo2<LT<int>>(LT<int>(),50)) << endl;
```

上例调用还是太复杂，可以用函数模式，让函数根据参数推断其类型，如例程7-27所示，调用大为简化。

例程7-27

```
template<typename DoSome>
DoDo2<DoSome> DooDoo(DoSome todo, typename DoSome::value_type N){
        return DoDo2<DoSome>(todo, N);
}
cout << endl << countIF(myVec.begin(), myVec.end(), DooDoo(LT<int>(), 50)) << endl;
```

STL支持代码countIF(myVec.begin(), myVec.end(), bind2nd(not2(less<int>()), 50))，其中not2的含义是对二元函数对象求反，其实现代码大致如例程7-28所示。

例程7-28

```
template<typename DoSome>
class csNot2{
public:
        typedef typename DoSome::value_type value_type;
        csNot2(DoSome todo) :ToDo(todo){ ; }
        bool operator()(typename DoSome::value_type a,typename DoSome::value_type b){
                return !ToDo(a, b);
        }
private:
        DoSome ToDo;
};
```

```
template<typename DoSome>
csNot2<DoSome> fxNot2(DoSome ToDo){
        return csNot2<DoSome>(ToDo);
}
cout << endl << countIF(myVec.begin(), myVec.end(), DooDoo(fxNot2(LT<int>()),50)) << endl;
```

获取类中数据成员的数据类型，可以采用另外一种更简单规范的方案，如例程7-29所示。

例程7-29

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

template<typename InputIterator, typename ToDo>
int countIF(InputIterator First, InputIterator Last, ToDo DoSome){
        int count = 0;
        while (First != Last)
                if (DoSome(*First++))++count;
        return (count);
}
template<typename arg1Type, typename rtnType>
struct uBase{//一元函数的基类，参考STL的unary_function
        typedef arg1Type uType1;//一元函数对象第一个参数数据类型
        typedef rtnType uType0;//一元函数对象返回值数据类型
};
template<typename arg1Type, typename arg2Type, typename rtnType>
struct bBase{//二元函数的基类，参考STL的binary_function基类
        typedef arg1Type bType1;//二元函数对象第一个参数数据类型
        typedef arg2Type bType2;//二元函数对象第二个参数数据类型
        typedef rtnType bType0;//二元函数对象返回值参数数据类型
};

template<typename T>
struct LT:public bBase<T,T,bool>{
        bool operator()(T a, T b){return a < b;}
};

template<typename T>
struct GT :public bBase<T, T, bool>{
        bool operator()(T a, T b){ return a> b; }
};

template<typename bFunctor>//bFunctor为二元函数对象
class csBinder2nd:public uBase<typename bFunctor::bType1,typename bFunctor::bType0>{
public:
```

```
第38行          csBinder2nd(bFunctor todo, typename bFunctor::bType2 N) :ToDo(todo), Num(N){ ; }
第39行          typename uBase::uType0 operator()(typename bFunctor::bType1 x){ return ToDo(x, Num); }
第40行  private:
第41行          bFunctor ToDo;
第42行          typename bFunctor::bType2 Num;
第43行  };
第44行  template<typename bFunctor>
第45行  csBinder2nd<bFunctor> fxBinding2nd(bFunctor todo, typename bFunctor::bType2 N){
第46行          return csBinder2nd<bFunctor>(todo, N);
第47行  }
第48行  template<typename bFunctor>
第49行  class csNot2:public bBase<typename bFunctor::bType1,typename bFunctor::bType2,typename bFunctor::bType0>{
第50行  public:
第51行          csNot2(bFunctor todo) :ToDo(todo){ ; }
第52行          typename bBase::bType0 operator()(typename bFunctor::bType1 a, typename bFunctor::bType2 b){
第53行                  return !ToDo(a, b);
第54行          }
第55行  private:
第56行          bFunctor ToDo;
第57行  };
第58行  template<typename bFunctor>
第59行  csNot2<bFunctor> fxNot2(bFunctor ToDo){
第60行          return csNot2<bFunctor>(ToDo);
第61行  }
第62行  template<typename uFunctor>
第63行  class csNot1 :public uBase<typename uFunctor::uType1,typename uFunctor::uType0>{
第64行  public:
第65行          csNot1(uFunctor todo) :ToDo(todo){ ; }
第66行          typename uBase::uType0 operator()(typename uFunctor::uType1 a){
第67行                  return !ToDo(a);
第68行          }
第69行  private:
第70行          uFunctor ToDo;
第71行  };
第72行  template<typename uFunctor>
第73行  csNot1<uFunctor> fxNot1(uFunctor ToDo){
第74行          return csNot1<uFunctor>(ToDo);
第75行  }
第76行
第77行  int main(){
第78行          vector<int> myVec = { 12, 43, 45, 12, 50, 67, 78, 2, 234 };
第79行          cout <<countIF(myVec.begin(), myVec.end(), fxNot1(fxBinding2nd(fxNot2(LT<int>()), 50))) << endl;
第80行          return 0;
第81行  }
```

第六节　算法

STL没有给标准容器添加大量的功能函数，而是选择提供多个算法，这些算法大都不依赖特定的容器类型，而是“泛型”的，可作用在不同类型的容器和不同类型的元素，可以称这些算法为反省算法(generic algorithm)，不仅可以用于vector，也可以用于list，而这些容器还可以容纳不同的数据类型，如int或double乃至各种class。对于自定义容器，只要与标准容器兼容，同样可以使用这些反省算法。

大多数算法是通过遍历由两个迭代器标记的一段元素来实现其功能，典型情况下，算法在遍历元素范围内，操纵其中的每一个元素。算法通过迭代器访问元素，这些迭代器标记了遍历元素的范围。

1、认识STL算法

例程7-30

```
第1行  //count()及count_if()函数，统计迭代器范围内满足条件的元素的个数
第2行  template<typename InputIterator,typename T>
第3行  int count(InputIterator Begin,InputIterator End,const T&Value);
第4行
第5行  template<typename InputIterator,typename boolFunction>
第6行  int count_if(InputIterator Begin,InputIterator End,boolFunction boolFUN);
```

2、不变算法

不变算法不改变容器中的内容，只是从容器中获取信息。

(1)　count、count_if：计数

count和count_if用于统计迭代器范围内满足指定条件的元素数量。count用于统计指定值在迭代器范围出现的次数；count_if用于统计满足指定条件的元素数量。count和count_if的格式如下，示例如例程7-31。

count()

```
template<typename InputIterator, typename T> typename iterator_traits<InputIterator>::difference_type
count(InputIterator First,InputIterator Last,const T&Value);
```

注：typename iterator_traits<InputIterator>::difference_type为函数count()返回值类型，此处可以理解为int。

count_if()

```
template<typename InputIterator,typename UnaryPredicate>
typename iterator_traits<InputIterator>::difference_type
count_if(InputIterator First,InputIterator Last,UnaryPredicate Pred);
```

注：typename iterator_traits<InputIterator>::difference_type为函数count_if()返回值类型，此处可以理解为int。

例程7-31

```
第1行  #include<iostream>
第2行  #include<algorithm>
第3行  #include<functional>
第4行  #include<iterator>
第5行  #include<vector>
第6行  using namespace std;
第7行  bool isOdd (int N) { return (N%2)==1;}
第8行  struct isEven{
第9行          bool operator()(int N){return (N%2)==0;}
第10行 };
第11行
第12行 int main(){
第13行         int myArr[]={9,7,2,2,5,7,7,2,3};
第14行         int arrSize=sizeof(myArr)/sizeof(myArr[0]);//计算数组数量
第15行         vector<int> myVec(myArr,myArr+arrSize);//用数组初始化myVec
第16行
第17行         int countNumA=count(myArr,myArr+arrSize,2);//2出现的次数
第18行         int countNumB=count_if(myVec.begin(),myVec.end(),bind2nd(less<int>(),5));//小于5的数量
```

```
第19行          int countNumC=count_if(myVec.begin(),myVec.end(),isOdd);//奇数的个数，调用函数
第20行          int countNumD=count_if(myVec.begin(),myVec.end(),isEven());//偶数的个数，调用结构isEven重载的括号运算符
第21行
第22行          cout<<"2的个数="<<countNumA<<" 比5小的个数="<<countNumB<<endl;
第23行          cout<<"奇数个数="<<countNumC<<" 偶数个数="<<countNumD<<endl;
第24行
第25行          return 0;
第26行  }
```

(2) lower_bound、upper_bound、equal_range：迭代器位置查找

lower_bound()用于查找指定值或条件首次出现的位置；upper_bound()用于最后一次出现的位置之后的位置；equal_range()则是两者的综合，返回值为pair类型，第一个相当于lower_bound()，第二个相当于upper_bound()。三个函数格式如下，示例如例程7-32。

```
lower_bound()
template<typename ForwardIterator,typename T> ForwardIterator
lower_bound(ForwardIterator First, ForwardIterator Last,const T&Value);
lower_bound()
template<typename ForwardIterator, typename T, typename BinayPredicate> ForwardIterator
lower_bound(ForwardIterator First,ForwardIterator Last,const T&Value,BinayPredicate Pred);
upper_bound()
template<typanem ForwardIterator, typename T> ForwardIterator
upper_bound(ForwardIterator First, ForwardIterator Last,const T&Value);
upper_bound()
template<typename ForwardIterator, typename T, typename BinaryPredicate> ForwardIterator
upper_bound(ForwardIterator First,ForwardIterator Last,const T&Value, BinaryPredicate Pred);
equal_range()
template<typename ForwardIterator,typename T> pair<ForwardIterator,ForwardIterator>
equal_range(ForwardIterator First, ForwardIterator Last,const T&Value);
equal_range()
template<typename ForwardIterator,typename T,typename BinaryPredicate> pair<ForwardIterator,ForwardIterator>
equal_range(ForwardIterator First,ForwardIterator Last,const T&Value,BinaryPredicate Pred);
```

例程7-32

```
第1行   #include<iostream>
第2行   #include<algorithm>
第3行   #include<functional>
第4行   #include<iterator>
第5行   #include<vector>
第6行   using namespace std;
第7行
第8行   bool myGT(int a,int b){return (a>b);}
第9行
第10行  int main(){
第11行          int myArr[]={9,7,2,2,5,7,7,2,3};
第12行          int arrSize=sizeof(myArr)/sizeof(myArr[0]);//计算数组数量
第13行          vector<int> myVec(myArr,myArr+arrSize);//用数组初始化myVec
第14行
第15行          sort(myVec.begin(),myVec.end());
第16行          sort(myArr,myArr+arrSize);
第17行
第18行          vector<int>::iterator pA=lower_bound(myVec.begin(),myVec.end(),2);//2在排序后首次出现的迭代器位置
第19行          vector<int>::iterator pB=upper_bound(myVec.begin(),myVec.end(),2);//2在最后一次出现的迭代器位置之后
第20行          cout<<*pA<<*pB<<endl;//输出2和3，注意upper_bound()在最后一次出现之后
第21行          cout<<"2的个数"<<pB-pA<<endl;
```

```
第22行
第23行          pair<vector<int>::iterator,vector<int>::iterator> iterPair;//iterPair为迭代器对，包括开始和结束
第24行          iterPair=equal_range(myVec.begin(),myVec.end(),2);
第25行          copy(iterPair.first,iterPair.second,ostream_iterator<int>(cout," "));//输出2的序列
第26行          cout<<"\n\n";
第27行
第28行          sort(myVec.begin(),myVec.end(),less<int>());//以less<int>排序
第29行          iterPair=equal_range(myVec.begin(),myVec.end(),7,less<int>());//less<int>查找，与排序标准保持一致
第30行          copy(iterPair.first,iterPair.second,ostream_iterator<int>(cout," "));//输出7的序列
第31行
第32行          vector<int>::iterator pX=lower_bound(myVec.begin(),myVec.end(),4,less<int>());
第33行          vector<int>::iterator pY=upper_bound(myVec.begin(),myVec.end(),8,less<int>());
第34行          copy(pX,pY,ostream_iterator<int>(cout," "));//输出5 7 7 7的序列
第35行
第36行          return 0;
第37行  }
```

(3) find、find_if：元素查找

find用于查找指定元素第一次出现的位置；find_if用于查找满足条件的第一个元素。函数格式如下，示例如例程7-33。

```
find
template<typename InputIterator,typename T>InputIterator
find(InputIterator First, InputIterator Last,const T&Value);
find_if
template<typename InputIterator,typename UnaryPredicate>InputIterator
find_if(InputIterator First, InputIterator Last,UnaryPredicate Pred);
```

例程7-33

```
第1行   #include<iostream>
第2行   #include<algorithm>
第3行   #include<functional>
第4行   #include<iterator>
第5行   #include<vector>
第6行   using namespace std;
第7行
第8行   bool isEven(int a){return (a%2==0);}//偶数判断
第9行
第10行  int main(){
第11行          int myArr[]={9,7,2,2,5,7,7,2,3};
第12行          int arrSize=sizeof(myArr)/sizeof(myArr[0]);//计算数组数量
第13行          vector<int> myVec(myArr,myArr+arrSize);//用数组初始化myVec
第14行
第15行          vector<int>::iterator pA=find(myVec.begin(),myVec.end(),5);
第16行          cout<<"5第一次出现位置:"<<pA-myVec.begin()<<endl;
第17行
第18行          vector<int>::iterator pB=find_if(myVec.begin(),myVec.end(),bind2nd(less<int>(),5));//小于5的数
第19行          cout<<"小于5的数第一次出现位置:"<<pB-myVec.begin()<<endl;
第20行
第21行          vector<int>::iterator pC=find_if(myVec.begin(),myVec.end(),isEven);//偶数
第22行          cout<<"偶数第一次出现位置:"<<pC-myVec.begin()<<endl;
```

```
第23行
第24行          return 0;
第25行  }
```

