第五章　数据结构与迭代器

本章导读

设计和实现高效的数据结构是计算机科学的重要课题之一。各种数据结构(如列表、堆栈、队列、集合以及二叉树等)在编译器构造、计算机操作系统以及文件管理等领域都有广泛的应用。在C++中，数据结构还和迭代器紧密相关，理解数据结构有助于理解迭代器的本质，从而更好地理解算法、函数对象等，也有助于更好地设计数据结构。

本章讲述几种常用数据结构的C++实现。

学习目标：

1. 认识迭代器及其分类；
2. 认识数组类；
3. 认识链表类；
4. 认识容器适配器；

本章目录

第一节　迭代器

迭代器(iterator)是用C++基本功能构建出来的重要应用性对象，能遍历STL标准容器中的部分或全部元素，是STL算法访问容器数据的通用而重要的手段。iterator不是C++关键字，也不是C++新技术，而是C++重要的实际应用。

迭代器是泛化的指针，STL算法利用迭代器对存储在容器中的元素进行遍历，迭代器提供了访问每个元素的通用方法。虽然指针也是一种迭代器，但迭代器却不是指针。指针可以访问内存中的一个地址，通过这个地址可以访问相应的内存单元；而迭代器更为抽象，可以指向容器中的一个位置，算法不必关心这个位置对应的真正物理地址。可以认为迭代器是容器和算法之间的“代理人(agent或proxy)”。

1、迭代器规范

iter_swap是STL算法中的一个函数，其功能是交换两个迭代器所指向的值。在说明该函数之前，首先看一个非迭代器版本的swap函数，如例程5-1所示，这段代码比较简单，无需赘述。

例程5-1

```
第1行  template<typename T>
第2行  void swap(T &leftVal,T &rightVal){
第3行      T tmp=leftVal;
第4行      leftVal=rightVal;
第5行      rightVal=tmp;
第6行  }
```

现在要实现迭代器版本，如例程5-2所示。一般说来，这段代码是不正确的！假定迭代器版本iterSwap适用于例程5-3场景，交换itrA和itrB的值即调用iterSwap(itrA,itrB)，结果会怎样呢？第3行*leftItr的值为int类型，而tmp是什么类型呢？Iter，不是int类型，常常是一个定义的类，类型在此不匹配，错误！如何解决呢？在解决这个问题之前，先来看看迭代器的形成，如例程5-4所示，这不是迭代器实现的典型方案。

例程5-2

```
template<typename Iter>
void iterSwap(Iter &leftIter,Iter &rightIter){
    Iter tmp=*leftIter;//*取得迭代器所指向的值
    *leftIter=*rightIter;
    *rightIter=tmp;
}
```

例程5-3

```
vector<int> myVec(50);
for(int i=0;i<50;++i)myVec[i]=rand()%100;
vector<int>::iterator itrA=myVec.begin();
vector<int>::iterator itrB=itrA+2;
```

例程5-4

```
template <typename T>
class Array{
public:
    Array(unsigned long size=100);
    T& operator[](unsigned long index){return *(this->addrDataStart+index);}
    T* begin(){return this->addrDataStart;}
    T* end(){return this->addrDataStart+this->memSize;}
    typedef T* iterator; //为T*建立类型别名为iterator
private:
    T* addrDataStart;//起始地址
    unsigned long memSize;//成员数量
};
template<typename T>
Array<T>::Array(unsigned long size):memSize(size){
    this->addrDataStart=new T[this->memSize];
}

#include<iostream>
using namespace std;

int main(){
    int arrSize=20;
    Array<int> myArr(arrSize);
    for(int i=0;i<arrSize;++i)
        myArr[i]=rand()%100;
    for(Array<int>::iterator p=myArr.begin();p!=myArr.end();++p){
        cout<<*p<<" ";
    }

    return 0;
}
```

在例程5-4中，仍然不能解决例程5-2所面临的问题，将其优化为例程5-5所示，注意第18行代码typedef T value_type，如果将例程5-2修改如例程5-6所示，tmp的类型就可以确定。

例程5-5

```
第1行  template <typename T>
第2行  class Array{
第3行  public:
第4行      Array(unsigned long size=100);
第5行      T& operator[](unsigned long index){return *(this->addrDataStart+index);}
第6行
第7行      class iterator;
第8行      iterator begin(){return iterator(this->addrDataStart);}
第9行      iterator end(){return iterator(this->addrDataStart+this->memSize);}
第10行
第11行 private:
第12行     T* addrDataStart;//起始地址
第13行     unsigned long memSize;//成员数量
第14行 };
第15行 template<typename T>
第16行 class Array<T>::iterator{//定义一个iterator类
第17行     public:
第18行         typedef T value_type;
第19行         iterator(T *p){this->nowPosition=p;}
第20行         iterator operator++(){nowPosition++; return *this;}
第21行         iterator operator+(int n){return iterator(this->nowPosition+n);}
第22行         T& operator*(){return *nowPosition;}
第23行         bool operator!=(const iterator & rightItr){return this->nowPosition!=rightItr.nowPosition;}
第24行     private:
第25行         T* nowPosition;
第26行 };
第27行 template<typename T>
第28行 Array<T>::Array(unsigned long size):memSize(size){
第29行     this->addrDataStart=new T[this->memSize];
第30行 }
第31行 template<typename Iter>
第32行 void iterSwap(Iter &leftIter,Iter &rightIter){
第33行     typename Iter::value_type tmp=*leftIter;//*取得迭代器所指向的值
第34行     *leftIter=*rightIter;
第35行     *rightIter=tmp;
第36行 }
第37行
第38行 #include<iostream>
第39行 using namespace std;
第40行
第41行 int main(){
```

```
第42行      int arrSize=20;
第43行      Array<int> myArr(arrSize);
第44行      for(int i=0;i<arrSize;++i)
第45行          myArr[i]=rand()%100;
第46行      for(Array<int>::iterator p=myArr.begin();p!=myArr.end();++p){
第47行          cout<<*p<<" ";
第48行      }
第49行
第50行      return 0;
第51行  }
```

例程5-6

```
第1行  template<typename Iter>
第2行  void iterSwap(Iter &leftIter,Iter &rightIter){
第3行      typename Iter::value_type tmp=*leftIter;//*取得迭代器所指向的值
第4行      *leftIter=*rightIter;
第5行      *rightIter=tmp;
第6行  }
```

通过在iterator类中定义value_type，然后函数中通过typename iter::value_type可以知道数据类型，但是value_type仅仅是一个标志符，其命名有什么要求呢？STL为iterator指定标准，命名为iterator_traits类，如例程5-7。除一些特定的迭代器外，需要定义5个要素。iterator_category是迭代器的类别，可以取值input_iterator_tag(输入迭代器), output_iterator_tag(输出迭代器), forward_iterator_tag(前向迭代器), bidirectional_iterator_tag(双向迭代器其)和random_access_iterator_tag(随机迭代器)；value_type是值的类型，使用*运算符返回的值；difference_type的含义是两个迭代器之间差值，即两个迭代器之间相差多个元素；pointer和reference分别指指针和引用。有了iterator_traits的帮助，例程5-6可以优化如例程5-8所示。

iterator_traits的定义。

例程5-7

```
第1行  template <class Iterator>
第2行  struct iterator_traits {
第3行      typedef typename Iterator::iterator_category iterator_category;
第4行      typedef typename Iterator::value_type value_type;
第5行      typedef typename Iterator::difference_type difference_type;
第6行      typedef typename Iterator::pointer pointer;
第7行      typedef typename Iterator::reference reference;
第8行  };
```

