

C++编程(15)

唐晓晟

北京邮电大学电信工程学院

第17章 标准容器

- 标准容器
- 序列
- 序列适配器
- 关联容器
- 拟容器
- 定义新容器
- 忠告

17.1 标准容器

- 标准库定义了两类容器：序列和关联容器
- 序列都很象**vector**，除了专门指明的情况之外，有关**vector**所论及的成员类型和函数都可以对任何其他容器使用，并产生同样效果
- 关联容器提供了基于关键码访问元素的功能

17.1.1 操作综述

□ 本节列出标准容器的公共的或者几乎公共的成员

Member Types (§16.3.1)	
<i>value_type</i>	Type of element.
<i>allocator_type</i>	Type of memory manager.
<i>size_type</i>	Type of subscripts, element counts, etc.
<i>difference_type</i>	Type of difference between iterators.
<i>iterator</i>	Behaves like <i>value_type*</i> .
<i>const_iterator</i>	Behaves like <i>const value_type*</i> .
<i>reverse_iterator</i>	View container in reverse order; like <i>value_type*</i> .
<i>const_reverse_iterator</i>	View container in reverse order; like <i>const value_type*</i> .
<i>reference</i>	Behaves like <i>value_type&</i> .
<i>const_reference</i>	Behaves like <i>const value_type&</i> .
<i>key_type</i>	Type of key (for associative containers only).
<i>mapped_type</i>	Type of <i>mapped_value</i> (for associative containers only).
<i>key_compare</i>	Type of comparison criterion (for associative containers only).

操作综述

- 容器可以看成是按照该容器的**iterator**所定义的顺序形成的序列，或者按反向顺序形成的序列，对于关联容器，这个顺序则基于容器的比较准则(默认为<)

Iterators (§16.3.2)	
<i>begin()</i>	Points to first element.
<i>end()</i>	Points to one-past-last element.
<i>rbegin()</i>	Points to first element of reverse sequence.
<i>rend()</i>	Points to one-past-last element of reverse sequence.

Element Access (§16.3.3)	
<i>front()</i>	First element.
<i>back()</i>	Last element.
<i>[]</i>	Subscripting, unchecked access (not for list).
<i>at()</i>	Subscripting, checked access (not for list).

操作综述

- 向量和双端队列提供了对它们的元素序列中后端元素的有效操作，表和双端队列还对它们的开始元素提供了等价的操作

Stack and Queue Operations (§16.3.5, §17.2.2.2)	
<i>push_back()</i>	Add to end.
<i>pop_back()</i>	Remove last element.
<i>push_front()</i>	Add new first element (for list and deque only).
<i>pop_front()</i>	Remove first element (for list and deque only).

- 各种容器提供如下的表操作

List Operations (§16.3.6)	
<i>insert(p,x)</i>	Add <i>x</i> before <i>p</i> .
<i>insert(p,n,x)</i>	Add <i>n</i> copies of <i>x</i> before <i>p</i> .
<i>insert(p,first,last)</i>	Add elements from [<i>first</i> : <i>last</i> [before <i>p</i> .
<i>erase(p)</i>	Remove element at <i>p</i> .
<i>erase(first,last)</i>	Erase [<i>first</i> : <i>last</i>].
<i>clear()</i>	Erase all elements.

操作综述

- 所有容器都提供了与元素个数有关的各种操作和若干其他操作

Other Operations (§16.3.8, §16.3.9, §16.3.10)	
<i>size()</i>	Number of elements.
<i>empty()</i>	Is the container empty?
<i>max_size()</i>	Size of the largest possible container.
<i>capacity()</i>	Space allocated for <i>vector</i> (for vector only).
<i>reserve()</i>	Reserve space for future expansion (for vector only).
<i>resize()</i>	Change size of container (for vector, list, and deque only).
<i>swap()</i>	Swap elements of two containers.
<i>get_allocator()</i>	Get a copy of the container's allocator.
<i>==</i>	Is the content of two containers the same?
<i>!=</i>	Is the content of two containers different?
<i><</i>	Is one container lexicographically before another?

