第 8 章　定制 STL 容器

本书的一个中心主题是 C++提供给程序员的原始功能。或许标准模板库(STL)最能说明这个问题，这个库改变了程序员编写程序的方式。STL 包含了一组成熟的模板类和函数，这些类和函数实现了常用的数据结构和算法。对涉及到数据存储及获取的各种编程问题，STL 提供了可以直接使用的解决方案，因此基于 STL 的代码变得越来越普遍。例如，第 2 章的垃圾回收子系统和第 3 章的线程控制面板都使用了 STL，从而极大地简化了代码。在过去，如果需要某个数据结构，如链表、堆栈或者队列，必须手工编码。现在，程序员可以简单地使用 STL 提供的某个解决方案。

STL 的核心是容器。容器是一个保存其他对象的对象。STL 提供了几种内建的容器以支持如堆栈、队列、链表以及向量之类的数据结构。由于容器是模板化的，因此能够存储任何类型的对象，包括您创建的类对象。

尽管内建的容器非常有用，但是您并不会受到它们的限制。STL 最引人注目的特征是允许您创建自己的容器。因此，STL 是可扩展的。一旦您创建了一个容器，它就自动与 STL 的其余部分完全兼容。

在这一章，您将会看到如何创建定制的 STL 容器。尽管定制容器并不困难，但是许多程序员开始时对这种想法还是有点恐惧。原因之一是 STL 基于模板的语法乍看上去是无法征服的。事实上，STL 是概念上非常整洁的、易于理解的子系统。如果您遵循几个规则，将毫无问题地创建您自己的容器。

本章开发的定制容器实现了一个名为 RangeArray 的范围可以选择的动态数组。当使用 RangeArray 时，您可以指定开始和结束的索引。例如，RangeArray 允许创建一个从 -10 到 10 的数组。

8.1　STL 的简要回顾

STL 是一个很大的话题，对其详细分析需要上百页的篇幅，而且对 STL 的完整分析远远超出了本书的范围。因此，本章假定您具有 STL 的基本知识。 在此给出了 STL 的主要术语及其组成部分，STL 基于 3 个主要的特性：容器、迭代器以及算法。如前所述，容器是包含其他对象的对象；迭代器是类似于指针的对象，通过迭代器，可以使用类似于用指针遍历数组的方法来遍历容器；算法是对容器中元素的操作，如修改、复制或者处理其他元素。

除了容器、迭代器以及算法，STL 还需要其他一些标准组件的支持，如分配器、适配器、谓词以及函数对象。下面部分进一步介绍主要的 STL 要素。

提示：

对于 STL 的完整讨论，作者推荐参考他的另一本书 *STL Programming From the Ground Up*，McGraw - Hill/Osborne。

8.1.1 容器

STL 定义了两种类型的容器：序列式容器和关联式容器。序列式容器拥有一个线性的对象链表。STL 提供了几种内建的序列式容器，包括 vector(定义了动态数组)、deque(创建了双端队列)以及 list(实现了一个双向链表)。关联式容器存储关键字/值对。因此，关联式容器允许基于关键字来高效地检索值。例如，map 可通过惟一的关键字访问值。因此，map 存储的关键字/值对，并允许检索值和对应的关键字。

每个容器类定义一组可能应用于容器的函数。例如，list 容器包含插入、删除和合并元素的函数。stack 包含压入和弹出值的函数。

8.1.2 算法

算法作用于容器。算法定义了对容器的内容进行操作的方法。其功能包括对容器内容的初始化、存储、搜索以及转换。许多算法只是针对容器中某个范围的元素而不是整个容器操作。算法的示例包括 copy()、remove()、replace()以及 find()。

8.1.3 迭代器

迭代器是类似于指针的对象，可以将其增加或者减小。您还能够对其应用*运算符。迭代器由不同容器定义的 iterator 类型声明。

STL 还支持反向迭代器。反向迭代器以相反的方向遍历一个序列。因此，如果反向迭代器指向某个序列的结尾，则增加这个迭代器会使得它指向结尾的前一个元素。

8.2 其他的 STL 实体

除了容器、迭代器以及算法之外，其他一些 STL 的元素也扮演着重要的角色：分配器、函数对象、谓词、适配器、绑定器以及否定器(negator)。

每一个容器都定义了一个分配器。分配器为容器管理内存的分配。默认的分配器是 allocator 类的一个对象，但是，如果特定的应用程序需要，就可以定义自己的分配器。在多数情况下，默认的分配器就足够了。

函数对象是定义 operator()的类。在此有几个预定义的函数对象，如 less()、greater()、plus()、minus()、multiplies()以及 divides()。或许使用最广泛的函数对象是 less()，这个函数用来判断一个对象什么时候小于另一个对象。当使用 STL 算法时，函数对象可以用来代替函数指针。

就最普遍的意义来讲，适配器将一个对象转换为另一个对象。适配器分为容器适配器、迭代器适配器以及函数适配器。queue 是容器适配器的一个示例，它使得 deque 容器可以像标准队列那样使用。适配器简化了许多复杂的情况。

有些算法和容器使用了一种特殊类型的称为谓词的函数。在此有两种版本的谓词：一元谓词和二元谓词。一元谓词具有一个参数，二元谓词具有两个参数。这些函数返回 true/false，但是它们返回 true 或 false 的准确条件是由您定义的。某些算法使用了特定类型的二元谓词来比较两个元素。如果第一个参数小于第二个参数，这些比较函数则返回 true。

在 STL 中常用的其他两个实体是绑定器和函数对象。绑定器将参数与函数对象绑定。STL 定义了两种绑定器：bind2nd()和 bind1st()。否定器返回谓词的对立值。在此有两个否定器：not1()

和 not2()。绑定器和否定器增加了 STL 功能的多样性。

8.3 定制容器的要求

在设计一个定制的容器之前，有必要准确知道它必须提供的功能。根据标准 C++，所有的容器都必须提供指定的一组类型、运算符以及成员函数。除了这些常用的元素之外，序列式容器和关联式容器都加入了一些特定的要求。如果您遵循这些规则，您定制的容器就可以被 STL 定义的所有其他实体(包括分配器、函数对象、谓词以及绑定器)使用。设计良好的定制容器可以完全整合到 STL 中。下面部分描述了对于容器的要求。

8.3.1 一般要求

每一种容器(序列式容器或者关联式容器)都必须通过分配器而不是 new 和 delete 来管理内存。因此，容器必须使用分配器的成员函数来分配并释放内存。当容器创建时，分配器作为一个参数被指定。然而，C++提供了一个默认的分配器类，名为 allocator，对于 allocator 类型的对象，allocator 参数可以采用默认值，所有的标准容器都是如此。所有的分配器都必须提供与 allocator 相同的成员函数(当然，可以有不同的实现)。通过这种方法，容器可以使用程序员提供的任何分配器。本章容器使用的分配器函数如表 8-1 所示。

表 8-1 本章使用的分配器函数

函 数	描 述
pointer allocate(size_type num,typename allocator<void>::const_pointer h=0);	返回一个指向已分配内存的指针，这块内存足够大，能够保存 T 类型的 num 个对象。h 的值是一个对函数的提示，该函数可以用来满足或者忽略请求
void construct(pointer ptr,const_reference val);	在 ptr 处构建一个 T 类型的对象
void deallocate(pointer ptr,size_type num) ;	释放在 ptr 处开始的 T 类型的 num 个对象。ptr 的值必须已经从 allocate()获取
void destroy(pointer ptr);	销毁 ptr 处的对象。会自动调用对象的析构函数
size_type max_size()const throw();	返回可分配的 T 类型对象的最大数量

