6

STL 容器

STL Container

本章延续第 5 章的讨论，详细讲解 STL 容器。首先对所有容器共通的能力和操作进行巡礼，然后详细讲解每一个容器，包括内部数据结构、操作（operations）、性能，以及各种操作的应用。如果某些操作值得深述，我还会给出相应的实例。每个容器的讲解都以一个典型应用实例作为结束。本章还讨论一个有趣的问题：各种容器的使用时机。比较各种容器的能力、优点、缺点之后，你便会了解如何选择最符合需求的容器。最后，本章详细介绍了每一个容器的所有成员。这一部分可视为参考手册，你可以在其中找到容器接口的细节和容器操作的确切标记式（signature）。必要的时候我会列出交叉索引，帮助你了解相似或互补的算法。

C++ 标准程序库还提供了一些特殊的容器类别——所谓的“容器配接器”（container adapters，包括 stack, queue, priority queue），以及 bitsets 和 valarrays。这些容器都有一些特殊接口，并不满足 STL 容器的一般要求，所以本书把它们放在其它章节讲解[1]。容器配接器和 bitsets 安排在第 10 章，valarrays 安排在 12.2 节，p547。

[1] 从历史沿革来说，容器配接器是 STL 的一部分。然而，从概念角度观之，它们并不属于 STL framework，它们只不过是“使用”STL。

6.1 容器的共通能力和共通操作

6.1.1 容器的共通能力

本节讲述 STL 容器的共通能力。其中大部分都是必要条件，所有 STL 容器都必须满足那些条件。三个最核心的能力是：

1. 所有容器提供的都是“value 语意”而非“reference 语意”。容器进行元素的安插操作时，内部实施的是拷贝操作，置于容器内。因此 STL 容器的每一个元素都必须能够被拷贝。如果你打算存放的对象不具有 public copy 构造函数，或者你要的不是副本（例如你要的是被多个容器共同容纳的元素），那么容器元素就只能是指针（指向对象）。5.10.2 节, p135 对此有所描述。
2. 总体而言，所有元素形成一个次序（order）。也就是说，你可以依相同次序一次或多次遍历每个元素。每个容器都提供“可返回迭代器”的函数，运用那些迭代器你就可以遍历元素。这是 STL 算法赖以生存的关键接口。
3. 一般而言，各项操作并非绝对安全。调用者必须确保传给操作函数的参数符合需求。违反这些需求（例如使用非法索引）会导致未定义的行为。通常 STL 自己不会抛出异常。如果 STL 容器所调用的使用者自定操作（user-defined operations）抛出异常，会导致各不相同的行为。参见 5.11.2 节, p139。

6.1.2 容器的共通操作

以下操作为所有容器共有，它们均满足上述核心能力。表 6.1 列出这些操作。后续各小节分别探讨这些共通操作。

初始化（initialization）

每个容器类别都提供了一个 default 构造函数，一个 copy 构造函数和一个析构函数。你可以以某个已知区间的内容作为容器初值——是的，负责此项工作的构造函数专门用来从另一个容器或数组或标准输入装置（standard input）得到元素并构造出容器。这些构造函数都是 member templates（p11），所以如果提供了从“来源端”到“目标端”的元素型别自动转换，那么不光是容器型别可以不同，元素型别也可以不同[2]。下面是个实例：

[2] 如果系统本身不支持 member templates，那就只能接受相同型别。你可以换用 copy() 算法，参见 p188 范例。

表 6.1 容器类别（Container Classes）的共通操作函数

操作	效果
ContType c	产生一个未含任何元素的空容器
ContType c1(c2)	产生一个同型容器
ContType c(beg,end)	复制 [beg;end] 区间内的元素，作为容器初值
c.~ContType()	删除所有元素，释放内存
c.size()	返回容器中的元素数量
c.empty()	判断容器是否为空（相当于 size()==0，但可能更快）
c.max_size()	返回元素的最大可能数量
c1 == c2	判断是否 c1 等于 c2
c1 != c2	判断是否 c1 不等于 c2，相当于 !(c1 == c2)
c1 < c2	判断是否 c1 小于 c2
c1 > c2	判断是否 c1 大于 c2，相当于 c2 < c1
c1 <= c2	判断是否 c1 小于等于 c2，相当于 !(c2 < c1)
c1 >= c2	判断是否 c1 大于等于 c2，相当于 !(c1 < c2)
c1 = c2	将 c2 的所有元素赋值给 c1
c1.swap(c2)	交换 c1 和 c2 的数据
swap(c1,c2)	同上，是个全局函数
c.begin()	返回一个迭代器，指向第一元素
c.end()	返回一个迭代器，指向最后元素的下一位置
c.rbegin()	返回一个逆向迭代器，指向逆向遍历时的第一元素
c.rend()	返回一个逆向迭代器，指向逆向遍历时的最后元素的下一位置
c.insert(pos,elem)	将 elem 的一份副本安插于 pos 处。返回值和 pos 的意义并不相同
c.erase(beg,end)	移除 [beg;end] 区间内的所有元素。某些容器会返回未被移除的第一个接续元素
c.clear()	移除所有元素，令容器为空
c.get_allocator()	返回容器的内存模型（memory model）

- 以另一个容器的元素为初值，完成初始化操作：

```
std::list<int> l; // l is a linked list of ints
...
// copy all elements of the list as floats into a vector
std::vector<float> c(l.begin(),l.end());
```

- 以某个数组的元素为初值，完成初始化操作：

```
int array[] = { 2, 3, 17, 33, 45, 77 };
...
// copy all elements of the array into a set
std::set<int> c(array,array+sizeof(array)/sizeof(array[0]));
```

- 以标准输入装置完成初始化操作：

```
// read all integer elements of the deque from standard input
std::deque<int> c((std::istream_iterator<int>(std::cin)),
                  (std::istream_iterator<int>()));
```

注意，不要遗漏了涵括“初始化参数”的那对“多余的”括号，否则这个表达式的意义会迥然不同，肯定让你匪夷所思，你会得到一堆奇怪的警告或错误。以下是不写括号的情形：

```
std::deque<int> c(std::istream_iterator<int>(std::cin),
                  std::istream_iterator<int>());
```

这种情况下 c 被视为一个函数，返回值是 deque<int>，第一参数的型别是 istream_iterator<int>，参数名为 cin，第二参数无名称，型别是“一个函数，不接受任何参数，返回值型别为 istream_iterator<int>”。以上结构不论作为声明或表达式，语法上都正确。根据 C++ 规则它被视为声明。只要加上一对括号，便可使参数(std::istream_iterator<int>(std::cin))不再符合声明语法[3]，也就消除了歧义。

原则上还有一些操作，可支持从另一区间获取数据、赋值、插入元素。不过这些操作的确切接口在各容器中彼此不同，有不同的附加参数。

与大小相关的操作函数（Size Operations）

所有容器都提供了三个和大小相关的操作函数：

1. **size()**

   返回当前容器的元素数量。

2. **empty()**

   这是 size()==0 表达式的一个快捷形式。empty()的实作可能比 size()==0 的更有效率，所以你应该尽可能使用它。

3. **max_size()**

   返回容器所能容纳的最大元素数量。其值因实作版本的不同而异。例如 vector 通常保有一个内存区块的全部元素，所以在 PCs 上可能会有相关限定。

---

[3] 感谢 EDG 的 John H. Spicer 给予的说明。

max_size()通常返回索引型别的最大值。

比较（Comparisons）

包括常用的比较操作符==, !=, <, <=, >, >=。它们的定义依据以下三条规则：

1. 比较操作的两端（两个容器）必须属于同一型别。
2. 如果两个容器的所有元素依序相等，那么这两个容器相等。采用 operator== 检查元素是否相等。
3. 采用字典式（lexicographical）顺序比较原则来判断某个容器是否小于另一个容器。参见 p360。

比较两个不同型别的容器，必须使用“比较”算法，参见 9.5.4 节, p356。

赋值（Assignments）和 swap()

当你对着容器赋值元素时，源容器的所有元素被拷贝到目标容器内，后者原本的所有元素全被移除。所以，容器的赋值操作代价比较高昂。

如果两个容器型别相同，而且拷贝后源容器不再被使用，那么我们可以使用一个简单的优化方法：swap()。swap()的性能比上述优异得多，因为它只交换容器的内部数据。事实上它只交换某些内部指针（指向实际数据如元素、配置器、排序准则——如果有的话），所以时间复杂度是“常数”，不像实际赋值操作的复杂度为“线性”。

6.2 Vectors

vector 模塑出一个动态数组。因此，它本身是“将元素置于动态数组中加以管理”的一个抽象概念（图 6.1）。不过请注意，*C++ Standard* 并未要求必须以动态数组实作 vector，只是规定了相应条件和操作复杂度。

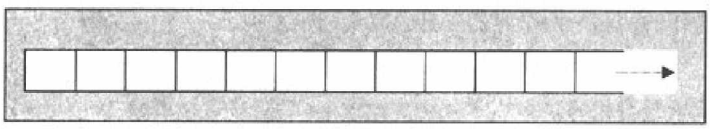

图 6.1 vector 的结构

使用 vector 之前，必须含入头文件 `<vector>`[4]

```
#include <vector>
```

其中，型别 `vector` 是一个定义于 namespace `std` 内的 template：

```
namespace std {
  template <class T,
            class Allocator = allocator<T> >
  class vector;
}
```

vector 的元素可以是任意型别 `T`，但必须具备 assignable 和 copyable 两个性质。第二个 template 参数可有可无，用来定义内存模型（memory model，参见 15 章）。缺省的内存模型是 C++ 标准程序库提供的 `allocator`[5]。

6.2.1 Vectors 的能力

vectors 将其元素复制到内部的 dynamic array 中。元素之间总是存在某种顺序，所以 vectors 是一种有序群集（ordered collection）。vector 支持随机存取，因此只要知道位置，你可以在常数时间内存取任何一个元素。vector 的迭代器是随机存取迭代器，所以对任何一个 STL 算法都可以奏效。

在末端附加或删除元素时，vector 的性能相当好。可是如果你在前端或中部安插或删除元素，性能就不怎么样了，因为操作点之后的每一个元素都必须移到另一个位置，而每一次移动都得调用 assignment（赋值）操作符。

---

[4] 早期的 STL 中，vectors 的定义头文件是 `<vector.h>`。

[5] 在不支持 default template parameters 的系统中，第二参数通常就没有了。

大小（Size）和容量（Capacity）

vector 优异性能的秘诀之一，就是配置比其所容纳的元素所需更多的内存。为了能够高效运用 vectors，你应该了解大小和容量之间的关系。

vectors 之中用于操作大小的函数有 `size()`,`empty()`,`max_size()`（6.1.2 节，p144）。另一个与大小有关的函数是 `capacity()`，它返回 vector 实际能够容纳的元素数量。如果超越这个数量，vector 就有必要重新配置内部存储器。

vector 的容量之所以很重要，有以下两个原因：

1. 一旦内存重新配置，和 vector 元素相关的所有 references、pointers、iterators 都会失效。
2. 内存重新配置很耗时间。

所以如果你的程序管理了和 vector 元素相关的 references、pointers、iterators，或如果执行速度对你而言至关重要，那么就必须考虑容量问题。

你可以使用 `reserve()`保留适当容量，避免一再重新配置内存。如此一来，只要保留的容量尚有余裕，就不必担心 references 失效。

```
std::vector<int> v;    // create an empty vector
v.reserve(80);         // reserve memory for 80 elements
```

另一种避免重新配置内存的方法是，初始化期间就向构造函数传递附加参数，构造出足够的空间。如果你的参数是个数值，它将成为 vector 的起始大小。

```
std::vector<T> v(5); // creates a vector and initializes it with five values
                     // (calls five times the default constructor of type T)
```

当然，要获得这种能力，此种元素型别必须提供一个 default 构造函数。请注意，如果型别很复杂，就算提供了 default 构造函数，初始化操作也很耗时。如果你这么做只不过是为了保留足够的内存，那倒不如使用 `reserve()`。

vectors 的容量，概念上和 strings 类似（参见 11.2.5 节，p485）。不过有一个大不同点：vector 不能使用 `reserve()`来缩减容量，这一点和 strings 不同。如果调用 `reserve()`所给的参数比当前 vector 的容量还小，不会引发任何反应。此外，如何达到时间和空间的最佳效率，系由实作版本决定。因此具体实作版本中，容量的增长幅度可能比你我料想的还大。事实上为了防止内存破碎，在许多实作方案中即使你不调用 `reserve()`，当你第一次安插元素时也会一口气配置整块内存（例如 2K）。如果你有一大堆 vectors，每个 vector 的实际元素却寥寥无几，那么浪费的内存会相当可观。

既然 vectors 的容量不会缩减，我们便可确定，即使删除元素，其 references、pointers、iterators 也会继续有效，继续指向动作发生前的位置。然而安插操作却可能使 references、pointers、iterators 失效（译注：因为安插可能导致 vector 重新配置空间）。

这里有一个间接缩减 vector 容量的小窍门。注意，两个 vectors 交换内容后，两者的容量也会互换，因此下面的例子虽然保留了元素，却缩减了容量：

```
template <class T>
void shrinkCapacity(std::vector<T>& v)
{
    std::vector<T> tmp(v); // copy elements into a new vector
    v.swap(tmp);           // swap internal vector data
}
```

你甚至可以利用下面的语句直接缩减容量[6]：

```
// shrink capacity of vector v for type T
std::vector<T>(v).swap(v);
```

不过请注意，swap()之后原先所有的 references、pointers、iterators 都换了指涉对象；它们仍然指向原本位置。换句话说，上述的 shrinkCapacity()会使所有 references、pointers、iterators 失效。

6.2.2 Vector 的操作函数

构造、拷贝和解构

表 6.2 列出 vectors 的所有构造函数和析构函数。你可以在构造时提供元素，也可以不。如果只指定大小，系统便会调用元素的 default 构造函数一一制造新元素。记住，即使对基本型别如 int，显式调用 default 构造函数进行初始化，也是一样可行（这个特性在 p14 介绍过）。请参考 6.1.2 节, p144 对于初始化的介绍。

表 6.2 Vectors 的构造函数和析构函数

操作	效果
vector<Elem> c	产生一个空 vector，其中没有任何元素
vector<Elem> c1(c2)	产生另一个同型 vector 的副本（所有元素都被拷贝）
vector<Elem> c(n)	利用元素的 default 构造函数生成一个大小为 n 的 vector
vector<Elem> c(n,elem)	产生一个大小为 n 的 vector，每个元素值都是 elem
vector<Elem> c(beg,end)	产生一个 vector，以区间[beg;end]做为元素初值
c.~vector<Elem>()	销毁所有元素，并释放内存