例程5-8

```
第1行  template<typename Iter>
第2行  void iterSwap(Iter &leftIter,Iter &rightIter){
第3行      iterator_traits<Iter>::value_type tmp=*leftIter;//*取得迭代器所指向的值
第4行      *leftIter=*rightIter;
第5行      *rightIter=tmp;
第6行  }
```

例程5-5并不太符合STL的迭代器标准，修改如例程5-9所示，特别注意第21-26行。其余项目都比较好理解，可能typedef std::random_access_iterator_tag iterator_category有些疑惑。首先迭代器分类已经在std中定义，可以直接饮用；其次iterator_category不能随意设置，每种类别都需要满足特定的条件。假定迭代器用于单向链表，则不能使用随机迭代器类别(std::random_access_iterator_tag)，单向链表不能回溯，而随机迭代器可以任意访问每个元素，可顺序依次向前，也可以依次向后，

还可跳跃多条记录等。至于每个迭代器的标准，将在下一小节阐述。另外，例程5-9也还不完全符合随机迭代器的标准，将在本章“数组类”明晰。

例程5-9

```
第1行  #include<iostream>
第2行  using namespace std;
第3行
第4行  template <typename T>
第5行  class Array{
第6行  public:
第7行      Array(unsigned long size=100);
第8行      T& operator[](unsigned long index){return *(this->addrDataStart+index);}
第9行
第10行     class iterator;
第11行     iterator begin(){return iterator(this->addrDataStart);}
第12行     iterator end(){return iterator(this->addrDataStart+this->memSize);}
第13行
第14行 private:
第15行     T* addrDataStart;//起始地址
第16行     unsigned long memSize;//成员数量
第17行 };
第18行 template<typename T>
第19行 class Array<T>::iterator{//定义一个iterator类
第20行     public:
第21行         //迭代器规范的定义
第22行         typedef T value_type;//值的类型
第23行         typedef T* pointer;//指针
第24行         typedef T& reference;//引用
第25行         typedef int difference_type;//两个迭代器之间的记录数
第26行         typedef std::random_access_iterator_tag iterator_category;//迭代器的类型
第27行
第28行         iterator(T *p){this->nowPosition=p;}
第29行         iterator operator++(){nowPosition++; return *this;}
第30行         iterator operator+(int n){return iterator(this->nowPosition+n);}
第31行         T& operator*(){return *nowPosition;}
第32行         bool operator!=(const iterator & rightItr){return this->nowPosition!=rightItr.nowPosition;}
第33行     private:
第34行         T* nowPosition;
第35行 };
第36行 template<typename T>
第37行 Array<T>::Array(unsigned long size):memSize(size){
第38行     this->addrDataStart=new T[this->memSize];
第39行 }
第40行 template<typename Iter>
```

```
第41行  void iterSwap(Iter &leftIter,Iter &rightIter){
第42行      iterator_traits<Iter>::value_type tmp=*leftIter;//*取得迭代器所指向的值
第43行      *leftIter=*rightIter;
第44行      *rightIter=tmp;
第45行  }
第46行
第47行  int main(){
第48行      int arrSize=20;
第49行      Array<int> myArr(arrSize);
第50行      for(int i=0;i<arrSize;++i)
第51行          myArr[i]=rand()%100;
第52行      for(Array<int>::iterator p=myArr.begin();p!=myArr.end();++p){
第53行          cout<<*p<<" ";
第54行      }
第55行
第56行      return 0;
第57行  }
```

2、迭代器分类

在C++的STL中，将迭代器共分为5中类型，如右图5-1所示。图中的箭头关系表明被指向的迭代器有箭头出发端迭代器的特征。如Forward Iterator具有Input Iterator和Output Iterator的特征。Random Access Iterator则具有其他迭代器所有特征。

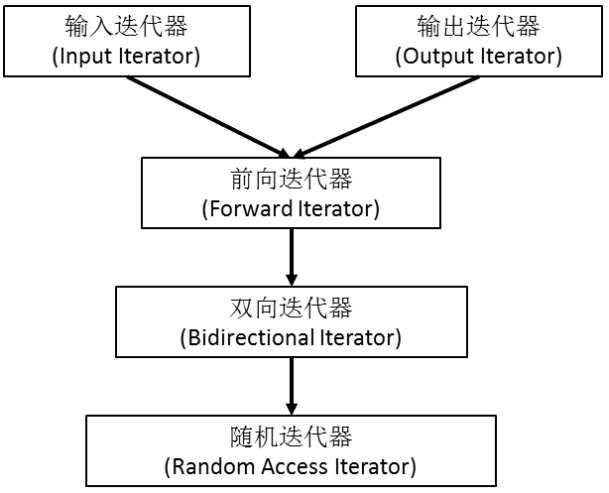

图5-1　5种类型的迭代器及其关系

迭代器的特征主要表现在迭代器的功能定义上，或者说迭代器必须且至少支持的运算符重载。迭代器能支持运算符重载，受制于迭代器所在的数据结构。如在标准单向链表中，迭代器就不能支持--运算符，但可以支持++运算符，而在双向链表中，则两者均可支持。

为描述方便，令itr代表某种迭代器类型，itrA和itrB表示itr类型的迭代器对象(实例)；T表示itr所指向元素的类型，t表示T类型的一个对象；mem表示T表示迭代器可以访问到的一个成员；i表示一个整数。

作为迭代器，都支持++运算符(包括前增和后增)，至少都支持前增运算符，即：
++pA：使迭代器指向下一个元素，并返回pA自身的引用；
pA++：使迭代器指向下一个元素，具体返回类型是迭代器具体情况而定。

⑴ 输入迭代器

输入迭代器用于从序列中读取数据，其iterator_category值为input_iterator_tag。输入迭代器支持对序列进行不可重复的单向遍历。在迭代器通过功能外，输入迭代器还应具备如下功能。

- itrA==itrB：两个输入迭代器可以用“==”判断是否相等；
- itrA!=itrB：连个输入迭代器可以用“!=”判断是否不等；
- *itrA：可以用“*”取得迭代器所指向的元素值，该值可以转换到T类型或T&、const T&等类型；
- itrA->mem：可以重载->运算符，以取得itrA所指向元素的mem成员值，相当于(*itrA).mem；
- *itrA++：等价于{T t=*itrA;++itrA;return t;}

输入迭代器的一个典型应用是输入流迭代器，请参考本章相关章节。

⑵ 输出迭代器

输出迭代器允许向序列中写入数据，其iterator_category值为input_iterator_tag。输出迭代器支持对序列进行单向遍历访问。在迭代器通用功能之外，输出迭代器还具备如下功能：