每个容器都必须提供以下这些类型：

```
iterator        const_iterator        reference              const_reference
value_type      size_type             difference_type
```

可反转的容器(支持双向迭代器)必须提供以下这些类型：

```
reverse_iterator      const_reverse_iterator
```

所有的容器都必须提供默认的构造函数和复制构造函数，默认的构造函数创建了一个长度为 0 的容器。此外还需要不同参数的构造函数，序列式容器和关联式容器的具体形式有所不同。

另外还需要析构函数。

必须支持下面的成员函数：

```
begin( )        clear( )        empty( )        end( )
erase( )        insert( )       max_size( )     rbegin( )
rend( )         size( )         swap( )
```

当然，只有可反转容器需要 rbegin()和 rend()函数。一些函数具有重载的形式。

提示：

所有 STL 内建的容器都提供函数 get_allocator()，但是这对于定制的容器并不需要。

容器必须定义迭代器函数(如 begin())的事实说明，容器必须支持所有需要的迭代器操作。

所有的容器都必须支持下面的运算符：

```
=               ==              !=              >
<               <=              >=
```

8.3.2 序列式容器的其他要求

除了默认的构造函数和复制构造函数之外，序列式容器必须提供构造函数来创建并初始化指定数量的元素。并且还必须提供构造函数来创建并初始化给定元素范围的对象。因此，必须支持如下形式的构造函数：

```
Cnt( )

Cnt(c)

Cnt(num, val)

Cnt(start, end)
```

在此，c 是 Cnt 类型的对象，num 是指定一个数量的整型数，val 是与存储在 Cnt 中的对象类型兼容的值，start 和 end 是用来初始化容器的元素范围的迭代器。定制的容器可以指定其他的构造函数。

提示：

除了复制构造函数之外，STL 序列式容器内建的构造函数接受一个参数来指定分配器(默认为 allocator)，但并不要求这么做。

标准 C++为序列式容器定义了如下可选的成员函数：

```
at( )           back( )         front( )        pop_back( )
pop_front( )    push_back( )    push_front( )
```

下标运算符[]也是可选的。当然，您可以加入自己设计的其他成员函数。

8.3.3 关联式容器的要求

所有的关联式容器必须定义这些附加的类型：

```
key_compare         key_type        value_compare
```

除了默认构造函数和复制构造函数之外，所有的关联式容器必须提供允许您指定比较函数的构造函数。您还必须定义一个构造函数，用来创建并初始化给定元素范围的对象。这种构造函数的版本之一必须使用默认的比较函数。其他的版本必须允许用户指定比较函数。也就是说，必须支持如下格式的构造函数：

```
Cnt( )

Cnt(c)

Cnt(comp)

Cnt(start, end)

Cnt(start, end, comp)
```

在此，c 是 Cnt 类型的对象，start 和 end 是用来初始化容器的元素范围的迭代器，comp 是比较函数。定制容器可以指定另外的构造函数。

提示：

除了复制构造函数之外，STL 关联式容器内建的构造函数接受了一个参数来指定分配器(allocator 默认的)，但并不要求这么做。

关联式容器必须提供这些附加的成员函数：

```
count( )          equal_range( )      find( )          key_comp( )
lower_bound( )    upper_bound( )      value_comp( )
```

8.4 创建范围可选的动态数组容器

本章的剩余部分开发了一个定制的序列式容器，名为 RangeArray，这个容器提供了一个范围可选的动态数组。尽管这里所显示的示例演示的是序列式容器的创建，然而其中大多数的概念也可以让用户来实现关联式容器。

8.4.1 RangeArray 的运行方式

正如您所知道的那样，在 C++中，所有的数组都从 0 开始，并且不允许负的索引。然而，允许程序员指定不同端点的数组对于某些应用程序是有意义的。例如笛卡儿坐标平面：每一个数轴都是具有正值和负值的直线。在程序中代表这种直线的简洁方法是使用允许正负索引的数组。例如，给定一条从 - 5 到 5 的线段，您可能会使用能够以如下方式索引的数组：

-5	-4	-3	-2	-1	0	1	2	3	4	5

在此开发的 RangeArray 容器可以创建这样的数组。更为常见的是，RangeArray 允许为数组索引指定任意的上界和下界。

RangeArray 是一个动态容器，允许数组在正负两个方向增长。在这点上，它类似于内建的容器 vector。RangeArray 支持序列式容器要求的所有操作，加上可选的[ ]运算符，以及可选的函数 at()、push_front()、pop_front()等。

下面给出了使用 RangeArray 的示例程序代码：

```
// This creates an array that runs from -3 to 4
// and is initialized to zero.
RangeArray<int> ob(-3, 4, 0);

// Load the values -3 to 4 into ob.
for(int i = -3; i < 5; i++) ob[i] = i;
// ...
cout << ob[-2];
// ...
ob[2] = ob[-1] % 2;
```

您可能已经猜到了，第一行创建了一个从-3 到 4 的 RangeArray 对象，数组中的每个元素都被初始化为 0。其余的行说明这个数组可以通过从-3 到 4 的索引来访问。

通常，RangeArray 允许通过指定数组的上下界以及用来初始化数组中每个元素的值来创建一个对象。也就是说，RangeArray 支持如下的构造函数：

```
RangeArray(lowerbound, upperbound, initvalue)
```

一旦创建了这样的数组，就可以使用其范围内的任何值来对其索引。这意味着允许任意的索引，包括负的索引。

由于 RangeArray 是动态的，因此它允许插入或者删除元素。当发生这样的操作时，数组需要增长或者收缩。然而，关键是数组可以在两个方向增长或者收缩。如果在负方向加入元素，则负方向增长。如果正方向的元素被删除，则正方向收缩。

在开始之前有必要指出，有意没有对 RangeArray 容器的实现进行高性能优化。相反，以最清晰的方式将它优化。这样做的目的是清楚地显示创建定制容器时需要的步骤。同样，如此设计也考虑到了理解的容易程度以及实现的直接性。