[6] 你（或你的编译器）大概会认为这个语句实在荒谬，居然针对一个临时对象调用一个 non-const 成员函数。然而标准 C++确实允许我们这么做。

非变动性操作（Nonmodifying Operations）

表6.3列出vectors的所有非变动性操作。参见6.1.2节，p14附注和6.2.1节，p149。

表 6.3 Vectors 的非变动性操作

操作	效果
`c.size()`	返回当前的元素数量
`c.empty()`	判断大小是否为零。等同于 `size()==0`，但可能更快
`c.max_size()`	返回可容纳的元素最大数量
`capacity()`	返回重新分配空间前所能容纳的元素最大数量
`reserve()`	如果容量不足，扩大之[7]
`c1 == c2`	判断 c1 是否等于 c2
`c1 != c2`	判断 c1 是否不等于 c2，等同于 `!(c1==c2)`
`c1 < c2`	判断 c1 是否小于 c2
`c1 > c2`	判断 c1 是否大于 c2，等同于 `c2<c1`
`c1 <= c2`	判断 c1 是否小于等于 c2，等同于 `!(c2<c1)`
`c1 >= c2`	判断 c1 是否大于等于 c2，等同于 `!(c1<c2)`

赋值（Assignments）

表 6.4 Vectors 的赋值操作

操作	效果
`c1 = c2`	将 c2 的全部元素赋值给 c1
`c.assign(n,elem)`	复制 n 个 elem，赋值给 c
`c.assign(beg,end)`	将区间[beg;end]内的元素赋值给 c
`c1.swap(c2)`	将 c1 和 c2 元素互换
`swap(c1,c2)`	同上。此为全局函数

表 6.4 列出“将新元素赋值给 vectors，并将旧元素全部移除”的方法。一系列 `assign()`函数和构造函数一一对应。你可以采用不同的内容赋值方式（来自容器、数组和标准输入装置），这和 p144 的构造函数情况类似。所有赋值操作都可能会调用元素型别的 default 构造函数、copy 构造函数、assignment 操作符和/或析构函数，视元素数量的变化而定。例如：

```
std::list<Elem> l;
std::vector<Elem> coll;
```

[7] `reserve()`的确会更易（变动，*modify*）vector。因为它造成所有 references、pointers 和 iterators 失效。但是从逻辑内容来说，容器并没有变化，所以还是把它列在这里。

```
...
// make coll be a copy of the contents of l
coll.assign(l.begin(),l.end());
```

元素存取（Element Access）

表 6.5 列出用来直接存取 vector 元素的全部操作函数。按照 C 和 C++ 的惯例，第一元素的索引为 0，最后元素的索引为 size()-1。所以第 n 个元素的索引是 n-1。对于 non-const vectors，这些函数都返回元素的 reference。也就是说你可以使用这些操作函数来更改元素内容（如果没有其它妨碍因素的话）。

表 6.5 直接用来存取 vectors 元素的各项操作

操作	效果
c.at(idx)	返回索引 idx 所标示的元素。如果 idx 越界，抛出 out_of_range
c[idx]	返回索引 idx 所标示的元素。不进行范围检查
c.front()	返回第一个元素。不检查第一个元素是否存在
c.back()	返回最后一个元素。不检查最后一个元素是否存在

对调用者来说，最重要的事情莫过于搞清楚这些操作是否进行范围检查。只有 at()会那么做。如果索引越界，at()会抛出一个 out_of_range 异常（详见 3.3 节，p25）。其它函数都不作检查。如果发生越界错误，会引发未定义行为。对着一个空 vector 调用 operator[],front(),back()，都会引发未定义行为。

```
std::vector<Elem> coll;     // empty!

coll[5] = elem;             // RUNTIME ERROR → undefined behavior
std::cout << coll.front(); // RUNTIME ERROR → undefined behavior
```

所以，调用 operator[]时，你必须心里有数，确定索引有效；调用 front()或 back()时必须确定容器不空：

```
std::vector<Elem> coll;     // empty!
if (coll.size() > 5) {
   coll[5] = elem;          // OK
}
if (!coll.empty()) {
   cout << coll.front();    // OK
}
coll.at(5) = elem;          // throws out_of_range exception
```

迭代器相关函数（Iterator Functions）

vectors 提供了一些常规函数来获取迭代器，如表 6.6。vector 迭代器是 random access iterators（随机存取迭代器；关于迭代器分类详见 7.2 节, p251），因此从理论上讲，你可以通过这个迭代器操作所有 STL 算法。

表 6.6 Vectors 的迭代器相关函数

操作	效果
`c.begin()`	返回一个随机存取迭代器，指向第一元素
`c.end()`	返回一个随机存取迭代器，指向最后元素的下一位置
`c.rbegin()`	返回一个逆向迭代器，指向逆向迭代的第一元素
`c.rend()`	返回一个逆向迭代器，指向逆向迭代的最后元素的下一位置

这些迭代器的确切型别由实作版本决定。对 vectors 来说，通常就是一般指针。一般指针就是随机存取迭代器，而 vector 内部结构通常也就是个数组，所以指针行为可以适用。不过你可不能仰仗这一点。例如也许有个 STL 安全版本，对所有区间范围和其它潜在错误实施检查，那么其 vector 迭代器可能就是个辅助类别。7.2.6 节, p258 展示了“以指针实作迭代器”和“以类别实作迭代器”之间的差异所引起的麻烦问题。

vector 迭代器持续有效，除非发生两种情况：(1) 使用者在一个较小索引位置上安插或移除元素，(2) 由于容量变化而引起内存重新分配（详见 6.2.1 节, p149）。

安插（insert）和移除（remove）元素

表 6.7 列出 vector 元素的安插、移除操作函数。依 STL 惯例，你必须保证传入的参数合法：(1) 迭代器必须指向一个合法位置、(2) 区间的起始位置不能在结束位置之后、(3) 决不能从空容器中移除元素。

关于性能，以下情况你可以预期安插操作和移除操作会比较快些：

- 在容器尾部安插或移除元素
- 容量一开始就够大
- 安插多个元素时，“调用一次”当然比“调用多次”来得快

安插元素和移除元素，都会使“作用点”之后的各元素的 references、pointers、iterators 失效。如果安插操作甚至引发内存重新分配，那么该容器身上的所有 references、pointers、iterators 都会失效。

表 6.7 vector 的安插、移除相关操作

操作	效果
c.insert(pos,elem)	在 pos 位置上插入一个 elem 副本，并返回新元素位置
c.insert(pos,n,elem)	在 pos 位置上插入 n 个 elem 副本。无回传值
c.insert(pos,beg, end)	在 pos 位置上插入区间[beg;end]内的所有元素的副本无回传值
c.push_back(elem)	在尾部添加一个 elem 副本
c.pop_back()	移除最后一个元素（但不回传）
c.erase(pos)	移除 pos 位置上的元素，返回下一元素的位置
c.erase(beg,end)	移除[beg, end]区间内的所有元素，返回下一元素的位置
c.resize(num)	将元素数量改为 num（如果 size()变大了，多出来的新元素都需以 default 构造函数构造完成）
c.resize(num,elem)	将元素数量改为 num（如果 size()变大了，多出来的新元素都是 elem 的副本）
c.clear()	移除所有元素，将容器清空

vectors 并未提供任何函数可以直接移除“与某值相等”的所有元素。这是算法发挥威力的时候。以下语句可将所有其值为 val 的元素移除：

```
std::vector<Elem> coll;
...
// remove all elements with value val
coll.erase(remove(coll.begin(),coll.end(),
                  val),
           coll.end());
```

具体解释详见 5.6.1 节, p111。

如果只是要移除“与某值相等”的第一个元素，可以这么做：

```
std::vector<Elem> coll;
...
// remove first element with value val
std::vector<Elem>::iterator pos;
pos = find(coll.begin(),coll.end(),
           val);
if (pos != coll.end()) {
   coll.erase(pos);
}
```

6.2.3 将 Vectors 当做一般 Arrays 使用

C++ 标准程序库并未明确要求 vector 的元素必须分布于连续空间中。但是一份标准规格缺陷报告显示，这个缺点将获得弥补，标准规格书中将明确保证上述论点。如此一来你可以确定，对于 vector v 中任意一个合法索引 i，以下表达式肯定为 true：

```
&v[i] == &v[0] + i
```

保证了这一点，就可推导出一系列重要结果。简单地说，任何地点只要你需要一个动态数组，你就可以使用 vector。例如你可以利用 vector 来存放常规的 C 字符串（型别为 char* 或 const char*）：

```
std::vector<char> v;          // create vector as dynamic array of chars
v.resize(41);           // make room for 41 characters (including '\0')
strcpy(&v[0],"hello, world");   // copy a C-string into the vector
printf("%s\n", &v[0]); // print contents of the vector as C-string
```

不过，这么运用 vector 你可得小心（和使用动态数组一样小心），例如你必须确保上述 vector 的大小足以容纳所有数据，如果你用的是 C-String，记住最后有个 '\0' 元素。这个例子说明，不管出于什么原因（例如为了和既有的 C 程序库打交道），只要你需要一个元素型别为 T 的数组，就可以采用 vector<T>，然后传递第一元素的地址给它。

注意，千万不要把迭代器当做第一元素的地址来传递。vector 迭代器是由实作版本定义的，也许并不是个一般指针。

```
printf("%s\n", v.begin()); // ERROR (might work, but not portable)
printf("%s\n", &v[0]);     // OK
```

6.2.4 异常处理（Exception Handling）

vector 只支持最低限度的逻辑错误检查。subscript（下标）操作符的安全版本 at()，是唯一被标准规格书要求可能抛出异常的一个函数（p152）。此外标准规格书也规定，只有一般标准异常（例如内存不足时抛出 bad_alloc），或用户自定操作函数的异常，才可能发生。

如果 vector 调用的函数（元素型别所提供的函数，或使用者提供的函数）抛出异常，C++ 标准程序库作出如下保证：

1. 如果 push_back() 安插元素时发生异常，该函数不起作用。
2. 如果元素的拷贝操作（包括 copy 构造函数和 assignment 操作符）不抛出异常，那么 insert() 要么成功，要么不生效用。
3. pop_back() 决不会抛出任何异常。

4. 如果元素拷贝操作（包括 copy 构造函数和 assignment 操作符）不抛出异常，erase()和 clear()就不抛出异常。
5. swap()不抛出异常。
6. 如果元素拷贝操作（包括 copy 构造函数和 assignment 操作符）绝对不会抛出异常，那么所有操作不是成功，就是不起作用。这类元素可被称为 **POD**（plain old data，简朴的老式数据）。POD 泛指那些无 C++特性的型别，例如 C structure 便是。

所有这些保证都基于一个条件：析构函数不得抛出异常。参见 5.11.2 节，p139 对于 STL 异常处理的一般性讨论。6.10.10 节，p249 列出对于异常给予特别保证的所有容器操作函数。

6.2.5 Vectors 运用实例

下面的例子展示了 vectors 的简单用法：

```
// cont/vector1.cpp

#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;

int main()
{
  // create empty vector for strings
  vector<string> sentence;

  // reserve memory for five elements to avoid reallocation
  sentence.reserve(5);

  // append some elements
  sentence.push_back("Hello,");
  sentence.push_back("how");
  sentence.push_back("are");
  sentence.push_back("you");
  sentence.push_back("?");

  // print elements separated with spaces
  copy (sentence.begin(), sentence.end(),
        ostream_iterator<string>(cout," "));
  cout << endl;
```

```
    // print "technical data"
    cout << " max_size(): " << sentence.max_size() << endl;
    cout << " size(): " << sentence.size() << endl;
    cout << " capacity(): " << sentence.capacity() << endl;

    // swap second and fourth element
    swap (sentence[1], sentence[3]);

    // insert element "always" before element "?"
    sentence.insert (find(sentence.begin(),sentence.end(),"?"),
                     "always");

    // assign "!" to the last element
    sentence.back() = "!";

    // print elements separated with spaces
    copy (sentence.begin(), sentence.end(),
          ostream_iterator<string>(cout," "));
    cout << endl;

    // print "technical data" again
    cout << " max_size(): " << sentence.max_size() << endl;
    cout << " size(): " << sentence.size() << endl;
    cout << " capacity(): " << sentence.capacity() << endl;
}
```

程序的输出可能像这样：

```
Hello, how are you ?
  max_size(): 268435455
  size(): 5
  capacity(): 5
Hello, you are how always !
  max_size(): 268435455
  size(): 6
  capacity(): 10
```

注意我说的是“可能”。是的，max_size()和capacity()的结果由实作版本决定。从这个例子中你可以看到，当容量不足时，此一实作版本将容量扩充一倍。

6.2.6 Class vector<bool>

C++ 标准程序库专门针对元素型别为 bool 的 vector 设计了一个特殊版本，目的是获取一个优化的 vector。其耗用空间远远小于以一般 vector 实作出来的 bool vector。一般 vector 的实作版本会为每个元素至少分配一个 byte 空间，而 vector<bool> 特殊版本内部只用一个 bit 来存储一个元素。所以通常小 8 倍之多。不过这里有个小麻烦：C++ 的最小可寻址值通常以 byte 为单位。所以上述的 vector 特殊版本需针对 references 和 iterators 作特殊考虑。

考虑结果是，vector<bool> 无法满足其它 vectors 必须的所有条件（例如 vector<bool>::reference 并不返回真正的 lvalue，vector<bool>::iterator 不是个真正的随机存取迭代器）。所以某些 template 程序代码可能适用于任何型别的 vector，唯独无法应付 vector<bool>。此外 vector<bool> 可能比一般 vectors 慢一些，因为所有元素操作都必须转化为 bit 操作。不过 vector<bool> 的具体方案也是由实作版本决定，所以性能（包括速度和空间消耗）也可能都有不同。

注意，vector<bool> 不仅仅是个特殊的 bool 版本，它还提供某些特殊的 bit 操作。你可以利用它们更方便地操作 bit 或标志（flags），而且由于 vector<bool> 的大小可动态改变，你还可以把它当成动态大小的 bitfield（位域）。如此一来你便可以添加或移除 bits。如果你需要静态大小的 bitfield，应当使用 bitset，而不是 vector<bool>。bitset 详见 10.4 节，p460。

表 6.8 vector<bool> 的特殊操作

操作	效果
c.flip()	将所有 bool 元素值取反值，亦即求补码
m[idx].flip()	将索引 idx 的 bit 元素取反值
m[idx] = val	令索引 idx 的 bit 元素值为 val（指定单一 bit）
m[idx1] = m[idx2]	令索引 idx1 的 bit 元素值为索引 idx2 的 bit 元素值