- *itrA=t：向迭代器所指向位置写入一个元素，要求*运算符返回必须是引用；

- *itrA++=t：等价于{*itrA=t;++itrA;}

⑶ 前向迭代器

前向迭代器包含输入迭代器和输出迭代器，既支持数据读取，也支持数据写入。前向迭代器支持对序列进行可重复的单向遍历，其iterator_category的取值为forward_iterator_tag。

前向迭代器是输入迭代器和输出迭代器的综合，要求支持输入迭代器和输入迭代器的全部功能。

⑷ 双向迭代器

双向迭代器在支持单向迭代器的基础上，支持迭代器反向移动，其iterator_category的取值为bidirectional_iterator_tag。在前向迭代器的功能外，双向迭代器还支持以下功能。

- --itrA：可以使用前置“--”使迭代器指向上一个元素，返回值为itrA自身引用；
- itrA--：可以使用后置“--”使迭代器指向上一个元素，相当于{itr itrB=itrA;--itrA;return itrB;}

STL容器中list的迭代器是典型的双向迭代器。

⑸ 随机迭代器

随机迭代器在双向迭代器的基础上，还支持直接将迭代器向前或向后移动i个元素，因此随机迭代器的功能几乎相当于指针，其iterator_category是random_access_iterator_tag。以下是随机访问迭代器新增的功能。

- itrA+=i：将迭代器itrA向前移动i个元素；
- itrA-=i：将迭代器itrA向后移动i个元素；
- itrA+i或i+itrA：获得指向itrA前i个元素的迭代器；
- itrA-i：获得指向itrA后i个元素的迭代器；
- itrA-itrB：获得一个整数i，表示两个迭代器之间相距元素个数；
- itrA Opertor itrB：Operator是<、<=、>或>=中的一个，用于比较itrA和itrB的前后关系；
- itrA[i]：相当于*(itrA+i)

STL容器中vector的迭代器是典型的随机迭代器，数组也可以视为随机迭代器。

3、输入输出流迭代器

输入输出流将在后续章节详细介绍。由于输入输出流与迭代器关系密切，因此在本章做初步介绍，初步认识到cin是输入流的一个实例，cout是输出流的实例。例程5-10是输入流cin的一个示例。注意：istream_iterator和ostream_iterator都在iterator头文件中定义。 输入流迭代器定义如下：

```
输入流迭代器模板定义：
template<typename T>istream_iterator<T>;

构造函数：
istream_iterator<dataType>(istream &in);
istream_iterator<dataType>();
```

dataType表示数据类型，如int、double等；参数in表示数据输入流，如cin，也可以是外部文件对象等；当调用没有参数的构造函数时，将定义为输入流结束。

例程5-10

```
第1行  #include<iostream>
第2行  #include<iterator>
第3行  #include<algorithm>
第4行  #include<vector>
第5行  using namespace std;
第6行  int main(){
第7行      vector<int> vecA,vecB(50);
第8行      for(int i=0;i<50;++i)vecA.push_back(rand()%100);
第9行      copy(vecA.begin(),vecA.end(),vecB.begin());//从头到尾将vecA逐一拷贝到从vecB开始
```

```
第10行      for(int i=0;i<50;++i)cout<<vecB[i]<<" ";//输出vecB中数据
第11行
第12行      istream_iterator<int> Beg(cin);//变量Beg以参数cin构造输入流迭代器，
第13行      istream_iterator<int> End;//当没有参数，则调用输入流迭代器默认构造函数，其值为结束标志
第14行
第15行      copy(Beg,End,vecB.begin());//与前面copy函数用法相似
第16行      for(int i=0;i<50;++i)cout<<vecB[i]<<" ";//输出vecB中数据
第17行
第18行      return 0;
第19行  }
```

cin和cout都是C++预定义的类，其中cin用于输入，cout用于输出。例程5-10第12行代码是以cin为作为参数调用istream_iterator构造函数。当以无参数方式申明istream_iterator时，则值为结束标志。在输入过程中，每个数据之间用空格分割，当输入结束时，产生结束符。在Windows系统中，按CTRL+Z和回车键，在Linux下则须CTRL+D，以表示输入结束。

用cin作为输入流时，可以认为在输入结束时，按下CTRL+Z或CTRL+D时，系统自动产生输入结束符。copy等函数执行时，逐步取出输入流中的数据，并与调用输入流迭代器无参构造函数产生的结束标志比较，如果不相等，则继续取数据，当取到相等结束标志时，相等并终止循环。

在C++中，可以像使用vector的迭代器一样使用istream_iterator，如例程5-10第15行所示。另外，在例程所示环境中，使用sort(Beg,End)是错误的，原因是输入迭代器只能被遍历一次。例程5-11是输入迭代器的一些习惯用法。注意第8行vecA后的代码((istream_iterator<int>(cin)),istream_iterator<int>())，其中第1个参数必须用括号包围，否则vecA将被视为函数原型申明；第2个参数最后的括号也不能省略，表明其为匿名对象。更明晰的申明办法如第13、20行所示。

例程5-11

```
第1行   #include<iostream>
第2行   #include<iterator>
第3行   #include<algorithm>
第4行   #include<vector>
第5行   using namespace std;
第6行   int main(){
第7行       cout<<"\n\n第一次输入:\n";
第8行       vector<int> vecA((istream_iterator<int>(cin)),istream_iterator<int>());
第9行       for(int i=0;i<vecA.size();++i)cout<<vecA[i]<<" ";
第10行
第11行      cin.clear();//清除原有输入流，可开始一个新的输入流；
第12行      cout<<"\n\n第二次输入:\n";
第13行      istream_iterator<int> inputBegin(cin),inputEnd;
第14行      vector<int> vecB(inputBegin,inputEnd);
第15行      sort(vecB.begin(),vecB.end());
第16行      for(vector<int>::iterator P=vecB.begin();P!=vecB.end();++P)cout<<*P<<" ";
第17行
第18行      cin.clear();//清除原有输入流，可开始一个新的输入流；
第19行      cout<<"\n\n第三次输入:\n";
第20行      istream_iterator<int> Beg(cin),End;
第21行      vector<int> vecC;
第22行      while(Beg!=End)
第23行          vecC.push_back(*Beg++);
```

```
第24行
第25行      sort(vecC.begin(),vecC.end());
第26行      for(vector<int>::iterator P=vecC.begin();P!=vecC.end();++P)cout<<*P<<" ";
第27行
第28行      return 0;
第29行  }
```

输出迭代器较输入迭代器简单，如例程5-12所示。输出流迭代器相关定义如下：

```
输出流迭代器模板定义：
template<typename T>ostream_iterator<T>;

构造函数：
ostream_iterator<dataType>(ostream &out);
ostream_iterator<dataType>(ostream &out,const char * delimiter);
```

dataType表示数据类型，如int、double等；参数out表示数据输出流，如cout等；delimiter是可选参数，表示两个输出数据之间的分隔符，支持C语言转移字符，如\t表示tab，也可以是汉字等。