8.4.2 完整的 RangeArray 类

在此首先给出 RangeArray 的全部代码。您将会发现，在创建您自己的容器类时，即使最简单的容器也会变为相当大的类。原因当然是必须满足一些要求。尽管单个的要求都不困难，但是将它们组合到一起就需要相当多的代码。不要被吓倒，在下面部分将逐步介绍每一个部分：

```
// A custom container that implements a
// range-selectable array.
//
// Call this file ra.h
//
#include <iostream>
#include <iterator>
#include <algorithm>
#include <cstdlib>
#include <stdexcept>

using namespace std;

// An exception class for RangeArray.
```

```
class RAExc {
  string err;
public:

  RAExc(string e) {
    err = e;
  }

  string geterr() { return err; }
};

// A range-selectable array container.
template<class T, class Allocator = allocator<T> >
class RangeArray {
  T *arrayptr; // pointer to array that underlies the container

  unsigned len; // holds length of the container
  int upperbound; // lower bound
  int lowerbound; // upper bound
  Allocator a; // allocator
public:

  // Required typedefs for container.
  typedef T value_type;
  typedef Allocator allocator_type;
  typedef typename Allocator::reference reference;
  typedef typename Allocator::const_reference const_reference;
  typedef typename Allocator::size_type size_type;
  typedef typename Allocator::difference_type difference_type;
  typedef typename Allocator::pointer pointer;
  typedef typename Allocator::const_pointer const_pointer;

  // Forward iterators.
  typedef T * iterator;
  typedef const T * const_iterator;

  // Note: This container does not support reverse
  // iterators, but you can add them if you like.

  // ***** Constructors and Destructor *****

  // Default constructor.
  RangeArray()
  {
    upperbound = lowerbound = 0;
    len = 0;
    arrayptr = a.allocate(0);
  }

  // Construct an array of the specified range
  // with each element having the specified initial value.
  RangeArray(int low, int high, const T &t);
```

```
// Construct zero-based array of num elements
// with the value t. This constructor is required
// for STL compatibility.
RangeArray(int num, const T &t=T());

// Construct from range of iterators.
RangeArray(iterator start, iterator stop);

// Copy constructor.
RangeArray(const RangeArray &o);

// Destructor.
~RangeArray();


// ***** Operator Functions *****

// Return reference to specified element.
T &operator[](int i)
{
  return arrayptr[i - lowerbound];
}

// Return const references to specified element.
const T &operator[](int i) const
{
  return arrayptr[i - lowerbound];
}

// Assign one container to another.
RangeArray &operator=(const RangeArray &o);


// ***** Insert Functions *****

// Insert val at p.
iterator insert(iterator p, const T &val);

// Insert num copies of val at p.
void insert(iterator p, int num, const T &val)
{
  for(; num>0; num--) p = insert(p, val) + 1;
}

// Insert range specified by start and stop at p.
void insert(iterator p, iterator start, iterator stop)
{
  while(start != stop) {
    p = insert(p, *start) + 1;
    start++;
  }
```

```
}

// ***** Erase Functions *****

// Erase element at p.
iterator erase(iterator p);

// Erase specified range.
iterator erase(iterator start, iterator stop)
{
  iterator p = end();

  for(int i=stop-start; i > 0; i--)
    p = erase(start);

  return p;
}


// ***** Push and Pop Functions *****

// Add element to end.
void push_back(const T &val)
{
  insert(end(), val);
}

// Remove element from end.
void pop_back()
{
  erase(end()-1);
}

// Add element to front.
void push_front(const T &val)
{
  insert(begin(), val);
}

// Remove element from front.
void pop_front()
{
  erase(begin());
}


// ***** front() and back() functions *****

// Return reference to first element.
T &front()
{
```

```
  return arrayptr[0];
}

// Return const reference to first element.
const T &front() const
{
  return arrayptr[0];
}

// Return reference to last element.
T &back()
{
  return arrayptr[len-1];
}

// Return const reference to last element.
const T &back() const
{
  return arrayptr[len-1];
}


// ***** Iterator Functions *****

// Return iterator to first element.
iterator begin()
{
  return &arrayptr[0];
}

// Return iterator to last element.
iterator end()
{
  return &arrayptr[upperbound - lowerbound];
}

// Return const iterator to first element.
const_iterator begin() const
{
  return &arrayptr[0];
}

// Return const iterator to last element.
const_iterator end() const
{
  return &arrayptr[upperbound - lowerbound];
}


// ***** Misc. Functions *****

// The at() function performs a range check.
```

```
// Return a reference to the specified element.
T &at(int i)
{
  if(i < lowerbound || i >= upperbound)
    throw out_of_range("Index Out of Range");

  return arrayptr[i - lowerbound];
}

// Return a const reference to the specified element.
const T &at(int i) const
{
  if(i < lowerbound || i >= upperbound)
    throw out_of_range("Index Out of Range");

return arrayptr[i - lowerbound];
}

// Return the size of the container.
size_type size() const
{
  return end() - begin();
}

// Return the maximum size of a RangeArray.
size_type max_size()
{
  return a.max_size();
}

// Return true if container is empty.
bool empty()
{
  return size() == 0;
}

// Exchange the values of two containers.
void swap(RangeArray &b)
{
  RangeArray<T> tmp;

  tmp = *this;
  *this = b;
  b = tmp;
}

// Remove and destroy all elements.
void clear()
{
  erase(begin(), end());
}
```

```
  // ***** Non-STL functions *****

  // Return endpoints.
  int getlowerbound()
  {
    return lowerbound;
  }

  int getupperbound()
  {
    return upperbound;
  }

};

// ***** Implementations of non-inline functions *****

// Construct an array of the specified range
// with each element having the specified initial value.
template <class T, class A>
RangeArray<T, A>::RangeArray(int low, int high,
                             const T &t)
{
  if(high <= low) throw RAExc("Invalid Range");

  high++;

  // Save endpoints.
  upperbound = high;
  lowerbound = low;

  // Allocate memory for the container.
  arrayptr = a.allocate(high - low);

  // Save the length of the container.
  len = high - low;

  // Construct the elements.
  for(size_type i=0; i < size(); i++)
    a.construct(&arrayptr[i], t);
}

// Construct zero-based array of num elements
// with the value t. This constructor is required
// for STL compatibility.
template <class T, class A>
RangeArray<T, A>::RangeArray(int num, const T &t) {

  // Save endpoints.
  upperbound = num;
  lowerbound = 0;
```

```
  // Allocate memory for the container.
  arrayptr = a.allocate(num);

  // Save the length of the container.
  len = num;

  // Construct the elements.
  for(size_type i=0; i < size(); i++)
    a.construct(&arrayptr[i], t);

}

// Construct zero-based array from range of iterators.
// This constructor is required for STL compatibility.
template <class T, class A>
RangeArray<T, A>::RangeArray(iterator start,
                             iterator stop)
{
  // Allocate sufficient memory.
  arrayptr = a.allocate(stop - start);

  upperbound = stop - start;
  lowerbound = 0;

  len = stop - start;

  // Construct the elements using those
  // specified by the range of iterators.
  for(size_type i=0; i < size(); i++)
    a.construct(&arrayptr[i], *start++);
}

// Copy constructor.
template <class T, class A>
RangeArray<T, A>::RangeArray(const RangeArray<T, A> &o)
{
  // Allocate memory for the copy.
  arrayptr = a.allocate(o.size());

  upperbound = o.upperbound;
  lowerbound = o.lowerbound;
  len = o.len;

  // Make the copy.
  for(size_type i=0; i < size(); i++)
    a.construct(&arrayptr[i], o.arrayptr[i]);
}

// Destructor.
template <class T, class A>
RangeArray<T, A>::~RangeArray()
{
```

```
  // Call destructors for elements in the container.
  for(size_type i=0; i < size(); i++)
    a.destroy(&arrayptr[i]);

  // Release memory.
  a.deallocate(arrayptr, size());
}

// Assign one container to another.
template <class T, class A> RangeArray<T, A> &
RangeArray<T, A>::operator=(const RangeArray<T, A> &o)
{
  // Call destructors for elements in target container.
  for(size_type i=0; i < size(); i++)
    a.destroy(&arrayptr[i]);

  // Release original memory.
  a.deallocate(arrayptr, size());

  // Allocate memory for new size.
  arrayptr = a.allocate(o.size());

  upperbound = o.upperbound;
  lowerbound = o.lowerbound;
  len = o.len;

  // Make copy.
  for(size_type i=0; i < size(); i++)
    arrayptr[i] = o.arrayptr[i];

  return *this;
}

// Insert val at p.
template <class T, class A>
typename RangeArray<T, A>::iterator
RangeArray<T, A>::insert(iterator p, const T &val)
{
  iterator q;
  size_type i, j;

  // Get sufficient memory.
  T *tmp = a.allocate(size() + 1);

  // Copy existing elements to new array,
  // inserting new element if possible.
  for(i=j=0; i < size(); i++, j++) {
    if(&arrayptr[i] == p) {
      tmp[j] = val;
      q = &tmp[j];
      j++;
    }
```

```
    tmp[j] = arrayptr[i];
  }

  // Otherwise, the new element goes on end.
  if(p == end()) {
    tmp[j] = val;
    q = &tmp[j];
  }

  // Adjust len and bounds.
  len++;
  if(p < &arrayptr[abs(lowerbound)])
    lowerbound--;
  else
    upperbound++;

  // Call destructors for elements in old container.
  for(size_type i=0; i < size()-1; i++)
    a.destroy(&arrayptr[i]);

  // Release memory for old container.
  a.deallocate(arrayptr, size()-1);

  arrayptr = tmp;

  return q;
}

// Erase element at p.
template <class T, class A>
typename RangeArray<T, A>::iterator
RangeArray<T, A>::erase(iterator p)
{
  iterator q = p;

  // Destruct element being erased.
  if(p != end()) a.destroy(p);

  // Adjust len and bounds.
  len--;
  if(p < &arrayptr[abs(lowerbound)])
    lowerbound++;
  else
    upperbound--;

  // Compact remaining elements.
  for( ; p < end(); p++)
    *p = *(p+1);

  return q;
}
```

```
// ******** Relational Operators **************

template<class T, class Allocator>
  bool operator==(const RangeArray<T, Allocator> &a,
                  const RangeArray<T, Allocator> &b)
{
  if(a.size() != b.size()) return false;

  return equal(a.begin(), a.end(), b.begin());
}

template<class T, class Allocator>
  bool operator!=(const RangeArray<T, Allocator> &a,
                  const RangeArray<T, Allocator> &b)
{
  if(a.size() != b.size()) return true;

  return !equal(a.begin(), a.end(), b.begin());
}

template<class T, class Allocator>
  bool operator<(const RangeArray<T, Allocator> &a,
                 const RangeArray<T, Allocator> &b)
{
  return lexicographical_compare(a.begin(), a.end(),
                                 b.begin(), b.end());
}

template<class T, class Allocator>
  bool operator>(const RangeArray<T, Allocator> &a,
                 const RangeArray<T, Allocator> &b)
{
  return b < a;
}

template<class T, class Allocator>
  bool operator<=(const RangeArray<T, Allocator> &a,
                  const RangeArray<T, Allocator> &b)
{
  return !(a > b);
}

template<class T, class Allocator>
  bool operator>=(const RangeArray<T, Allocator> &a,
                  const RangeArray<T, Allocator> &b)
{
  return !(a < b);
}
```