(4) binary_search：折半查找

binary_search()函数用折半法在有序序列中查找指定值或指定条件的元素。函数格式如下，示例如例程7-34。

```
binary_search
template<typename ForwardIterator,typename T>bool
binary_search(ForwardIterator First, ForwardIterator Last,const T&Value);

template<typename ForwardIterator, typename T, typename BinaryPredicate>bool
binary_search(ForwardIterator First,ForwardIterator Last,const T&Value,BinaryPredicate Pred);
```

例程7-34

```
第1行   #include<iostream>
第2行   #include<algorithm>
第3行   #include<functional>
第4行   #include<iterator>
第5行   #include<vector>
第6行   using namespace std;
第7行
第8行   bool isEven(int a){return (a%2==0);}//偶数判断
第9行
第10行  int main(){
第11行          int myArr[]={9,7,2,2,5,7,7,2,3};
第12行          int arrSize=sizeof(myArr)/sizeof(myArr[0]);//计算数组数量
第13行          vector<int> myVec(myArr,myArr+arrSize);//用数组初始化myVec
第14行
第15行          sort(myVec.begin(),myVec.end(),greater<int>());
第16行          copy(myVec.begin(),myVec.end(),ostream_iterator<int>(cout," "));
第17行
第18行          bool bFind;
第19行
第20行          bFind=binary_search(myVec.begin(),myVec.end(),5);
第21行          cout<<"找到否？"<<(bFind?"Yes":"No")<<endl;//输出Yes
第22行
第23行          bFind=binary_search(myVec.begin(),myVec.end(),6);
第24行          cout<<"找到否？"<<(bFind?"Yes":"No")<<endl;//输出No
第25行
第26行          bFind=binary_search(myVec.begin(),myVec.end(),2,greater<int>());
第27行          cout<<"找到否？"<<(bFind?"Yes":"No")<<endl;//输出Yes
第28行
第29行          return 0;
第30行  }
```

(5) for_each：元素遍历

for_each()函数用于遍历迭代器范围内的元素，并通知执行指定操作，其格式如下，其实现代码如例程7-35所示，其应用示例如例程7-36。

```
for_each
template<typename InputIterator,typename UnaryFunction>Function
for_each(InputIterator First,InputIterator Last, UnaryFunction Fun);
```

例程7-35

```
template<class InputIterator,class Function>
Function for_each(InputIterator First,InputIterator Last,Function Fun){
        while(first!=last){
                Fun(*first);
                ++first;
        }
        return Fun;
}
```

例程7-36

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

void Print(int a){cout<<a<<" ";}
template<typename T>
struct myPrint{
        void operator()(T a){cout<<a<<"\t";}
};
int main(){
        int myArr[]={9,7,2,2,5,7,7,2,3};
        int arrSize=sizeof(myArr)/sizeof(myArr[0]);//计算数组数量
        vector<int> myVec(myArr,myArr+arrSize);//用数组初始化myVec

        for_each(myVec.begin(),myVec.end(),Print);
        cout<<"\n\n";
        for_each(myVec.begin(),myVec.end(),myPrint<int>());
        cout<<"\n\n";
        for_each(myVec.begin(),myVec.end(),bind2nd(less<int>(),5));//不会有任何显示
        //上一个语句虽然被执行，但不会被输出

        return 0;
}
```

(6) mismatch：两个序列匹配比较

mismatch()函数用于比较两个序列的元素值是否相等(匹配)，或相应元素输入匹配元素后是否匹配。其格式如下，其实现代码如例程7-37所示，其应用示例如例程7-38。

```
mismatch
template<typename InputIterator1,typename InputIterator2> pair<InputIterator1,InputIterator2>
mismatch(InputIterator1 First1,InputIterator1 Last1, InputIterator2 First2);

template<typename InputIterator1,typename InputIterator2,typename BinaryPredicate>
pair<InputIterator1, InputIterator2>
mismatch (InputIterator1 First1,InputIterator1 Last1,InputIterator2 First2, BinaryPredicate Pred);
```

例程7-37

```
template<typename InputIterator1,typename InputIterator2,typename BinaryPredicate>
pair<InputIterator1,InputIterator2>
mismatch (InputIterator1 First1,InputIterator1 Last1,InputIterator2 First2,BinaryPredicate Pred){
        while((First1!=Last1)&&Pred(*First1,*First2)){++First1;++First2; }
        //如是格式1，可将Pred(*first1,*first2)替换为*first1==*first2
        return std::make_pair(First1,First2);
}
```

例程7-38

```
#include<iostream>
#include<algorithm>
#include<functional>
#include<iterator>
#include<vector>
using namespace std;
struct Compare{
        bool operator()(int a,int b){
                return ((a+b)>10);//相应搭档大于10否
        }
};
int main(){
        int myArr[]={9,7,2,2,5,7,7,2,3};
        int arrSize=sizeof(myArr)/sizeof(myArr[0]);//计算数组数量
        vector<int> myVec(myArr,myArr+arrSize);//用数组初始化myVec

        pair<vector<int>::iterator,int*>pBound=mismatch(myVec.begin(),myVec.end(),myArr);
        if(pBound.first==myVec.end()){
                cout<<"全部匹配";
        }else{
                cout<<"第"<<pBound.first-myVec.begin()<<"开始不匹配";
        }//输出全部匹配

        cout<<"\n\n";
        sort(myVec.begin(),myVec.end());
        pair<vector<int>::iterator,int*>pPair=mismatch(myVec.begin(),myVec.end(),myArr);
        if(pPair.second==myArr+arrSize){
                cout<<"全部匹配";
        }else{
                cout<<"第"<<pPair.second-myArr<<"开始不匹配";//首次不匹配出现在编号为0的位置
        }

        cout<<"\n\n";
        //注意myVec已经排序
        pair<vector<int>::iterator,int*>pPairA=mismatch(myVec.begin(),myVec.end(),myArr,Compare());
        if(pPairA.first==myVec.end()){
```

```
第37行                  cout<<"全部匹配";
第38行          }else{
第39行                  cout<<"第"<<pPairA.first-myVec.begin()<<"开始不匹配";//首次不匹配出现在编号为1的位置
第40行          }
第41行
第42行          return 0;
第43行  }
```

(7) max_element/min_element以及max/min：最大/最小元素及最大/最小

max_element()/min_element()用于返回迭代器范围最大/最小元素所在的迭代器，其格式如下所示，其实现代码大致如例程7-39，其应用示例如例程7-40。另外，STL还提供了max()和min()函数，用于比较两个元素的大小。

```
max
template<typename dataType>const dataType
max(const dataType&leftSide,const dataType&rightSide);

template<typename dataType,typename BinaryPredicate>const dataType
max(const dataType&leftSide,const dataType&rightSide,BinaryPredicate Pred);
max_element
template<typename ForwardIterator> ForwardIterator
max_element(ForwardIterator First, ForwardIterator Last);

template<typename ForwardIterator,typename BinaryPredicate>ForwardIterator
max_element(ForwardIterator first, ForwardIterator last,BinaryPredicate Pred);
min
template<typename dataType>const dataType
min(const dataType&leftSide,const dataType&rightSide);

template<typename dataType,typename BinaryPredicate>const dataType
min(const dataType&leftSide,const dataType&rightSide,BinaryPredicate Pred);
min_element
template<typename ForwardIterator> ForwardIterator
min_element(ForwardIterator First, ForwardIterator Last);

template<typename ForwardIterator,typename BinaryPredicate>ForwardIterator
min_element(ForwardIterator first, ForwardIterator last,BinaryPredicate Pred);
```

例程7-39

```
第1行   template<typename ForwardIterator,typename BinaryPredicate>
第2行   ForwardIterator max_element(ForwardIterator First,ForwardIterator Last,BinaryPredicate Pred){
第3行           if (First==Last)return Last;
第4行           ForwardIterator Largest=First;
第5行
第6行           while(++First!=Last)
第7行                   if(Pred(*Largest,*First))Largest=First;
第8行                   //或将Pred(*Largest,*First)替换为*Largest<*First
第9行           return Largest;
第10行  }
```

例程7-40

```
第1行   #include<iostream>
第2行   #include<algorithm>
第3行   #include<functional>
第4行   #include<iterator>
第5行   #include<vector>
第6行
第7行   using namespace std;
第8行   struct Compare{
```

```
第9行          bool operator()(int a,int b){
第10行                 return (-a<-b);//相反数大小比较
第11行         }
第12行 };
第13行 int main(){
第14行         int myArr[]={9,7,2,2,5,7,7,2,3};
第15行         int arrSize=sizeof(myArr)/sizeof(myArr[0]);//计算数组数量
第16行         vector<int> myVec(myArr,myArr+arrSize);//用数组初始化myVec
第17行
第18行         vector<int>::iterator pVecMax=max_element(myVec.begin(),myVec.end());
第19行         vector<int>::iterator pVecMin=min_element(myVec.begin(),myVec.end());
第20行         cout<<"myVec中的最大值="<<*pVecMax<<" 最小值="<<*pVecMin<<endl;
第21行
第22行         int *pArrMax=max_element(myArr,myArr+arrSize,Compare());
第23行         int *pArrMin=min_element(myArr,myArr+arrSize,Compare());
第24行         cout<<"myArr中的相反数最大值="<<-*pArrMax<<" 相反数最小值="<<-*pArrMin<<endl;
第25行
第26行         return 0;
第27行 }
```

(8) search_n、search：连续性查找

search_n

```
template<typename ForwardIterator,typename Size,typename T>ForwardIterator
search_n(ForwardIterator First,ForwardIterator Last,Size count,const T&Value);

template<typename ForwardIterator,typename Size,typename T,typensame BinaryPredicate> ForwardIterator
search_n(ForwardIterator First,ForwardIterator Last,Size count,const T&Value,BinaryPredicate Pred);
```

search

```
template<typename ForwardIterator1,typename ForwardIterator2> ForwardIterator1
search(ForwardIterator1 First1,ForwardIterator1 Last1,ForwardIterator2 First2,ForwardIterator2 Last2);

template<typename ForwardIterator1,typename ForwardIterator2,typename BinaryPredicate> ForwardIterator1
search(ForwardIterator1 First1,ForwardIterator1 Last1,ForwardIterator2 First2,ForwardIterator2
Last2,BinaryPredicate Pred);
```

例程7-41

```
第1行  //search的定义
第2行  template<typename ForwardIterator1,typename ForwardIterator2>
第3行  ForwardIterator1 search(ForwardIterator1 First1,ForwardIterator1 Last1,
第4行                                                  ForwardIterator2 First2,ForwardIterator2 Last2){
第5行          if (First2==Last2)return First1;
第6行
第7行          while (First1!=Last1){
第8行                  ForwardIterator1 it1 = First1;
第9行                  ForwardIterator2 it2 = First2;
第10行                 while (*it1==*it2){
第11行                         ++it1; ++it2;
第12行                         if (it2==Last2) return First1;
第13行                         if (it1==Last1) return Last1;
第14行                 }
第15行                 ++First1;
```

```
第16行          }
第17行          return Last1;
第18行  }
```

例程7-42

```
第1行   #include<iostream>
第2行   #include<algorithm>
第3行   #include<functional>
第4行   #include<iterator>
第5行   #include<vector>
第6行
第7行   using namespace std;
第8行   struct Compare{
第9行           bool operator()(int a,int b){
第10行                  return (a>=b);
第11行          }
第12行  };
第13行  int main(){
第14行          int myArr[]={9,7,2,2,5,7,7,2,3};
第15行          int arrSize=sizeof(myArr)/sizeof(myArr[0]);//计算数组数量
第16行          vector<int> myVec(myArr,myArr+arrSize);//用数组初始化myVec
第17行
第18行          //search_n示例-------------开始
第19行          vector<int>::iterator iterA=search_n(myVec.begin(),myVec.end(),2,7);//7出现两次的位置
第20行          if(iterA!=myVec.end())
第21行                  cout<<"2个7首次出现的位置="<<iterA-myVec.begin()<<endl;
第22行          else
第23行                  cout<<"Not Found!"<<endl;
第24行
第25行          vector<int>::iterator iterB=search_n(myVec.begin(),myVec.end(),3,3,Compare());//7出现两次的位置
第26行          if(iterB!=myVec.end())
第27行                  cout<<"连续3个大于等于3的数字出现位置="<<iterB-myVec.begin()<<endl;
第28行          else
第29行                  cout<<"Not Found!"<<endl;
第30行          //search_n示例-------------结束
第31行
第32行          //search示例-------------开始
第33行          vector<int> tmpVec(myArr+1,myArr+3);
第34行          //tmpVec中7,2这个序列首次出现在myVec中的位置
第35行          vector<int>::iterator iterC=search(myVec.begin(),myVec.end(),tmpVec.begin(),tmpVec.end());
第36行          if(iterC!=myVec.end())
第37行                  cout<<"tmpVec首次出现在myVec中的位置="<<iterC-myVec.begin()<<endl;
第38行          else
第39行                  cout<<"Not Found!"<<endl;
第40行          //search示例-------------结束
```