操作综述

□ 容器提供了各种构造函数和赋值操作

Constructors, etc. (§16.3.4)	
<i>container()</i>	Empty container.
<i>container(n)</i>	<i>n</i> elements default value (not for associative containers).
<i>container(n,x)</i>	<i>n</i> copies of <i>x</i> (not for associative containers).
<i>container(first,last)</i>	Initial elements from [<i>first</i> : <i>last</i> [.
<i>container(x)</i>	Copy constructor; initial elements from container <i>x</i> .
<i>~container()</i>	Destroy the container and all of its elements.

Assignments (§16.3.4)	
<i>operator=(x)</i>	Copy assignment; elements from container <i>x</i> .
<i>assign(n,x)</i>	Assign <i>n</i> copies of <i>x</i> (not for associative containers).
<i>assign(first,last)</i>	Assign from [<i>first</i> : <i>last</i> [.

操作综述

□ 关联容器提供如下基于关键码的检索操作

Associative Operations (§17.4.1)	
<i>operator[]</i> (<i>k</i>)	Access the element with key <i>k</i> (for containers with unique keys).
<i>find</i> (<i>k</i>)	Find the element with key <i>k</i> .
<i>lower_bound</i> (<i>k</i>)	Find the first element with key <i>k</i> .
<i>upper_bound</i> (<i>k</i>)	Find the first element with key greater than <i>k</i> .
<i>equal_range</i> (<i>k</i>)	Find the <i>lower_bound</i> and <i>upper_bound</i> of elements with key <i>k</i> .
<i>key_comp</i> ()	Copy of the key comparison object.
<i>value_comp</i> ()	Copy of the <i>mapped_value</i> comparison object.

17.1.2 容器综述

Standard Container Operations					
	[]	List Operations	Front Operations	Back (Stack) Operations	Iterators
	§16.3.3 §17.4.1.3	§16.3.6 §20.3.9	§17.2.2.2 §20.3.9	§16.3.5 §20.3.12	§19.2.1
<i>vector</i>	const	O(n)+		const+	Ran
<i>list</i>		const	const	const	Bi
<i>deque</i>	const	O(n)	const	const	Ran
<i>stack</i>				const+	
<i>queue</i>			const	const+	
<i>priority_queue</i>			O(log(n))	O(log(n))	
<i>map</i>	O(log(n))	O(log(n))+			Bi
<i>multimap</i>		O(log(n))+			Bi
<i>set</i>		O(log(n))+			Bi
<i>multiset</i>		O(log(n))+			Bi
<i>string</i>	const	O(n)+	O(n)+	const+	Ran
<i>array</i>	const				Ran
<i>valarray</i>	const				Ran
<i>bitset</i>	const				

容器综述

- ❑ 上页表中，**Iterators**列的**Ran**表示随机访问、**Bi**表示双向迭代
- ❑ 表中其他项目表示的都是操作的效率，**const**表示该操作所用的时间不依赖于容器中元素的数目(相当于 $O(1)$)，后缀**+**表明偶尔会出现显著的额外时间开销，注意基本操作中没有非常糟的操作，比如说 $O(n^2)$
- ❑ 有关复杂性和代价的度量值都是上界

17.1.3 表示

- 标准并没有为各种标准容器预先设定任何特定的实现方式。相反，标准只是描述了各种容器的界面，并提出了一些复杂性要求
- 实现者将选择适当的，一般也是经过最聪明地优化过的实现方式，以满足这些普遍要求
- 例如：**vector**(数组)、**list**(链表)、**map**(平衡树)、**string**(11章所描述的方式或者数组)

17.1.4 对元素的要求

- ❑ 容器里的元素是被插入对象的副本
- ❑ 一个对象要想成为容器的元素，它就必须属于某个允许容器的实现对他进行复制的类型
- ❑ 容器可以利用复制构造函数或者赋值做元素的赋值工作，在两种情况下，都要求复制结果是一个等价的对象

17.1.4.1 比较

- ❑ 关联容器要求其元素是有序的，许多可以应用于容器的操作也是这样(例如,**sort()**)
- ❑ 按照默认方式，这个序由<运算符定义，如果<不合适，程序员必须另外提供合适的比较操作
- ❑ 排序准则必须定义一种严格的弱顺序(**strict weak ordering**)，即小于和等于操作必须具有传递性