例程5-12

```
第1行   #include<iostream>
第2行   #include<vector>
第3行   #include<iterator>
第4行   #include<algorithm>
第5行   using namespace std;
第6行
第7行   int main(){
第8行       vector<int> myVec;
第9行       for(int i=0;i<50;++i)
第10行          myVec.push_back(rand()%100);
第11行
第12行      copy(myVec.begin(),myVec.end(),ostream_iterator<int>(cout,"\t"));
第13行      ostream_iterator<int> outA(cout);
第14行      *outA=1111;//相当于执行cout<<1111;
第15行
第16行      ostream_iterator<int> outB(cout,"\n");
第17行      *outB=9999;//相当于执行cout<<1111<<"\n";
第18行
第19行      return 0;
第20行  }
```

第二节　数据结构示例

深入理解数据结构，不但能综合应用已学的C++知识，还能对运算符重载、迭代器、数据结构的行为及其效率有更多理解，对实际工作中如何正确选择合适的数据结构提供更多支持。

例程5-13是数组类数据结构。C++提供标准数组其长度固定，例程所示数组可以根据需要调整数组长度，同时也能通过下标运算符读写元素，与此同时，例程还提供了较为完整的迭代器支持，能应用于各种STL算法。

1、数组类

```
#ifndef ARRAY_H
#define ARRAY_H
#include<cassert>

template<typename T>
class Array{
public:

    typedef unsigned int size_type;
    class ArrIterator;
    class constArrIterator;
    typedef ArrIterator iterator;
    typedef constArrIterator const_iterator;
    typedef std::reverse_iterator<iterator> reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

    Array(size_type size);
    size_type size()const;
    T& operator[](size_type index);
    const T& operator[](size_type index)const;
    reverse_iterator rbegin(){return (reverse_iterator(end()));}
    const_reverse_iterator rbegin() const{return (const_reverse_iterator(end()));}
    reverse_iterator rend(){return (reverse_iterator(begin()));}
    const_reverse_iterator rend()const{return (const_reverse_iterator(begin()));}
    iterator begin(){return iterator(this->addrData);}
    iterator end(){return iterator(this->addrData +this->memberCount);}
    const_iterator begin()const{return const_iterator(this->addrData);}
    const_iterator end()const{return const_iterator(this->addrData +this->memberCount);}

    void resize(int newSize);//调整数组成员数量
private:
    T* addrData;//数据块指针
    size_type memberCount;//数据数量
};
template<typename T>
struct Array<T>::ArrIterator{

//Iterator Specification
        typedef T value_type;//必须：迭代器标准

        typedef T& reference;//引用

        typedef T* pointer;//指针
        typedef std::random_access_iterator_tag iterator_category;//必须：迭代器标准。定义迭代器类型
        typedef int difference_type;//必须：迭代器标准。表示两迭代器间距，常用于计数
```

```
第43行
第44行   //Function Member
第45行           ArrIterator(constArrIterator constArrIter){this->ptrNow=constArrIter.ptrNow;}
第46行           ArrIterator():ptrNow(NULL){}
第47行           ArrIterator(pointer ptr):ptrNow(ptr){}
第48行           ArrIterator operator++(){++this->ptrNow; return *this;}//前增
第49行           ArrIterator operator++(int Nothing){ArrIterator tmp = *this; ++*this;return tmp;}//后增
第50行           ArrIterator operator--(){--this->ptrNow; return *this;}//前减
第51行           ArrIterator operator--(int Nothing){ArrIterator tmp= *this; --*this;return tmp;}//后减
第52行
第53行           ArrIterator operator+(int n){ArrIterator tmp=*this;tmp.ptrNow+=n;return tmp;}
第54行           ArrIterator operator-(int n){ArrIterator tmp=*this;tmp.ptrNow-=n;return tmp;}
第55行           T& operator[](int n){return *(ptrNow+n);}
第56行           reference operator*(){return *this->ptrNow;}
第57行           pointer operator->(){return ptrNow;}
第58行           bool operator==(const ArrIterator& rightValue)const{return this->ptrNow==rightValue.ptrNow;}
第59行           bool operator!=(const ArrIterator& rightValue)const{return this->ptrNow!=rightValue.ptrNow;}
第60行           bool operator<(const ArrIterator & rightValue)const{return this->ptrNow<rightValue.ptrNow;}
第61行           bool operator<=(const ArrIterator & rightValue)const{return this->ptrNow<=rightValue.ptrNow;}
第62行           bool operator>(const ArrIterator & rightValue)const{return this->ptrNow>rightValue.ptrNow;}
第63行           bool operator>=(const ArrIterator & rightValue)const{return this->ptrNow>=rightValue.ptrNow;}
第64行           difference_type operator-(const ArrIterator & rightValue)const{return this->ptrNow-rightValue.ptrNow
;}
第65行
第66行   //Data Member
第67行           pointer ptrNow;
第68行   };
第69行   template<typename T>
第70行   struct Array<T>::constArrIterator{
第71行
第72行   //Iterator Specification
第73行           typedef T value_type;
第74行           typedef const T& reference;
第75行           typedef const T* pointer;
第76行           typedef int difference_type;
第77行           typedef std::random_access_iterator_tag iterator_category;
第78行
第79行   //Function Member
第80行           constArrIterator():ptrNow(NULL){}
第81行           constArrIterator(pointer ptr):ptrNow(ptr){}
第82行           constArrIterator(ArrIterator arrIterator){this->ptrNow=arrIterator.ptrNow;}
第83行           ArrIterator operator++()const{++this->ptrNow; return *this;}//前增
第84行           ArrIterator operator++(int Nothing)const{constArrIterator tmp = *this; ++*this;return tmp;}//后增
第85行           ArrIterator operator--()const{--this->ptrNow; return *this;}//前减
```

```
第86行          ArrIterator operator--(int Nothing)const{constArrIterator tmp= *this; --*this;return tmp;}//后减
第87行
第88行          ArrIterator operator+(int n)const{constArrIterator tmp=*this;tmp.ptrNow+=n;return tmp;}
第89行          ArrIterator operator-(int n)const{constArrIterator tmp=*this;tmp.ptrNow-=n;return tmp;}
第90行          T& operator[](int n)const{return *(ptrNow+n);}
第91行          reference operator*()const{return *ptrNow;}
第92行          pointer operator->()const{return ptrNow;}
第93行          bool operator==(const constArrIterator& rightValue)const{return ptrNow == rightValue.ptrNow;}
第94行          bool operator!=(const constArrIterator& rightValue)const{return ptrNow != rightValue.ptrNow;}
第95行          bool operator<(const constArrIterator & rightValue)const{return this->ptrNow<rightValue.ptrNow;}
第96行          bool operator<=(const ArrIterator & rightValue)const{return this->ptrNow<=rightValue.ptrNow;}
第97行          bool operator>(const ArrIterator & rightValue)const{return this->ptrNow>rightValue.ptrNow;}
第98行          bool operator>=(const ArrIterator & rightValue)const{return this->ptrNow>=rightValue.ptrNow;}
第99行          difference_type operator-(const ArrIterator & rightValue)const{return this->ptrNow-rightValue.ptrNow
;}
第100行
第101行 //Data Member
第102行         pointer ptrNow;
第103行 };
第104行
第105行 template<typename T>
第106行 Array<T>::Array(size_type size):memberCount(size){//Array构造函数
第107行     this->addrData = new T[memberCount];
第108行 }
第109行 template<typename T>
第110行 typename Array<T>::size_type Array<T>::size()const{
第111行     return memberCount;
第112行 }
第113行 template<typename T>
第114行 T& Array<T>::operator[](size_type index){
第115行     assert(index<memberCount);
第116行     return addrData[index];
第117行 }
第118行 template<typename T>
第119行 const T& Array<T>::operator[](size_type index)const{
第120行     assert(index<memberCount);
第121行     return addrData[index];
第122行 }
第123行 template<typename T>
第124行 void Array<T>::resize(int newSize){
第125行     assert(newSize>=0);
第126行     if(newSize==this->memberCount)return;//直接退出函数，什么也不做；
第127行
第128行     T *newAddrData=new T[newSize];
```

```
第129行        int N=(newSize<this->memberCount)?newSize:this->memberCount;
第130行        //将原有数组中前N个元素复制到新数组中
第131行        for(int i=0;i<N;++i)newAddrData[i]=addrData[i];
第132行        delete []addrData;
第133行        this->addrData=newAddrData;
第134行        this->memberCount=newSize;
第135行    }
第136行    #endif
```