正如代码开始的注释所指出的那样，您应该将所有的代码都放到一个文件 ra.h 中。在后面所示的示例程序中将会使用它。

前面的代码包含两个类。第一个是 RAExc 异常类。如果使用无效的边界创建一个 RangeArray，如下界大于上界，则会抛出这种类型的异常。第二个是 RangeArray 容器类，在下面部分将详细介绍这个类。

8.4.3 详细讨论 RangeArray 类

如同所有内建的 STL 序列式容器一样，RangeArray 以如下的模板说明开始：

```
template<class T, class Allocator=allocator<T>>
```

T 是容器中存储的数据类型，Allocator 是分配器，默认情况下是标准分配器。

1. 私有成员

RangeArray 数组类以如下的私有声明开始：

```
T *arrayptr; // pointer to array that underlies the container

unsigned len;    // holds length of the container
int upperbound;  // lower bound
int lowerbound;  // upper bound

Allocator a; // allocator
```

指针 arrayptr 存储了一个指针，这个指针指向的内存将包含元素类型为 T 的数组。这块内存将存储类型为 RangeArray 的对象所保存的元素。这个数组的索引通常从 0 开始。RangeArray 对象的索引将从基于 0 的索引转换为 arrayptr 所指向的数组。

RangeArray 当前的长度存储在 len 中。数组的上界和下界分别存储在 upperbound 和 lowerbound 中。对于 RangeArray 来说，0 被认为是正数。容器的分配器存储在 a 中。

2. 所需的类型定义

在私有成员之后，RangeArray 定义了所有序列式容器需要的各种 typedef，如下所示：

```
// Required typedefs for container.
typedef T value_type;
typedef Allocator allocator_type;
typedef typename Allocator::reference reference;
typedef typename Allocator::const_reference const_reference;
typedef typename Allocator::size_type size_type;
typedef typename Allocator::difference_type difference_type;
typedef typename Allocator::pointer pointer;
typedef typename Allocator::const_pointer const_pointer;

// Forward iterators.
typedef T * iterator;
typedef const T * const_iterator;
```

这些 typedef 类似于内建的 STL 容器使用的 typedef。注意，前向迭代器只是指向类型 T 的对象的一些指针。对于 RangeArray 而言，这就足够了，但是对于更加复杂的容器，可能并非如此。另外，在此没有提供反向迭代器，但是您可以将其作为一个有趣的练习试着加入它们。

3. RangeArray 的构造函数和析构函数

为了创建一个兼容的容器，您必须支持前面描述的为序列式容器定义的 4 个构造函数。RangeArray 还定义了第 5 个构造函数，允许您创建一个指定范围的数组。这些构造函数如下所示：

```
// Default constructor.
RangeArray()
{
  upperbound = lowerbound = 0;
  len = 0;
  arrayptr = a.allocate(0);
}

// Construct an array of the specified range
// with each element having the specified initial value.
template <class T, class A>
RangeArray<T, A>::RangeArray(int low, int high,
                             const T &t)
{
  if(high <= low) throw RAExc("Invalid Range");

  high++;

  // Save endpoints.
  upperbound = high;
  lowerbound = low;

  // Allocate memory for the container.
  arrayptr = a.allocate(high - low);

  // Save the length of the container.
  len = high - low;

  // Construct the elements.
  for(size_type i=0; i < size(); i++)
    a.construct(&arrayptr[i], t);
  }

  // Construct zero-based array of num elements
  // with the value t. This constructor is required
  // for STL compatibility.
  template <class T, class A>
  RangeArray<T, A>::RangeArray(int num, const T &t) {

    // Save endpoints.
    upperbound = num;
    lowerbound = 0;

    // Allocate memory for the container.
    arrayptr = a.allocate(num);
```

```
  // Save the length of the container.
  len = num;

  // Construct the elements.
  for(size_type i=0; i < size(); i++)
    a.construct(&arrayptr[i], t);
}

// Construct zero-based array from range of iterators.
// This constructor is required for STL compatibility.
template <class T, class A>
RangeArray<T, A>::RangeArray(iterator start,
                             iterator stop)
{
  // Allocate sufficient memory.
  arrayptr = a.allocate(stop - start);

  upperbound = stop - start;
  lowerbound = 0;

  len = stop - start;

  // Construct the elements using those
  // specified by the range of iterators.
  for(size_type i=0; i < size(); i++)
    a.construct(&arrayptr[i], *start++);
}

// Copy constructor.
template <class T, class A>
RangeArray<T, A>::RangeArray(const RangeArray<T, A> &o)
{
  // Allocate memory for the copy.
  arrayptr = a.allocate(o.size());

  upperbound = o.upperbound;
  lowerbound = o.lowerbound;
  len = o.len;

  // Make the copy.
  for(size_type i=0; i < size(); i++)
    a.construct(&arrayptr[i], o.arrayptr[i]);
}
```