比较示例

```
template<class Ran> void sort(Ran first, Ran last);
template<class Ran, class Cmp> void sort(Ran first,
                                         Ran last, Cmp cmp);

class Nocase{
public: bool operator()(const string&, const string&) const;
};

bool Nocase::operator()(const string& x, const string& y) const{
    string::const_iterator p = x.begin();
    string::const_iterator q = y.begin();
    while(p!=x.end() && q != y.end() && toupper(*p)==toupper(*q)){
        ++p; ++q;
    }
    if(p == x.end() ) return q!=y.end();
    if(q == y.end() ) return false;
    return toupper(*p) < toupper(*q);
}
```

比较示例

- ❑ 使用上面的比较准则调用`sort()`，例如，给定fruit: apple pear Apple Pear lemon
- ❑ 用`sort(fruit.begin(), fruit.end(), Nocase())`将产生
fruit: Apple apple lemon Pear pear
- ❑ 而简单的用`sort(fruit.begin(), fruit.end())`将给出
fruit: Apple Pear apple lemon pear

17.1.4.2 其他关系运算符

- ❑ 按默认规定，容器和算法在需要小于比较时都采用<，如果默认方式不何时，程序员必须另行提供一个比较准则
- ❑ 标准库在名字空间**std::rel_ops**里定义了各种比较运算符，并通过**<utility>**给出

```
template<class T>bool rel_ops::operator!=(const T& x, const T& y)
    {return !(x==y);}
template<class T>bool rel_ops::operator>(const T& x, const T& y)
    {return y<x;}
template<class T>bool rel_ops::operator<=(const T& x, const T& y)
    {return !(y<x);}
template<class T>bool rel_ops::operator>=(const T& x, const T& y)
    {return !(x<y);} // 注意以上运算符都是定义在<运算符基础之上的
```

17.2 序列

- ❑ 序列遵循vector所描述的模式
- ❑ 标准库提供的基本序列包括: `vector` `list` `deque`
- ❑ 从它们出发, 通过定义适当的界面, 生成了 `stack` `queue` `priority_queue`
- ❑ 这几个序列被称为容器适配器(container adapters)、序列适配器(sequence adapters), 或者简称适配器

17.2.1 向量—vector

- ❑ 前面已经描述了vector的细节
- ❑ 只有vector提供了预留空间的功能
- ❑ 按默认规定，用[]的下标操作不做范围检查，如果需要检查，请用at()
- ❑ vector提供随机访问迭代器

17.2.2 表—list

- ❑ **list**是一种最合适于做元素插入和删除的序列
- ❑ 由于下标访问操作太慢(与**vector**相比), **list**没有提供下标操作, 也因为这个原因, **list**提供的是双向迭代器
- ❑ **list**给出了**vector**所提供的几乎所有成员类型和操作, 例外的就是下标、**capacity()**、和**reserve()**

17.2.2.1 粘接、排序和归并

- **list**提供了若干特别适合于对表进行处理的操作，这些操作都是稳定的

```
template <class T, class A = allocator<T> > class list {
public:
    // 表的特殊操作
    void splice(iterator pos, list& x);
    // 将x的所有元素移到本表的pos之前，且不作复制
    void splice(iterator pos, list& x, iterator p);
    // 将x中的*p移到本表的pos之前，且不作复制
    void splice(iterator pos, list& x, iterator first, iterator last);
    void merge(list&); // 归并排序的表
    template <class Cmp> void merge(list&,Cmp);
    void sort();
    template <class Cmp> void sort(Cmp);
};
```

粘接、排序和归并

例如：给出

fruit: apple pear

citrus: orange grapefruit lemon

我们可以

```
list<string>::iterator p = find_if(fruit.begin(), fruit.end(), initial('p'));  
fruit.splice(p, citrus, citrus.begin());
```

将orange从citrus粘接入fruit，这样做的效果是从citrus删除了第一个元素，并将它放到fruit里的第一个名字以p开始的元素之前

注意，这里的迭代器参数必须是合法的迭代器，确实指向应该指向的那个list

```
list<string>::iterator p = find_if(fruit.begin(), fruit.end(), initial('p'));  
fruit.splice(p, citrus, citrus.begin()); // ok  
fruit.splice(p, citrus, fruit.begin()); //错误: fruit.begin()并不指向citrus  
citrus.splice(p, fruit, fruit.begin()); //错误: p不是指向citrus
```