```
第41行
第42行          return 0;
第43行  }
```

(9) adjacent_find：查找连续两个相同元素

adjacent_find()函数用于查找连续两个相同的元素，返回第1个元素所在的迭代器位置，其格式如下：

```
adjacent_find
template<typename ForwardIterator>ForwardIterator
adjacent_find(ForwardIterator First,ForwardIterator Last);

template<class ForwardIterator, class BinaryPredicate>ForwardIterator
adjacent_find(ForwardIterator First,ForwardIterator Last,BinaryPredicate pred);
```

例程7-43是adjacent_find()函数的模拟等效实现。

例程7-43

```
第1行  template<typename ForwardIterator>
第2行  ForwardIterator adjacent_find(ForwardIterator First, ForwardIterator Last){
第3行          if (First != Last){
第4行                  ForwardIterator next=First;++next;
第5行                  while(next!=Last){
第6行                          if(*First==*next)return First;//针对第2形式：可换成Pred(*First,*next)
第7行                          ++First; ++next;
第8行                  }
第9行          }
第10行         return Last;
第11行 }
```

例程7-44是adjacent_find()函数的应用示例。

例程7-44

```
第1行  #include<iostream>
第2行  #include<algorithm>
第3行  using namespace std;
第4行
第5行  bool exactDivision5(int i,int j){
第6行          return (i%5==0&&j%5==0);
第7行  }
第8行
第9行  int main(){
第10行         int arrNum[]={11,12,17,17,98,95,85,11};
第11行         int arrSize=sizeof(arrNum)/sizeof(arrNum[0]);
第12行
第13行         int *iter;
第14行         iter=adjacent_find(arrNum,arrNum+arrSize);
第15行         cout<<*iter<<"在编号为"<<(iter-arrNum)<<"的位置，至少出现2次。"<<endl;
第16行
第17行         iter=adjacent_find(arrNum,arrNum+arrSize,exactDivision5);
第18行         cout<<"在编号"<<(iter-arrNum)<<"位置出现的"<<*iter
第19行                 <<"及其后的"<<*(iter+1)<<"能被5整除"<<endl;
第20行
```

```
第21行          return 0;
第22行  }
```

(10) find_end：子序列查找

find_end()函数用于在序列1[First1,Last1)中查找序列2[First2,Last2)最后一次出现的位置。如果序列1中不存在完全匹配序列2的子序列，则返回Last1。此算法被重载为两个版本，如下所示：

```
find_end
template<typename ForwardIterator1,typename ForwardIterator2>ForwardIterator1
find_end(ForwardIterator1 First1,ForwardIterator1 Last1,
ForwardIterator2 First2,ForwardIterator2 Last2);

template<typename ForwardIterator1,typename ForwardIterator2,typename BinaryPredicate>ForwardIterator1
find_end(ForwardIterator1 First1,ForwardIterator1 Last1,
ForwardIterator2 First2,ForwardIterator2 Last2,BinaryPredicate Pred);
```

例程7-45是find_end()第2种格式的等效模拟实现。

例程7-45

```
第1行  template<typename ForwardIterator1,typename ForwardIterator2,typename BinaryPredicate>
第2行  ForwardIterator1 find_end(ForwardIterator1 First1,ForwardIterator1 Last1,
                                                        ForwardIterator2 First2,ForwardIterator2 Last2,Binar
第3行  yPredicate Pred){
第4行          if(First2==Last2)return Last1;
第5行
第6行          ForwardIterator1 rtnIter=Last1;
第7行
第8行          while(First1!=Last1){
第9行                  ForwardIterator1 iter1=First1;
第10行                 ForwardIterator2 iter2=First2;
第11行                 while(pred(*it1,*it2)) {//针对第1种格式，替换*iter1==*iter2
第12行                         ++iter1; ++iter2;
第13行                         if (iter2==Last2){rtnIter=First1;break;}
第14行                         if (iter1==Last1)return rtnIter;
第15行                 }
第16行                 ++First1;
第17行         }
第18行         return rtnIter;
第19行 }
```

例程7-46是find_end()函数的应用示例，其功能是查找dividendList中最后能被divisorList中所列数3、5、7整除的数。注意要求dividendList三个连续的数，至于其原因，请查看其等效模拟代码。

例程7-46

```
第1行  #include<iostream>
第2行  #include<algorithm>
第3行
第4行  using namespace std;
第5行
第6行  bool exactDivision(int a,int b){return (a%b==0);}
第7行
第8行  int main(){
第9行          //判断dividendList最后能被divisorList整除的数
```

```
第10行          int dividendList[]={12,15,14,19,210,30,99,105,112,123,124};
第11行          int arrSizeA=sizeof(dividendList)/sizeof(dividendList[0]);
第12行
第13行          int divisorList[]={3,5,7};
第14行          int arrSizeB=sizeof(divisorList)/sizeof(divisorList[0]);
第15行
第16行          int * lastPosition;
第17行
第18行          lastPosition=find_end(dividendList,dividendList+arrSizeA,divisorList,divisorList+arrSizeB,exactDivis
ion);
第19行          if(lastPosition!=(dividendList+arrSizeA)){
第20行                  for(int i=0;i<arrSizeB;++i)
第21行                          cout<<*(lastPosition+i)<<" ";
第22行          }
第23行
第24行          return 0;
第25行  }
```

(11) find_first_of：元素出现在另外序列中

find_first_of()函数用于搜索一个序列的元素在第二个序列中第一次出现的位置，其格式如下：

```
find_first_of
template<typename ForwardIterator1,typename ForwardIterator2>ForwardIterator1
find_first_of(ForwardIterator1 First1,ForwardIterator1 Last1,
ForwardIterator2 First2,ForwardIterator2 Last2);

template<typename ForwardIterator1,typename ForwardIterator2,typename BinaryPredicate>ForwardIterator1
find_first_of(ForwardIterator1 First1,ForwardIterator1 Last1,
ForwardIterator2 First2,ForwardIterator2 Last2,BinaryPredicate Pred);
```

例程7-47是函数find_first_of()的等效模拟实现。

例程7-47

```
第1行   template<typename ForwardIterator1,typename ForwardIterator2,typename BinaryPredicate>
第2行   InputIterator find_first_of(InputIterator First1,InputIterator Last1,
                                                        ForwardIterator First2,ForwardIterator Last2,BinaryP
第3行   redicate Pred){
第4行           while(First1!=Last1){
第5行                   for(ForwardIterator iter=First2;it!=Last2; ++it){
第6行                           if(pred(*iter,*First1))//对于第1种格式，请调整为*iter==*First1
第7行                                   return First1;
第8行                   }
第9行                   ++First1;
第10行          }
第11行          return Last1;
第12行  }
```

例程7-48是find_first_of()函数的应用示例，功能是判断元音首次出现的位置。

例程7-48

```
第1行   #include<iostream>
第2行   #include<algorithm>
第3行   #include<string>
```

```
第4行  using namespace std;
第5行
第6行  int main(){
第7行          //判断myWord中第一个元音出现的位置
第8行          string myWord="China";
第9行          string vowelList="aeiou";
第10行         string::iterator firstPosition;
第11行         firstPosition=find_first_of(myWord.begin(),myWord.end(),vowelList.begin(),vowelList.end());
第12行         cout<<myWord<<"中的字母"<<*firstPosition<<"是元音，是编号为"
第13行                         <<(firstPosition-myWord.begin())<<"的字符"<<endl;
第14行
第15行         return 0;
第16行 }
```

(12) equal：两个序列相等比较

equal()函数用于判断[First1,Last1)与[First2,Last2)是否相等，如果相等返回值为true，否则为false。如果[First2,Last2)序列中的元素较[First1,Last1)中的元素个数较多，则多出的元素不予考虑。如果希望两个序列在指定区间内相等，且元素数量也相等，可以在应用equal()之前判断两个容器元素数量是否相等。

```
equal
template<typename InputIterator1,typename InputIterator2>bool
equal(InputIterator1 First1, InputIterator1 Last1,InputIterator2 First2);

template<typename InputIterator1,tyepname InputIterator2,typename BinaryPredicate>bool
equal(InputIterator1 First1,InputIterator1 Last1,InputIterator2 First2,BinaryPredicate Pred);
```

例程7-49函数equal()的等效模拟实现。

例程7-49

```
第1行  template<typename InputIterator1,typename InputIterator2,typename BinaryPredicate>
第2行  bool equal(InputIterator1 First1, InputIterator1 Last1, InputIterator2 First2,BinaryPredicate Pred){
第3行          while (First1!=Last1){
第4行                  if(!Pred(*First1,*First2))return false;//第1种格式替换为*First1 == *First2
第5行                  ++First1; ++First2;
第6行          }
第7行          return true;
第8行  }
```

例程7-50函数equal()的应用示例，比较两个字符串在忽略大小写情况下是否相等。

例程7-50

```
第1行  #include<iostream>
第2行  #include<algorithm>
第3行  #include<string>
第4行  using namespace std;
第5行
第6行  char toLower(char charA){//转换为小写字母
第7行          return (charA>=65 && charA<=90)?charA+32:charA;
第8行  }
第9行  bool comp(char a,char b){
第10行         return (toLower(a)==toLower(b));
第11行 }
```

```
第12行
第13行  int main(){
第14行          string strA="China is the world's most populous country!";
第15行          string strB="China Is The World's Most Populous Country!";
第16行          bool isEqual=equal(strA.begin(),strA.end(),strB.begin(),comp);
第17行
第18行          cout<<"strA与strB"<<(isEqual?"相等!":"不相等!");
第19行
第20行          return 0;
第21行  }
```

3、改变算法

4、排序算法

将杂乱无章的数据元素，通过一定的方法按照某种规则顺序排列元素的过程叫做排序。假定在待排序的元素中，存在多个相同值的元素，若经过排序，这些记录的相对次序保持不变，称之为稳定排序，在STL中，稳定排序都含有stable单词；否则称为非稳定排序。

对于元素序列，可以对迭代器范围内的全部元素排序，称之为“全排序”；也可以仅对部分元素排序，称之为“部分排序”。在STL中，部分排序含有单词partial。

大部分排序算法都仅对随机迭代器容器有效(注：数组被视为随机迭代器)，部分排序算法也适用于前向迭代器(必然也对随机迭代器有效)。有关STL排序算法简单介绍，如表7-12所示。

表7-12：STL排序算法简表

函数名	适用迭代器	功能描述
sort	随机迭代器	对迭代器区间内所有元素排序
stable_sort	随机迭代器	对迭代器区间内所有元素稳定排序
partial_sort	随机迭代器	对给定迭代器区间部分元素排序
partial_sort_copy	随机迭代器	对给定区间的元素复制并排序
nth_element	随机迭代器	找出给定区间的某个位置对应的元素
partition	双向迭代器	满足设定条件的元素放在前面
stable_partition	双向迭代器	(稳定排序)设定条件的元素放在前面

在STL算法中，sort()函数应用频率很高，是主要的排序函数。在STL中，sort()函数主要采用“快速排序(Quick Sort)”算法，是目前排序效率相当高的一种算法。另外，在很多STL算法实现中，还结合“内插排序算法”，使之效率更高。

(1) sort：全排序

在STL中，最常用的排序函数是sort()，其格式如下。

```
sort
template<typename RandomAccessIterator>void
sort(RandomAccessIterator First,RandomAccessIterator Last);

template<typename RandomAccessIterator,typename BinaryPredicate>void
sort(RandomAccessIterator First,RandomAccessIterator Last,BinaryPredicate Pred);
```

在第一种格式中，排序标准依赖元素自身的＜运算符。如果是C++基础数据类型，则默认支持＜运算符，如果是自定义结构或类，则须重载＜运算符，如例程7-51所示。在UpDown类中，重载＜运算符，否则第33行将不能排序。

例程7-51

```
第1行   #include<iostream>
第2行   #include<algorithm>
第3行   #include<functional>
第4行   #include<iterator>
第5行   #include<vector>
第6行   using namespace std;
第7行   class UpDown{//分数类
第8行   public:
第9行           UpDown(int U=1,int D=1):Up(U),Down(D){}
第10行          //重载＜运算符
```