表 6.8 列出 vector<bool> 的特殊操作。flip() 对 vector 中的每一个 bit 取补码。注意，你竟然可以对单一 bool 元素调用 flip()。是不是很惊讶？也许你觉得让 subscript 操作符返回 bool，再对如此基本型别调用 flip() 是不可能的。然而这里 vector<bool> 用了一个常见技巧，称作 **proxy**[8]，对于 vector<bool>，subscript 操作符（及其它返回单一元素的操作符）的返回型别实际上是个辅助类别，一旦你

[8] proxy 可让你控制一般无法控制的东西，通常用来获取更好的安全性。上述情形中，此技术施行某种控制，使某种操作成为可能。原则上其返回的对象行为类似 bool。

要求返回值为 bool，便会触发一个自动型别转换函数。表 6.8 的其它操作由成员函数支持。vector<bool> 的相关声明如下：

```
namespace std {
class vector<bool> {
   public:
      // auxiliary type for subscript operator
      class reference {
         ...
         public:
            // automatic type conversion to bool
            operator bool() const;

            // assignments
            reference& operator= (const bool);
            reference& operator= (const reference&);

            // bit complement
            void flip();
      }
      ...

      // operations for element access
      // - return type is reference instead of bool
      reference operator[](size_type n);
      reference at(size_type n);
      reference front();
      reference back();
      ...
   };
}
```

你会发现，所有用于元素存取的函数，返回的都是 reference 型别。所以，你可以使用以下语句：

```
c.front().flip(); // negate first Boolean element
c[5] = c.back();  // assign last element to element with index 5
```

一如往常，为了避免未定义的行为，调用者必须确保第一、第六和最后一个元素存在。

只有在 non-const vector<bool> 容器中才会用到内部型别 reference。存取元素用的 const member function 会返回型别为 bool 的普通数值。

6.3 Deques

容器 deque（发音为 "deck"）和 vector 非常相似。它也采用动态数组来管理元素，提供随机存取，并有着和 vector 几乎一模一样的接口。不同的是 deque 的动态数组头尾都开放，因此能在头尾两端进行快速安插和删除（图 6.2）。

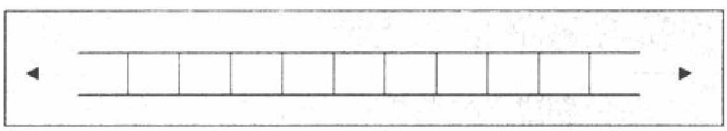

图 6.2 deque 的逻辑结构

为了获取这种能力，deque 通常实作为一组独立区块，第一区块朝某方向扩展，最后一个区块朝另一方向扩展，如图 6.3。

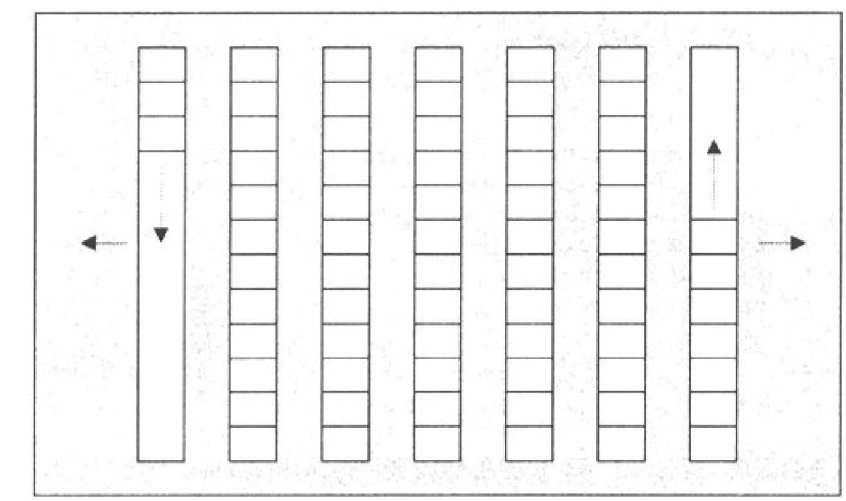

图 6.3 deque 的内部结构

使用 deque 之前，必须先含入头文件 <deque>[9]：

```
#include <deque>
```

在其中，deque 型别是定义于命名空间 std 内的一个 class template：

```
namespace std {
    template <class T,
```

[9] 早期的 STL 中，deque 的头文件是 <deque.h>。

```
            class Allocator = allocator<T> >
    class deque;
}
```

和 vector 相同，第一个 template 参数用来表明元素型别——只要是 assignable 和 copyable 都可以胜任。第二个 template 参数可有可无，用来指定内存模型（memory model），缺省为 allocator（详见第 15 章）[10]。

6.3.1 Deques 的能力

与 vectors 相比，deques 功能上的不同处在于：

- 两端都能快速安插元素和移除元素（vector 只在尾端逞威风）。这些操作可以在分期摊还的常数时间（amortized constant time）内完成。
- 存取元素时，deque 的内部结构会多一个间接过程，所以元素的存取和迭代器的动作会稍稍慢一些。
- 迭代器需要在不同区块间跳转，所以必须是特殊的智能型指针，非一般指针。
- 在对内存区块有所限制的系统中（例如 PC 系统），deque 可以内含更多元素，因为它使用不止一块内存。因此 deque 的 max_size()可能更大。
- deque 不支持对容量和内存重分配时机的控制。特别要注意的是，除了头尾两端，在任何地方安插或删除元素，都将导致指向 deque 元素的任何 pointers、references、iterators 失效。不过，deque 的内存重分配优于 vectors，因为其内部结构显示，deques 不必在内存重分配时复制所有元素。
- deque 的内存区块不再被使用时，会被释放。deque 的内存大小是可缩减的。不过，是不是这么做，以及究竟怎么做，由实作版本定义之。

deques 的下述特性跟 vectors 差不多：

- 在中段部分安插、移除元素的速度相对较慢，因为所有元素都需移动以腾出或填补空间。
- 迭代器属于 random access iterator（随机存取迭代器）。

总之，如果是以下情形，最好采用 deque：

- 你需要在两端安插和移除元素（这是 deque 的拿手好戏）。
- 无需引用（*refer to*）容器内的元素。
- 要求容器释放不再使用的元素（不过，标准规格上并没有保证这一点）。

vectors 和 deques 的接口几乎一样，所以如果无需什么特殊性质，两者都可试试。

---

[10] 在尚未支持 default template parameters 的系统中，第二参数通常会被省略。

6.3.2 Deques 的操作函数

表 6.9 至表 6.11 列出了 deques 的所有操作函数：

表 6.9 deques 的构造函数和析构函数

操作	效果
`deque<Elem> c`	产生一个空的 deque
`deque<Elem> c1(c2)`	针对某个 deque 产生同型副本（所有元素都被拷贝）
`deque<Elem> c(n)`	产生一个 deque，含有 n 个元素，这些元素均以 default 构造函数产生出来
`deque<Elem> c(n,elem)`	产生一个 deque，含有 n 个元素，这些元素均是 elem 的副本
`deque<Elem> c(beg,end)`	产生一个 deque，以区间[beg;end]内的元素为初值
`c.~deque<Elem>()`	销毁所有元素，释放内存

表 6.10 deques 的非变动性操作（nonmodifying operations）

操作	效果
`c.size()`	返回容器的实际元素个数
`c.empty()`	判断容器大小是否为零。等同于 `size()==0`，但可能更快
`c.max_size()`	返回可容纳的最大元素数量
`c1 == c2`	判断是否 c1 等于 c2
`c1 != c2`	判断是否 c1 不等于 c2。等同于 `!(c1 == c2)`
`c1 < c2`	判断是否 c1 小于 c2
`c1 > c2`	判断是否 c1 大于 c2。等同于 `c2 < c1`
`c1 <= c2`	判断是否 c1 小于等于 c2 。等同于 `!(c2 < c1)`
`c1 >= c2`	判断 c1 是否大于等于 c2 。等同于 `!(c1 < c2)`
`c.at(idx)`	返回索引 idx 所标示的元素。如果 idx 越界，抛出 out_of_range
`c[idx]`	返回索引 idx 所标示的元素，不进行范围检查
`c.front()`	返回第一个元素。不检查元素是否存在
`c.back()`	返回最后一个元素。不检查元素是否存在
`c.begin()`	返回一个随机迭代器，指向第一元素
`c.end()`	返回一个随机迭代器，指向最后元素的下一位置
`c.rbegin()`	返回一个逆向迭代器，指向逆向迭代时的第一个元素
`c.rend()`	返回一个逆向迭代器，指向逆向迭代时的最后元素的下一位置

表 6.11 deques 的变动性操作（modifying operations）

操作	效果
c1 = c2	将 c2 的所有元素赋值给 c1
c.assign(n,elem)	将 n 个 elem 副本赋值给 c
c.assign(beg,end)	将区间[beg;end]中的元素赋值给 c
c1.swap(c2)	将 c1 和 c2 的元素互换
swap(c1,c2)	同上。此为全局函数
c.insert(pos,elem)	在 pos 位置插入一个 elem 副本，并返回新元素的位置
c.insert(pos,n, elem)	在 pos 位置插入 elem 的 n 个副本，无返回值。
c.insert(pos,beg, end)	在 pos 位置插入在区间[beg;end)所有元素的副本，无返回值
c.push_back(elem)	在尾部添加 elem 的一个副本
c.pop_back()	移除最后一个元素（但不回传）
c.push_front(elem)	在头部插入 elem 的一个副本
c.pop_front()	移除头部元素（但不回传）
c.erase(pos)	移除 pos 位置上的元素，返回下一元素位置
c.erase(beg,end)	移除[beg, end)区间内的所有元素，返回下一元素位置
c.resize(num)	将大小（元素个数）改为 num。如果 size()增长了，新增元素都以 default 构造函数产生出来
c.resize(num,elem)	将大小（元素个数）改为 num。如果 size()增长了，新增元素都是 elem 的副本
c.clear()	移除所有元素，将容器清空

deques 的各项操作只有以下数点和 vectors 不同：

1. deques 不提供容量操作（capacity()和 reserve()）。
2. deques 直接提供函数，用以完成头部元素的安插和删除（push_front()和 pop_front()）。

其它操作都相同，所以这里不重复。它们的具体描述请见 p150, 6.2.2 节。

还有一些值得考虑的事情：

1. 除了 at()，没有任何成员函数会检查索引或迭代器是否有效。
2. 元素的插入或删除可能导致内存重新分配，所以任何插入或删除动作都会使所有指向 deques 元素的 pointers、references 和 iterators 失效。唯一例外是在头部或尾部插入元素，动作之后，pointers 和 references 仍然有效（但 iterators 就没这么幸运）。

6.3.3 异常处理（Exception Handling）

原则上 deques 提供的异常处理和 vectors 提供的一样（p155）。新增的操作函数 push_front()和 pop_front()分别对应于 push_back()和 pop_back()。因此，C++ 标准程序库保证下列行为：

- 如果以 push_back()或 push_front()安插元素时发生异常，则该操作不带来任何效应。
- pop_back()和 pop_front()不会抛出任何异常。

STL 的异常处理一般原则请见 p139, 5.11.2 节。异常发生时，提供特殊保障的所有容器操作函数均列于 p248, 6.10.10 节。

6.3.4 Deques 运用实例

以下程序以简单的例子说明 deques 的功用：

```
// cont/deque1.cpp

#include <iostream>
#include <deque>
#include <string>
#include <algorithm>
using namespace std;

int main()
{
    // create empty deque of strings
    deque<string> coll;

    // insert several elements
    coll.assign (3, string("string"));
    coll.push_back ("last string");
    coll.push_front ("first string");

    // print elements separated by newlines
    copy (coll.begin(), coll.end(),
          ostream_iterator<string>(cout,"\n"));
    cout << endl;

    // remove first and last element
    coll.pop_front();
    coll.pop_back();
```

```
    // insert "another" into every element but the first
    for (int i=1; i<coll.size(); ++i) {
        coll[i] = "another " + coll[i];
    }

    // change size to four elements
    coll.resize (4, "resized string");

    // print elements separated by newlines
    copy (coll.begin(), coll.end(),
          ostream_iterator<string>(cout,"\n"));
}
```

程序输出如下：

```
first string
string
string
string
last string

string
another string
another string
resized string
```

6.4 Lists

Lists 使用一个 doubly linked list（双向链表）来管理元素，如图 6.4。按惯例，C++ 标准程序库并未明定实作方式，只是遵守 list 的名称、限制和规格。

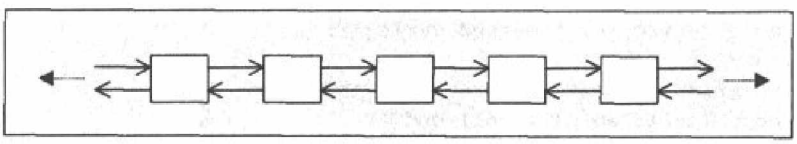

图 6.4 List 的结构

使用 list 时必须含入头文件 `<list>`[11]：

```
#include <list>
```

其中 list 型别系定义于 namespace `std` 中，是个 class template：

```
namespace std {
    template <class T,
              class Allocator = allocator<T> >
    class list;
}
```

任何型别 `T` 只要具备 assignable 和 copyable 两性质，就可以作为 list 的元素。第二个 template 参数可有可无，用来指定内存模型（详见第 15 章）。缺省的内存模型是 C++ 标准程序库所提供的 `allocator`[12]。

6.4.1 Lists 的能力

Lists 的内部结构和 vector 或 deque 截然不同，所以在几个主要方面与前述二者存在明显区别：

- Lists 不支持随机存取。如果你要存取第 5 个元素，就得顺着串链一一爬过前 4 个元素。所以，在 list 中随机遍历任意元素，是很缓慢的行为。
- 任何位置上（不只是两端）执行元素的安插和移除都非常快，始终都是常数时间内完成，因为无需移动任何其它元素。实际上内部只是进行了一些指针操作而已。

---

[11] 早期 STL 中，list 的定义头文件是 `<list.h>`

[12] 如果系统不支持 default template parameters，通常省略第二参数。

- 安插和删除动作并不会造成指向其它元素的各个 pointers、references、iterators 失效。
- Lists 对于异常有着这样的处理方式：要么操作成功，要么什么都不发生。你决不会陷入“只成功一半”这种前不着村后不巴店的尴尬境地。