例程5-14

```
第1行   #include<iostream>
第2行   #include<vector>
第3行   #include<algorithm>
第4行   #include<iterator>
第5行   #include"Array.h"
第6行
第7行   using namespace std;
第8行
第9行   int main(){
第10行      Array<int> arrData(50);
第11行      for(Array<int>::iterator p=arrData.begin();p!=arrData.end();++p)
第12行          *p=rand()%100;
第13行
第14行      cout<<"\n\n交换前原始数据:\n";
第15行      for(Array<int>::iterator p=arrData.begin();p!=arrData.end();++p)
第16行          cout<<*p<<" ";
第17行
第18行
第19行      Array<int>::iterator pA=arrData.begin();
第20行      Array<int>::iterator pB=arrData.end();
第21行      Array<int>::const_iterator pC=arrData.begin();
第22行      Array<int>::iterator pTmp=pA+2;
第23行      iter_swap(pA,pTmp);
第24行      cout<<"\n\n交换后原始数据:\n";
第25行      for(Array<int>::iterator p=arrData.begin();p!=arrData.end();++p)
第26行          cout<<*p<<" ";
第27行
第28行      cout<<"\n\n排序后数据:\n";
第29行      sort(arrData.begin(),arrData.end());
第30行      for(Array<int>::iterator p=arrData.begin();p!=arrData.end();++p)
第31行          cout<<*p<<" ";
第32行      cout<<"\n\n排序后反序数据:\n";
第33行      copy(arrData.rbegin(),arrData.rend(),ostream_iterator<int>(cout," "));
第34行
```

```
第35行      cout<<"\n\n";
第36行      return 0;
第37行  }
```

数组类数据结构较为简单，根据需要通过new运算符开辟连续内存空间(第105-108行)。数组成员可以通过下标运算符进行读写(第113-122行)，也可以通过迭代器采用类似指针方式进行读写(第56行、第91行分别分布在ArrIterator和constArrIterator)。

本例所示代码嵌套有两种迭代器分别是ArrIterator和constArrIterator，经过typedef别名定义后分别代表iterator和const_iterator。const_iterator有两种实现方式，一是采用目前方式独立定义，也可以通过const iterator方式，即const arrIterator方式。采用独立方式的好处显而易见，即可以对不同行为单独定义，互不影响，关系简单；而采用const ArrIterator需要考虑两者之间的协调。注意constArrIterator的成员函数大都有const关键字，表明其值不能被修改。

例程存在诸多不足。

首先，Array没有析构函数。类可以没有析构函数，编译器将根据需要决定是否自动产生析构函数，但一般说来，一旦有在堆内的动态内存分配，尤其是在构造函数中有动态内存分配，就要在析构函数中回收已分配内存，不然，在以下情况，将导致内存泄露，如例程5-15所示。

例程5-15

```
第1行  template<typename T>
第2行  void Print(){
第3行      Array<T> tmpArr(100);
第4行      //赋值：给每个数组成员赋值随机数
第5行      for(int i=0;i<100;++i)tmpArr[i]=rand()%100;
第6行      //输出数组
第7行      for(int i=0;i<100;++i)cout<tmpArr[i]<<" ";
第8行  }
```

在例程5-15所示的程序中，当执行函数时，将通过Array的构造函数动态分配100单元的内存，当结束程序执行后，内存一直被占据并没有释放内存，由此可以造成严重的内存泄露。析构函数的代码如例程5-16所示。

例程5-16

```
第1行  template<typename T>
第2行  Array<T>::~Array(){
第3行      delete []this->addrData;
第4行  }
```

其次，Array类没有拷贝构造函数。类可以没有拷贝构造函数，编译器将根据需要决定是否自动产生拷贝构造函数，当执行例程5-17时，将会发生问题。即输出arrA与arrB相同，都是arrB的序列，其原因是arrB与arrA都指向相同的内容。

例程5-17

```
第1行  void showCopyConstruct(){
第2行      Array<int> arrA(5);
第3行      arrA[0]=0;arrA[1]=1;arrA[2]=2;arrA[3]=3;arrA[4]=4;
第4行      Array<int> arrB(arrA);
第5行      arrB[0]=10;arrB[1]=9;arrB[2]=8;arrB[3]=7;arrB[4]=6;
第6行      copy(arrA.begin(),arrA.end(),ostream_iterator<int>(cout," "));
第7行      copy(arrB.begin(),arrB.end(),ostream_iterator<int>(cout," "));
第8行  }
```

当没有拷贝构造函数时，编译器会自动根据需要产生拷贝构造函数。当类中没有指针时，一般没有问题，编译器产生的拷贝构造函数会简单地将数据成员复制到目标类，是为浅拷贝(Shallow Copy)；当有指针时，指针成员同样复制，如在本例中arrA的addrData值将复制给arrB的addrData，由于addrData值相同，因此指向相同内存区块，很明显，在本例这将导致错误。因此需要显式申明拷贝构造函

数，实现深拷贝(Deep Copy)，其代码如例程5-18所示。

例程5-18

```
template<typename T>
Array<T>::Array(Array<T> ©Array){
    this->memberCount=copyArray.memberCount;
    this->addrData=new T[this->memberCount];

    for(int i=0;i<this->memberCount;++i)
        *(this->addrData+i)=copyArray.addrData[i];
}
```

Array还不支持形如arrB=arrA的赋值形式，这种形式的赋值，需要重载等号运算符，如例程5-19所示。赋值运算符重载，其本质与拷贝构造函数相似。另外，执行代码Array<int> arrB=arrA时，即申明变量同时赋初值时，执行拷贝构造函数，而不是执行赋值运算符重载。

例程5-19

```
template<typename T>
Array<T> & Array<T>::operator=(const Array &assignArray){
    this->memberCount=assignArray.memberCount;
    this->addrData=new T[this->memberCount];
    for(int i=0;i<this->memberCount;++i)
        *(this->addrData+i)=assignArray.addrData[i];
    return *this;
}
```