```
第11行          bool operator<(const UpDown &UD)const{//未考虑负数情况
第12行                  return this->Up*UD.Down<UD.Up*this->Down;
第13行          }
第14行          //重载<运算符
第15行          friend ostream& operator<<(ostream &out,const UpDown &UD);
第16行  private:
第17行          int Up,Down;
第18行  };
第19行  ostream& operator<<(ostream &out,const UpDown &UD){
第20行          return out<<UD.Up<<"/"<<UD.Down;
第21行  }
第22行  int main(){
第23行          int myArr[]={9,7,2,2,5,7,7,2,3};
第24行          int arrSize=sizeof(myArr)/sizeof(myArr[0]);//计算数组数量
第25行          vector<int> myVec(myArr,myArr+arrSize);//用数组初始化myVec
第26行
第27行          sort(myVec.begin(),myVec.end());
第28行          copy(myVec.begin(),myVec.end(),ostream_iterator<int>(cout," "));
第29行          UpDown myUD[]={UpDown(rand()%10,rand()%10),UpDown(rand()%10,rand()%10),
第30行                                          UpDown(rand()%10,rand()%10),UpDown(rand()%10,rand()%10),
第31行                                          UpDown(rand()%10,rand()%10)};//分数数组
第32行          arrSize=sizeof(myUD)/sizeof(myUD[0]);
第33行          sort(myUD,myUD+arrSize);
第34行          copy(myUD,myUD+arrSize,ostream_iterator<UpDown>(cout," "));
第35行
第36行          return 0;
第37行  }
```

sort()函数还支持函数或函数对象作为比较函数，可以自定义，也可以用STL提供的仿函数，如例程7-52所示。

例程7-52

```
第1行   #include<iostream>
第2行   #include<algorithm>
第3行   #include<functional>
第4行   #include<iterator>
第5行   #include<string>
第6行   #include<vector>
第7行   using namespace std;
第8行
第9行   struct NameList{
第10行          string ID;//学号
第11行          string Name;//姓名
第12行          string Sex;//性别
第13行          NameList(string id,string name,string sex):ID(id),Name(name),Sex(sex){}
第14行          //＜则按ID排序
第15行          bool operator<(const NameList&rightSide)const{
```

```
第16行                  return this->ID<rightSide.ID;
第17行          }
第18行          //＞则按Name排序
第19行          bool operator>(const NameList&rightSide)const{
第20行                  return this->Name>rightSide.Name;
第21行          }
第22行  };
第23行  //先按性别排序，然后按姓名排序，可以用class或struc实现
第24行  bool sortNameListRule(const NameList&leftSide,const NameList&rightSide){
第25行          if(leftSide.Sex!=rightSide.Sex){
第26行                  return (leftSide.Sex=="男"?1:0);
第27行          }else{
第28行                  return leftSide.Name<rightSide.Name;
第29行          }
第30行  }
第31行  void printA(const NameList &NL){cout<<NL.ID<<"-"<<NL.Name<<"-"<<NL.Sex<<endl;}
第32行  void printB(const NameList &NL){cout<<NL.Name<<"-"<<NL.ID<<"-"<<NL.Sex<<endl;}
第33行  void printC(const NameList &NL){cout<<NL.Sex<<"-"<<NL.Name<<"-"<<NL.ID<<endl;}
第34行  int main(){
第35行          NameList myNameList[]={
第36行                  NameList("JIA00","贾宝玉","男"),NameList("JIA01","贾政","男"),
第37行                  NameList("JIA02","贾迎春","女"),NameList("JIA03","贾元春","女"),
第38行                  NameList("XUE00","薛宝钗","女"),NameList("XUE01","薛蟠","男"),
第39行                  NameList("LIN00","林黛玉","女"),NameList("LIN01","林如海","男")
第40行          };
第41行
第42行          sort(myNameList,myNameList+8);
第43行          for_each(myNameList,myNameList+8,printA);
第44行
第45行          cout<<"\n\n";
第46行          sort(myNameList,myNameList+8,not2(less<NameList>()));
第47行          for_each(myNameList,myNameList+8,printA);
第48行
第49行          cout<<"\n\n";
第50行          sort(myNameList,myNameList+8,greater<NameList>());
第51行          for_each(myNameList,myNameList+8,printB);
第52行
第53行          cout<<"\n\n";
第54行          sort(myNameList,myNameList+8,sortNameListRule);
第55行          for_each(myNameList,myNameList+8,printC);
第56行
第57行          return 0;
第58行  }
```

在例程7-52的NameList中，定义了三个数据成员，分别是ID、Name和Sex，都是string类型。在其中重载了＜和＞运算符，分别按ID和Name。sort()函数默认按＜排序，如第42行所示，按ID升序排列，如果需要按ID降序排列，可执行46行代码。less默认是按＜排序，

not2则对其取反，即按升序排列。第50行，则geater方式排序，greater是按＞排序，在NameList中按Name排序，因此必须重载＞运算。第54行是sortNameListRule()排序，在该函数中，首先按性别排序，然后按姓名排序。

(2) stable_sort：稳定排序

stable_sort()是稳定排序函数。稳定排序时，如果两个排序关键字相同，则元素原有位置不发生调整，而非稳定排序则不能保证(可能调整，也可能不调整)。stable_sort的格式如下，其应用示例如例程7-53。

```
stable_sort
template<typename RandomAccessIterator>void
stable_sort(RandomAccessIterator First,RandomAccessIterator Last);

template<typename RandomAccessIterator,typename BinaryPredicate> void
stable_sort(RandomAccessIterator First,RandomAccessIterator Last,BinaryPredicate Pred);
```

例程7-53

```
第1行  #include<iostream>
第2行  #include<algorithm>
第3行  #include<functional>
第4行  #include<iterator>
第5行  #include<string>
第6行  #include<vector>
第7行  using namespace std;
第8行
第9行  struct NameList{
第10行         string ID;//学号
第11行         string Name;//姓名
第12行         string Sex;//性别
第13行         NameList(string id,string name,string sex):ID(id),Name(name),Sex(sex){}
第14行         //＜则按ID排序
第15行         bool operator<(const NameList&rightSide)const{
第16行                 return this->Name.length()<rightSide.Name.length();
第17行         }
第18行 };
第19行 void printC(const NameList &NL){cout<<NL.Sex<<"-"<<NL.Name<<"-"<<NL.ID<<endl;}
第20行 int main(){
第21行         NameList myNameList[]={
第22行                 NameList("JIA00","贾宝玉","男"),NameList("LIN01","林如海","男"),
第23行                 NameList("JIA02","贾迎春","女"),NameList("JIA03","贾元春","女"),
第24行                 NameList("XUE00","薛宝钗","女"),NameList("JIA01","贾政","男"),
第25行                 NameList("LIN00","林黛玉","女"),NameList("XUE01","薛蟠","男")
第26行         };
第27行         sort(myNameList,myNameList+8);
第28行         for_each(myNameList,myNameList+8,printC);
第29行
第30行         cout<<"\n\n";
第31行         stable_sort(myNameList,myNameList+8);
第32行         for_each(myNameList,myNameList+8,printC);
第33行
第34行         return 0;
第35行 }
```

(3) partial_sort：部分排序

partial_sort()函数用于部分排序，部分排序仅需部分元素有序，其余元素可以无需。如仅需成绩前5名且有序就属于部分排序，其余成绩无需关系。虽然可以用全排序，然后取出其中需要部分，但parital_sort()效率更高，当数据量很大时，影响就比较明显。

parital_sort()格式如下，其应用示例如例程7-54。

```
partial_sort
template<typename RandomAccessIterator>void
partial_sort(RandomAccessIterator First,RandomAccessIterator Middle,RandomAccessIterator Last);

template<typename RandomAccessIterator,typename BinaryPredicate>void
partial_sort(RandomAccessIterator First,RandomAccessIterator Middle,
RandomAccessIterator Last,BinaryPredicate Pred);
```

例程7-54

```
第1行  #include<iostream>
第2行  #include<algorithm>
第3行  #include<functional>
第4行  #include<iterator>
第5行  #include<vector>
第6行  using namespace std;
第7行
第8行
第9行  int main(){
第10行         //partial_sort示例
第11行         vector<int> scoreList;
第12行         for(int i=0;i<100;++i)scoreList.push_back(rand()%100);//产生100个0-99之间的数据
第13行         copy(scoreList.begin(),scoreList.end(),ostream_iterator<int>(cout," "));
第14行
第15行         cout<<"\n\n";
第16行         partial_sort(scoreList.begin(),scoreList.begin()+10,scoreList.end());//从头开始10个数据从小到大排列
第17行         copy(scoreList.begin(),scoreList.end(),ostream_iterator<int>(cout," "));
第18行
第19行         cout<<"\n\n";
第20行         //从头开始10个数据从大到小排列
第21行         partial_sort(scoreList.begin(),scoreList.begin()+10,scoreList.end(),greater<int>());
第22行         copy(scoreList.begin(),scoreList.end(),ostream_iterator<int>(cout," "));
第23行
第24行         //partial_sort_copy示例
第25行         cout<<"\n\n";
第26行         vector<int> topList(10);
第27行         random_shuffle(scoreList.begin(),scoreList.end());//随机打乱顺序
第28行         partial_sort_copy(scoreList.begin(),scoreList.end(),topList.begin(),topList.end());
第29行         copy(topList.begin(),topList.end(),ostream_iterator<int>(cout," "));
第30行
第31行         cout<<"\n\n";
第32行         random_shuffle(scoreList.begin(),scoreList.end());//随机打乱顺序
第33行         partial_sort_copy(scoreList.begin(),scoreList.end(),topList.begin(),topList.end(),greater<int>());
第34行         copy(topList.begin(),topList.end(),ostream_iterator<int>(cout," "));
第35行
第36行         return 0;
```

```
第37行  }
```

在格式中Middle用于指定排序部分的范围，如需要排序10个，则可以用迭代器开始加上10即可。如例程第15、第20行所示。另外，partial_sort()同样支持指定比较函数或函数对象，如例程第20行所示。

(4) partial_sort_copy：部分排序并拷贝

partial_sort_copy()与partial_sort()相似，但前者还后者的基础上将排序部分排序的结果拷贝到制定的容器，如例程7-54第28行、第33行所示。partial_sort_copy的格式如下。另外，无需指定partial_sort_copy()中部分排序数量，算法可从目标迭代器的范围自动确定。

```
partial_sort_copy
template<typename InputIterator,typename RandomAccessIterator>RandomAccessIterator
partial_sort_copy(InputIterator First,InputIterator Last,
RandomAccessIterator resultFirst,RandomAccessIterator resultLast);

template<typename InputIterator,typename RandomAccessIterator,typename BinaryPredicate> RandomAccessIterator
partial_sort_copy(InputIterator First,InputIterator Last,
RandomAccessIterator resultFirst,RandomAccessIterator resultLast,BinaryPredicate Pred);
```

(5) partition：分组排序

将partition()函数归为排序有些牵强，其功能是按将元素分为两组，分组方法由函数或仿函数确定，其格式如下，应用示例如例程7-55。partition()函数适用于双向迭代器(即也适用于list容器)，其返回值类型为双向迭代器，指向分组后第二组第一个元素。

```
partition
template<typename BidirectionalIterator,typename UnaryPredicate>BidirectionalIterator
partition(BidirectionalIterator First,BidirectionalIterator Last,UnaryPredicate Pred);
```

例程7-55

```
第1行  #include<iostream>
第2行  #include<algorithm>
第3行  #include<functional>
第4行  #include<iterator>
第5行  #include<vector>
第6行  using namespace std;
第7行
第8行  bool Group(int a){return a%2;}
第9行
第10行 int main(){
第11行         //partial_sort示例
第12行         vector<int> scoreList;
第13行         for(int i=0;i<100;++i)scoreList.push_back(rand()%100);//产生100个0-99之间的数据
第14行         copy(scoreList.begin(),scoreList.end(),ostream_iterator<int>(cout," "));
第15行
第16行         cout<<"\n\n";
第17行         vector<int>::iterator iterA=partition(scoreList.begin(),scoreList.end(),Group);
第18行         copy(scoreList.begin(),scoreList.end(),ostream_iterator<int>(cout," "));
第19行
第20行         cout<<"\n\n";
第21行         sort(iterA,scoreList.end());
第22行         sort(scoreList.begin(),iterA);
第23行         copy(scoreList.begin(),scoreList.end(),ostream_iterator<int>(cout," "));
第24行
第25行         return 0;
第26行 }
```

例程7-55第21、22行利用partition()函数返回值迭代器，分别对两组进行排序，并通过第23行显示器结果。

(6) stable_partition：稳定分组排序

stable_partition()的用于与partition()相似，都是用于分组，stable则表示稳定，即根据Pred分为两组后，数的前后关系不变。相当于把元素抽走，如例程7-56所示。

```
stable_partition template<typename BidirectionalIterator,typename UnaryPredicate>BidirectionalIterator
stable_partition(BidirectionalIterator First,BidirectionalIterator Last,UnaryPredicate Pred);
```

例程7-56

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

bool Group(int a){return a%2;}

int main(){
        //stable_partition示例
        int myArr[]={9,7,2,8,5,6,7,2,3};
        vector<int> scoreList(myArr,myArr+9);

        vector<int>::iterator iterA=stable_partition(scoreList.begin(),scoreList.end(),Group);
        copy(scoreList.begin(),scoreList.end(),ostream_iterator<int>(cout," "));
        //输出为：9 7 5 7 3 2 8 6 2

        return 0;
}
```

(7) nth_element：排名n的元素

nth_element()仅能获得排名为n的元素，比如：排名第5的元素。在统计学中，中位数是指刚好将元素分为两半的元素(此处假设元素个数为奇数)。一般要求将数据排好序，然后找出该数，但如果仅仅要求知道该数，就可以使用nth_element()提升效率。nth_element()函数格式如下，其应用示例如例程7-57。

```
nth_element
template<typename RandomAccessIterator>void
nth_element(RandomAccessIterator First, RandomAccessIterator nth, RandomAccessIterator Last);

template<class RandomAccessIterator,typename BinaryPredicate>void
nth_element(RandomAccessIterator First,RandomAccessIterator nth, RandomAccessIterator Last,BinaryPredicate Pred);
```