Lists 所提供的成员函数反映出它和 vectors 以及 deques 的不同：

- 由于不支持随机存取，lists 既不提供 subscript（下标）操作符，也不提供 `at()`。
- Lists 并未提供容量、空间重新分配等操作函数，因为全无必要。每个元素都有自己的内存，在被删除之前一直有效。
- Lists 提供了不少特殊的成员函数，专门用于移动元素。较之同名的 STL 通用算法，这些函数执行起来更快，因为它们无需拷贝或移动，只需调整若干指针即可。

6.4.2 Lists 的操作函数

生成（Creation），复制（Copy）和销毁（Destroy）

lists 的生成、复制和销毁动作，和所有序列式容器相同，详见表 6.12。关于初值来源（initialization sources）的若干注意事项，可参考 6.1.2 节, p144。

表 6.12 Lists 的构造函数和析构函数

操作	效果
`list<Elem> c`	产生一个空的 list
`list<Elem> c1(c2)`	产生一个与 `c2` 同型的 list（每个元素都被复制）
`list<Elem> c(n)`	产生拥有 `n` 个元素的 list，这些元素都以 default 构造函数初始化
`list<Elem> c(n,elem)`	产生拥有 `n` 个元素的 list，每个元素都是 `elem` 的副本
`list<Elem> c(beg,end)`	产生一个 list 并以 `[beg;end]` 区间内的元素为初值
`c.~list<Elem>()`	销毁所有元素，释放内存

非变动性操作（Nonmodifying Operations）

Lists 也提供诸如“询问大小”和“两相比较”等等一般性操作。详见表 6.13 和 6.1.2 节, p144。

表 6.13 Lists 的非变动性操作（Nonmodifying Operations）

操作	效果
c.size()	返回元素个数
c.empty()	判断容器大小是否为零。等同于 size()==0，但可能更快
c.max_size()	返回元素的最大可能数量
c1 == c2	判断是否 c1 等于 c2
c1 != c2	判断是否 c1 不等于 c2。等同于!(c1 == c2)
c1 < c2	判断是否 c1 小于 c2
c1 > c2	判断是否 c1 大于 c2。等同于与 c2 < c1 相同
c1 <= c2	判断是否 c1 小于等于 c2。等同于 !(c2 < c1)
c1 >= c2	判断是否 c1 大于等于 c2。等同于 !(c1 < c2)

赋值（Assignment）

和其它序列式容器一样，lists 也提供常用的赋值操作，如表 6.14。

表 6.14 Lists 的 assignment（赋值）操作函数

操作	效果
c1 = c2	将 c2 的全部元素赋值给 c1
c.assign(n,elem)	将 elem 的 n 个拷贝赋值给 c
c.assign(beg,end)	将区间[beg;end)的元素赋值给 c
c1.swap(c2)	将 c1 和 c2 的元素互换
swap(c1,c2)	同上。此为全局函数

…如往常，安插操作和构造函数一一匹配，如此一来就有能力提供不同的初值来源（initialization sources），详见 6.1.2 节, p144。

元素存取（Element Access）

lists 不支持随机存取，只有 front()和 back()能够直接存取元素，如表 6.15。

表 6.15 Lists 元素的直接存取

操作	效果
c.front()	返回第一个元素。不检查元素存在与否
c.back()	返回最后一个元素。不检查元素存在与否

一如以往，这些操作并不检查容器是否为空。对着空容器执行任何操作，都会导致未定义的行为。所以调用者必须确保容器至少含有一个元素。例如：

```
std::list<Elem> coll; // empty!

std::cout << coll.front(); // RUNTIME ERROR → undefined behavior

if (!coll.empty()) {
    std::cout << coll.back();  // OK
}
```

迭代器相关函数（Iterator Functions）

只有运用迭代器，才能够存取 list 中的各个元素。Lists 提供的迭代器函数如表 6.16。然而由于 list 不能随机存取，这些迭代器只是双向（而非随机）迭代器。所以凡是用到随机存取迭代器的算法（所有用来操作元素顺序的算法——特别是排序算法——都归此类）你都不能调用。不过你可以拿 list 的特殊成员函数 `sort()`取而代之，详见 p245。

表 6.16 Lists 的迭代器相关函数

操作	效果
`c.begin()`	返回一个双向迭代器，指向第一个元素
`c.end()`	返回一个双向迭代器，指向最后元素的下一位置
`c.rbegin()`	返回一个逆向迭代器，指向逆向迭代的第一个元素
`c.rend()`	返回一个逆向迭代器，指向逆向迭代的最后元素的下一位置

元素的安插（Inserting）和移除（Removing）

表 6.17 列出 lists 元素的安插和移除操作。Lists 提供 deques 的所有功能，还增加了 `remove()`和 `remove_if()`算法应用于 list 身上的特殊版本。

和一般运用 STL 时相似，你必须确保参数的正确。迭代器必须指向合法位置，区间终点不能位于区间起点的前头；还有，你不能从空容器中移除元素。

如果想要安插或移除多个元素，你可以对它们进行单一调用，比多次调用来得快。

为了移除元素，lists 特别配备了 `remove()`算法（9.7.1 节, p378）的特别版本。这些成员函数比 `remove()`算法的速度更快，因为它们只进行内部指目标操作，无需顾及元素。所以，面对 list，你应该调用成员函数 `remove()`，而不是像面对 vectors 和 deques 那样调用 STL 算法（如 p154 所示）。

表 6.17 Lists 的安插、移除操作函数

操作	效果
c.insert(pos,elem)	在迭代器 pos 所指位置上安插一个 elem 副本，并返回新元素的位置
c.insert(pos,n,elem)	在迭代器 pos 所指位置上安插 n 个 elem 副本，无返回值
c.insert(pos,beg,end)	在迭代器 pos 所指位置上安插 [beg;end]区间内的所有元素的副本，无返回值
c.push_back(elem)	在尾部追加一个 elem 副本
c.pop_back()	移除最后一个元素（但不返回）
c.push_front(elem)	在头部安插一个 elem 副本
c.pop_front()	移除第一元素（但不返回）
c.remove(val)	移除所有其值为 val 的元素
c.remove_if(op)	移除所有“造成 op(elem)结果为 true”的元素
c.erase(pos)	移除迭代器 pos 所指元素，返回下一元素位置
c.erase(beg,end)	移除区间[beg;end]内的所有元素，返回下一元素位置
c.resize(num)	将元素容量变为 num。如果 size()变大，则以 default 构造函数构造所有新增元素
c.resize(num,elem)	将元素容量变为 num。如果 size()变大，则以 elem 副本作为新增元素的初值
c.clear()	移除全部元素，将整个容器清空

想要将所有“与某值相等”的元素移除，可以用如下语句（进一步细节详见 5.6.3 节，p116）：

```
std::list<Elem> coll;
...
// remove all elements with value val
coll.remove(val);
```

如果只是想移除“与某值相等”的第一个元素，你得使用诸如 p154 中针对 vectors 所用的算法。

如果使用 remove_if()，你可以通过一个函数或仿函数[13]来定义元素移除原则；它可以将每一个“令传入之操作结果为 true”的元素移除：

```
list.remove_if (not1(bind2nd(modulus<int>(),2)));
```

如果你对以上语句感到头晕，别急，到 p306 看看详细解释。关于 remove()和 remove_if()的其它例子，详见 p378。

13 一个不支持 member templates 的系统，通常不会提供 remove_if()成员函数。

Splice 函数

Linked lists 的一大好处就是不论在任何位置，元素的安插和移除都只需要常数时间。如果你有必要将若干元素从 A 容器转放到 B 容器，那么上述好处就更见其效了，因为你只需要重新定向某些指针即可，如图 6.5。

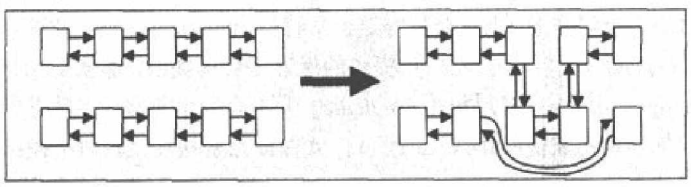

图 6.5 Splice 操作函数用以改变 list 元素的次序

为了利用这个优势，lists 不仅提供 `remove()`，还提供其它一些成员函数，用来改变元素和区间次序，或是用来重新串链。我们不仅可以调用这些函数，移动单一 list 内的元素，也可以移动两个 lists 之间的元素——只要 lists 的型别一致即可。表 6.18 列举出这些函数。6.10.8 节, p244 另有详细说明，实例可见 6.4.4 节, p172。

表 6.18 Lists 的特殊变动性操作（Special Modifying Operations）

操作	效果
`c.unique()`	如果存在若干相邻而数值相等的元素，就移除重复元素，只留下一个
`c.unique(op)`	如果存在若干相邻元素，都使 `op()` 的结果为 `true`，则移除重复元素，只留下一个
`c1.splice(pos,c2)`	将 `c2` 内的所有元素转移到 `c1` 之内、迭代器 `pos` 之前
`c1.splice(pos,c2, c2pos)`	将 `c2` 内的 `c2pos` 所指元素转移到 `c1` 内的 `pos` 所指位置上（`c1` 和 `c2` 可相同）
`c1.splice(pos,c2, c2beg,c2end)`	将 `c2` 内的 `[c2beg;c2end]` 区间内所有元素转移到 `c1` 内的 `pos` 之前（`c1` 和 `c2` 可相同）
`c.sort()`	以 `operator<` 为准则，对所有元素排序
`c.sort(op)`	以 `op()` 为准则，对所有元素排序
`c1.merge(c2)`	假设 `c1` 和 `c2` 容器都包含已序（*sorted*）元素，将 `c2` 的全部元素转移到 `c1`，并保证合并后的 list 仍为已序
`c1.merge(c2,op)`	假设 `c1` 和 `c2` 容器都包含 `op()` 原则下的已序（*sorted*）元素，将 `c2` 的全部元素转移到 `c1`，并保证合并后的 list 在 `op()` 原则下仍为已序
`c.reverse()`	将所有元素反序（reverse the order）

6.4.3 异常处理（Exception Handling）

所有 STL 标准容器中，lists 对于异常安全性（**exception safety**）提供了最佳支持。几乎所有操作都是不成功便成仁：要么成功，要么无效。仅有少数几个操作没有如此保证，包括赋值（赋值）运算和成员函数 `sort()`，不过它们也有基本保证：异常发生时不会泄漏资源，也不会与容器恒常特性（invariants）发生冲突。`merge()`，`remove()`,`remove_if()`,`unique()` 提供的保证是有前提的，那就是元素间的比较动作（采用 `operator==` 或判断式 *predicate*）并不会抛出异常。用数据库编程术语来说，只要你不调用赋值操作或 `sort()`，并保证元素相互比较时不抛出异常，那么 lists 便可说是“事务安全（**transaction safe**）”。表 6.19 列出异常状况下提供特殊保证的所有操作函数。STL 异常处理的一般性讨论，请见 5.11.2 节, p139。

表 6.19 Lists 的各种操作在异常发生时提供的特殊保证

操作	保证
`push_back()`	如果不成功，就是无任何作用
`push_front()`	如果不成功，就是无任何作用
`insert()`	如果不成功，就是无任何作用
`pop_back()`	不抛出异常
`pop_front()`	不抛出异常
`erase()`	不抛出异常
`clear()`	不抛出异常
`resize()`	如果不成功，就是无任何作用
`remove()`	只要元素比较操作不抛出异常，它就不抛出异常
`remove_if()`	只要判断式 *predicate* 不抛出异常，它就不抛出异常
`unique()`	只要元素比较操作不抛出异常，它就不抛出异常
`splice()`	不抛出异常
`merge()`	只要元素比较时不抛出异常，它便保证“要么不成功，要么无任何作用”
`reverse()`	不抛出异常
`swap()`	不抛出异常

6.4.4 Lists 运用实例

下面这个例子突显出 list 特殊成员函数的用法:

```
// cont/list1.cpp

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

void printLists (const list<int>& l1, const list<int>& l2)
{
    cout << "list1: ";
    copy (l1.begin(), l1.end(), ostream_iterator<int>(cout," "));
    cout << endl << "list2: ";
    copy (l2.begin(), l2.end(), ostream_iterator<int>(cout," "));
    cout << endl << endl;
}

int main()
{
    // create two empty lists
    list<int> list1, list2;

    // fill both lists with elements
    for (int i=0; i<6; ++i) {
         list1.push_back(i);
         list2.push_front(i);
    }
    printLists(list1, list2);

    // insert all elements of list1 before the first element with value 3 of list2
    // - find() returns an iterator to the first element with value 3
    list2.splice(find(list2.begin(),list2.end(), // destination position
                 3),
                 list1);             // source list
    printLists(list1, list2);

    // move first element to the end
    list2.splice(list2.end(),        // destination position
                 list2,              // source list
                 list2.begin());     // source position
    printLists(list1, list2);

    // sort second list, assign to list1 and remove duplicates
    list2.sort();
    list1 = list2;
```

```
    list2.unique();
    printLists(list1, list2);

    // merge both sorted lists into the first list
    list1.merge(list2);
    printLists(list1, list2);
}
```

程序输出如下：

```
list1: 0 1 2 3 4 5
list2: 5 4 3 2 1 0

list1:
list2: 5 4 0 1 2 3 4 5 3 2 1 0

list1:
list2: 4 0 1 2 3 4 5 3 2 1 0 5

list1: 0 0 1 1 2 2 3 3 4 4 5 5
list2: 0 1 2 3 4 5

list1: 0 0 0 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5
list2:
```

6.5 Sets 和 Multisets

set 和 multiset 会根据特定的排序准则，自动将元素排序。两者不同处在于 multisets 允许元素重复而 sets 不允许（请参考 6.6 节和第 5 章关于本主题的讨论）。

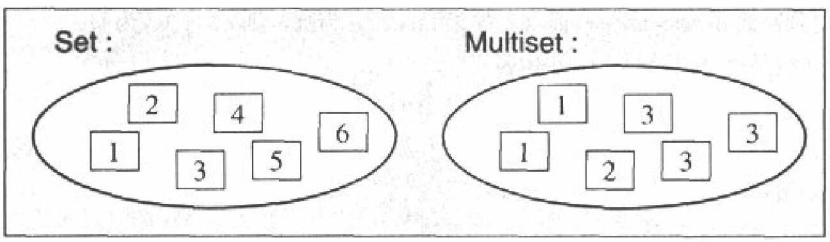

图 6.6 Sets 和 Multisets

使用 set 或 multiset 之前，必须先含入头文件 <set>[14]：

```
#include <set>
```

在这个头文件中，上述两个型别都被定义为命名空间 std 内的 class templates：

```
namespace std {
    template <class T,
              class Compare = less<T>,
              class Allocator = allocator<T> >
    class set;

    template <class T,
          class Compare = less<T>,
          class Allocator = allocator<T> >
    class multiset;
}
```