例程5-14第21行代码Array<int>::const_iterator pC=arrData.begin()似乎存在一定矛盾。虽然arrData.begin()的返回值为iterator或const_iterator，但在此处arrData.begin()的返回值是iterator，其原因是arrData的数据类型并没有const修饰符，而返回const_iterator需要有const修饰符，即便使用const_iterator类型，也不会调用const_iterator begin()const函数。因此，需要将iterator类型转换为const_iterator。例程5-13第82行constArrIterator(ArrIterator arrIterator){this->ptrNow=arrIterator.ptrNow;}完成该功能，其本质是构造函数，输入参数为ArrIterator即interator，其功能是将输入ArrIterator转换为constArrIterator。

如同字符串能拼接，如果两个Array能通过加法运算符拼接，将更加便于使用，形如：arrC=arrA+arrB。在拼接时要注意arrA的原值不发生变化，如果发生变化，则变成arrB追加在arrA之后。另外，能实现该操作的前提除了重载加法运算符外，还要支持赋值运算符，即将arrA+arrB的结果赋值给arrC。加号运算符重载代码如例程5-20所示。

例程5-20

```
template<typename T>
Array<T> Array<T>::operator+(const Array<T> &rightArray){
    int totalCount=this->memberCount+rightArray.memberCount;
    Array<T> leftArray(totalCount);
    for(int i=0;i<this->memberCount;++i)
        *(leftArray.addrData+i)=*(this->addrData+i);

    for(int i=0;i<rightArray.memberCount;++i)
        *(leftArray.addrData+i+this->memberCount)=*(rightArray.addrData+i);
    return leftArray;
}
```

经过上述改造，虽然程序更加完备，但还不能增加在尾部追加数据、中间插入数据等，尤其是外部追加数据(push_back)要考虑周全，如每当追加时，即调用resize()一个成员数据，则每次增加都将调用resize()，降低效率。例程5-21是为实现上述功能而进行的较大修改，是其功能类似STL的vector，例程5-22是其应用示例。