例程7-57

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

int main(){
        //nth_element
        vector<int> myVec;
        for(int i=0;i<=100;++i)myVec.push_back(rand()%100);
        int midNum=myVec.size()/2;

```

```
第13行          cout<<"\n\n";
第14行          copy(myVec.begin(),myVec.end(),ostream_iterator<int>(cout," "));
第15行
第16行          nth_element(myVec.begin(),myVec.begin()+midNum,myVec.end());//取得中位数
第17行
第18行          cout<<"\n\n";
第19行          copy(myVec.begin(),myVec.end(),ostream_iterator<int>(cout," "));
第20行          cout<<"\n\n中位数="<<*(myVec.begin()+midNum)<<endl;
第21行
第22行          return 0;
第23行  }
```

5、搜索算法

6、集合算法

(1) set_difference：差集

用于求两个已排序集合的差集，结果集合中包含属于第一个集合但不属于第二个集合的元素，如例程7-58中第26-32行所示。

```
set_difference
template<typename InputIterator1,typename InputIterator2,typename OutputIterator>OutputIterator
set_difference(InputIterator1 First1,InputIterator1 Last1,InputIterator2 First2,InputIterator2 Last2,OutputIterator
Result);

template<typename InputIterator1,typename InputIterator2,typename OutputIterator,typename BinaryPredicate>
OutputIterator
set_difference(InputIterator1 First1,InputIterator1 Last1,
InputIterator2 First2,InputIterator2 Last2,OutputIterator Result,BinaryPredicate Pred);
```

例程7-58

```
第1行   #include<iostream>
第2行   #include<algorithm>
第3行   #include<iterator>
第4行   #include<vector>
第5行   using namespace std;
第6行
第7行   int main(){
第8行           vector<int> primeVec;//保存质数
第9行           vector<int> oddVec;//奇数
第10行
第11行          primeVec.push_back(2);primeVec.push_back(3);primeVec.push_back(5);
第12行          primeVec.push_back(7);primeVec.push_back(11);primeVec.push_back(13);
第13行          primeVec.push_back(17);primeVec.push_back(19);primeVec.push_back(23);
第14行          random_shuffle(primeVec.begin(),primeVec.end());
第15行
第16行          oddVec.push_back(3);oddVec.push_back(5);oddVec.push_back(7);
第17行          oddVec.push_back(9);oddVec.push_back(11);oddVec.push_back(13);
第18行          oddVec.push_back(15);oddVec.push_back(17);oddVec.push_back(19);
第19行          oddVec.push_back(21);oddVec.push_back(23);oddVec.push_back(25);
第20行          random_shuffle(oddVec.begin(),oddVec.end());
第21行
第22行          //以用set_difference需排序
第23行          sort(primeVec.begin(),primeVec.end());
```

```
第24行          sort(oddVec.begin(),oddVec.end());
第25行
第26行          ////set_difference示例
第27行          vector<int>resultA(oddVec.size());//结果集合，最大为第一个的长度
第28行          vector<int>::iterator rtnIter;//返回迭代器
第29行          rtnIter=set_difference(oddVec.begin(),oddVec.end(),primeVec.begin(),primeVec.end(),resultA.begin());
第30行          resultA.resize(rtnIter-resultA.begin());//根据返回迭代器截取多余长度
第31行          copy(resultA.begin(),resultA.end(),ostream_iterator<int>(cout," "));
第32行          //输出：9 15 21 25
第33行
第34行          cout<<"\n\n";
第35行          //set_intersection交集
第36行          vector<int>resultB(primeVec.size()<oddVec.size()?primeVec.size():oddVec.size());
第37行          rtnIter=set_intersection(oddVec.begin(),oddVec.end(),primeVec.begin(),primeVec.end(),resultB.begin());
第38行          resultB.resize(rtnIter-resultB.begin());//根据返回迭代器截取多余长度
第39行          copy(resultB.begin(),resultB.end(),ostream_iterator<int>(cout," "));
第40行          //输出：3 5 7 11 13 17 19 23
第41行
第42行          cout<<"\n\n";
第43行          //set_union交集，去除重复
第44行          vector<int>resultC(primeVec.size()+oddVec.size());
第45行          rtnIter=set_union(oddVec.begin(),oddVec.end(),primeVec.begin(),primeVec.end(),resultC.begin());
第46行          resultC.resize(rtnIter-resultC.begin());//根据返回迭代器截取多余长度
第47行          copy(resultC.begin(),resultC.end(),ostream_iterator<int>(cout," "));
第48行          //输出：2 3 5 7 9 11 13 15 17 19 21 23 25
第49行
第50行          cout<<"\n\n";
第51行          //set_symmetric_difference
第52行          vector<int>resultD(primeVec.size()+oddVec.size());
第53行          rtnIter=set_symmetric_difference(oddVec.begin(),oddVec.end(),
第54行                                                  primeVec.begin(),primeVec.end(),resultD.begin());
第55行          resultD.resize(rtnIter-resultD.begin());//根据返回迭代器截取多余长度
第56行          copy(resultD.begin(),resultD.end(),ostream_iterator<int>(cout," "));
第57行          //输出：2 9 15 21 25
第58行
第59行          cout<<"\n\n";
第60行          //merge：合并
第61行          vector<int>resultE(primeVec.size()+oddVec.size());
第62行          rtnIter=merge(oddVec.begin(),oddVec.end(),primeVec.begin(),primeVec.end(),resultE.begin());
第63行          resultD.resize(rtnIter-resultE.begin());//根据返回迭代器截取多余长度，此处多此一举
第64行          copy(resultE.begin(),resultE.end(),ostream_iterator<int>(cout," "));
第65行          //输出：2 3 3 5 5 7 7 9 11 11 13 13 15 17 17 19 19 21 23 23 25
第66行
```

```
第67行          //inplace_merge：合并
第68行          cout<<"\n\n";
第69行          sort(primeVec.begin(),primeVec.end());
第70行          sort(oddVec.begin(),oddVec.end());
第71行          vector<int>resultF(primeVec.size()+oddVec.size());
第72行          copy(primeVec.begin(),primeVec.end(),resultF.begin());
第73行          copy(oddVec.begin(),oddVec.end(),resultF.begin()+primeVec.size());
第74行          copy(resultF.begin(),resultF.end(),ostream_iterator<int>(cout," "));
第75行          //输出：2 3 5 7 11 13 17 19 23 3 5 7 9 11 13 15 17 19 21 23 25
第76行
第77行          cout<<"\n\n";
第78行          inplace_merge(resultF.begin(),resultF.begin()+primeVec.size(),resultF.end());
第79行          copy(resultF.begin(),resultF.end(),ostream_iterator<int>(cout," "));
第80行          //输出：2 3 3 5 5 7 7 9 11 11 13 13 15 17 17 19 19 21 23 23 25
第81行
第82行          return 0;
第83行      }
```

(2) set_intersection：交集

用于求两个已排序集合的交集，即并存于两个集合的元素，如例程7-58中第35-40行所示。

```
set_intersection
template<typename InputIterator1,typename InputIterator2,typename OutputIterator> OutputIterator
set_intersection(InputIterator1 First1,InputIterator1 Last1,
InputIterator2 First2,InputIterator2 Last2,OutputIterator Result);

template<typename InputIterator1,typename InputIterator2,typename OutputIterator,typename BinaryPredicate>
OutputIterator
set_intersection(InputIterator1 First1,InputIterator1 Last1,
InputIterator2 First2,InputIterator2 Last2,OutputIterator Result,BinaryPredicate Pred);
```

(3) set_union：并集

用于求两个已排序集合的并集，即合并两个集合并取出重复部分元素，如例程7-58中第44-48行所示。

```
template<typename InputIterator1,typename InputIterator2,typename OutputIterator> OutputIterator
set_union(InputIterator1 First1,InputIterator1 Last1,
InputIterator2 First2,InputIterator2 Last2,OutputIterator Result);

template<typename InputIterator1,typename InputIterator2,typename OutputIterator,typename BinaryPredicate>
OutputIterator
set_union(InputIterator1 First1,InputIterator1 Last1,
InputIterator2 First2,InputIterator2 Last2,OutputIterator Result,BinaryPredicate Pred);
```

(4) set_symmetric_difference：对称差集

在第1个集合但不在第2个集合与在第2个集合但不在第1个集合的元素合集，如例程7-58中第52-57行所示。

```
template<typename InputIterator1,typename InputIterator2,typename OutputIterator> OutputIterator
set_symmetric_difference(InputIterator1 First1,InputIterator1 Last1,
InputIterator2 First2,InputIterator2 Last2,OutputIterator Result);

template<typename InputIterator1,typename InputIterator2,typename OutputIterator,typename BinaryPredicate>
OutputIterator
set_symmetric_difference(InputIterator1 First1,InputIterator1 Last1,
InputIterator2 First2,InputIterator2 Last2,OutputIterator Result,BinaryPredicate Pred);
```

(5) merge：合并

merge()函数用于合并两个有序序列，被合并两个序列所有元素都将被合并到目标容器中，且不删除重复元素(这与set_union()函数不同)，目标容器将自动生成有序序列(稳定排序)，函数执行后的返回值为迭代器，指向结果容器最后一个元素之后，其格式如下，其示例如例程7-58第61-64行所示。

```
merge
template<typename InputIterator1,typename InputIterator2,typename OutputIterator>OutputIterator
merge(InputIterator1 First1,InputIterator1 Last1,
InputIterator2 First2,InputIterator2 Last2,OutputIterator Result);
```

```
template<typename InputIterator1,typename InputIterator2,typename OutputIterator,typename BinaryPredicate>
OutputIterator merge(InputIterator1 first1,InputIterator1 last1,
InputIterator2 first2,InputIterator2 last2,OutputIterator result,BinaryPredicate Pred);
```

(6) inplace_merge：归并排序

如果一个序列中的前后两部分都已经按照相同标准排序，相当于[first,middle)和[middle,last)，则inplace_merge()函数可将之稳定有序归并在一起，且遵守原有排序规则，如例程7-58第67-80行所示。resultF由primeVec和oddVec组成，并按相同标准排序(升序)，middle为resultF.begin()+primeVec.size()，在这个位置之前为primeVec元素，在此及其之后为oddVec。inplace_merge()即照此排序。

inplace_merge()的格式如下：

```
inplace_merge
template<typename BidirectionalIterator>void
inplace_merge(BidirectionalIterator First, BidirectionalIterator Middle,BidirectionalIterator Last);

template<typename BidirectionalIterator,typename BinaryPredicate>void
inplace_merge(BidirectionalIterator First,BidirectionalIterator Middle,
BidirectionalIterator Last,BinaryPredicate Pred);
```

(7) includes：子序列包含判断

includes()函数用于判断第2个迭代器范围内的元素是否被包含第1个迭代器范围之内，如果在则返回true，否则返回false，即子序列判断。

```
includes
template<typename InputIterator1,typename InputIterator2> bool
includes(InputIterator1 First1,InputIterator1 Last1,InputIterator2 First2,InputIterator2 Last2);

template<typename InputIterator1,typename InputIterator2,typename BinaryPredicate> bool
includes(InputIterator1 First1,InputIterator1 Last1,
InputIterator2 First2,InputIterator2 Last2,BinaryPredicate Pred);
```

例程7-59

```
第1行  #include<iostream>
第2行  #include<algorithm>
第3行  #include<functional>
第4行  #include<iterator>
第5行  #include<vector>
第6行  using namespace std;
第7行
第8行  int main(){
第9行          int myArr[]={9,7,2,2,5,7,7,2,3};
第10行         int arrSize=sizeof(myArr)/sizeof(myArr[0]);//计算数组数量
第11行         vector<int> myVec(myArr,myArr+arrSize);//用数组初始化myVec
第12行         vector<int>tmpVec(myArr+4,myArr+7);
第13行         sort(myVec.begin(),myVec.end());//includes要求是排序序列
第14行         sort(tmpVec.begin(),tmpVec.end());//includes要求是排序序列
第15行
第16行         bool bInclude=includes(myVec.begin(),myVec.end(),tmpVec.begin(),tmpVec.end());
第17行         cout<<(bInclude?"myVec含有tmpVec":"myVec不含有tmpVec")<<endl;
第18行
第19行         return 0;
第20行 }
```

7、移出元素

移出元素的算法都含有remove单词，分别是remove()、remove_copy()、remove_copy_if()、remove_if()函数。从单词组合能看出，移出元素时还能执行拷贝，并可按条件移出元素。 如例程7-60所示。

例程7-60

```
第1行  #include<iostream>
```

```
#include<algorithm>
#include<iterator>
#include<vector>
using namespace std;
struct isEven{
        bool operator()(int a){return a%2;}
};
int main(){
        vector<int> vecNum,vecTmp;
        for(int i=0;i<50;++i)vecNum.push_back(i);//0-49
        vecTmp=vecNum;
        copy(vecTmp.begin(),vecTmp.end(),ostream_iterator<int>(cout," "));
        //移走元素11
        remove(vecTmp.begin(),vecTmp.end(),11);
        cout<<"\n\n";
        copy(vecTmp.begin(),vecTmp.end(),ostream_iterator<int>(cout," "));