只要是 assignable、copyable、comparable（根据某个排序准则）的型别 T，都可以成为 set 或 multiset 的元素型别。可有可无的第二个 template 参数用来定义排序准则。如果没有传入特别的排序准则，就采用缺省准则 less——这是一个仿函数，以 operator< 对元素进行比较，以便完成排序（less[15]的内容详见 p305）。可有可无的第三参数用来定义内存模型（参见第 15 章）。缺省的内存模型是

---

[14] 早期 STL 中，sets 的头文件是 <set.h>，multisets 的头文件是 <mulitiset.h>

[15] 在不支持 default template arguments 的系统中，第二参数通常会被省略。

allocator，由 C++ 标准程序库提供[16]。

所谓“排序准则”，必须定义 **strict weak ordering**，其意义如下：

1. 必须是“反对称的（**antisymmetric**）”。
   对 operator< 而言，如果 x < y 为真，则 y < x 为假。
   对判断式 *predicate* op() 而言，如果 op(x,y) 为真，则 op(y,x) 为假。
2. 必须是“可传递的（**transitive**）”。
   对 operator< 而言，如果 x < y 为真且 y < z 为真，则 x < z 为真。
   对判断式 op() 而言，如果 op(x,y) 为真且 op(y,z) 为真，则 op(x,z) 为真。
3. 必须是“非自反的（**irreflexive**）”。
   对 operator< 而言，x < x 永远为假。
   对判断式 *predicate* op() 而言，op(x,x) 永远为假。

基于这些特性，排序准则也可用于相等性检验，也就是说，如果两个元素都不小于对方（或说 op(x,y) 和 op(y,x) 都为假），则两个元素相等。

译注：以上种种性质（及其它各种相关性质）是 STL 学术理论的一部分。STL 先建立起一个抽象概念阶层体系，形成一个软件组件分类学，最后再以实际工具（C++ template）将各个概念实作出来。这些理论架构的最佳描述书籍是《*Generic Programming and the STL*》，by Matthew H. Austern, Addison Wesley 1998；中译本《泛型程序设计与 STL》，侯捷/黄俊尧合译，碁峰 2000。

6.5.1 Sets 和 Multisets 的能力

和所有标准关联式容器类似，sets 和 multisets 通常以平衡二叉树（balanced binary tree，图 6.7）完成。C++ 标准规格书并未明定，但由 sets 和 multisets 各项操作的复杂度可以得出这样的结论[17]。

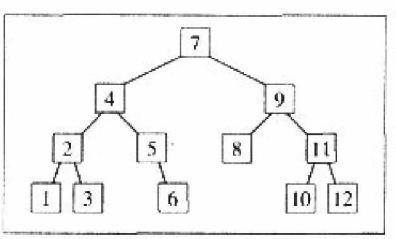

图 6.7 Sets 和 Multisets 的内部结构

[16] 在不支持 default template arguments 的系统中，第三参数通常会被省略。

[17] 事实上 sets 和 multisets 通常以红黑树（red-black tree）实作而成。红黑树在改变元素数量和元素搜寻方面都很出色，它保证节点安插时最多只会作两个重新连接（relink）动作，而且到达某一元素的最长路径深度，最多只是最短路径深度的两倍。

自动排序的主要优点在于使二叉树于搜寻元素时具有良好性能。其搜寻函数算法具有对数（logarithmic）复杂度。在拥有 1000 个元素的 sets 或 multisets 中搜寻元素，二叉树搜寻动作（由成员函数执行）的平均时间为线性搜寻（由 STL 算法执行）时间的 1/50。关于复杂度的讨论，详见 p21, 2.3 节。

但是，自动排序造成 sets 和 multisets 的一个重要限制：你不能直接改变元素值，因为这样会打乱原本正确的顺序。因此，要改变元素值，必须先删除旧元素，再插入新元素。这里提供的接口正反映了这种行为：

- sets 和 multisets 不提供用来直接存取元素的任何操作函数。
- 通过迭代器进行元素间接存取，有一个限制：从迭代器的角度来看，元素值是常数。

6.5.2 Sets 和 Multisets 的操作函数

生成（Create）、复制（Copy）和销毁（Destroy）

表 6.20 列出 sets 和 multisets 的构造函数和析构函数。

表 6.20 Sets 和 Multisets 的构造函数和析构函数

操作	效果
set `c`	产生一个空的 set/multiset，其中不含任何元素
set `c(op)`	以 `op` 为排序准则，产生一个空的 set/multiset
set `c1(c2)`	产生某个 set/multiset 的副本，所有元素均被复制
set `c(beg,end)`	以区间 `[beg;end)` 内的元素产生一个 set/multiset
set `c(beg,end,op)`	以 `op` 为排序准则，利用 `[beg;end)` 内的元素生成一个 set/multiset
`c.~`***set***`()`	销毁所有元素，释放内存

其中 ***set*** 可为下列形式：

set	效果
`set<Elem>`	一个 set，以 `less<>`（`operator<`）为排序准则
`set<Elem,Op>`	一个 set，以 `op` 为排序准则
`multiset<Elem>`	一个 multiset，以 `less<>`（`operator<`）为排序准则
`multiset<Elem,Op>`	一个 multiset，以 `op` 为排序准则

有两种方式可以定义排序准则：

1. 以 template 参数定义之。

   例如[18]：

   ```
   std::set<int, std::greater<int> > coll;
   ```

   这种情况下，排序准则就是型别的一部分。因此型别系统确保“只有排序准则相同的容器才能被合并”。这是排序准则的通常指定法。更精确地说，第二参数是排序准则的型别，实际的排序准则是容器所产生的函数对象（*function object*，或称*functor*）。为了产生它，容器构造函数会调用“排序准则型别”的 default 构造函数。p294 有一个“使用者自定之排序准则”的运用实例。

2. 以构造函数参数定义之。

   这种情况下，同一个型别可以运用不同的排序准则，而排序准则的初始值或状态也可以不同。如果执行期才获得排序准则，而且需要用到不同的排序准则（但数据型别必须相同），此一方式可派上用场。p191 有个完整例子。

如果使用者没有提供特定的排序准则，那么就采用缺省准则——仿函数 `less<>`。`less<>` 系透过 `operator<` 对元素进行排序[19]。

请注意，排序准则也被用于元素相等性检验工作。当采用缺省排序准则时，两元素的相等性检验语句如下：

```
if (! (elem1 < elem2 || elem2 < elem1))
```

这样做有三点好处：

1. 只需传递一个参数作为排序准则。
2. 不必针对元素型别提供 `operator==`。
3. 你可以对“相等性”有截然相反的定义（即使表达式中 `operator==` 的行为有所不同，也无关紧要）。不过当心造成混淆。

这种相等性检验方式会花费比较长的时间，因为评估上述表达式可能需要两次比较。注意，如果第一次比较结果为 true，就不用进行第二次比较了。

看到这里，如果容器型别名称让你很烦，采用“型别定义式”不失为一个好办法。在使用容器型别（以及迭代器型别）的任何地方都可以采用这种便捷之道，例

---

[18] 注意，两个 > 之间需加上一个空格，因为 >> 会被编译器视为移位操作符，导致本处语法错误。

[19] 在不支持 default template parameters 的系统中，通常必须这么设定排序准则：

```
set<int,less<int> > coll;
```

如：

```
typedef std::set<int,std::greater<int> > IntSet;
...
IntSet coll
IntSet::iterator pos;
```

“利用区间的起点和终点来构造容器”的构造函数，可以从其它型别的容器中，或是从 array、或是从标准输入装置（standard input）中接受元素来源。详见 6.1.2 节，p144。

非变动性操作（Nonmodifying Operations）

sets 和 multisets 提供常见的非变动性操作，用来查询大小、相互比较。

表 6.21 Sets 和 Multisets 的非变动性操作（Nonmodifying Operations）

操作	效果
`c.size()`	返回容器的大小
`c.empty()`	判断容器大小是否为零。等同于 `size()==0`，但可能更快
`c.max_size()`	返回可容纳的最大元素数量
`c1 == c2`	判断是否 c1 等于 c2
`c1 != c2`	判断是否 c1 不等于 c2。等同于 `!(c1 == c2)`
`c1 < c2`	判断是否 c1 小于 c2
`c1 > c2`	判断是否 c1 大于 c2。等同于 `c2 < c1`
`c1 <= c2`	判断是否 c1 小于等于 c2。等同于 `!(c2 < c1)`
`c1 >= c2`	判断是否 c1 大于等于 c2。等同于 `!(c1 < c2)`

元素比较动作只能用于型别相同的容器。换言之，元素和排序准则必须有相同的型别，否则编译时期会产生型别方面的错误。

```
std::set<float> c1;    // sorting criterion: std::less<>
std::set<float,std::greater<float> > c2;
...
if (c1 == c2) {        // ERROR: different types
   ...
}
```

比较动作系以“字典（lexicographical）顺序”来检查某个容器是否小于另一个容器。如果要比较不同型别（拥有不同排序准则）的容器，你必须采用 p356, 9.5.4 节的 “比较算法（comparing algorithms）”。

特殊的搜寻函数（Special Search Operations）

sets 和 multisets 在元素快速搜寻方面有优化设计，所以提供了特殊的搜寻函数，如表 6.22。这些函数是同名的 STL 算法的特殊版本。面对 sets 和 multisets，你应该优先采用这些优化算法，如此可获得对数复杂度，而非 STL 算法的线性复杂度。举个例子，在 1,000 个元素中搜寻，平均 10 次比较之后便可得出结果，如果是线性复杂度，平均 500 次比较才能有结果（参见 2.3 节, p21）。

表 6.22 Sets 和 Multisets 的搜寻操作函数

操作	效果
`count(elem)`	返回“元素值为 `elem`”的元素个数
`find(elem)`	返回“元素值为 `elem`”的第一个元素，如果找不到就返回 `end()`
`lower_bound(elem)`	返回 `elem` 的第一个可安插位置，也就是“元素值 `>= elem`”的第一个元素位置
`upper_bound(elem)`	返回 `elem` 的最后一个可安插位置，也就是“元素值 `> elem`”的第一个元素位置
`equal_range(elem)`	返回 `elem` 可安插的第一个位置和最后一个位置，也就是“元素值 `== elem`”的元素区间

成员函数 `find()` 搜寻出与参数值相同的第一个元素，并返回一个迭代器，指向该位置。如果没找到这样的元素，就返回容器的 `end()`。

`lower_bound()` 和 `upper_bound()` 分别返回元素可安插点的第一个和最后一个位置。换言之，`lower_bound()` 返回大于等于参数值的第一个元素所处位置，`upper_bound()` 返回大于参数值的第一个元素位置。`equal_range()` 则是将 `lower_bound()` 和 `upper_bound()` 的返回值做成一个 `pair` 返回（型别 `pair` 在 p33, 4.1 节介绍），所以它返回的是“与参数值相等”的元素所形成的区间。如果 `lower_bound()` 或 “`equal_range()` 的第一值”等于“`equal_range()` 的第二值”或 `upper_bound()`，则此 sets 或 multisets 内不存在相同数值的元素。这是当然啦，同值区间中至少也得包含一个元素嘛！

下面的例子说明如何使用 `lower_bound()`, `upper_bound()` 和 `equal_range()`：

```
// cont/set2.cpp

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

```
int main ()
{
  set<int> c;

  c.insert(1);
  c.insert(2);
  c.insert(4);
  c.insert(5);
  c.insert(6);

  cout << "lower_bound(3): " << *c.lower_bound(3) << endl;
  cout << "upper_bound(3): " << *c.upper_bound(3) << endl;
  cout << "equal_range(3): " << *c.equal_range(3).first << " "
                             << *c.equal_range(3).second << endl;

  cout << endl;
  cout << "lower_bound(5): " << *c.lower_bound(5) << endl;
  cout << "upper_bound(5): " << *c.upper_bound(5) << endl;
  cout << "equal_range(5): " << *c.equal_range(5).first << " "
                             << *c.equal_range(5).second << endl;
}
```

程序输入如下：

```
lower_bound(3): 4
upper_bound(3): 4
equal_range(3): 4 4

lower_bound(5): 5
upper_bound(5): 6
equal_range(5): 5 6
```

上例如果使用 multisets 而不是 sets，程序输出相同。

赋值（Assignments）

sets 和 multisets 只提供所有容器都提供的基本赋值操作（表 6.23），详见 p147。

这些操作函数中，赋值操作的两端容器必须具有相同型别。尽管“比较准则”本身可能不同，但其型别必须相同。p191 列出一个“排序准则不同，但型别相同”的例子。如果准则不同，准则本身也会被赋值（assigned）或交换（swapped）。

表 6.23 Sets 和 Multisets 的赋值操作

操作	效果
`c1 = c2`	将 c2 中所有元素赋值给 c1
`c1.swap(c2)`	将 c1 和 c2 的元素互换
`swap(c1,c2)`	同上。此为全局函数

迭代器相关函数（Iterator Functions）

sets 和 multisets 不提供元素直接存取，所以只能采用迭代器。Sets 和 multisets 也提供了一些常见的迭代器函数（表 6.24）。

表 6.24 Sets 和 multisets 的迭代器相关操作函数

操作	效果
`c.begin()`	返回一个双向迭代器（将元素视为常数），指向第一元素
`c.end()`	返回一个双向迭代器（将元素视为常数），指向最后元素的下一位置
`c.rbegin()`	返回一个逆向迭代器，指向逆向遍历时的第一个元素
`c.rend()`	返回一个逆向迭代器，指向逆向遍历时的最后元素的下一位置

和其它所有关联式容器类似，这里的迭代器是双向迭代器（参见 p255, 7.2.4 节）。所以，对于只能用于随机存取迭代器的 STL 算法（例如排序或随机乱序 *random shuffling* 算法），sets 和 multisets 就无福消受了。

更重要的是，对迭代器操作而言，所有元素都被视为常数，这可确保你不会人为改变元素值，从而打乱既定顺序。然而这也使得你无法对 sets 或 multisets 元素调用任何变动性算法（modifying algorithms）。例如你不能对它们调用 `remove()`，因为 `remove()`算法实际上是以一个参数值覆盖被移除的元素（详细讨论见 p115, 5.6.2 节）。如果要移除 sets 和 multisets 的元素，你只能使用它们所提供的成员函数。

元素的安插（Inserting）和移除（Removing）

表 6.25 列出 sets 和 multisets 的元素安插和删除函数。

按 STL 惯例，你必须保证参数有效：迭代器必须指向有效位置、序列起点不能位于终点之后、不能从空容器中删除元素。

安插和移除多个元素时，单一调用（一次处理）比多次调用（逐一处理）快得多。

表 6.25 Sets 和 Multisets 的元素安插和移除

操作	效果
c.insert(elem)	安插一份 elem 副本，返回新元素位置（不论是否成功——对 sets 而言）
c.insert(pos,elem)	安插一份 elem 副本，返回新元素位置（pos 是个提示，指出安插操作的搜寻起点。如果提示恰当，可大大加快速度）
c.insert(beg,end)	将区间[beg;end]内所有元素的副本安插到 c（无返回值）
c.erase(elem)	移除“与 elem 相等”的所有元素，返回被移除的元素个数
c.erase(pos)	移除迭代器 pos 所指位置上的元素，无返回值
c.erase(beg,end)	移除区间[beg;end]内的所有元素，无返回值
c.clear()	移除全部元素，将整个容器清空

注意，安插函数的返回值型别不尽相同：

- sets 提供如下接口：