第一个构造函数是默认构造函数，它创建一个空的对象。STL 指定的限制之一是对默认对象调用 size()的结果必须为 0。因此，默认构造函数将上界和下界设置为 0。另外，它还将 len 设置为 0，并且使用分配器提供的 allocate()函数创建了一个长度为 0 的数组。最后两步确保在所有情况下都存在完整的对象。

第二个构造函数是 RangeArray 特有的。它创建了一个指定范围的对象，每个元素都具有一

定的初始值。下界在第一个参数中指定；上界在第二个参数中指定。如果上界小于或者等于下界，就会抛出 RAExc 异常。第三个参数传递了初始值。因此，这条语句：

```
RangeArray<char> ch(-2,10, 'X');
```

创建了一个字符数组，从 -2 开始一直到 10，每个元素的初始值都是 X。这个数组使用 allocate() 来分配，allocate()是 allocator 类的一个成员函数。为了创建一个兼容的容器，您必须使用分配器函数而不是 new 函数来获取容器需要的内存。一旦完成分配，数组中的每个元素都使用作为第三个参数传递的值来创建。(如果可以使用默认的初始化值，可能会更方便，因为不需要指定这个值。但是这样做会与第三个构造函数混淆，STL 需要第三个构造函数)。这个创建过程是通过使用另一个分配器函数 construct()完成的。

第三个构造函数创建了一个基数为 0 的数组，这个数组具有指定数量的元素，使用指定的值初始化每个元素。可以使用默认的初始化值。这个构造函数不是只对 RangeArray 有用，而且是被 STL 容器规范要求的。

第四个构造函数创建了一个给定值范围的、基数为 0 的数组。这个范围是由传递的指向范围的开头以及结尾的迭代器指定的。这个构造函数不仅对于 RangeArray 是有用的，而且是被 STL 容器规范所要求的。

最后一个是复制构造函数。它为一个新的对象分配内存，并从源对象复制边界、长度以及元素。因此，这个副本具有自己的内存，但是其他方面与源对象相同。

RangeArray 的析构函数如下所示：

```
// Destructor.
template <class T, class A>
RangeArray<T, A>::~RangeArray()
{
  // Call destructors for elements in the container.
  for(size_type i=0; i < size(); i++)
    a.destroy(&arrayptr[i]);

  // Release memory.
  a.deallocate(arrayptr, size());
}
```

首先，它通过调用分配器的 destroy()函数销毁数组中的每个元素。然后通过调用分配器的 deallocate()函数释放数组使用的内存。

记住，RangeArray 的构造函数没有使用 new 分配内存，析构函数也不会使用 delete 释放内存。相反，它们使用了分配器提供的合适的成员函数。这种方式使得用户可以指定管理容器内存的另一种方法。

4. RangeArray 的运算符函数

RangeArray 定义了 3 个成员运算符函数。前两个 operator[]()函数如下所示：

```
// Return reference to specified element.
T &operator[](int i)
{
  return arrayptr[i - lowerbound];
```

```
}

// Return const references to specified element.
const T &operator[](int i) const
{
  return arrayptr[i - lowerbound];
}
```

对于完整的容器，需要[]运算符的 const 和非 const 版本。这些函数提供了检索 RangeArray 的机制。特别注意检索 arrayptr 的方式。记住，arrayptr 指向一个标准的 C++数组。因此，i 中传递的索引必须转换为 arrayptr 中基数为 0 的索引。lowerbound 保存了对应于 RangeArray 中最低索引的值。因此，为了获取基数为 0 的索引，从 i 中减去 lowerbound。

另一个需要注意的问题：operator[]()没有执行边界检查。STL 没有要求这个运算符执行边界检查，在此也没有包括。这是合理的，因为除了使得用户指定的范围之外， RangeArray 的行为类似于普通的数组。(记住，普通的 C++数组没有提供边界检查)。当然，如果您愿意，可以在 operator[]()中加入边界检查。

下面所示的 operator=()函数有一点复杂：

```
// Assign one container to another.
template <class T, class A> RangeArray<T, A> &
RangeArray<T, A>::operator=(const RangeArray<T, A> &o)
{
  // Call destructors for elements in target container.
  for(size_type i=0; i < size(); i++)
    a.destroy(&arrayptr[i]);

  // Release original memory.
  a.deallocate(arrayptr, size());

  // Allocate memory for new size.
  arrayptr = a.allocate(o.size());

  upperbound = o.upperbound;
  lowerbound = o.lowerbound;
  len = o.len;

  // Make copy.
  for(size_type i=0; i < size(); i++)
    arrayptr[i] = o.arrayptr[i];

  return *this;
}
```

在这个程序中，首先销毁并释放目标对象中存在的任何对象。然后分配足够的内存来保存源对象中的内容。随后恰当地设置成员变量，然后复制元素。最后返回一个目标对象的引用。

5. insert()函数

STL 要求序列式容器支持三种形式的 insert()：一个用来插入值，一个用来插入一个值的多

个副本，还有一个用来确定范围。第一个版本的 insert()如下所示：

```
// Insert val at p.
template <class T, class A>
typename RangeArray<T, A>::iterator
RangeArray<T, A>::insert(iterator p, const T &val)
{
  iterator q;
  size_type i, j;

  // Get sufficient memory.
  T *tmp = a.allocate(size() + 1);

  // Copy existing elements to new array,
  // inserting new element if possible.
  for(i=j=0; i < size(); i++, j++) {
    if(&arrayptr[i] == p) {
      tmp[j] = val;
      q = &tmp[j];
      j++;
    }
    tmp[j] = arrayptr[i];
  }

  // Otherwise, the new element goes on end.
  if(p == end()) {
    tmp[j] = val;
    q = &tmp[j];
  }

  // Adjust len and bounds.
  len++;
  if(p < &arrayptr[abs(lowerbound)])
    lowerbound--;
  else
    upperbound++;

  // Call destructors for elements in old container.
  for(size_type i=0; i < size()-1; i++)
    a.destroy(&arrayptr[i]);

  // Release memory for old container.
  a.deallocate(arrayptr, size()-1);

  arrayptr = tmp;

  return q;
}
```

第一个版本以一个迭代器作为第一个参数，在迭代器指定的位置插入一个元素。这个函数返回一个插入元素的迭代器。为了保存现有的元素和新加入的元素，它分配了一个足够大的内

存片断。然后将现有的元素复制到新分配的内存中，并将新的元素插入到合适的位置。随后恰当地更新了 len 以及 upperbound(或者 lowerbound)。在下一步，它从原来的 RangeArray 销毁并释放了对象，并将新内存的地址赋给 arrayptr。最后，返回一个指向插入对象的指针。

现在，仔细观察更新 lowerbound 或者 upperbound 变量的代码。数组可以向正方向或者负方向增长，这取决于新的元素是在数组的正方向还是负方向插入。因此，有必要判断插入发生的位置，并改变合适的值。这个判断是通过比较 p 中传递的迭代器以及 arrayptr[lowerbound]中元素的地址而做出的。如果迭代器小于这个元素，负方向扩展；否则，正方向扩展。

第二个版本的 insert()如下所示：

```
// Insert num copies of val at p.
void insert(iterator p, int num, const T &val)
{
  for(; num>0; num--) p = insert(p, val) + 1;
}

// Insert range specified by start and stop at p.
void insert(iterator p, iterator start, iterator stop)
{
  while(start != stop) {
    p = insert(p, *start) + 1;
    start++;
  }
}
```