        //移走所有奇数
        vecTmp=vecNum;//恢复vecTmp为vecNum
        vector<int>::iterator iter;
        iter=remove_if(vecTmp.begin(),vecTmp.end(),isEven());
        cout<<"\n\n";
        vecTmp.resize(iter-vecTmp.begin());
        copy(vecTmp.begin(),vecTmp.end(),ostream_iterator<int>(cout," "));

        //移走所有奇数并拷贝
        vecTmp=vecNum;//恢复vecTmp为vecNum
        vector<int> vecCopy(50);//定义拷贝目标容器
        iter=remove_copy_if(vecTmp.begin(),vecTmp.end(),vecCopy.begin(),isEven());
        cout<<"\n\nremove_copy_if后的vecTmp\n";
        copy(vecTmp.begin(),vecTmp.end(),ostream_iterator<int>(cout," "));
        cout<<"\n\nremove_copy_if后的vecCopy\n";
        vecCopy.resize(iter-vecCopy.begin());
        copy(vecCopy.begin(),vecCopy.end(),ostream_iterator<int>(cout," "));

        //移走11
        vecTmp=vecNum;//恢复vecTmp为vecNum
        vector<int> vecDest(50);//copy目标容器
        iter=remove_copy(vecTmp.begin(),vecTmp.end(),vecDest.begin(),11);
        cout<<"\n\nremove_copy后的vecTmp\n";
        copy(vecTmp.begin(),vecTmp.end(),ostream_iterator<int>(cout," "));
        cout<<"\n\nremove_copy后的vecDest\n";
        vecDest.resize(iter-vecDest.begin());
        copy(vecDest.begin(),vecDest.end(),ostream_iterator<int>(cout," "));

```

```
第47行          return 0;
第48行  }
```

8、元素替换

元素替换涉及到的函数有：replace()、replace_if()、replace_copy()、replace_copy_if()。

例程7-61

```
第1行  #include<iostream>
第2行  #include<algorithm>
第3行  #include<iterator>
第4行  #include<vector>
第5行  using namespace std;
第6行  struct isEven{
第7行          bool operator()(int a){return a%2;}
第8行  };
第9行  int main(){
第10行         vector<int> vecNum,vecTmp;
第11行         for(int i=0;i<50;++i)vecNum.push_back(i);//0-49
第12行         vecTmp=vecNum;
第13行         replace(vecTmp.begin(),vecTmp.end(),11,1111);
第14行         copy(vecTmp.begin(),vecTmp.end(),ostream_iterator<int>(cout," "));
第15行
第16行         vecTmp=vecNum;
第17行         replace_if(vecTmp.begin(),vecTmp.end(),isEven(),1111);
第18行         cout<<"\n\n";
第19行         copy(vecTmp.begin(),vecTmp.end(),ostream_iterator<int>(cout," "));
第20行
第21行         vecTmp=vecNum;
第22行         vector<int>vecCopyA(50);
第23行         replace_copy_if(vecTmp.begin(),vecTmp.end(),vecCopyA.begin(),isEven(),1111);
第24行         cout<<"\n\n";
第25行         copy(vecTmp.begin(),vecTmp.end(),ostream_iterator<int>(cout," "));
第26行         copy(vecCopyA.begin(),vecCopyA.end(),ostream_iterator<int>(cout," "));
第27行
第28行         vecTmp=vecNum;
第29行         vector<int>vecCopyB(50);
第30行         replace_copy(vecTmp.begin(),vecTmp.end(),vecCopyB.begin(),11,1111);
第31行         cout<<"\n\n";
第32行         copy(vecTmp.begin(),vecTmp.end(),ostream_iterator<int>(cout," "));
第33行         copy(vecCopyB.begin(),vecCopyB.end(),ostream_iterator<int>(cout," "));
第34行
第35行         return 0;
第36行 }
```

9、元素交换算法

元素交换算法有3个函数，分别是swap()、iter_swap()、swap_ranges()，其原型如下，例程7-62是swap_ranges()函数的模拟定义，例程7-63是这3个函数的应用示例。

swap

template<typename T>void
swap(T& valueA,T& valueB);
iter_swap
template<typename ForwardIterator1,typename ForwardIterator2>void
iter_swap(ForwardIterator1 iterA,ForwardIterator2 iterB);
swap_ranges
template<typename ForwardIterator1,typename ForwardIterator2>ForwardIterator2
swap_ranges(ForwardIterator1 First1,ForwardIterator1 Last1,ForwardIterator2 First2);

例程7-62

```
template<typename ForwardIterator1,typename ForwardIterator2>
ForwardIterator2 swap_ranges(ForwardIterator1 First1,ForwardIterator1 Last1,ForwardIterator2 First2){
        while (First1!=Last1) {
                swap(*First1, *First2);
                ++First1;++First2;
        }
        return First2;
}
```

例程7-63

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

int main(){
        //swap()函数示例
        int a=99,b=11;
        cout<<"a="<<a<<" b="<<b<<endl;
        swap(a,b);
        cout<<"a="<<a<<" b="<<b<<endl;

        int arrNum[]={12,14,15,3,7};
        int arrSize=sizeof(arrNum)/sizeof(arrNum[0]);
        vector<int> vecNum(arrNum,arrNum+arrSize);
        copy(arrNum,arrNum+arrSize,ostream_iterator<int>(cout," "));cout<<"\n\n";
        copy(vecNum.begin(),vecNum.end(),ostream_iterator<int>(cout," "));cout<<"\n\n";

        //iter_swap()示例
        iter_swap(arrNum,vecNum.begin()+2);//将arrNum中的第1个与vecNum第3个交换
        copy(arrNum,arrNum+arrSize,ostream_iterator<int>(cout," "));cout<<"\n\n";
        copy(vecNum.begin(),vecNum.end(),ostream_iterator<int>(cout," "));cout<<"\n\n";

        //swap_ranges()示例
        //arrNum序列：15 14 15 3 7
        //vecNum序列：12 14 12 3 7
        swap_ranges(arrNum,arrNum+3,vecNum.begin()+1);
        copy(arrNum,arrNum+arrSize,ostream_iterator<int>(cout," "));cout<<"\n\n";
```

```
第30行          copy(vecNum.begin(),vecNum.end(),ostream_iterator<int>(cout," "));cout<<"\n\n";
第31行
第32行          return 0;
第33行  }
```

10、填充算法

填充算法包括4个函数，分别是：fill()、fill_n()、generate()、generate_n()，例程7-64是generate()函数的模拟定义，例程7-65是其示例，其格式如下所示。

```
fill
template<typename ForwardIterator,typename T>void
fill(ForwardIterator First,ForwardIterator Last,const T&Value);
fill_n
template<typename OutputIterator,typename Count,typename T>void
fill_n(OutputIterator First,Count n,const T&Value);
generate
template<typename ForwardIterator,typename Generator>void
generate(ForwardIterator First,ForwardIterator Last,Generator genFun);
generate
template<typename OutputIterator,typename Count,typename Generator>void
generate_n(OutputIterator First,Count n,Generator gen);
```

例程7-64

```
第1行   template<typename ForwardIterator,typename Generator>
第2行   void generate(ForwardIterator First,ForwardIterator Last,Generator gen){
第3行           while(First!=Last){
第4行                   *First=gen();
第5行                   ++First;
第6行           }
第7行   }
```

例程7-65

```
第1行   #include<iostream>
第2行   #include<algorithm>
第3行   #include<iterator>
第4行   #include<vector>
第5行   using namespace std;
第6行   struct uniNum{
第7行           int nowNum;
第8行           uniNum(){nowNum=0;}
第9行           int operator()(){return ++nowNum;};
第10行  };
第11行  int randNum(){return rand()%100;}
第12行  int main(){
第13行          vector<int> myVec(10);
第14行
第15行          //fill()示例
第16行          fill(myVec.begin(),myVec.end(),99);
第17行          copy(myVec.begin(),myVec.end(),ostream_iterator<int>(cout," "));cout<<"\n\n";
第18行          //输出：99 99 99 99 99 99 99 99 99 99
第19行
第20行          //fill_n()示例
第21行          fill_n(myVec.begin(),5,11);
```

```
第22行          copy(myVec.begin(),myVec.end(),ostream_iterator<int>(cout," "));cout<<"\n\n";
第23行          //输出：11 11 11 11 11 99 99 99 99 99
第24行
第25行          //generate()示例
第26行          generate(myVec.begin(),myVec.end(),uniNum());
第27行          copy(myVec.begin(),myVec.end(),ostream_iterator<int>(cout," "));cout<<"\n\n";
第28行          //输出：1 2 3 4 5 6 7 8 9 10
第29行
第30行          //generate_n()示例
第31行          generate_n(myVec.begin(),5,randNum);
第32行          copy(myVec.begin(),myVec.end(),ostream_iterator<int>(cout," "));cout<<"\n\n";
第33行          //输出：41 67 34 0 69 6 7 8 9 10，随机值可能不同
第34行
第35行          return 0;
第36行  }
```

11、reverse、reverse_copy：反序及反序拷贝

reverse()函数将[First,Last)范围内的元素反序，而reverse_copy()则是将反序后的元素拷贝到目标容器，原容器范围内的元素维持原序，其格式如下所示，应用示例如例程7-66。

```
reverse
template<typename BidirectionalIterator>void
reverse(BidirectionalIterator First,BidirectionalIterator Last);
reverse_copy
template<typename BidirectionalIterator,typename OutputIterator>OutputIterator
reverse_copy(BidirectionalIterator First,BidirectionalIterator Last,OutputIterator Result);
```

例程7-66

```
第1行   #include<iostream>
第2行   #include<algorithm>
第3行   #include<iterator>
第4行   #include<vector>
第5行   using namespace std;
第6行
第7行   int main(){
第8行           int arrNum[]={1,3,4,8,2,9,45,23};
第9行           int arrSize=sizeof(arrNum)/sizeof(arrNum[0]);
第10行          vector<int> myVec(arrSize);
第11行          cout<<"\n\n原序\n";
第12行          copy(arrNum,arrNum+arrSize,ostream_iterator<int>(cout," "));
第13行          reverse(arrNum,arrNum+arrSize);
第14行          cout<<"\n\n反序\n";
第15行          copy(arrNum,arrNum+arrSize,ostream_iterator<int>(cout," "));
第16行
第17行          reverse_copy(arrNum,arrNum+arrSize,myVec.begin());
第18行          cout<<"\n\n再反序\n";
第19行          copy(myVec.begin(),myVec.end(),ostream_iterator<int>(cout," "));
第20行
第21行          return 0;
第22行  }
```

12、rotate、rotate_copy：旋转及旋转拷贝

rotate()函数用于围绕指定元素旋转序列；rotate_copy()则将选择结果复制到制定容器中，其格式如下，应用示例如例程7-67。

```
rotate
template<typename ForwardIterator>void
rotate(ForwardIterator First,ForwardIterator Middle,ForwardIterator Last);
rotate_copy
template<typename ForwardIterator,typename OutputIterator>OutputIterator
rotate_copy(ForwardIterator First,ForwardIterator Middle,ForwardIterator Last,OutputIterator Result);
```

例程7-67

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

int main(){
        int arrNum[]={1,3,4,8,2,9,45,23};
        int arrSize=sizeof(arrNum)/sizeof(arrNum[0]);
        vector<int> myVec(arrSize);

        cout<<"\n\n旋转前\n";
        copy(arrNum,arrNum+arrSize,ostream_iterator<int>(cout," "));
        //输出：1 3 4 8 2 9 45 23
        rotate(arrNum,arrNum+3,arrNum+arrSize);
        cout<<"\n\n旋转后\n";
        copy(arrNum,arrNum+arrSize,ostream_iterator<int>(cout," "));
        //输出：8 2 9 45 23 1 3 4

        cout<<"\n\n旋转拷贝前的arrNum\n";
        copy(arrNum,arrNum+arrSize,ostream_iterator<int>(cout," "));
        //输出：8 2 9 45 23 1 3 4
        rotate_copy(arrNum,arrNum+3,arrNum+arrSize,myVec.begin());
        cout<<"\n\n旋转拷贝后的arrNum\n";
        copy(arrNum,arrNum+arrSize,ostream_iterator<int>(cout," "));
        //输出：8 2 9 45 23 1 3 4
        cout<<"\n\n旋转拷贝后的myVec\n";
        copy(myVec.begin(),myVec.end(),ostream_iterator<int>(cout," "));
        //输出：45 23 1 3 4 8 2 9

        return 0;
}
```

13、unique、unique_copy、去重及去重拷贝

unique()函数的功能是删除连续重复的元素，仅留下一个；unique_copy()函数与unique()功能相同，但将结果拷贝到目标容器。示例如例程7-68。

```
unique
template<typename ForwardIterator>ForwardIterator
unique(ForwardIterator First,ForwardIterator Last);

template<typename ForwardIterator,typename BinaryPredicate>ForwardIterator
unique(ForwardIterator First,ForwardIterator Last,BinaryPredicate Pred);
```

unique_copy
template<typename InputIterator,typename OutputIterator>OutputIterator
unique_copy (InputIterator First, InputIterator Last,OutputIterator Result);

template<typename InputIterator,typename OutputIterator,typename BinaryPredicate>OutputIterator
unique_copy(InputIterator First, InputIterator Last,OutputIterator Result,BinaryPredicate Pred);

例程7-68