粘接、排序和归并

- ❑ **merge**组合起两个排好序的表，方式是将一个**list**的元素都取出来，将它们都放入另外一个表，且维持正确的顺序
- ❑ 如果一个表未排序，**merge()**仍然能够产生一个表，其中包含两个表的元素的并集，只是顺序不作保证
 - f1: apple quince pear
 - f2: lemon grapefruit orange lime
 - 可以按：
f1.sort();
f2.sort();
f1.merge(f2); // merge也不复制元素
 - 其结果是：
f1: apple grapefruit lemon lime orange pear quince
f2: <empty>

17.2.2.2 前端操作

- **list**也提供一些针对第一个元素的操作，与每个容器都提供的针对最后元素的操作相对应
- 对**list**而言，前端操作与后端操作一样方便而高效，如果可以选择的话，最好是用后端操作，这样写出来的代码也对**vector**适用

```
template <class T, class A = allocator<T> > class list {  
public:  
    // 元素访问  
    reference front(); // 引用第一个元素  
    const_reference front() const;  
    void push_front(const T&); // 加入新的第一个元素  
    void pop_front(); // 删除第一个元素  
};
```


17.2.2.3 其他操作

- 对**list**的插入和删除操作效率特别高，当这类操作非常频繁时，会使得人们倾向于使用**list**，反过来，又使直接支持某些删除元素的通用方法变得很有价值

```
template <class T, class A = allocator<T> > class list {  
public:  
    void remove(const T& val); // 删除所有值为val的元素  
    template <class Pred> void remove_if(Pred p);  
    void unique(); // 根据==删除连续的重复元素  
    template <class BinPred> void unique(BinPred b);  
    // 根据b删除重复元素  
    void reverse(); // reverse order of elements  
};
```

17.2.3 双端队列—deque

- ❑ deque(其发音起伏就像check)就是一种双端的队列，也就是说，deque是一个优化了的序列，对其两端的操作效率类似于list，而其下标操作具有接近vector的效率
- ❑ 注意：在deque的中间插入和删除元素具有与vector一样的低效率，而不是类似list的效率

17.3 序列适配器

- ❑ **vector**、**list**和**deque**序列不可能互为实现的基础而同时又不损失效率，另一方面，**stack**和**queue**则都可以在这三种基本序列的基础上优雅而高效地实现
- ❑ 因此，**stack**和**queue**就没有定义为独立的容器，而是作为基本容器的适配器(**adapter**)
- ❑ 容器适配器所提供的是原来容器的一个受限的界面(特别是适配器不提供迭代器，提供它们的意图就是为了只经由它们的专用界面使用)

17.3.1 堆栈—stack

□ stack容器在<stack>中定义

```
template <class T, class C = deque<T> > class std::stack{
protected: C c;
public:
    typedef typename C::value_type value_type;
    typedef typename C::size_type size_type;
    typedef C container_type;
    explicit stack(const C& a =C()) : c(a) { }
    bool empty() const { return c.empty(); }
    size_type size() const { return c.size(); }
    value_type& top() { return c.back(); }
    const value_type& top() const { return c.back(); }
    void push(const value_type& x) { c.push_back(x); }
    void pop() { c.pop_back(); }
};
```

堆栈—stack

- ❑ **stack**就是某容器的一个简单界面，容器的类型作为模板参数传递给**stack**。**stack**所做的全部事情就是从它的容器的界面中删除所有非**stack**操作，并将**back()**、**push_back()**和**pop_back()**改为人们所习惯的名字**top()**、**push()**和**pop()**
- ❑ **top()**函数只读取、**pop()**操作只删除栈顶元素
- ❑ 按默认规定，**stack**用一个**deque**来保存自己的元素，但也可以采用任何提供了**back()**、**push_back()**和**pop_back()**的序列