例程5-21

```
第1行  #ifndef USERVECTOR_H
第2行  #define USERVECTOR_H
第3行  #include<cassert>
第4行  template<typename T>
第5行  class userVector{
第6行  public:
第7行      //迭代器
第8行      struct UVIterator;//非常量迭代器
第9行      struct constUVIterator;//常量迭代器
第10行
第11行     //类型别名
第12行     typedef unsigned long int uLongInt;//无符号长整数
第13行     typedef UVIterator iterator;
第14行     typedef constUVIterator const_iterator;
第15行     typedef std::reverse_iterator<iterator> reverse_iterator;//定义反向迭代器
第16行     typedef std::reverse_iterator<const_iterator> const_reverse_iterator;//定义常量反向迭代器
第17行     typedef T& reference;//引用的别名
第18行     typedef const T& const_reference;//常量引用的别名
第19行
第20行     userVector():addrData(NULL),intoMember(0),ontoMember(0){;}//无参构造函数
第21行     userVector(int size);//构造函数
第22行     userVector(const userVector<T>& UV);//拷贝构造函数
第23行     ~userVector();//析构函数
第24行     userVector<T>& operator=(const userVector<T> &rightUV);//重载赋值运算符(=)
第25行     T&operator[](int Index);//重载下标运算符
第26行     const T&operator[](int Index)const;//重载下标运算符
第27行     operator T*();//重载到T*类型的转变
第28行     operator const T*()const;//重载到const T*的转换
第29行
第30行     uLongInt size()const;//数据数量
第31行     uLongInt capacity()const;//当前容量
第32行     void push_back(const T&val);//追加数据
第33行     void pop_back();//删除尾部数据
第34行     bool isEmpty();//是否为空
第35行     reference at(int index);
第36行     const_reference at(int index)const;
第37行     void clear();//清空数据
第38行     iterator erase(iterator seat);//删除iterator seat位置处element
第39行     iterator erase(iterator begin,iterator end);//删除指定范围内的元素;
第40行
```

```
第41行      iterator begin();
第42行      const_iterator begin()const;
第43行      iterator end();
第44行      const_iterator end()const;
第45行      reverse_iterator rbegin(){return (reverse_iterator(end()));}
第46行      const_reverse_iterator rbegin() const{return (const_reverse_iterator(end()));}
第47行      reverse_iterator rend(){return (reverse_iterator(begin()));}
第48行      const_reverse_iterator rend()const{return (const_reverse_iterator(begin()));}
第49行
第50行  private:
第51行      T *addrData;//数据起点地址
第52行      uLongInt intoMember;//使用空间，已装入记录数
第53行      uLongInt ontoMember;//可用空间，可装入记录数
第54行  };
第55行
第56行  //=======================UVIterator定义，非常量迭代器定义
第57行  template<typename T>
第58行  struct userVector<T>::UVIterator{
第59行
第60行  //Iterator Specification
第61行          typedef T value_type;//必须：迭代器标准
第62行          typedef T& reference;//引用
第63行          typedef T* pointer;//指针
第64行          typedef std::random_access_iterator_tag iterator_category;//必须：迭代器标准。定义迭代器类型
第65行          typedef int difference_type;//必须：迭代器标准。表示两迭代器间距，常用于计数
第66行
第67行  //Function Member
第68行          UVIterator(constUVIterator &constUVIter){this->pSeat=constArrIter.pSeat;}
第69行          UVIterator():pSeat(NULL){}
第70行          UVIterator(pointer ptr):pSeat(ptr){}
第71行          UVIterator operator++(){++this->pSeat; return *this;}//前增
第72行          UVIterator operator++(int Nothing){UVIterator tmp = *this; ++*this;return tmp;}//后增
第73行          UVIterator operator--(){--this->pSeat; return *this;}//前减
第74行          UVIterator operator--(int Nothing){UVIterator tmp= *this; --*this;return tmp;}//后减
第75行
第76行          UVIterator operator+(int n){UVIterator tmp=*this;tmp.pSeat+=n;return tmp;}
第77行          UVIterator operator-(int n){UVIterator tmp=*this;tmp.pSeat-=n;return tmp;}
第78行          T& operator[](int n){return *(pSeat+n);}
第79行          reference operator*(){return *this->pSeat;}
第80行          pointer operator->(){return pSeat;}
第81行          bool operator==(const UVIterator& rightValue)const{return this->pSeat==rightValue.pSeat;}
第82行          bool operator!=(const UVIterator& rightValue)const{return this->pSeat!=rightValue.pSeat;}
第83行          bool operator<(const UVIterator & rightValue)const{return this->pSeat<rightValue.pSeat;}
第84行          bool operator<=(const UVIterator & rightValue)const{return this->pSeat<=rightValue.pSeat;}
```

```
第85行          bool operator>(const UVIterator & rightValue)const{return this->pSeat>rightValue.pSeat;}
第86行          bool operator>=(const UVIterator & rightValue)const{return this->pSeat>=rightValue.pSeat;}
第87行          difference_type operator-(const UVIterator & rightValue)const{return this->pSeat-rightValue.pSeat;}
第88行
第89行  //Data Member
第90行          pointer pSeat;//迭代器所指向的位置
第91行  };
第92行
第93行  //====================constUVIterator定义，常量迭代器
第94行  template<typename T>
第95行  struct userVector<T>::constUVIterator{
第96行
第97行  //Iterator Specification
第98行          typedef T value_type;
第99行          typedef const T& reference;
第100行         typedef const T* pointer;
第101行         typedef int difference_type;
第102行         typedef std::random_access_iterator_tag iterator_category;
第103行
第104行 //Function Member
第105行         constUVIterator():pSeat(NULL){}
第106行         constUVIterator(pointer ptr):pSeat(ptr){}
第107行         constUVIterator(UVIterator UVIter){this->pSeat=arrIterator.pSeat;}
第108行         constUVIterator operator++()const{++this->pSeat; return *this;}//前增
第109行         constUVIterator operator++(int Nothing)const{constUVIterator tmp = *this; ++*this;return tmp;}//后增
第110行         constUVIterator operator--()const{--this->pSeat; return *this;}//前减
第111行         constUVIterator operator--(int Nothing)const{constUVIterator tmp= *this; --*this;return tmp;}//后减
第112行
第113行         constUVIterator operator+(int n)const{constUVIterator tmp=*this;tmp.pSeat+=n;return tmp;}
第114行         constUVIterator operator-(int n)const{constUVIterator tmp=*this;tmp.pSeat-=n;return tmp;}
第115行         T& operator[](int n)const{return *(pSeat+n);}
第116行         reference operator*()const{return *pSeat;}
第117行         pointer operator->()const{return pSeat;}
第118行         bool operator==(const constUVIterator& rightValue)const{return pSeat == rightValue.pSeat;}
第119行         bool operator!=(const constUVIterator& rightValue)const{return pSeat != rightValue.pSeat;}
第120行         bool operator<(const constUVIterator & rightValue)const{return this->pSeat<rightValue.pSeat;}
第121行         bool operator<=(const constUVIterator & rightValue)const{return this->pSeat<=rightValue.pSeat;}
第122行         bool operator>(const constUVIterator & rightValue)const{return this->pSeat>rightValue.pSeat;}
第123行         bool operator>=(const constUVIterator & rightValue)const{return this->pSeat>=rightValue.pSeat;}
第124行         difference_type operator-(const constUVIterator & rightValue)const{return this->pSeat-rightValue.pSeat;}
第125行
第126行 //Data Member
第127行         pointer pSeat;//迭代器所指向的位置
```

```
第128行 };
第129行
第130行
第131行 //析构函数
第132行 template<typename T>
第133行 userVector<T>::~userVector(){
第134行     if(this->ontoMember!=0){
第135行         delete []this->addrData;//释放内存
第136行         this->intoMember=0;
第137行         this->ontoMember=0;
第138行     }
第139行 }
第140行 //带参构造函数
第141行 template<typename T>
第142行 userVector<T>::userVector(int size){
第143行     this->intoMember=size;
第144行     this->ontoMember=size*2;
第145行     this->addrData=new T[this->ontoMember];
第146行 }
第147行 template<typename T>
第148行 userVector<T>::userVector(const userVector<T>& uvCopy){
第149行     this->intoMember=uvCopy.intoMember;
第150行     this->ontoMember=uvCopy.ontoMember;
第151行     this->addrData=new T[this->ontoMember];
第152行     for(uLongInt i=0;i<this->intoMember;++i){
第153行         *(this->addrData+i)=*(uvCopy.addrData+i);
第154行     }
第155行 }
第156行 template<typename T>
第157行 T&userVector<T>::operator[](int Index){//没有做边界检查
第158行     return *(this->addrData+Index);
第159行 }
第160行 template<typename T>
第161行 const T&userVector<T>::operator[](int Index)const{//没有做边界检查
第162行     return (this->addrData+index);
第163行 }
第164行 template<typename T>
第165行 userVector<T>::operator T*(){
第166行     return this->addrData;
第167行 }
第168行 template<typename T>
第169行 userVector<T>::operator const T*()const{
第170行     return this->addrData;
```

```
第171行  }
第172行  //重载赋值运算符
第173行  template<typename T>

第174行  userVector<T>& userVector<T>::operator=(const userVector<T> &rightUV){
第175行      if(this->ontoMember==rightUV.ontoMember){
第176行          if(this->ontoMember!=NULL){
第177行              for(uLongInt i=0;i<rightUV.intoMember;++i)*(this->addrData+i)=*(rightUV.addrData+i);
第178行          }else{

第179行              ;//此处不处理，表明都还没有分配内存
第180行          }
第181行      }else{
第182行          if(this->ontoMember==0||rightUV.ontoMember==0){
第183行              if(this->ontoMember==0){
第184行                  this->intoMember=rightUV.intoMember;
第185行                  this->ontoMember=rightUV.ontoMember;
第186行                  this->addrData=new T[this->ontoMember];
第187行                  for(uLongInt i=0;i<this->intoMember;++i)*(this->addrData+i)=*(rightUV.addrData+i);
第188行              }
第189行              if(rightUV.ontoMember==0){
第190行                  this->intoMember=0;
第191行                  this->ontoMember=0;
第192行                  delete []this->addrData;
第193行              }
第194行          }else{
第195行              if(this->ontoMember!=0 && rightUV.ontoMember!=0){
第196行                  delete []this->addrData;
第197行                  this->intoMember=rightUV.intoMember;
第198行                  this->ontoMember=rightUV.ontoMember;
第199行                  this->addrData=new T[this->ontoMember];
第200行                  for(uLongInt i=0;i<this->intoMember;++i)*(this->addrData+i)=*(rightUV.addrData+i);
第201行              }
第202行          }
第203行      }
第204行      return *this;
第205行  }
第206行  template<typename T>
第207行  void userVector<T>::push_back(const T&val){
第208行      if(this->ontoMember==0){//尚未分配空间
第209行          this->ontoMember=100;//分配100个数据空间
第210行          this->addrData=new T[100];
第211行          this->intoMember=1;
第212行          *(this->addrData)=val;
第213行      }else{
```

```
第214行            if(this->intoMember==this->ontoMember){//空间已满
第215行                uLongInt newSize=this->ontoMember*2;
第216行                T *newAddrData=new T[newSize];//分配新的数据空间
第217行                for(uLongInt i=0;i<this->ontoMember;++i)*(newAddrData+i)=*(this->addrData+i);
第218行                *(newAddrData+this->ontoMember)=val;
第219行                delete []this->addrData;
第220行                this->addrData=newAddrData;
第221行                this->intoMember++;
第222行                this->ontoMember=newSize;
第223行            }else{//空间未满
第224行                *(this->addrData+this->intoMember)=val;
第225行                this->intoMember++;
第226行            }
第227行        }
第228行    }
第229行    template<typename T>
第230行    void userVector<T>::pop_back(){
第231行        assert(this->intoMember>0);
第232行        this->intoMember--;//无需重构内存空间，让其不能使用即可。
第233行    }
第234行    //是否为空
第235行    template<typename T>
第236行    bool userVector<T>::isEmpty(){
第237行        return (this->intoMember==0);
第238行    }
第239行    template<typename T>
第240行    void userVector<T>::clear(){
第241行        if(this->ontoMember!=0){
第242行            delete []this->addrData;
第243行            this->intoMember=0;
第244行            this->ontoMember=0;
第245行        }
第246行    }
第247行    template<typename T>
第248行    typename userVector<T>::iterator userVector<T>::erase(typename userVector<T>::iterator seat){
第249行        int eraseSeat=seat-(*this).begin();
第250行        assert(eraseSeat>=0);
第251行        T * newAddrData=new T[this->ontoMember];//注意：删除元素的空间没有回收
第252行        for(int i=0;i<eraseSeat-1;++i)*(newAddrData+i)=*(this->addrData+i);
第253行        for(int i=eraseSeat;i<this->intoMember;++i)*(newAddrData+i-1)=*(this->addrData+i);
第254行        this->intoMember--;
第255行        delete []this->addrData;
第256行        this->addrData=newAddrData;
```

```
第257行         seat.pSeat=NULL;
第258行         return this->begin()+eraseSeat-1;
第259行     }
第260行     template<typename T>
第261行     typename userVector<T>::iterator userVector<T>::erase(typename userVector<T>::iterator begin,typename userVector<T>::iterator end){
第262行         int eraseStartNo=begin-(*this).begin();
第263行         int eraseStopNo=end-(*this).begin();
第264行         assert(eraseStartNo>=0 && eraseStopNo>=0 &&(eraseStopNo-eraseStartNo)>=0);

第265行         T * newAddrData=new T[this->ontoMember];//注意：删除元素的空间没有回收
第266行         for(int i=0;i<eraseStartNo-1;++i)*(newAddrData+i)=*(this->addrData+i);
第267行         for(int i=eraseStopNo;i<this->intoMember;++i)*(newAddrData+i-eraseStartNo-1)=*(this->addrData+i);
第268行         this->intoMember-=eraseStopNo-eraseStartNo+1;
第269行         delete []this->addrData;
第270行         this->addrData=newAddrData;
第271行         return this->begin()+eraseStopNo-1;
第272行     }

第273行     template<typename T>
第274行     typename userVector<T>::reference userVector<T>::at(int index){
第275行         assert(index<this->intoMember && index>=0);
第276行         return *(this->addrData+index);
第277行     };
第278行     template<typename T>
第279行     typename userVector<T>::const_reference userVector<T>::at(int index)const{
第280行         assert(index<this->intoMember);
第281行         return *(this->addrData+index);
第282行     };
第283行     template<typename T>
第284行     typename userVector<T>::uLongInt userVector<T>::capacity()const{
第285行         return (this->ontoMember);
第286行     }
第287行     template<typename T>
第288行     typename userVector<T>::uLongInt userVector<T>::size()const{
第289行         return (this->intoMember);
第290行     }
第291行     template<typename T>
第292行     typename userVector<T>::iterator userVector<T>::begin(){
第293行         return iterator(this->addrData);
第294行     }
第295行     template<typename T>
第296行     typename userVector<T>::const_iterator userVector<T>::begin()const{
第297行         return const_iterator(this->addrData);
第298行     }
```

```
template<typename T>
typename userVector<T>::iterator userVector<T>::end(){
    return iterator(this->addrData+this->intoMember);
}
template<typename T>
typename userVector<T>::const_iterator userVector<T>::end()const{
    return const_iterator(this->addrData+this->intoMember);
}
#endif
```