```
第1行  #include<iostream>
第2行  #include<algorithm>
第3行  #include<iterator>
第4行  #include<vector>
第5行  using namespace std;
第6行
第7行  int main(){
第8行          int arrNum[]={2,3,4,4,7,4,5,3,3,7};
第9行          int arrSize=sizeof(arrNum)/sizeof(arrNum[0]);
第10行         vector<int> myVec(arrNum,arrNum+arrSize),tmpVec;
第11行         vector<int>::iterator iter;
第12行
第13行         tmpVec=myVec;
第14行         cout<<"\n\nunique()函数执行前的tmpVec\n";
第15行         copy(tmpVec.begin(),tmpVec.end(),ostream_iterator<int>(cout," "));
第16行
第17行         iter=unique(tmpVec.begin(),tmpVec.end());//iter指向去重后最后一个元素
第18行         cout<<"\n\nunique()函数执行后，tmpVec执行resize()前\n";
第19行         copy(tmpVec.begin(),tmpVec.end(),ostream_iterator<int>(cout," "));
第20行         tmpVec.resize(iter-tmpVec.begin());
第21行         cout<<"\n\nunique()函数执行后，tmpVec执行resize()后\n";
第22行         copy(tmpVec.begin(),tmpVec.end(),ostream_iterator<int>(cout," "));
第23行         //tmpVec.end()可以调整为iter，同为去重后的序列
第24行
第25行         tmpVec=myVec;
第26行         vector<int> copyDest(arrSize);
第27行
第28行         cout<<"\n\nunique_copy()函数执行前的tmpVec\n";
第29行         copy(tmpVec.begin(),tmpVec.end(),ostream_iterator<int>(cout," "));
第30行
第31行         iter=unique_copy(tmpVec.begin(),tmpVec.end(),copyDest.begin());//iter指向去重后最后一个元素
第32行         cout<<"\n\nunique()函数执行后，tmpVec元素列表\n";
第33行         copy(tmpVec.begin(),tmpVec.end(),ostream_iterator<int>(cout," "));
第34行
第35行         cout<<"\n\nunique()函数执行后，copyDest元素列表\n";
第36行         copy(copyDest.begin(),copyDest.end(),ostream_iterator<int>(cout," "));
第37行         //注意：请分析最后两个0产生的原因
第38行
第39行         return 0;
第40行 }
```

14、transform：数据变换

transform()函数有两种形式的重载。第1种形式是对一个容器迭代器范围内的元素用一元函数对象对象处理后输出到Result所指向的迭代器内。第2种形式对两个容器迭代器范围的相应元素用二元函数对象处理后输出到Result所指向的得带起范围内。transform()的实现代码如例程7-69所示，应用示例如例程progSTLTransform01。

```
transform
template<typename InputIterator,typename OutputIterator,typename UnaryOperation>OutputIterator
transform(InputIterator First1,InputIterator Last1,OutputIterator Result,UnaryOperation OP);

template<typename InputIterator1,typename InputIterator2,typename OutputIterator,typename BinaryOperation>
OutputIterator
transform (InputIterator1 First1,InputIterator1 Last1,InputIterator2 First2,
OutputIterator Result,BinaryOperation BinaryOP);
```

例程7-69

```
template<typename InputIterator,typename OutputIterator,typename UnaryOperator>
OutputIterator transform(InputIterator First1,InputIterator Last1,OutputIterator Result,UnaryOperator OP){
        while(First1!=First1){
                *Result=OP(*First1);//or: *Result=BinaryOP(*First1,*First2++);
                ++Result;++First1;
        }
        return Result;
}
```

例程7-70

```
#include<iostream>
#include<algorithm>
#include<functional>
#include<iterator>
#include<vector>
#include<string>
using namespace std;
struct createNum{
        int startNum;//起点数
        int intervalNum;//间隔数
        int orderNum;//序号
        createNum(int firstNum,int interval):startNum(firstNum),intervalNum(interval){
                orderNum=0;
        };
        int operator()(){
                ++orderNum;
                return startNum+orderNum*intervalNum;
        }
};
int main(){
        string myStr="Everywhere!Everytime!Everything!Everyone!";
        transform(myStr.begin(),myStr.end(),myStr.begin(),toupper);

        vector<int> vecNumA(10);
        vector<int> vecNumB(10);
        vector<int> vecResult(10);
```

```
第27行          cout<<"\n\n奇数序列\n";
第28行          generate(vecNumA.begin(),vecNumA.end(),createNum(1,2));
第29行          copy(vecNumA.begin(),vecNumA.end(),ostream_iterator<int>(cout," "));
第30行          //输出：3 5 7 9 11 13 15 17 19 21
第31行
第32行          cout<<"\n\n偶数序列\n";
第33行          generate(vecNumB.begin(),vecNumB.end(),createNum(0,2));
第34行          copy(vecNumB.begin(),vecNumB.end(),ostream_iterator<int>(cout," "));
第35行          //输出：2 4 6 8 10 12 14 16 18 20
第36行
第37行          //将奇数和偶数序列相加，并将结果写入vecResult中。
第38行          cout<<"\n\n结果序列\n";
第39行          transform(vecNumA.begin(),vecNumA.end(),vecNumB.begin(),vecResult.begin(),plus<int>());
第40行          copy(vecResult.begin(),vecResult.end(),ostream_iterator<int>(cout," "));
第41行          //输出：5 9 13 17 21 25 29 33 37 41
第42行
第43行          return 0;
第44行  }
```

15、copy、copy_backward：拷贝与反向拷贝

copy()函数用于复制容器中数据到指定容器；copy_backward()的功能是反向拷贝，使用方法与之类似，其格式如下，例程7-71是其copy()函数的实现代码，应用示例如例程7-73。值得注意的是，copy_backward()与反序拷贝函数reverse_copy()不同，copy_backward()拷贝结果与原序相同，可以参考例程7-72。

```
copy
template<typename InputIterator,typename OutputIterator>OutputIterator
copy(InputIterator First, InputIterator Last, OutputIterator Result);
copy_backward
template <typename BidirectionalIterator1,typename BidirectionalIterator2>BidirectionalIterator2
copy_backward(BidirectionalIterator1 First,BidirectionalIterator1 Last,BidirectionalIterator2 Result);
```

例程7-71

```
第1行   template<typename InputIterator,typename OutputIterator>
第2行   OutputIterator copy(InputIterator First,InputIterator Last,OutputIterator Result){
第3行           while (First!=Last){
第4行                   *Result=*First;
第5行                   ++Result;++First;
第6行           }
第7行           return Result;
第8行   }
```

例程7-72

```
第1行   template<typename BidirectionalIterator1,typename BidirectionalIterator2>
第2行   BidirectionalIterator2 copy_backward(BidirectionalIterator1 First,BidirectionalIterator1 Last,BidirectionalIterator2 Result){
第3行           while(Last!=First)*(--Result)=*(--Last);
第4行           return Result;
第5行   }
```

例程7-73

```
第1行   #include<iostream>
```

```
第2行  #include<algorithm>
第3行  #include<functional>
第4行  #include<iterator>
第5行  #include<vector>
第6行  using namespace std;
第7行
第8行  int main(){
第9行          int arrNum[]={1,3,5,7,9};
第10行         int arrSize=sizeof(arrNum)/sizeof(arrNum[0]);
第11行         vector<int> myVec(arrSize);
第12行         copy(arrNum,arrNum+arrSize,myVec.begin());
第13行         copy(myVec.begin(),myVec.end(),ostream_iterator<int>(cout," "));
第14行
第15行         cout<<"\n\n";
第16行         myVec.clear();myVec.resize(arrSize);
第17行         copy_backward(arrNum,arrNum+arrSize,myVec.end());
第18行         copy(myVec.begin(),myVec.end(),ostream_iterator<int>(cout," "));
第19行
第20行         return 0;
第21行 }
```

16、random_shuffle：随机乱序

random_shuffle()有2种重载方式，用于对随机迭代器容器随机乱序，其格式如下，例程7-74是第2种格式的实现方式，应用示例如例程7-75。

```
random_shuffle
template<typename RandomAccessIterator>void
random_shuffle(RandomAccessIterator First,RandomAccessIterator Last);

template<typename RandomAccessIterator,typename RandomNumberGenerator>void
random_shuffle(RandomAccessIterator First,RandomAccessIterator Last,RandomNumberGenerator& Gen);
```

例程7-74

```
第1行  template<typename RandomAccessIterator,typename RandomNumberGenerator>
第2行  void random_shuffle(RandomAccessIterator First,RandomAccessIterator Last,RandomNumberGenerator&Gen){
第3行          iterator_traits<RandomAccessIterator>::difference_type i,n;
第4行          n =(last-first);
第5行          for(i=n-1; i>0; --i){
第6行                  swap(First[i],First[Gen(i+1)]);
第7行          }
第8行  }
```

例程7-75

```
第1行  #include<iostream>
第2行  #include<algorithm>
第3行  #include<functional>
第4行  #include<iterator>
第5行  #include<vector>
第6行  using namespace std;
第7行  int myRand(int i){return std::rand()%i;}
第8行  int main(){
```

```
第9行          int arrNum[]={0,1,2,3,4,5,6,7,8,9};
第10行         int arrSize=sizeof(arrNum)/sizeof(arrNum[0]);
第11行         vector<int> myVec(arrNum,arrNum+arrSize);
第12行         random_shuffle(arrNum,arrNum+arrSize);
第13行         copy(arrNum,arrNum+arrSize,ostream_iterator<int>(cout," "));
第14行
第15行         cout<<"\n\n用自定义函数myRand()乱序\n";
第16行         random_shuffle(myVec.begin(),myVec.end(),myRand);
第17行         copy(myVec.begin(),myVec.end(),ostream_iterator<int>(cout," "));
第18行
第19行         return 0;
第20行     }
```

17、堆操作

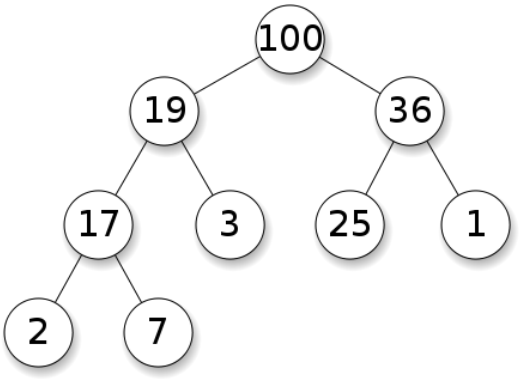

图7-1　大根堆示意图

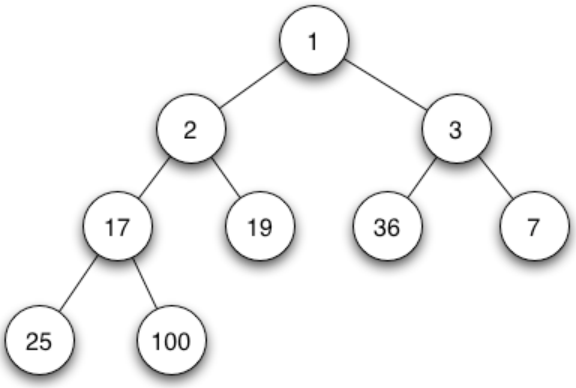

图7-2　小根堆示意图

在讨论算法时，堆(heap)通常是指组织序列元素的一种方式，根据第一个元素值的不同，堆分为大根堆(max heap)和小根堆(min heap)，或者称为大堆、小堆。在大根堆中，第一个元素具有最大元素值，而小根堆则与之相反。图7-1是大根堆示意图，图7-2是小根堆示意图。

在大根堆中，第1个元素为最大值，其子节点的值都小于父节点，以此类推，其孙节点的值小于其子节点的值。在STL的heap算法中，新加入节点一定要放在最下一层作为叶节点。由于新增元素的大小不同，可能不满足大根堆或者小根堆的定义，因此，为使新元素满足要求，需要执行上溯程序，即将新节点与其父节点相比较，如果其值比父节点大，就父子交换位置，如此一直上溯，直到不需要对换位置或直到根节点为止。小根堆的思路与之类似。

在STL中，堆算法可以独立使用，但更多是做幕后英雄，扮演priority_queue(优先队列)的助手。顾名思义，priority_queue允许以任何次序将元素置入容器中，但取出时一定是从优先级最高的元素开始取。另外，堆是一种二叉树(binary tree)且是一种完全二叉树(complete binary tree)，即除最底层叶节点外，不得有空隙，且从左至右也不得有空隙。

在STL中，有关STL的算法共有4个，分别是：make_heap()，用于将现有序列数据转换为heap；push_heap()用于向heap推入数据；pop_heap()用于从heap中弹出第1个元素；sort_heap()用于排序heap，执行后，序列就不再具有heap特征。

例程7-76是heap4个函数的应用示例。第14行通过执行make_heap()函数，使得vecNum中的数据符合max heap特征；在执行push_heap()之前，需要用push_back()函数向vecNum中追加一个数据(第19行)，然后执行push_heap()函数(第21行)，再次使数据符合max heap特征；pop_heap()函数将把最大元素移出heap，不过该元素在STL中，将被移动到容器最后，如第25-28行所示，如需继续维持heap特征，可以删除该元素(第30行)；sort_heap()用于对heap进行排序，排序后容器中的数据将不再具有heap特征，sort_heap()只能对heap进行排序，如果pop_heap()后没有删除被移植到最后的元素，将不能使用sort_heap()函数，除非对迭代器范围进行限制。

例程7-76