`stack<char> s1; // 用deque<char>保存char类型的元素`

`stack<int, vector<int>> s2; // 用vector<int>保存int类型的元素`

17.3.2 队列—queue

- queue在<queue>里定义，它也是一个容器的界面，该容器应该允许在back()处插入新元素，且能从front()提取出来

```
template <class T, class C = deque<T> > class std::queue {  
protected: C c;  
public:  
    typedef typename C::value_type value_type;  
    typedef typename C::size_type size_type;  
    typedef C container_type;  
    explicit queue(const C& a =C()) : c(a) { }  
    bool empty() const { return c.empty(); }  
    size_type size() const { return c.size(); }  
    value_type& front() { return c.front(); }  
    const value_type& front() const { return c.front(); }  
    value_type& back() { return c.back(); }  
    const value_type& back() const { return c.back(); }  
    void push(const value_type& x) { c.push_back(x); }  
    void pop() { c.pop_front(); } };
```

队列—queue

- ❑ 按照默认规定，queue用deque保存自己的元素，但是任何提供了front()、back()、push_back()和pop_front()的序列都可以用(vector没有提供pop_front())，所以不能作为queue的基础容器)
- ❑ 对于queue，也是使用push()添加元素，使用pop()删除元素

17.3.3 优先队列—priority_queue

```
template <class T, class C = vector<T>,  
    class Cmp = less<typename C::value_type>> class std::priority_queue{  
protected: C c; Cmp cmp;  
public:  
    typedef typename C::value_type value_type;  
    typedef typename C::size_type size_type;  
    typedef C container_type;  
    explicit priority_queue(const Cmp& a1 =Cmp() , const C& a2 =C())  
        : c(a2) , cmp(a1) { }  
    template <class In>  
        priority_queue(In first, In last, const Cmp& =Cmp() , const C& =C());  
    bool empty() const { return c.empty(); }  
    size_type size() const { return c.size(); }  
    const value_type& top() const { return c.front(); }  
    void push(const value_type&);  
    void pop();  
};
```


优先队列—priority_queue

- ❑ 按默认规定，priority_queue简单地用<运算符做比较，top()返回最大的元素
- ❑ 实现priority_queue的一种有用方式是采用一个树结构保存元素的相对位置，这能给出 $O(\log(n))$ 代价的push()和pop()操作
- ❑ 按照默认规定，priority_queue用一个vector保存它的元素，对任何能提供front()、push_back()和pop_back()，并能使用随机访问迭代器的序列都可以用
- ❑ priority_queue最可能是用一个heap实现

17.4 关联容器

- ❑ 关联容器是用户定义类型中最常见的也最有用的一种
- ❑ 关联数组也常被称为映射(**map**), 有时也被称为字典(**dictionary**), 其中保存的是值的对
- ❑ 给定一个称为关键码的值, 我们就能访问一个称为映射值的值
- ❑ 可以将关联数组想象为一个下标不必是整数的数组
- ❑ **map**是传统的关联数组, **multiset**允许元素中出现重复关键码, **set**和**multiset**可以看成是退化的关联数组, 其中没有与关键码相关联的值

```
template<class K, class V> class Assoc {  
public:  
    V& operator[](const K&); // 返回对应于K的V的引用  
};
```

17.4.1 映射—map

- 一个map就是一个(关键码—值)对偶的序列
- map提供基于关键码的快速提取操作，关键码具有唯一性，map提供双向迭代器
- map要求其关键码类型提供一个小于操作，以保持自己元素的有序性
- 对于没有明显顺序的元素，或者不必保持容器有序的情况，可以考虑用hash_map

17.4.1.1 类型

```
template <class Key, class T, class Cmp = less<Key>,  
    class A = allocator< pair<const Key,T> > > class std::map  
{  
public:  
    typedef Key key_type;    // 关键码类  
    typedef T mapped_type;  // 映射值类  
    typedef pair<const Key, T> value_type; // 注意: 是pair  
    typedef Cmp key_compare;  
    typedef A allocator_type;  
    typedef typename A::reference reference;  
    typedef typename A::const_reference const_reference;  
    typedef implementation_defined1 iterator;  
    typedef implementation_defined2 const_iterator;  
    typedef typename A::size_type size_type;  
    typedef typename A::difference_type difference_type;  
    typedef std::reverse_iterator<iterator> reverse_iterator;  
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;  
};
```