6. erase 函数

序列式容器必须支持两种版本的 erase()函数。第一种版本的 erase()函数删除了迭代器所指的元素。当删除了某个元素之后，立刻返回指向这个元素的迭代器，如果最后一个元素被删除，则返回 end()。这个版本的 erase()函数如下所示：

```
// Erase element at p.
template <class T, class A>
typename RangeArray<T, A>::iterator
RangeArray<T, A>::erase(iterator p)
{
  iterator q = p;

  // Destruct element being erased.
  if(p != end()) a.destroy(p);

  // Adjust len and bounds.
  len--;
  if(p < &arrayptr[abs(lowerbound)])
    lowerbound++;
  else
    upperbound--;

  // Compact remaining elements.
  for( ; p < end(); p++)
    *p = *(p+1);
```

```
  return q;
}
```

这个版本的 erase()函数通过销毁被删除的元素、合适地调整上下界，然后压缩剩余的元素来完成操作。

第二个版本采用了第一个版本的框架，如下所示。它删除了多个元素。它返回指向这个范围中最后元素的迭代器(也就是说，stop 指向的那个元素)。因此，它删除了在 start 和 stop - 1 之间的元素。

```
// Erase specified range.
iterator erase(iterator start, iterator stop)
{
  iterator p = end();

  for(int i=stop-start; i > 0; i--)
    p = erase(start);

  return p;
}
```

7. 压入与弹出函数

RangeArray 实现了 push_back()、pop_back()、push_front()以及 pop_front()，如下所示。正如您所看到的那样，它们是通过 insert()和 erase()实现的，并且它们的操作方式相当直接：

```
// Add element to end.
void push_back(const T &val)
{
  insert(end(), val);
}

// Remove element from end.
void pop_back()
{
  erase(end()-1);
}

// Add element to front.
void push_front(const T &val)
{
  insert(begin(), val);
}

// Remove element from front.
void pop_front()
{
  erase(begin());
}
```

8. front()和 back()函数

front()和 back()函数各自简单地返回了指向数组开始和结尾的迭代器，如下所示。它们的实现相当直接。然而要注意，const 和非 const 的两个版本都需要。

```
// Return reference to first element.
T &front()
{
  return arrayptr[0];
}

// Return const reference to first element.
const T &front() const
{
  return arrayptr[0];
}

// Return reference to last element.
T &back()
{
  return arrayptr[len-1];
}

// Return const reference to last element.
const T &back() const
{
  return arrayptr[len-1];
}
```

9. 迭代器函数

由于 RangeArray 容器的迭代器只是简单的指针，保存在 arrayptr 指向的内存中。迭代器函数 begin()和 end()很小。注意 const 和非 const 的两个版本都需要。

```
// Return iterator to first element.
iterator begin()
{
  return &arrayptr[0];
}

// Return iterator to last element.
iterator end()
{
  return &arrayptr[upperbound - lowerbound];
}

// Return const iterator to first element.
const_iterator begin() const
{
  return &arrayptr[0];
}
```

```
// Return const iterator to last element.
const_iterator end() const
{
  return &arrayptr[upperbound - lowerbound];
}
```

10. 其他函数

所有的序列式容器都必须实现 size()、max_size()、empty()、swap()以及 clear()。这些函数如下所示：

```
// Return the size of the container.
size_type size() const
{
  return end() - begin();
}

// Return the maximum size of a RangeArray.
size_type max_size()
{
  return a.max_size();
}

// Return true if container is empty.
bool empty()
{
  return size() == 0;
}

// Exchange the values of two containers.
void swap(RangeArray &b)
{
  RangeArray<T> tmp;
  tmp = *this;
  *this = b;
  b = tmp;
}

// Remove and destroy all elements.
void clear()
{
  erase(begin(), end());
}
```

通常，凭直觉就可以操作这些函数。然而，max_size()值得花一些文字来描述。它返回能够创建的最大容器所能包含的 T 类型元素的数量。这个值通过调用分配器定义的 max_size()函数获得。

RangeArray 还实现了可选的 at()函数，如下所示：

```
// The at() function performs a range check.
// Return a reference to the specified element.
```

```
T &at(int i)
{
  if(i < lowerbound || i >= upperbound)
    throw out_of_range("Index Out of Range");

  return arrayptr[i - lowerbound];
}

// Return a const reference to the specified element.
const T &at(int i) const
{
  if(i < lowerbound || i >= upperbound)
    throw out_of_range("Index Out of Range");

  return arrayptr[i - lowerbound];
}
```

at()函数返回指定索引的位置处元素的引用。它与 operator[]()仅有的区别是它对索引的范围执行检查。如果这个索引越界，at()就会抛出 out_of_range 异常。这个异常在 C++的头文件<stdexcept>中定义。

RangeArray 还提供了非 STL 函数 getlowerbound()和 getupperbound()。这两个函数分别获取 RangeArray 的上界和下界，如下所示：

```
// Return endpoints.
int getlowerbound()
{
  return lowerbound;
}

int getupperbound()
{
  return upperbound;
}
```

11. 关系运算符

为 RangeArray 定义的关系运算符如下所示：

```
template<class T, class Allocator>
  bool operator==(const RangeArray<T, Allocator> &a,
                  const RangeArray<T, Allocator> &b)
{
  if(a.size() != b.size()) return false;

  return equal(a.begin(), a.end(), b.begin());
}

template<class T, class Allocator>
  bool operator!=(const RangeArray<T, Allocator> &a,
                  const RangeArray<T, Allocator> &b)
{
```

```
  if(a.size() != b.size()) return true;

  return !equal(a.begin(), a.end(), b.begin());
}

template<class T, class Allocator>
  bool operator<(const RangeArray<T, Allocator> &a,
                 const RangeArray<T, Allocator> &b)
{
  return lexicographical_compare(a.begin(), a.end(),
                                 b.begin(), b.end());
}

template<class T, class Allocator>
  bool operator>(const RangeArray<T, Allocator> &a,
                 const RangeArray<T, Allocator> &b)
{
  return b < a;
}

template<class T, class Allocator>
  bool operator<=(const RangeArray<T, Allocator> &a,
                  const RangeArray<T, Allocator> &b)
{
  return !(a > b);
}

template<class T, class Allocator>
  bool operator>=(const RangeArray<T, Allocator> &a,
                  const RangeArray<T, Allocator> &b)
{
  return !(a < b);
}
```

operator=()和 operator!=()都使用了 equal()算法来判断等同性。如同 equal()定义的那样，如果两个对象以相同的顺序包含了相同的元素，这两个对象就是相等的。

运算符“<”使用了 lexicographical_compare()来判断一个对象何时小于另一个对象。标准 C++推荐使用这个函数。它通过比较两个序列中对应的元素，并查找第一个不相等的元素。如果能找到，当第一个范围的元素比第二个范围的元素小时，返回 true。否则返回 false。