```
第1行  #include<iostream>
第2行  #include<algorithm>
第3行  #include<functional>
第4行  #include<iterator>
```

```
第5行  #include<vector>
第6行  using namespace std;
第7行
第8行  int main(){
第9行          int arrNum[]={25,7,1,100,36,2,3,17,19};
第10行         int arrSize=sizeof(arrNum)/sizeof(arrNum[0]);
第11行         vector<int> vecNum(arrNum,arrNum+arrSize);
第12行
第13行         //形成堆
第14行         make_heap(vecNum.begin(),vecNum.end());
第15行         cout<<"\n\nmake_heap()执行后\n";
第16行         copy(vecNum.begin(),vecNum.end(),ostream_iterator<int>(cout," "));
第17行         //输出：100 36 3 19 25 2 1 17 7
第18行
第19行         vecNum.push_back(123);//增加一个数后
第20行         cout<<"\n\npush_back(123)后执行push_heap()\n";
第21行         push_heap(vecNum.begin(),vecNum.end());
第22行         copy(vecNum.begin(),vecNum.end(),ostream_iterator<int>(cout," "));
第23行         //输出：123 100 3 19 36 2 1 17 7 25
第24行
第25行         pop_heap(vecNum.begin(),vecNum.end());
第26行         cout<<"\n\n执行pop_heap()后\n";
第27行         copy(vecNum.begin(),vecNum.end(),ostream_iterator<int>(cout," "));
第28行         //输出：100 36 3 19 25 2 1 17 7 123
第29行
第30行         vecNum.pop_back();//如需继续维持heap特征，需删除最后一个元素
第31行
第32行         sort_heap(vecNum.begin(),vecNum.end());
第33行         cout<<"\n\n执行sort_heap()后\n";
第34行         copy(vecNum.begin(),vecNum.end(),ostream_iterator<int>(cout," "));
第35行         //输出：1 2 3 7 17 19 25 36 100
第36行
第37行         return 0;
第38行 }
```

有关heap算法的4个函数格式如下：

```
make_heap()
template<typename RandomAccessIterator>void
make_heap(RandomAccessIterator First,RandomAccessIterator Last);

template<typename RandomAccessIterator,typename BinaryPredicate>void
make_heap(RandomAccessIterator First,RandomAccessIterator Last,BinaryPredicate Pred);
push_heap()
template<typename RandomAccessIterator>void
push_heap(RandomAccessIterator First,RandomAccessIterator Last);

template<typename RandomAccessIterator,typename BinaryPredicate>void
push_heap(RandomAccessIterator First,RandomAccessIterator Last,BinaryPredicate Pred);
pop_heap()
template<typename RandomAccessIterator>void
pop_heap(RandomAccessIterator First,RandomAccessIterator Last);

template<typename RandomAccessIterator,typename BinaryPredicate>void
```

pop_heap(RandomAccessIterator First,RandomAccessIterator Last,BinaryPredicate Pred);
sort_heap()
template<class RandomAccessIterator>void
sort_heap(RandomAccessIterator First,RandomAccessIterator Last);

template<typename RandomAccessIterator,typename BinaryPredicate>void
sort_heap (RandomAccessIterator First,RandomAccessIterator Last,BinaryPredicate Pred);

例程7-77sort_heap()函数的等效模拟实现。

例程7-77

```
template<typename RandomAccessIterator>
void sort_heap(RandomAccessIterator First,RandomAccessIterator Last){
        while(Last-First>1)
                pop_heap(First,Last--);
}
```

18、prev_permutation、next_permutation：排列算法

prev_permutation()和next_permutation()函数用于排列。如{1,2,3}有多少种排列方式？从数学中知道其为6种，如：{2,3,1}、{3,1,2}……。根据排序，可以将{1,2,3}视为排列中的第一个(最小)，然后是{1,3,2}，然后是{2,1,3}、{2,3,1}，最后是{3,1,2}、{3,2,1}共计6个，对于{1,2,3}就没有prev_permutation()，对于{3,2,1}就没有next_permutation()。两个函数的应用示例如下：

例程7-78

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

int main(){
        int arrNum[]={5,2,8};//在该排列中{2,5,8}最小
        int arrSize=sizeof(arrNum)/sizeof(arrNum[0]);
        do{
                cout<<"{"<<arrNum[0]<<" "<<arrNum[1]<<" "<<arrNum[2]<<"}"<<endl;
        }while(next_permutation(arrNum,arrNum+arrSize));
        //输出：{5,2,8}、{5,8,2}、{8,2,5}、{8,5,2}

        //将arrNum的数据恢复到{5,2,8}
        cout<<"\n\n";
        arrNum[0]=5;arrNum[1]=2;arrNum[2]=8;
        do{
                cout<<"{"<<arrNum[0]<<" "<<arrNum[1]<<" "<<arrNum[2]<<"}"<<endl;
        }while(prev_permutation(arrNum,arrNum+arrSize));
        //输出：{5,2,8}、{2,8,5}、{2,5,8}

        return 0;
}
```

有关prev_permutation()、next_premutation()函数的格式如下：

next_permutation
template<class BidirectionalIterator>bool
next_permutation(BidirectionalIterator First,BidirectionalIterator Last);

template<typename BidirectionalIterator,typename Compare>bool
next_permutation(BidirectionalIterator First,BidirectionalIterator Last,BinaryPredicate Pred);
prev_permutation
template<class BidirectionalIterator>bool
prev_permutation(BidirectionalIterator First,BidirectionalIterator Last);

```
template<typename BidirectionalIterator,typename Compare>bool
prev_permutation(BidirectionalIterator First,BidirectionalIterator Last,BinaryPredicate Pred);
```

例程7-79

19、数值算法

在C++STL中，提供了4个常用的数值算法函数，分别是accumulate()、adjacent_difference()、inner_product()、partial_sum()，这些函数在头文件numeric中。

(1) accumulate：元素总和

acuumulate()的功能默认是求所有元素之和，其仿函数版本可以定义其他规则，其格式如下：

acuumulate

```
template<typename InputIterator,typename T>T
accumulate(InputIterator First,InputIterator Last,T Init);

template<typename InputIterator,typename T,typename BinaryOperation>T
accumulate(InputIterator First,InputIterator Last,T Init,BinaryOperation OP)
```

例程7-80是accumulate()函数的等效模拟实现。

例程7-80

```
template<typename InputIterator,typename T>
T accumulate(InputIterator First,InputIterator Last,T Init){
        while(First!=Last) {
                Init=Init+*First;//第2种格式需替换为Init=OP(Init,*First)
                ++First;
        }
        return Init;
}
```

例程7-81是accumulate()函数的应用示例。

例程7-81

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

int main(){
        int numList[]={1,3,5,7,9};
        int arrSize=sizeof(numList)/sizeof(numList[0]);
        long resultComputingA=accumulate(numList,numList+arrSize,0);
        cout<<resultComputingA<<endl;//输出25

        long resultComputingB=accumulate(numList,numList+arrSize,1,multiplies<int>());
        cout<<resultComputingB<<endl;//输出945

        return 0;
}
```

(2) inner_product：元素内积

inner_product()的功能默认是求[First1,Last1)与[First2,Last2)的内积，其仿函数版本可以定义其他规则，其格式如下：

inner_product

template<typename InputIterator1,typename InputIterator2,typename T>T
inner_product(InputIterator1 First1,InputIterator1 Last1,InputIterator2 First2,T Init);

template<typename InputIterator1,typename InputIterator2,typename T,
typename BinaryOperation1,typename BinaryOperation2>T
inner_product(InputIterator1 First1,InputIterator1 Last1,
InputIterator2 First2,T Init,BinaryOperation1 OP1,BinaryOperation2 OP2);

例程7-82是inner_product()函数的等效模拟实现。

例程7-82

```
template<typename InputIterator1,typename InputIterator2,typename T,typename BinaryOP1,typename BinaryOP2>
T inner_product(InputIterator1 First1,InputIterator1 Last1,
                                    InputIterator2 First2,T Init,BinaryOP1 OP1,BinaryOP2,OP2){
        while(First1!=last1) {
                Init=OP1(Init,OP2(*First1,*First2));//第1种格式可替换为Init=Init+(*First1)*(*First2);
                ++First1; ++First2;
        }
        return Init;
}
```

例程7-83是inner_product()函数的应用示例。

例程7-83

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

int main(){
        int numList[]={1,3,5,7,9};
        int arrSize=sizeof(numList)/sizeof(numList[0]);
        vector<int> vecNum(numList,numList+arrSize);
        long resultComputingA=inner_product(numList,numList+arrSize,vecNum.begin(),0);
        cout<<resultComputingA<<endl;//输出：165

        long resultComputingB=inner_product(numList,numList+arrSize,vecNum.begin(),0,
                                                                        minus<float>(),plus<int>());
        cout<<resultComputingB<<endl;//输出：-50

        return 0;
}
```

(3) adjacent_difference：相邻差

adjacent_difference()的功能默认是求两个相邻元素的差，即后一个元素与前一个元素的差值，其仿函数版本可以定义其他规则，其格式如下：

adjacent_difference
template<typename InputIterator,typename OutputIterator>OutputIterator
adjacent_difference(InputIterator First,InputIterator Last,OutputIterator Result);

template<typename InputIterator,typename OutputIterator,typename BinaryOperation>OutputIterator
adjacent_difference(InputIterator First,InputIterator Last,OutputIterator result,BinaryOperation OP);

例程7-84是adjacent_difference()函数的等效模拟实现。

例程7-84

```
template<typename InputIterator,typename OutputIterator>
OutputIterator adjacent_difference(InputIterator First,InputIterator Last,OutputIterator Result){
        if(First!=Last){
                typename iterator_traits<InputIterator>::value_type Val,Prev;
                *Result=Prev= *First;
                while(++First!=Last) {
                        Val=*First;
                        *++Result=Val-Prev;//对仿函数版本，可改为*++Result++=OP(Val,Prev)
                        Prev=Val;
                }
                ++Result;
        }
        return Result;
}
```

例程7-85是adjacent_difference()函数的应用示例。

例程7-85

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

int main(){
        int numList[]={1,3,5,7,9};
        int arrSize=sizeof(numList)/sizeof(numList[0]);
        vector<int> vecNum(numList,numList+arrSize);
        vector<int>::iterator iterA=adjacent_difference(numList,numList+arrSize,vecNum.begin());
        copy(vecNum.begin(),iterA,ostream_iterator<int>(cout," "));//输出：1,2,2,2,2

        cout<<"\n\n";
        vector<int> vecData(arrSize);
        vector<int>::iterator iterB=
                                adjacent_difference(numList,numList+arrSize,vecData.begin(),multiplies<int>
());
        copy(vecData.begin(),iterB,ostream_iterator<int>(cout," "));//输出：1,3,15,35,63

        return 0;
}
```

(4) partial_sum：累积和

partial_sum()的功能默认是求累积，即当前元素与之前元素之和，其仿函数版本可以定义其他规则，其格式如下：

partial_sum
```
template<typename InputIterator,typename OutputIterator>OutputIterator
partial_sum(InputIterator First,InputIterator Last,OutputIterator Result);

template<typename InputIterator,typename OutputIterator,typename BinaryOperation>OutputIterator
partial_sum(InputIterator First,InputIterator Last,OutputIterator Result,BinaryOperation OP);
```

例程7-87是partial_sum()函数的等效模拟实现。

例程7-86

```
template<typename InputIterator,typename OutputIterator>
OutputIterator partial_sum(InputIterator First,InputIterator Last,OutputIterator Result){
        typename iterator_traits<InputIterator>::value_type Val = *first;
        if(First!=Last){
                *Result =Val;
                while (++First!=Last){
                        Val=Val + *First;//对仿函数版本可改为：Val=OP(Val,*First)
                        *++Result =Val;
                }
                ++Result;
        }
        return Result;
}
```

例程7-87是partial_sum()函数的应用示例。

例程7-87

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

int main(){
        int numList[]={1,3,5,7,9};
        int arrSize=sizeof(numList)/sizeof(numList[0]);
        vector<int> vecNum(arrSize);
        vector<int>::iterator iterA=partial_sum(numList,numList+arrSize,vecNum.begin());
        copy(vecNum.begin(),iterA,ostream_iterator<int>(cout," "));//输出：1,4,9,16,25

        cout<<"\n\n";
        vector<int>::iterator iterB=partial_sum(numList,numList+arrSize,vecNum.begin(),multiplies<int>());
        copy(vecNum.begin(),iterB,ostream_iterator<int>(cout," "));//输出：1,3,15,105,945

        return 0;
}
```

第七节　小结

第八节　阅读材料

1、STL的版本

2、STL内存配置器

3、C++11中的STL