17.4.1.2 迭代器的对偶

- map提供了一组在以pair<const Key, mapped_type>为元素类型的迭代器

```
template <class Key, class T, class Cmp = less<Key>,
          class A = allocator< pair<const Key,T> > > class map
{
public:
    // 迭代器
    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;
    reverse_iterator rbegin();
    const_reverse_iterator rbegin() const;
    reverse_iterator rend();
    const_reverse_iterator rend() const;
    // ...
};
```

迭代器的对偶

我们可以用如下方式打印出一个电话簿的各项内容

```
void f(map<string,number>& phone_book)
{
    typedef map<string,number>::const_iterator CI;
    for (CI p = phone_book.begin(); p!=phone_book.end(); ++p)
        cout << p-> first << '\t' << p-> second << '\n';
}
```

```
template <class T1, class T2> struct std::pair {
    typedef T1 first_type;
    typedef T2 second_type;
    T1 first;
    T2 second; // 下面为构造函数
    pair() :first(T1()), second(T2()) { }
    pair(const T1& x, const T2& y) :first(x), second(y) { }
    template<class U, class V> pair(const pair<U,V>& p) :first(p.first),
        second(p.second) { }
}; // 对任何pair, 总用first和second索引第一个和第二个元素(关键码和映射值)
```

17.4.1.3 下标

- **map**的特征性操作就是采用下标运算符提供的关联查找，当没有找到关键码时，**map**的下标操作将加进一个默认元素，如果仅仅希望进行查找操作，可以使用**find()**

```
template <class Key, class T, class Cmp = less<Key>,  
         class A = allocator< pair<const Key,T> > > class map  
{  
    public:  
    mapped_type& operator[](const key_type& k); // 用关键码k访问元素  
};  
void f() {  
    map<string,int>m; // map开始为空  
    int x =m["Henry"]; // 创建新项Henry，初始化为0，返回0  
    m["Harry"] = 7; // 创建新项Harry，其值为7  
    int y =m["Henry"] ; // 返回Henry的值  
    m["Harry"] = 9; // Harry关键码对应的值改为9  
}
```

17.4.1.4 构造函数

- ❑ 复制一个容器隐含着为它的所有元素分配空间，并完成每个元素的复制工作，代价可能非常高

```
template <class Key, class T, class Cmp =less<Key>,
        class A =allocator<pair<const Key,T> > > class map
{
public:
    // 构造、复制、析构等函数
    explicit map(const Cmp& =Cmp(), const A& =A());
    template <class In> map(In first, In last, const Cmp& =Cmp(),
        const A& =A());
    map(const map&);
    ~map();
    map& operator=(const map&);
};
```


17.4.1.5 比较

- ❑ 为了能在`map`中找到对应于给定关键码的元素，`map`必须能够做关键码比较，此外，迭代器也会按照递增的方式遍历`map`，因此，插入元素时通常也需要元素比较
- ❑ 按默认规定，关键码比较采用`<`，`map`的`value_type`是(关键码、值)对偶，因此提供了`value_comp()`来进行比较

比较

```
template <class Key, class T, class Cmp = less<Key>,  
    class A = allocator< pair<const Key,T> > > class map  
{  
public:  
    typedef Cmp key_compare;  
    class value_compare : public  
        binary_function<value_type,value_type,bool>{  
friend class map;  
protected:  
    Cmp cmp;  
    value_compare(Cmp c) : cmp(c) {}  
public:  
    bool operator()(const T& x, const T& y) const  
        { return cmp(x.first, y.first) ; }  
};  
key_compare key_comp() const;  
value_compare value_comp() const;  
};
```

比较

```
map<string,int> m1;  
map<string,int,Nocase> m2; // 描述比较类型 (§ 17.1.4.1)  
map<string,int,String_cmp> m3; // 明确比较类型 (§ 17.1.4.1)  
map<string,int> m4(String_cmp(literary)); // 传递比较对象
```

有了key_comp()和value_comp()成员函数，就可以要求对一个map的关键码和值做各种比较

常见做法是将另外某个容器或者算法的比较准则传递给map

```
void f(map<string,int>&m)  
{  
    map<string,int>mm; // 按默认方式使用<做比较  
    map<string,int> mmm(m.key_comp()); // 使用m的方式比较  
    // ...  
}
```