```
pair<iterator,bool> insert(const value_type& elem);
iterator            insert(iterator pos_hint,
                           const value_type& elem);
```

- multisets 提供如下接口：

```
iterator insert(const value_type& elem);
iterator insert(iterator pos_hint,
                const value_type& elem);
```

返回值型别不同的原因是：multisets 允许元素重复，而 sets 不允许。因此如果将某元素安插至一个 set 内，而该 set 已经内含同值元素，则安插操作将告失败。所以 set 的返回值型别是以 pair 组织起来的两个值（关于 pair 详见 p33, 4.1 节）：

1. pair 结构中的 second 成员表示安插是否成功。
2. pair 结构中的 first 成员返回新元素的位置，或返回现存的同值元素的位置。

其它任何情况下，函数都返回新元素位置（如果 sets 已经内含同值元素，则返回同值元素的位置）。

以下例子把数值 3.3 的元素安插到 set c 中，藉此说明如何使用上述接口：

```
std::set<double> c;
...
if (c.insert(3.3).second) {
    std::cout << "3.3 inserted" << std::endl;
}
else {
    std::cout << "3.3 already exists" << std::endl;
}
```

如果你还想处理新位置或旧位置，程序代码得更复杂些：

```
// define variable for return value of insert()
std::pair<std::set<float>::iterator,bool> status;

// insert value and assign return value
status = c.insert(value);

// process return value
if (status.second) {
    std::cout << value << " inserted as element "
}
else {
    std::cout << value << " already exists as element "
}
std::cout << std::distance(c.begin(),status.first) + 1
          << std::endl;
```

对于此一序列的两次调用结果可能如下：

```
8.9 inserted as element 4
7.7 already exists as element 3
```

注意，所有拥有“位置提示参数”的安插函数，其返回值型别都一样，不论是 sets 或 multisets，这些函数都只返回一个迭代器。这些函数的效果与“无位置提示参数”的函数一样，只不过性能略有差异。你可以传进一个迭代器，该位置将作为一个提示，用来提升性能。事实上如果被安插元素的位置恰好紧贴于提示位置之后，那么时间复杂度就会从“对数”一变而为“分期摊还常数（amortized constant）”（复杂度的介绍请见 p21, 2.3 节）。和“单参数安插函数”不同的是，带有“额外提示位置”的若干安插函数，都具有相同的返回值型别，这就确保你至少有了一个通用型安插函数，在各种容器中有共同接口。事实上通用型安插迭代器（general inserters）就是靠这个接口的支持才得以实现的。

要删除“与某值相等”的元素，只需调用 erase()：

```
std::set<Elem> coll;
...
// remove all elements with passed value
coll.erase(value);
```

和 lists 不同的是，erase() 并非取名为 remove()（后者的讨论请见 p170）。是的，它的行为不同，它返回被删除元素的个数，用在 sets 身上，返回值非 0 即 1。

如果 multisets 内含重复元素，你不能使用 erase() 来删除这些重复元素中的第一个。你可以这么做：

```
std::multiset<Elem> coll;
...
// remove first element with passed value
std::multiset<Elem>::iterator pos;
pos = coll.find (elem);
if (pos != coll.end()) {
   coll.erase(pos);
}
```

这里应该采用成员函数 find()，而非 STL 算法 find()，因为前者速度更快（参见 p154 的例子）。

注意，还有一个返回值不一致的情况。作用于序列式容器和关联式容器的 erase() 函数，其返回值有以下不同：

1. 序列式容器提供下面的 erase() 成员函数：
```
iterator erase(iterator pos);
iterator erase(iterator beg, iterator end);
```
2. 关联式容器提供下面的 erase() 成员函数：
```
void erase(iterator pos);
void erase(iterator beg, iterator end);
```

存在这种差别，完全是为了性能。在关联式容器中“搜寻某元素并返回后继元素”可能颇为耗时，因为这种容器的底部是以二叉树完成，所以如果你想编写对所有容器都适用的程序代码，你必须忽略返回值。

6.5.3 异常处理（Exception Handling）

sets 和 multisets 是“以节点（nodes）为基础”的容器。如果节点构造失败，容器仍保持原样。此外，由于析构函数通常并不抛出异常，所以节点的移除不可能失败。

然而，面对多重元素安插操作，“保持元素次序”这一条件会造成“异常抛出时能够完全复原”这一需求变得不切实际。因此只有“单一元素安插操作”才支持“成功，否则无效”的操作原则。至于“多元素删除操作”总是能够成功。如果排序准则之复制/赋值操作会抛出异常，则 swap() 也会抛出异常。

STL 异常处理的一般性讨论见于 p139, 5.11.2 节。p248 的 6.10.10 节列出“异常出现时会给予特殊保证”的所有容器操作函数。

6.5.4 Sets 和 Multisets 运用实例

以下程序展示 sets 的一些能力[20]：

```
// cont/set1.cpp

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

int main()
{
    /* type of the collection:sets:
    * - no duplicates
    * - elements are integral values
    * - descending order
    */
    typedef set<int,greater<int> > IntSet;

    IntSet coll1; // empty set container

    // insert elements in random order
    coll1.insert(4);
    coll1.insert(3);
    coll1.insert(5);
    coll1.insert(1);
    coll1.insert(6);
    coll1.insert(2);
    coll1.insert(5);

    // iterate over all elements and print them
    IntSet::iterator pos;
    for (pos = coll1.begin(); pos != coll1.end(); ++pos) {
        cout << *pos << ' ';
    }
    cout << endl;
```

[20] distance()的定义已有改变。早期 STL 版本中，你必须含入 distance.hpp（见 p263）

```
    // insert 4 again and process return value
    pair<IntSet::iterator,bool> status = coll1.insert(4);
    if (status.second) {
        cout << "4 inserted as element "
             << distance(coll1.begin(),status.first) + 1
             << endl;
    }
    else {
        cout << "4 already exists" << endl;
    }

    // assign elements to another set with ascending order
    set<int> coll2(coll1.begin(),
                   coll1.end());

    // print all elements of the copy
    copy (coll2.begin(), coll2.end(),
          ostream_iterator<int>(cout," "));
    cout << endl;

    // remove all elements up to element with value 3
    coll2.erase (coll2.begin(), coll2.find(3));

    // remove all elements with value 5
    int num;
    num = coll2.erase (5);
    cout << num << " element(s) removed" << endl;

    // print all elements
    copy (coll2.begin(), coll2.end(),
          ostream_iterator<int>(cout," "));
    cout << endl;
}
```

首先，以下的型别定义：

```
typedef set<int,greater<int> > IntSet;
```

定义了一个 set，其中容纳降序（递减）排列的 ints。产生一个空的 sets 之后，首先利用 insert()安插数个元素：

```
IntSet coll1;

coll1.insert(4);
...
```

注意数值为 5 的元素被安插两次，但第二次安插操作会被程序忽略，因为 sets 不允许数值重复的元素。

打印所有元素后，程序再次安插元素 4。这次按照 p183 的方法来处理 insert() 返回值。

以下语句：

```
set<int> coll2(coll1.begin(),coll1.end());
```

产生一个新的 set，其中容纳升序（递增）排列的 ints，并以原本那个 sets 的元素作为初值[21]。

两个容器有不同的排序准则，所以它们的型别不同，不能直接相互赋值或比较。但只要元素型别相同或彼此可以转型，你就可以使用某些“有能力处理不同容器型别”的算法来达到目的。

以下语句：

```
coll2.erase (coll2.begin(), coll2.find(3));
```

移除了数值为 3 的元素之前的所有元素。注意数值为 3 的元素位于序列尾端，所以没被移除。

最后，所有数值为 5 的元素都被移除：

```
int num;
num = coll2.erase (5);
cout << num << " element(s) removed" << endl;
```

程序输出如下：

```
6 5 4 3 2 1
4 already exists
1 2 3 4 5 6
1 element(s) removed
3 4 6
```

---

[21] 这行语句需要两个语言新性质：member templates 和 default template arguments。如果系统不支持，你必须改成这样：

```
set<int,less<int> > coll2;
copy (coll1.begin(), coll1.end(),
      inserter(coll2,coll2.begin()));
```

对于 multisets，上一个程序需要些微改变，并产生不同结果：

```
// cont/mset1.cpp

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

int main()
{
    /* type of the collection: sets
    * - duplicates allowed
    * - elements are integral values
    * - descending order
    */
    typedef multiset<int,greater<int> > IntSet;

    IntSet coll1; // empty multiset container

    // insert elements in random order
    coll1.insert(4);
    coll1.insert(3);
    coll1.insert(5);
    coll1.insert(1);
    coll1.insert(6);
    coll1.insert(2);
    coll1.insert(5);

    // iterate over all elements and print them
    IntSet::iterator pos;
    for (pos = coll1.begin(); pos != coll1.end(); ++pos) {
        cout << *pos << ' ';
    }
    cout << endl;

    // insert 4 again and process return value
    IntSet::iterator ipos = coll1.insert(4);
    cout << "4 inserted as element "
         << distance(coll1.begin(),ipos) + 1
         << endl;
```

```
    // assign elements to another multiset with ascending order
    multiset<int> coll2(coll1.begin(),
                        coll1.end());

    // print all elements of the copy
    copy (coll2.begin(), coll2.end(),
    ostream_iterator<int>(cout," "));
    cout << endl;

    // remove all elements up to element with value 3
    coll2.erase (coll2.begin(), coll2.find(3));

    // remove all elements with value 5
    int num;
    num = coll2.erase (5);
    cout << num << " element(s) removed" << endl;

    // print all elements
    copy (coll2.begin(), coll2.end(),
          ostream_iterator<int>(cout," "));
    cout << endl;
}
```

这个程序把所有的 set 都改为 multiset，此外 insert()返回值的处理也有所不同：

```
IntSet::iterator ipos = coll1.insert(4);
cout << "4 inserted as element "
     << distance(coll1.begin(),ipos) + 1
     << endl;
```

由于 multisets 可能包含重复元素，所以安插操作只在异常被抛出时才失败。因此，返回值型别只是一个迭代器，指向新元素位置。

程序输出如下：

```
6 5 5 4 3 2 1
4 inserted as element 5
1 2 3 4 4 5 5 6
2 element(s) removed
3 4 4 6
```

6.5.5 执行期指定排序准则

无论是将排序准则作为第二个 template 参数传入，或是采用缺省的排序准则 less<>，通常你都会将排序准则定义为型别的一部分。但有时必须在执行期处理排序准则，或者你可能需要对同一种数据型别采用不同的排序准则。此时你就需要一个“用来表现排序准则”的特殊型别，使你能够在执行期间传递某个准则。以下范例程序说明了这种做法：

```
// cont/setcmp.cpp

#include <iostream>
#include <set>
#include "print.hpp"
using namespace std;

// type for sorting criterion
template <class T>
class RuntimeCmp {
   public:
      enum cmp_mode {normal, reverse};

   private:
      cmp_mode mode;

   public:
      // constructor for sorting criterion
      // - default criterion uses value normal
      RuntimeCmp (cmp_mode m=normal) : mode(m) {
      }

      // comparision of elements
      bool operator() (const T& t1, const T& t2) const {
         return mode == normal ? t1 < t2 : t2 < t1;
      }

      // comparision of sorting criteria
      bool operator== (const RuntimeCmp& rc) {
         return mode == rc.mode;
   }
};

// type of a set that uses this sorting criterion
typedef set<int,RuntimeCmp<int> > IntSet;
```

```
// forward declaration
void fill (IntSet& set);

int main()
{
    // create, fill, and print set with normal element order
    // - uses default sorting criterion
    IntSet coll1;
    fill(coll1);
    PRINT_ELEMENTS (coll1, "coll1: ");

    // create sorting criterion with reverse element order
    RuntimeCmp<int> reverse_order(RuntimeCmp<int>::reverse);

    // create, fill, and print set with reverse element order
    IntSet coll2(reverse_order);
    fill(coll2);
    PRINT_ELEMENTS (coll2, "coll2: ");

    // assign elements AND sorting criterion
    coll1 = coll2;
    coll1.insert(3);
    PRINT_ELEMENTS (coll1, "coll1: ");

    // just to make sure...
    if (coll1.value_comp() == coll2.value_comp()) {
        cout << "coll1 and coll2 have same sorting criterion"
             << endl;
    }
    else {
        cout << "coll1 and coll2 have different sorting criterion"
             << endl;
    }
}

void fill (IntSet& set)
{
    // fill insert elements in random order
    set.insert(4);
    set.insert(7);
    set.insert(5);
    set.insert(1);
    set.insert(6);
    set.insert(2);
    set.insert(5);
}
```

在这个程序中，RuntimeCmp<> 是一个简单的 template，提供“执行期间面对任意型别定义一个排序准则”的泛化能力。其 default 构造函数采用默认值 normal，按升序排序；你也可以将 RuntimeCmp<>::reverse 传递给构造函数，便能按降序排序。

程序输出如下：