8.4.4 一些 RangeArray 示例程序

为了演示 RangeArray，下面给出了 3 个示例程序。第一个演示不同的成员函数，如下所示。这个程序还使用了 3 个算法，一个函数对象以及一个绑定器。这些元素的使用说明了 RangeArray 是功能齐全的、与 STL 其余部分兼容的容器。

```
// Demonstrate basic RangeArray operations.
#include <iostream>
#include <algorithm>
#include <functional>
```

```
#include "ra.h"
using namespace std;

// Display integers -- for use by for_each.
void display(int v)
{
  cout << v << " ";
}

int main()
{
  RangeArray<int> ob(-5, 5, 0);
  RangeArray<int>::iterator p;
  int i, sum;
  cout << "Size of ob is: " << ob.size() << endl;

  cout << "Initial contents of ob:\n";
  for(i=-5; i <= 5; i++) cout << ob[i] << " ";
  cout << endl;

  // Give ob some values.
  for(i=-5; i <= 5; i++) ob[i] = i;

  cout << "New values for ob: \n";
  p = ob.begin();
  do {
    cout << *p++ << " ";
  } while (p != ob.end());
  cout << endl;

  // Display sum of negative indexes.
  sum = 0;
  for(i = ob.getlowerbound(); i < 0; i++)
    sum += ob[i];
  cout << "Sum of values with negative subscripts is: ";
  cout << sum << "\n\n";

  // Use copy() algorithm to copy one object to another.
  cout << "Copy ob to ob2 using copy() algorithm.\n";

  RangeArray<int> ob2(-5, 5, 0);
  copy(ob.begin(), ob.end(), ob2.begin());

  // Use for_each() algorithm to display ob2.
  cout << "Contents of ob2: \n";
  for_each(ob2.begin(), ob2.end(), display);
  cout << "\n\n";

  // Use replace_copy_if() algorithm to remove values less than zero.
  cout << "Replace values less than zero with zero.\n";
  cout << "Put the result into ob3.\n";
  RangeArray<int> ob3(ob.begin(), ob.end());
```

```
// The next line uses the function object less() and
// the binder bind2nd().
replace_copy_if(ob.begin(), ob.end(), ob3.begin(),
                bind2nd(less<int>(), 0), 0);
cout << "Contents of ob3: \n";
for_each(ob3.begin(), ob3.end(), display);
cout << "\n\n";

cout << "Swap ob and ob3.\n";
ob.swap(ob3); // swap ob and ob3
cout << "Here is ob3:\n";
for_each(ob3.begin(), ob3.end(), display);
cout << endl;
cout << "Swap again to restore.\n";
ob.swap(ob3); // restore
cout << "Here is ob3 after second swap:\n";
for_each(ob3.begin(), ob3.end(), display);
cout << "\n\n";

// Use insert() member functions.
cout << "Element at ob[0] is " << ob[0] << endl;
cout << "Insert values into ob.\n";
ob.insert(ob.end(), -9999);
ob.insert(&ob[1], 99);
ob.insert(&ob[-3], -99);
for_each(ob.begin(), ob.end(), display);
cout << endl;
cout << "Element at ob[0] is " << ob[0] << "\n\n";

cout << "Insert -7 three times to front of ob.\n";
ob.insert(ob.begin(), 3, -7);
for_each(ob.begin(), ob.end(), display);
cout << endl;
cout << "Element at ob[0] is " << ob[0] << "\n\n";

// Use push_back() and pop_back().
cout << "Push back the value 40 onto ob.\n";
ob.push_back(40);
for_each(ob.begin(), ob.end(), display);
cout << endl;
cout << "Pop back two values from ob.\n";
ob.pop_back(); ob.pop_back();
for_each(ob.begin(), ob.end(), display);
cout << "\n\n";

// Use push_front() and pop_front().
cout << "Push front the value 19 onto ob.\n";
ob.push_front(19);
for_each(ob.begin(), ob.end(), display);
cout << endl;
cout << "Pop front two values from ob.\n";
```

```
ob.pop_front(); ob.pop_front();
for_each(ob.begin(), ob.end(), display);
cout << "\n\n";

// Use front() and back()
cout << "ob.front(): " << ob.front() << endl;
cout << "ob.back(): " << ob.back() << "\n\n";

// Use erase().
cout << "Erase element at 0.\n";
p = ob.erase(&ob[0]);
for_each(ob.begin(), ob.end(), display);
cout << endl;
cout << "Element at ob[0] is " << ob[0] << endl;
cout << endl;

cout << "Erase many elements in ob.\n";
p = ob.erase(&ob[-2], &ob[3]);
for_each(ob.begin(), ob.end(), display);
cout << endl;
cout << "Element at ob[0] is " << ob[0] << endl;
cout << endl;

cout << "Insert ob4 into ob.\n";
RangeArray<int> ob4(0, 2, 0);
for(i=0; i < 3; i++) ob4[i] = i+100;
ob.insert(&ob[0], ob4.begin(), ob4.end());
for_each(ob.begin(), ob.end(), display);
cout << endl;
cout << "Element at ob[0] is " << ob[0] << endl;
cout << endl;

cout << "Here is ob shown with its indices:\n";
for(i=ob.getlowerbound(); i<ob.getupperbound(); i++)
cout << "[" << i << "]: " << ob[i] << endl;
cout << endl;

// Use the at() function.
cout << "Use the at() function.\n";
for(i=ob.getlowerbound(); i < ob.getupperbound(); i++)
  ob.at(i) = i * 11;

for(i=ob.getlowerbound(); i < ob.getupperbound(); i++)
  cout << ob.at(i) << " ";
cout << "\n\n";

// Use the clear() function.
cout << "Clear ob.\n";
ob.clear();
for_each(ob.begin(), ob.end(), display); // no effect!
cout << "Size of ob after clear: " << ob.size()
     << "\nBounds: " << ob.getlowerbound()
```

```
        << " to " << ob.getupperbound() << "\n\n";

  // Create a copy of an object.
  cout << "Make a copy of ob2.\n";
  RangeArray<int> ob5(ob2);
  for_each(ob5.begin(), ob5.end(), display);
  cout << "\n\n";

  // Construct a new object from a range.
  cout << "Construct object from a range.\n";
  RangeArray<int> ob6(&ob2[-2], ob2.end());
  cout << "Size of ob6: " << ob6.size() << endl;
  for_each(ob6.begin(), ob6.end(), display);
  cout << endl;

  return 0;
}
```

这个程序的输出如下所示：

```
Size of ob is: 11
Initial contents of ob:
0 0 0 0 0 0 0 0 0 0 0
New values for ob:
-5 -4 -3 -2 -1 0 1 2 3 4 5
Sum of values with negative subscripts is: -15

Copy ob to ob2 using copy() algorithm.
Contents of ob2:
-5 -4 -3 -2 -1 0 1 2 3 4 5

Replace values less than zero with zero.
Put the result into ob3.
Contents of ob3:
0 0 0 0 0 0 1 2 3 4 5

Swap ob and ob3.
Here is ob3:
-5 -4 -3 -2 -1 0 1 2 3 4 5
Swap again to restore.

Here is ob3 after second swap:
0 0 0 0 0 0 1 2 3 4 5

Element at ob[0] is 0
Insert values into ob.
-5 -4 -99 -3 -2 -1 0 99 1 2 3 4 5 -9999
Element at ob[0] is 0

Insert -7 three times to front of ob.
-7 -7 -7 -5 -4 -99 -3 -2 -1 0 99 1 2 3 4 5 -9999
Element at ob[0] is 0
```

```
Push back the value 40 onto ob.
-7 -7 -7 -5 -4 -99 -3 -2 -1 0 99 1 2 3 4 5 -9999 40
Pop back two values from ob.
-7 -7 -7 -5 -4 -99 -3 -2 -1 0 99 1 2 3 4 5

Push front the value 19 onto ob.
19 -7 -7 -7 -5 -4 -99 -3 -2 -1 0 99 1 2 3 4 5
Pop front two values from ob.
-7 -7 -5 -4 -99 -3 -2 -1 0 99 1 2 3 4 5

ob.front(): -7
ob.back(): 5

Erase element at 0.
-7 -7 -5 -4 -99 -3 -2 -1 99 1 2 3 4 5
Element at ob[0] is 99

Erase many elements in ob.
-7 -7 -5 -4 -99 -3 3 4 5
Element at ob[0] is 3

Insert ob4 into ob.
-7 -7 -5 -4 -99 -3 100 101 102 3 4 5
Element at ob[0] is 100

Here is ob shown with its indices:
[-6]: -7
[-5]: -7
[-4]: -5
[-3]: -4
[-2]: -99
[-1]: -3
[0]: 100
[1]: 101
[2]: 102
[3]: 3
[4]: 4
[5]: 5

Use the at() function.
-66 -55 -44 -33 -22 -11 0 11 22 33 44 55

Clear ob.
Size of ob after clear: 0
Bounds: 0 to 0

Make a copy of ob2.
-5 -4 -3 -2 -1 0 1 2 3 4 5

Construct object from a range.
Size of ob6: 8
-2 -1 0 1 2 3 4 5
```