17.4.1.6 映射操作

```
template <class Key, class T, class Cmp = less<Key>,
class A = allocator< pair<const Key,T> > > class map {
public:
    // 映射操作
    iterator find(const key_type& k); // 查找关键码为k的元素
    const_iterator find(const key_type& k) const;
    size_type count(const key_type& k) const; // 关键码为k的元素个数
    iterator lower_bound(const key_type& k); // 查找第一个关键码为k的元素
    const_iterator lower_bound(const key_type& k) const;
    iterator upper_bound(const key_type& k); // 找第一个关键码大于k的元素
    const_iterator upper_bound(const key_type& k) const;
    pair<iterator,iterator> equal_range(const key_type& k);
    pair<const_iterator,const_iterator> equal_range(const key_type& k)
    const;
    // ...
};
```

映射操作

```
void f(multimap<string,int>&m)
{
    multimap<string,int>::iterator lb =m.lower_bound("Gold");
    multimap<string,int>::iterator ub =m.upper_bound("Gold");
    for (multimap<string,int>::iterator p = lb; p!=ub; ++p) {
        // ...
    }
    // 使用两个函数找到Gold出现的上界和下届
}
void f(multimap<string,int>&m)
{
    typedef multimap<string,int>::iterator MI;
    pair<MI,MI> g =m.equal_range("Gold");
    for (MI p = g.first; p!=g.second; ++p) {
        // ...
    }
} // 使用range_equal()一下子算出两个结果
```

17.4.1.7 表操作

```
template <class Key, class T, class Cmp = less<Key>,
         class A = allocator< pair<const Key,T> > > class map
{
    public:// 表操作
    pair<iterator, bool> insert(const value_type& val);
    // 插入(关键码、值)对
    iterator insert(iterator pos, const value_type& val);
    // pos只是一个提示, 暗示insert从pos开始查找val的关键码
    template <class In> void insert(In first, In last); // 从序列中插入
    void erase(iterator pos); // 删除被指向的元素
    size_type erase(const key_type& k); // 删除关键码为k的所有元素, 返回个数
    void erase(iterator first, iterator last); // 删除区间
    void clear(); // 删除所有元素
};
// 将一个值放入关联容器的最方便方式就是使用下标操作直接赋值
// phone_book["Order department"] = 8226339;
// m[k]的结果等价于(*(m.insert(makepair(k,V()))).first)).second
// []操作总需要V(), 因此, 若map的值类型没有默认值, 无法使用下标操作
```

表操作

```
void f(map<string,int>&m)
{
    pair<string,int> p99("Paul",99);
    pair<map<string,int>::iterator,bool> p =m.insert(p99);
    if (p.second) {
        // “Paul”被插入
    }
    else {
        // “Paul”已经存在
    }
    map<string,int>::iterator i = p.first; // 指向 m["Paul"]
    // ...
}
// m.insert(val)的返回值是pair<iterator,bool>, 如果val被实际插入, bool值为true, iterator引用的是m中一个元素, 保存着val的关键码val.first
```

17.4.1.8 其他函数

- ❑ `map`提供了一些处理元素个数的常用函数
- ❑ 此外，`map`还提供了`==`，`!=`，`<`，`>`，`<=`，`>=`和`swap()`，它们都作为非成员函数

```
template <class Key, class T, class Cmp = less<Key>,
class A = allocator< pair<const Key,T> > > class map
{
public:
    // capacity:
    size_type size() const; // number of elements
    size_type max_size() const; // size of largest possible map
    bool empty() const { return size()==0; }
    void swap(map&);
};
```


17.4.2 多重映射—multimap

- ❑ multimap很象map，但是允许重复的键码
- ❑ 注意到insert函数总是真正的插入，所以返回值为iterator，而不是pair

```
template <class Key, class T, class Cmp = less<Key>,  
    class A = allocator< pair<const Key,T> > > class std::multimap  
{  
public:  
    // 类似map，除了：  
    iterator insert(const value_type&); // 返回iterator，不是pair  
    // 无[]运算符  
};
```

17.4.3 集合—set

- 一个**set**也可以被看作是一个**map**，其中的值是无关紧要的，所以只保存了关键码，这样做只引起了用户界面的少许改动
- 将**value_type**定义为**Key**类型，使得**map**和**set**的代码在大部分情况下完全一样

```
template <class Key, class Cmp = less<Key>,  
         class A = allocator<Key> > class std::set  
{  
public:  
    // 类似map，除了：  
    typedef Key value_type; // 关键码本身就是值  
    typedef Cmp value_compare;  
    // 无[]运算符  
};
```