```
coll1: 1 2 4 5 6 7
coll2: 7 6 5 4 2 1
coll1: 7 6 5 4 3 2 1
coll1 and coll2 have same sorting criterion
```

注意，coll1 和 coll2 拥有相同型别，该型别即 fill() 函数的参数型别。再请注意，assignment 操作符不仅赋值了元素，也赋值了排序准则（否则任何一个赋值操作岂不会轻易危及排序准则！）。

6.6 Maps 和 Multimaps

Map 和 multimap 将 *key/value* pair（键值/实值 对组）当做元素，进行管理。它们可根据 *key* 的排序准则自动将元素排序。multimaps 允许重复元素，maps 不允许，见图 6.8。

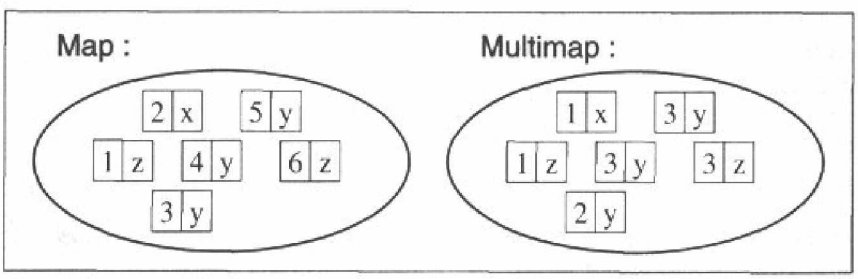

图 6.8 Maps 和 Multimaps

使用 map 和 multimap 之前，你必须先含入头文件 <map>[22]：

```
#include <map>
```

在其中，map 和 multimap 被定义为命名空间 std 内的 class templates：

```
namespace std {
    template <class Key, class T,
              class Compare = less<Key>,
              class Allocator = allocator<pair<const Key,T> > >
    class map;

    template <class Key, class T,
              class Compare = less<Key>,
              class Allocator = allocator<pair<const Key,T> > >
    class multimap;
}
```

第一个 template 参数被当做元素的 *key*，第二个 template 参数被当做元素的 *value*。Map 和 multimap 的元素型别 Key 和 T，必须满足以下两个条件：

22 在早期 STL 中，maps 被定义于<map.h>而 multimaps 被定义于<multimap.h>。

1. *key/value* 必须具备 assignable（可赋值的）和 copyable（可复制的）性质。
2. 对排序准则而言，*key* 必须是 comparable（可比较的）。

第三个 template 参数可有可无，用它来定义排序准则。和 sets 一样，这个排序准则必须定义为 **strict weak ordering**（参见 p176）。元素的次序由它们的 *key* 决定，和 *value* 无关。排序准则也可以用来检查相等性：如果两个元素的 *key* 彼此都不小于对方，则两个元素被视为相等。如果使用者未传入特定排序准则，就使用缺省的 `less` 排序准则——以 `operator<` 来进行比较[23]（`less` 的详细资料请见 p305）。

第四个 template 参数也是可有可无，用它来定义内存模型（详见第 15 章）。缺省的内存模型是 `allocator`，由 C++ 标准程序库提供[24]。

6.6.1 Maps 和 Multimaps 的能力

和所有标准的关联式容器一样，maps/multimpas 通常以平衡二叉树完成，如图 6.9。

标准规范并未明定这一点，但是从 map 和 multimap 各项操作的复杂度可以得出这一结论。典型情况下，set, multisets, map, multimaps 使用相同的内部数据结构。因此你可以把 set 和 multisets 分别视为特殊的 map 和 multimaps，只不过 sets 元素的 *value* 和 *key* 是指同一对象。因此 map 和 multimaps 拥有 set 和 multisets 的所有能力和所有操作函数。当然某些细微差异还是有的：首先，它们的元素是 *key/value* pair，其次，map 可作为关联式数组来运用。

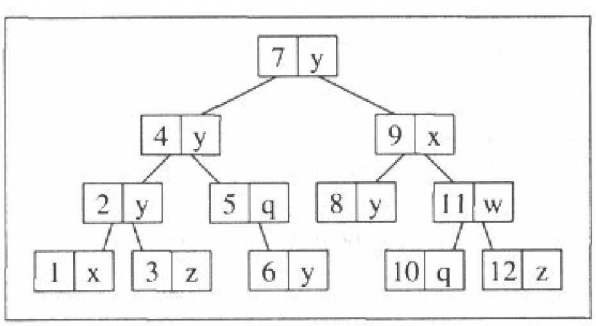

图 6.9 Maps 和 Multimaps 的内部结构

[23] 在不支持 default template parameters 的系统中，第三个参数通常会被省略。

[24] 在不支持 default template parameters 的系统中，第四个参数通常会被省略。

Map 和 multimaps 根据元素的 *key* 自动对元素进行排序。这么一来，根据已知的 *key* 搜寻某个元素时，就能够有很好的性能，而根据已知 *value* 搜寻元素时，性能就很糟糕。“自动排序”这一性质使得 map 和 multimaps 身上有了一条重要的限制：你不可以直接改变元素的 *key*，因为这会破坏正确次序。要修改元素的 *key*，你必须先移除拥有该 *key* 的元素，然后插入拥有新的 *key/value* 的元素（详见 p201）。从迭代器的观点来看，元素的 *key* 是常数。至于元素的 *value* 倒是可以直接修改的，当然，前提是 *value* 并非常数型态。

6.6.2 Maps 和 Multimaps 的操作函数

生成（Create）、复制（Copy）和销毁（Destroy）

表 6.26 列出 maps 和 multimaps 的生成、复制、销毁等各项操作。

表 6.26 Maps 和 Multimaps 的构造函数和析构函数

操作	效果
map `c`	产生一个空的 map/multimap，其中不含任何元素
map `c(op)`	以 op 为排序准则，产生一个空的 map/multimap
map `c1(c2)`	产生某个 map/multimap 的副本，所有元素均被复制
map `c(beg,end)`	以区间 [beg;end] 内的元素产生一个 map/multimap
map `c(beg,end,op)`	以 op 为排序准则，利用 [beg;end] 内的元素生成一个 map/multimap
`c.~`***map***`()`	销毁所有元素，释放内存

其中，***map*** 可为下列型式：

map	效果
`map<Key,Elem>`	一个 map，以 `less<>`（`operator<`）为排序准则
`map<Key,Elem,Op>`	一个 map，以 op 为排序准则
`multimap<Key,Elem>`	一个 multimap，以 `less<>`（`operator<`）为排序准则
`multimap<Key,Elem,Op>`	一个 multimap，以 op 为排序准则

有两种方式可以定义排序准则：

1. 以 template 参数定义。

   例如[25]：

   ```
   std::map<float, std::string, std::greater<float> > coll;
   ```

   这种情况下，排序准则就是型别的一部分。因此型别系统确保“只有排序准则相同的容器才能被合并”。这是比较常见的排序准则指定法。更精确地说，第三参数是排序准则的型别。实际的排序准则是容器所产生的函数对象（*function object*，或称*functor*）。为了产生它，容器构造函数会调用“排序准则型别”的 default 构造函数。p294 有一个“使用者自定之排序准则”的运用实例。

2. 以构造函数参数定义。

   在这种情况下，你可以有一个“排序准则型别”并为它指定不同的排序准则（也就是说让该型别所产生出来的对象（代表一个排序准则）的初值或状态不同）。如果执行期才获得排序准则，而且程序需要用到不同的排序准则（但其数据型别必须相同），此一方式可派上用场。一个典型的例子是在执行期指定“*key* 的型别为 `string`”的排序准则。完整例子见 p213。

如果使用者没有提供特定排序准则，就采用缺省准则——仿函数 `less<>`。`less<>`系透过 `operator<` 对元素进行排序[26]。

你应当做一些型别定义（typedef），从而简化繁琐的型别表达式：

```
typedef std::map<std::string,float,std::greater<std::string> >
                 StringFloatMap;
...
StringFloatMap coll;
```

某些构造函数使用区间起点和终点作为参数，它们可以使用不同型别的容器、数组、标准输入装置（standard input）来进行初始化，详见 6.1.2 节，p144。然而由于元素是 *key/value* pair，因此你必须确定来自源区间的元素型别也是 `pair<key, value >`，或至少可转化成 `pair<key, value>`。

非变动性操作（Nonmodifying Operations）

maps 和 multimaps 提供常见的非变动性操作，用来查询大小、相互比较。如表 6.27。

---

[25] 注意，两个 > 之间需加上一个空格，因为 >> 会被编译器视为移位操作符，导致本处语法错误。

[26] 在不支持 default template parameters 的系统中，通常必须这样设定排序准则：

```
map<float, string, less<float> > coll;
```

表 6.27 Maps 和 Multimaps 的非变动性操作（Nonmodifying Operations）

操作	效果
`c.size()`	返回容器的大小。
`c.empty()`	判断容器大小是否为零。等同于 `size()==0`，但可能更快。
`c.max_size()`	返回可容纳的最大元素数量。
`c1 == c2`	判断是否 `c1` 等于 `c2`。
`c1 != c2`	判断是否 `c1` 不等于 `c2`。等同于 `!(c1 == c2)`。
`c1 < c2`	判断是否 `c1` 小于 `c2`。
`c1 > c2`	判断是否 `c1` 大于 `c2`。等同于 `c2 < c1`。
`c1 <= c2`	判断是否 `c1` 小于等于 `c2`。等同于 `!(c2 < c1)`。
`c1 >= c2`	判断是否 `c1` 大于等于 `c2`。等同于 `!(c1 < c2)`。

元素比较动作只能用于型别相同的容器。换言之，容器的 *key*、*value*、排序准则都必须有相同的型别，否则编译期会产生型别方面的错误。例如：

```
std::map<float,std::string> c1;     // sorting criterion: less<>
std::map<float,std::string,std::greater<float> > c2;
...
if (c1 == c2) { // ERROR: different types
    ...
}
```

比较动作系以“字典（lexicographical）顺序”来检查某个容器是否小于另一个容器（详见 p360）。如果要比较不同型别（拥有不同排序准则）的容器，你必须采用 p356, 9.5.4 节的“比较算法（comparing algorithms）”。

特殊的搜寻动作（Special Search Operations）

就像 set 和 multisets 一样，map 和 multimaps 也提供特殊的搜寻函数，以便利用内部树状结构获取较好的性能，见表 6.28。

成员函数 `find()`用来搜寻拥有某个 *key* 的第一个元素，并返回一个迭代器，指向该位置。如果没找到这样的元素，就返回容器的 `end()`。你不能以 `find()`搜寻拥有某特定 *value* 的元素，你必须改用通用算法如 `find_if()`，或干脆写一个显式循环。下面这个例子便是利用一个简单循环，对拥有特定 *value* 的所有元素进行某项操作：

```
std::multimap<std::string,float> coll;
...
// do something with all elements having a certain value
std::multimap<std::string,float>::iterator pos;
for (pos = coll.begin(); pos != coll.end(); ++pos) {
```

表 6.28 Maps 和 Multimaps 的特殊搜寻操作函数

操作	效果
count(key)	返回“键值等于 key”的元素个数
find(key)	返回“键值等于 key”的第一个元素，找不到就返回 end()
lower_bound(key)	返回“键值为 key”之元素的第一个可安插位置，也就是“键值 >= key”的第一个元素位置
upper_bound(key)	返回“键值为 key”之元素的最后一个可安插位置，也就是“键值 > key”的第一个元素位置
equal_range(key)	返回“键值为 key”之元素的第一个可安插位置和最后一个可安插位置，也就是“键值 == key”的元素区间

```
    if (pos->second == value) {
        do_something();
    }
}
```

当你要以此类循环来移除元素时，请特别当心。因为可能会发生一些意料之外的事。细节详见 p204。如果使用 find_if()算法做类似的搜寻操作，会比写一个循环更复杂，因为你必须提供仿函数（*functor*，亦即函数对象 *function object*），将元素的 *value* 拿来和某个 *value* 比较。详见 p211 实例。

至于 lower_bound(), upper_bound(), equal_range()，其行为和 sets（见 p180）的相应函数十分相似，唯一的不同就是：元素是个 *key/value* pair。

赋值（Assignments）

maps 和 multimaps 只支持所有容器都提供的基本赋值操作（表 6.29），详见 p147。

表 6.29 Maps 和 Multimaps 的赋值（赋值）操作

操作	效果
c1 = c2	将 c2 中所有元素赋值给 c1
c1.swap(c2)	将 c1 和 c2 的元素互换
swap(c1,c2)	同上。此为全局函数

这些操作函数中，赋值操作的两端容器必须具有相同型别。尽管“比较准则”本身可能不同，但其型别必须相同。p213 列出一个“排序准则不同，但型别相同”的例子。如果准则不同，准则本身也会被赋值（assigned）或交换（swapped）。

迭代器函数（Iterator Functions）和元素存取（Element Access）

Map 和 multimaps 不支持元素直接存取，因此元素的存取通常是经由迭代器进行。不过有个例外：map 提供 subscript（下标）操作符，可直接存取元素，详见 6.6.3 节，p205。表 6.30 列出 maps 和 multimaps 所支持的迭代器相关函数。

表 6.30 Maps 和 Multimaps 的迭代器相关操作函数

操作	效果
`c.begin()`	返回一个双向迭代器（*key* 被视为常数），指向第一元素
`c.end()`	返回一个双向迭代器（*key* 被视为常数），指向最后元素的下一位置
`c.rbegin()`	返回一个逆向迭代器，指向逆向遍历时的第一个元素
`c.rend()`	返回一个逆向迭代器，指向逆向遍历时的最后元素的下一位置

和其它所有关联式容器类似，这里的迭代器是双向迭代器（参见 p255，7.2.4 节）。所以，对于只能用于随机存取迭代器的 STL 算法（例如排序或随机乱序算法 *random shuffling*），maps 和 multimaps 就无福消受了。

更重要的是，在 map 和 multimap 中，所有元素的 *key* 都被视为常数。因此元素的实质型别是 `pair<const key,T>`。这个限制是为了确保你不会因为变更元素的 *key* 而破坏业已排好的元素次序。所以你不能针对 map 或 multimap 调用任何变动性算法（modifying algorithms）。例如你不能对它们调用 `remove()`，因为 `remove()` 算法实际上是以一个参数值覆盖被移除的元素（详细讨论见 p115，5.6.2 节）。如果要移除 map 和 multimaps 的元素，你只能使用它们所提供的成员函数。

下面是 map 迭代器运用实例：

```
std::map<std::string,float> coll;
...
std::map<std::string,float>::iterator pos;
for (pos = coll.begin(); pos != coll.end(); ++pos) {
     std::cout << "key: " << pos->first << "\t"
               << "value: " << pos->second << std::endl;
}
```

其中迭代器 pos 遍历了 string/float pair 所组成的序列。以下表达式：

```
pos->first
```

获得元素的 *key*，而以下表达式：

```
pos->second
```

获得元素的 *value*[27]。

如果你尝试改变元素的 *key*，会引发错误：

```
pos->first = "hello"; // ERROR at compile time
```

不过如果 *value* 本身的型别并非 const，则改变 *value* 没有问题：