例程5-22

```
#include<iostream>
#include<vector>
#include<iterator>
#include<algorithm>
#include"userVector.h"
using namespace std;
void printUV(int *a,int size){
    a[0]=100;
    for(int i=0;i<size;++i)
        cout<<a[i]<<" ";
}
void showCopyConstruct(){
    userVector<int> arrA(5);
    arrA[0]=0;arrA[1]=1;arrA[2]=2;arrA[3]=3;arrA[4]=4;
    userVector<int> arrB(5);//=arrA;
    arrB=arrA;
    copy(arrB.begin(),arrB.end(),ostream_iterator<int>(cout," "));
    arrB[0]=10;arrB[1]=9;arrB[2]=8;arrB[3]=7;arrB[4]=6;
    copy(arrA.begin(),arrA.end(),ostream_iterator<int>(cout," "));
    copy(arrB.begin(),arrB.end(),ostream_iterator<int>(cout," "));
}
int main(){
    int uvSize=50;
    userVector<int> myUserVec(uvSize);
    for(int i=0;i<uvSize;++i)
        myUserVec[i]=rand()%100;

    for(int i=0;i<uvSize;++i)
        cout<<myUserVec[i]<<" ";

    cout<<endl<<endl;;

    printUV(myUserVec,50);
    cout<<endl<<endl;
```

```
第35行        userVector<int> myUV(10);
第36行        vector<int> myVec(50);
第37行
第38行        for(int i=0;i<100;++i)
第39行            myUV.push_back(rand()%100);
第40行
第41行        for(unsigned long int i=0;i<myUV.size();++i)
第42行            cout<<myUV[i]<<" ";
第43行        myVec.push_back(10);
第44行        for(unsigned long int i=0;i<myVec.size();++i)
第45行            cout<<myVec[i]<<" ";
第46行        cout<<endl<<endl;
第47行        showCopyConstruct();cout<<"\n\n";
第48行        userVector<int>::iterator pStart=myUV.begin();
第49行        userVector<int>::iterator pEnd=myUV.end();
第50行        generate(pStart,pStart+10,rand);
第51行        copy(pStart,pEnd,ostream_iterator<int>(cout,"\t"));
第52行        for(int i=0;i<500;++i)
第53行            myUV.push_back(rand()%100);
第54行
第55行        cout<<"\n\n";
第56行        copy(myUV.begin(),myUV.end(),ostream_iterator<int>(cout,"\t"));cout<<"\n\n";
第57行        //myVec.pop_back();
第58行        myUV.pop_back();myUV.pop_back();myUV.pop_back();//执行三次pop_back()
第59行        copy(myUV.begin(),myUV.end(),ostream_iterator<int>(cout,"\t"));cout<<"\n\n";
第60行        copy(myUV.rbegin(),myUV.rend(),ostream_iterator<int>(cout,"\t"));cout<<"\n\n";
第61行        myUV.at(100)=555;
第62行        copy(myUV.rbegin(),myUV.rend(),ostream_iterator<int>(cout,"\t"));cout<<"\n\n";
第63行        cout<<"\n\n";
第64行        myUV.clear();
第65行        copy(myUV.begin(),myUV.end(),ostream_iterator<int>(cout,"\t"));cout<<"\n\n";
第66行        for(int i=0;i<100;++i)
第67行            myUV.push_back(rand()%100);
第68行        copy(myUV.begin(),myUV.end(),ostream_iterator<int>(cout,"\t"));cout<<"\n\n";
第69行        userVector<int>::iterator pA=myUV.begin()+5;
第70行        userVector<int>::iterator pB=myUV.begin()+10;
第71行        cout<<"*pA="<<*pA<<"*pB="<<*pB<<endl;
第72行        myUV.erase(pA,pB);
第73行        copy(myUV.begin(),myUV.end(),ostream_iterator<int>(cout,"\t"));cout<<"\n\n";
第74行
第75行
第76行        cout<<"\n\n";
第77行        return 0;
第78行    }
```

2、链表类