17.4.4 多重集合—multiset

□ multiset是一种允许重复关键码的set

```
template <class Key, class T, class Cmp = less<Key>,
          class A = allocator<Key> > class std::multiset
{
public:
    // 类似set, 除了:
    iterator insert(const value_type&); // 返回iterator, 不是pair
};
```

17.5 拟容器—Almost Containers

- 内部数组、`string`、`valarray`和`bitset`里也保存元素，因此，很多情况下也可以将它们看作容器
- 它们中的每个都缺乏标准容器的这些或那些特性，所以，这些“拟容器”不像开发完整的容器那样具有完全互换性

17.5.1 串—string

- ❑ `basic_string`提供下标操作、随机访问迭代器以及容器所能提供的几乎所有记法上的方便之处
- ❑ 不像容器那样支持广泛的元素类型选择
- ❑ 特别为作为字符串的使用做了优化

17.5.2 值向量—`valarray`

- ❑ 为数值计算而优化了的向量
- ❑ 提供了许多有用的数值操作
- ❑ 只提供了标准容器中的`size()`和下标操作
- ❑ 到`valarray`中的元素指针是一种随机访问迭代器

17.5.3 位集合—bitset

- ❑ C++可以通过整数上的按位运算，有效地支持小的标志集合的概念
- ❑ 类**bitset<N>**推广了这个概念，并通过提供了在N个二进制位的集合(下标从0到N-1)上的各种操作提供进一步的方便，这里的N需要在编译时已知
- ❑ 对于无法放进**long int**的二进制位的集合，用一个**bitset**比直接用一些整数方便的多，对于更小的集合，这里就有一个效率上的权衡问题
- ❑ 如果希望给二进制位命名，而不是给它们编号，请考虑使用**set**、枚举、或者位域(C.8.1)

位集合—bitset

- ❑ **bitset<N>**是N个二进制位的数组，与**vector<bool>**的不同之处在于它的大小是固定的；与**set**的不同点在于它通过下标索引其中的二进制位，而不是通过关键码关联；与**vector<bool>**和**set**都不同的地方在于它提供了同时对许多二进制位进行处理的操作
- ❑ 通过内部指针不能直接对单个的位寻址，因此，**bitset**提供了一种位引用类型，这是一种一般性的技术，用于对由于某些原因而使内部指针不能使用的对象寻址
- ❑ **bitset**模板在**std**名字空间定义，通过**<bitset>**给出

17.7 忠告

- [1] 如果需要用容器，首先考虑用 **vector**
- [2] 了解你经常使用的每个操作的代价(复杂性，大O度量)
- [3] 容器的界面、实现和表示是不同的概念，不要混淆
- [4] 你可以依据多种不同准则去排序和搜索
- [5] 不要用C风格的字符串作为关键码，除非你提供了一种适当的比较准则
- [6] 你可以定义这样的比较准则，使等价的但是不相同的关键码值映射到同一个关键码

忠告

- ❑ [7] 在插入和删除元素时，最好是使用序列末端的操作 (***back***操作)
- ❑ [8] 当你需要在容器的前端或中间做许多插入和删除时，请用***list***
- ❑ [9] 当你主要通过关键码访问元素时，请用***map***或***multimap***
- ❑ [10] 尽量用最小的操作集合，以取得最大的灵活性
- ❑ [11] 如果要保持元素的顺序性，选用***map***而不是***hash_map***
- ❑ [12] 如果查找速度极其重要，选***hash_map***而不是***map***

忠告

- ❑ [13] 如果无法对元素定义小于操作时，选 ***hash_map*** 而不是 ***map***
- ❑ [14] 当你需要检查某个关键码是否在关联容器里的时候，用 ***find()***
- ❑ [15] 用 ***equal_range()*** 在关联容器里找出所有具有给定关键码的所有元素
- ❑ [16] 当具有同样关键码的多个值需要保持顺序时，用 ***multimap***
- ❑ [17] 当关键码本身就是你需要保存的值时，用 ***set*** 或 ***multiset***