下一个示例程序演示了关系运算符：

```
// Demonstrate the relational operators.
#include <iostream>
#include "ra.h"

using namespace std;

// Display integers -- for use by for_each.
void display(int v)
{
  cout << v << " ";
}

int main()
{
  RangeArray<int> ob1(-3, 2, 0), ob2(-3, 2, 0), ob3(-4, 4, 0);
  int i;

  // Give ob1 and ob2 some values.
  for(i = -3; i < 3; i++) {
    ob1[i] = i;
    ob2[i] = i;
  }

  cout << "Contents of ob1 and ob2:\n";
  for(i=-3; i < 3; i++)
    cout << ob1[i] << " ";
  cout << endl;

  for(i=-3; i < 3; i++)
    cout << ob2[i] << " ";
  cout << "\n\n";

  if(ob1 == ob2) cout << "ob1 == ob2\n";
  if(ob1 != ob2) cout << "error\n";
  cout << endl;

  cout << "Assign ob1[-1] the value 99\n";
  ob1[-1] = 99;
  cout << "Contents of ob1 are now:\n";
  for(i=-3; i < 3; i++)
    cout << ob1[i] << " ";
  cout << endl;

  if(ob1 == ob2) cout << "error\n";
  if(ob1 != ob2) cout << "ob1 != ob2\n";
  cout << endl;

  if(ob1 < ob2) cout << "ob1 < ob2\n";
```

```
  if(ob1 <= ob2) cout << "ob1 <= ob2\n";
  if(ob1 > ob2) cout << "ob1 > ob2\n";
  if(ob1 >= ob2) cout << "ob1 >= ob2\n";

  if(ob2 < ob1) cout << "ob2 < ob1\n";
  if(ob2 <= ob1) cout << "ob2 <= ob1\n";
  if(ob2 > ob1) cout << "ob2 > ob1\n";
  if(ob2 >= ob1) cout << "ob2 >= ob1\n";
  cout << endl;

  // Compare objects of differing sizes.
  if(ob3 != ob1) cout << "ob3 != ob1\n";
  if(ob3 == ob1) cout << "ob3 == ob1\n";

  return 0;
}
```

其输出如下所示：

```
Contents of ob1 and ob2:
-3 -2 -1 0 1 2
-3 -2 -1 0 1 2

ob1 == ob2

Assign ob1[-1] the value 99
Contents of ob1 are now:
-3 -2 99 0 1 2
ob1 != ob2

ob1 > ob2
ob1 >= ob2
ob2 < ob1
ob2 <= ob1

ob3 != ob1
```

第三个程序将类对象存储在 RangeArray 中。这个程序还说明了当发生不同的操作时，调用构造函数和析构函数的时刻：

```
// Store class objects in a RangeArray.
#include <iostream>
#include "ra.h"

using namespace std;

class test {
public:
  int a;

  test() { cout << "Constructing\n"; a=0; }
```

```
  test(const test &o) {
    cout << "Copy Constructor\n";
    a = o.a;
  }

  ~test() { cout << "Destructing\n"; }
};

int main()
{
  RangeArray<test> t(-3, 1, test());
  int i;

  cout << "Original contents of t:\n";
  for(i=-3; i < 2; i++) cout << t[i].a << " ";
  cout << endl;

  // Give t some new values.
  for(i=-3; i < 2; i++) t[i].a = i;

  cout << "New contents of t:\n";
  for(i=-3; i < 2; i++) cout << t[i].a << " ";
  cout << endl;

  // Copy to new container.
  RangeArray<test> t2(-7, 3, test());
  copy(t.begin(), t.end(), &t2[-2]);

  cout << "Contents of t2:\n";
  for(i=-7; i < 4; i++) cout << t2[i].a << " ";
  cout << endl;

  RangeArray<test> t3(t.begin()+1, t.end()-1);
  cout << "Contents of t3:\n";
  for(i=t3.getlowerbound(); i < t3.getupperbound(); i++)
    cout << t3[i].a << " ";
  cout << endl;

  t.clear();

  cout << "Size after clear(): " << t.size() << endl;

  // Assign container objects.
  t = t3;
  cout << "Contents of t:\n";
  for(i=t.getlowerbound(); i<t.getupperbound(); i++)
  cout << t[i].a << " ";
    cout << endl;

  return 0;
}
```

这个程序的输出如下所示：

```
Constructing
Copy Constructor
Copy Constructor
Copy Constructor
Copy Constructor
Copy Constructor
Destructing
Original contents of t:
0 0 0 0 0
New contents of t:
-3 -2 -1 0 1
Constructing
Copy Constructor
Copy Constructor
Copy Constructor
Copy Constructor
Copy Constructor
Copy Constructor
Copy Constructor
Copy Constructor
Copy Constructor
Copy Constructor
Copy Constructor
Destructing
Contents of t2:
0 0 0 0 0 -3 -2 -1 0 1 0
Copy Constructor
Copy Constructor
Copy Constructor
Contents of t3:
-2 -1 0
Destructing
Destructing
Destructing
Destructing
Destructing
Size after clear(): 0
Contents of t:
-2 -1 0
Destructing
Destructing
Destructing
Destructing
Destructing
Destructing
Destructing
Destructing
Destructing
Destructing
Destructing
```

```
Destructing
Destructing
Destructing
Destructing
Destructing
Destructing
```

8.4.5 尝试完成以下任务

您可能想要完善并试验 RangeArray 类。例如，您可以加入反向迭代器以及 rbegin()和 rend()函数。在此还有一些其他的想法，您可以试着优化这个容器。如前所述，RangeArray 中的代码是为了操作的透明而不是速度设计的，因此可以轻易地提升速度。例如，当创建对象时，试着分配比所需多一点点的内存，从而并不是所有的插入操作都强制重新分配。试着创建一个成员函数将 RangeArray 转换为标准的、基数为 0 的数组。试着加入采用标准数组和索引作为参数的构造函数，让这个构造函数将数组转换为 RangeArray，将这个索引作为 0 的位置。最后，试着使用一个 vector 而不是标准的数组来保存 RangeArray 的元素。观察它是否简化了实现(您将会得到一个惊喜)。