```
pos->second = 13.5; // OK
```

如果你一定得改变元素的 *key*，只有一条路：以一个“*value* 相同”的新元素替换掉旧元素。下面是个泛化函数：

```
// cont/newkey.hpp

namespace MyLib {
    template <class Cont>
    inline
    bool replace_key (Cont& c,
                      const typename Cont::key_type& old_key,
                      const typename Cont::key_type& new_key)
    {
        typename Cont::iterator pos;
        pos = c.find(old_key);

        if (pos != c.end()) {
            // insert new element with value of old element
            c.insert(typename Cont::value_type(new_key,
                                               pos->second));
            // remove old element
            c.erase(pos);
            return true;
        }
        else {
            // key not found
            return false;
        }
    }
}
```

关于 insert()和 erase()成员函数，请见下一节讨论。

[27] pos->first 是(*pos).first 的简写形式。有些程序库只支持后一种形式。

这个泛型函数的用法很简单，把旧的 *key* 和新的 *key* 传递进去就行。例如：

```
std::map<std::string,float> coll;
...
MyLib::replace_key(coll,"old key","new key");
```

如果你面对的是 multimaps，情况也一样。

注意，maps 提供了一种非常方便的手法让你改变元素的 *key*。只需如此这般：

```
// insert new element with value of old element
coll["new_key"] = coll["old_key"];
// remove old element
coll.erase("old_key");
```

关于 maps 的 subscript（下标）操作符使用细节，详见 6.6.3 节, p205。

元素的安插（Inserting）和移除（Removing）

表 6.31 列出 maps 和 multimaps 所支持的元素安插和删除函数。

表 6.31 Maps 和 Multimaps 的元素安插和移除

操作	效果
`c.insert(elem)`	安插一份 `elem` 副本，返回新元素位置（不论是否成功——对 maps 而言）
`c.insert(pos,elem)`	安插一份 `elem` 副本，返回新元素位置（`pos` 是个提示，指出安插操作的搜寻起点。如果提示恰当，可大大加快速度）
`c.insert(beg,end)`	将区间 `[beg;end)` 内所有元素的副本安插到 `c`（无返回值）
`c.erase(elem)`	移除“实值（*value*）与 `elem` 相等”的所有元素，返回被移除的元素个数
`c.erase(pos)`	移除迭代器 `pos` 所指位置上的元素，无返回值
`c.erase(beg,end)`	移除区间 `[beg;end)` 内的所有元素，无返回值
`c.clear()`	移除全部元素，将整个容器清空

p182 之中关于 set 和 multisets 的说明，此处依然适用。上述操作函数的返回值型别有些差异，其情况与 set 和 multisets 的情况完全相同。当然，这里的元素是 *key/value* pair。所以这里的用法更复杂些。

安插一个 *key/value* pair 的时候，你一定要记住，在 map 和 multimaps 内部，*key* 被视为常数。你要不就提供正确型别，要不就得提供隐式或显式型别转换。有三个不同的方法可以将 *value* 传入 map：

1. 运用 value_type

为了避免隐式型别转换，你可以利用 value_type 明白传递正确型别。value_type 是容器本身提供的型别定义。例如：

```
std::map<std::string,float> coll;
...
coll.insert(std::map<std::string,float>::value_type("otto",
                                                    22.3));
```

2. 运用 pair<>

另一个作法是直接运用 pair<>。例如：

```
std::map<std::string,float> coll;
...
// use implicit conversion:
coll.insert(std::pair<std::string,float>("otto",22.3));
// use no implicit conversion:
coll.insert(std::pair<const std::string,float>("otto",22.3));
```

上述第一个 insert()语句内的型别并不正确，所以会被转换成真正的元素型别。为了做到这一点，insert()成员函数被定义为 member template[28]。

3. 运用 make_pair()

最方便的办法是运用 make_pair()函数（详见 p36）。这个函数根据传入的两个参数构造出一个 pair 对象：

```
std::map<std::string,float> coll;
...
coll.insert(std::make_pair("otto",22.3));
```

和作法 2 一样，也是利用 member template insert()来执行必要的型别转换。

下面是个简单例子，在 map 中安插一个元素，然后检查是否成功：

```
std::map<std::string,float> coll;
...
if (coll.insert(std::make_pair("otto",22.3)).second) {
    std::cout << "OK, could insert otto/22.3" << std::endl;
}
else {
    std::cout << "Oops, could not insert otto/22.3 "
              << "(key otto already exists)" << std::endl;
}
```

[28] 如果你的系统不支持 member template，你必须传递型别正确的元素，通常必须因此进行显式型别转换（explicit conversions）。

关于 insert()返回值的讨论，请见 p182，那儿有更多例子，也适用于 maps。注意此处仍然透过 map 的 subscript(下标)操作符提供较为方便的元素安插和设定操作。这一点将在 6.6.3 节, p205 讨论。

如果要移除“拥有某个 *value*”的元素，调用 erase()即可办到：

```
std::map<std::string,float> coll;
...
// remove all elements with the passed key
coll.erase(key);
```

erase()返回移除元素的个数。对 maps 而言其返回值非 0 即 1。

如果 multimap 内含重复元素，你不能使用 erase()来删除这些重复元素中的第一个。你可以这么做：

```
typedef std::multimap<std::string,float> StringFloatMMap;
StringFloatMMap coll;
...
// remove first element with passed key
StringFloatMMap::iterator pos;
pos = coll.find(key);
if (pos != coll.end()) {
    coll.erase(pos);
}
```

这里应该采用成员函数 find()，而非 STL 算法 find()，因为前者速度更快(参见 p154 的例子)。然而你不能使用成员函数 find()来移除“拥有某个 *value*（而非某个 *key*）”的元素。详细讨论请见 p198。

移除元素时，当心发生意外状况。当你移除迭代器所指对象时，有一个很大的危险，看看这个例子：

```
typedef std::map<std::string,float> StringFloatMap;
StringFloatMap coll;
StringFloatMap::iterator pos;
...
for (pos = coll.begin(); pos != coll.end(); ++pos) {
    if (pos->second == value) {
        coll.erase(pos); // RUNTIME ERROR !!!
    }
}
```

对 pos 所指元素实施 erase()，会使 pos 不再成为一个有效的 coll 迭代器。如果此后你未对 pos 重新设值就径行使用 pos，前途未卜！事实上只要一个 ++pos 操作就会导致未定义的行为。

如果 erase()总是返回下一元素的位置，那就好办了：

```
typedef std::map<std::string,float> StringFloatMap;
StringFloatMap coll;
StringFloatMap::iterator pos;
...
for (pos = coll.begin(); pos != coll.end(); ) {
   if (pos->second == value) {
      pos = coll.erase(pos); // would be fine, but COMPILE TIME ERROR
   }
   else {
      ++pos;
   }
}
```

可惜 STL 设计过程中否决了这种想法，因为万一用户并不需要这一特性，就会耗费不必要的执行时间。我个人不太赞成这项决定，因为这样一来代码会变得更复杂，更容易出错，长期而言，恐怕得不偿失！

下面是移除“迭代器所指元素”的正确作法：

```
typedef std::map<std::string,float> StringFloatMap;
StringFloatMap coll;
StringFloatMap::iterator pos;
...
// remove all elements having a certain value
for (pos = coll.begin(); pos != coll.end(); ) {
   if (pos->second == value) {
      coll.erase(pos++);
   }
   else {
      ++pos;
   }
}
```

注意，pos++ 会将 pos 移向下一元素，但返回其原始值（指向原位置）的一个副本。因此，当 erase()被调用，pos 已经不再指向那个即将被移除的元素了。

6.6.3 将 Maps 视为关联式数组（Associated Arrays）

通常，关联式容器并不提供元素的直接存取，你必须依靠迭代器。不过 maps 是个例外。Non-const maps 提供下标操作符，支持元素的直接存取，如表 6.32。

不过，下标操作符的索引值并非元素整数位置，而是元素的 *key*。也就是说，索引可以是任意型别，而非局限为整数型别。这种接口正是我们所说的关联式数组（associative array）。

表 6.32 Maps 的直接元素存取（透过 operator[]）

操作	效果
m[key]	返回一个 reference，指向键值为 key 的元素。如果该元素尚未存在，就安插该元素

和一般数组之间的区别还不仅仅在于索引型别。其它的区别包括：你不可能用上一个错误索引。如果你使用某个 *key* 作为索引，而容器之中尚未存在对应元素，那么就会自动安插该元素。新元素的 *value* 由 default 构造函数构造。如果元素的 *value* 型别没有提供 default 构造函数，你就没这个福分了。再次提醒你，所有基本数据型别都提供有 default 构造函数，以零为初值（见 p14）。

关联式数组的行为方式可说是毁誉参半：

- 优点是你可以透过更方便的接口对着 map 安插新元素。例如：

```
std::map<std::string,float> coll; // empty collection
/* insert "otto"/7.7 as key/value pair
* - first it inserts "otto"/float()
* - then it assigns 7.7
*/
coll["otto"] = 7.7;
```

其中的语句：

```
coll["otto"] = 7.7;
```

处理如下：

1. 处理 coll["otto"]：
   - 如果存在键值为 "otto" 的元素，以上式子返回该元素的 reference。
   - 如果没有任何元素的键值是 "otto"，以上式子便为 map 自动安插一个新元素，键值 *key* 为 "otto"，实值 *value* 则以 default 构造函数完成，并返回一个 reference 指向新元素。
2. 将 7.7 赋值给 *value*：
   - 紧接着，将 7.7 赋值给上述刚刚诞生的新元素。

这样，map 之内就包含了一个键值（*key*）为 "otto" 的元素，其实值（*value*）为 7.7。

- 缺点是你可能会不小心误置新元素。例如下面的语句可能会做出一些意想不到的事情：

```
std::cout << coll["ottto"];
```

它会安插一个键值为 "otto" 的新元素，然后打印其实值，缺省情况下是 0。然而，按道理它应该产生一条错误信息，告诉你你把 "otto" 拼写错了。

同时亦请注意，这种元素安插方式比一般的 maps 安插方式来得慢，p202 曾经谈过这个主题。原因是新元素必须先使用 default 构造函数将实值（*value*）初始化，而这个初值马上又被真正的 *value* 给覆盖了。

6.6.4 异常处理（Exception Handling）

就异常处理而言，Maps 和 multimaps 的行为与 sets 和 multisets 一样。参见 p185。

6.6.5 Maps 和 Multimaps 运用实例

将 Map 当做关联式数组

下面这个例子将 map 当成一个关联式数组来使用，用来反映股票行情。元素的键值（*key*）是股票名称，实值（*value*）是股票价格：

```
// cont/map1.cpp

#include <iostream>
#include <map>
#include <string>
using namespace std;

int main()
{
    /* create map / associative array
    * - keys are strings
    * - values are floats
    */
    typedef map<string,float> StringFloatMap;

    StringFloatMap stocks; // create empty container

    // insert some elements
    stocks["BASF"] = 369.50;
```

```
    stocks["VW"] = 413.50;
    stocks["Daimler"] = 819.00;
    stocks["BMW"] = 834.00;
    stocks["Siemens"] = 842.20;

    // print all elements
    StringFloatMap::iterator pos;
    for (pos = stocks.begin(); pos != stocks.end(); ++pos) {
        cout << "stock: " << pos->first << "\t"
             << "price: " << pos->second << endl;
    }
    cout << endl;

    // boom (all prices doubled)
    for (pos = stocks.begin(); pos != stocks.end(); ++pos) {
        pos->second *= 2;
    }

    // print all elements
    for (pos = stocks.begin(); pos != stocks.end(); ++pos) {
    cout << "stock: " << pos->first << "\t"
        << "price: " << pos->second << endl;
    }
    cout << endl;

    /* rename key from "VW" to "Volkswagen"
    * - only provided by exchanging element
    */
    stocks["Volkswagen"] = stocks["VW"];
    stocks.erase("VW");

    // print all elements
    for (pos = stocks.begin(); pos != stocks.end(); ++pos) {
    cout << "stock: " << pos->first << "\t"
        << "price: " << pos->second << endl;
    }
}
```

程序输出如下:

```
stock: BASF price: 369.5
stock: BMW price: 834
stock: Daimler price: 819
stock: Siemens price: 842.2
stock: VW price: 413.5

stock: BASF price: 739
stock: BMW price: 1668
stock: Daimler price: 1638
stock: Siemens price: 1684.4
stock: VW price: 827

stock: BASF price: 739
stock: BMW price: 1668
stock: Daimler price: 1638
stock: Siemens price: 1684.4
stock: Volkswagen price: 827
```

将 Multimap 当做字典

下面例子展示如何将 multimap 当成一个字典来使用:

```
// cont/mmap1.cpp

#include <iostream>
#include <map>
#include <string>
#include <iomanip>
using namespace std;

int main()
{
    // define multimap type as string/string dictionary
    typedef multimap<string,string> StrStrMMap;

    // create empty dictionary
    StrStrMMap dict;

    // insert some elements in random order
    dict.insert(make_pair("day","Tag"));
```

```
    dict.insert(make_pair("strange","fremd"));
    dict.insert(make_pair("car","Auto"));
    dict.insert(make_pair("smart","elegant"));
    dict.insert(make_pair("trait","Merkmal"));
    dict.insert(make_pair("strange","seltsam"));
    dict.insert(make_pair("smart","raffiniert"));
    dict.insert(make_pair("smart","klug"));
    dict.insert(make_pair("clever","raffiniert"));

    // print all elements
    StrStrMMap::iterator pos;
    cout.setf (ios::left, ios::adjustfield);
    cout << ' ' << setw(10) << "english "
         << "german " << endl;
    cout << setfill('-') << setw(20) << ""
         << setfill(' ') << endl;
    for (pos = dict.begin(); pos != dict.end(); ++pos) {
         cout << ' ' << setw(10) << pos->first.c_str()
              << pos->second << endl;
    }
    cout << endl;

    // print all values for key "smart"
    string word("smart");
    cout << word << ": " << endl;
    for (pos = dict.lower_bound(word);
         pos != dict.upper_bound(word); ++pos) {
         cout << "      " << pos->second << endl;
    }

    // print all keys for value "raffiniert"
    word = ("raffiniert");
    cout << word << ": " << endl;
    for (pos = dict.begin(); pos != dict.end(); ++pos) {
         if (pos->second == word) {
              cout << " " << pos->first << endl;
         }
    }
}
```

程序输出如下:

```
  english    german
-------------------------
  car        Auto
  clever     raffiniert
  day        Tag
  smart      elegant
  smart      raffiniert
  smart      klug
  strange    fremd
  strange    seltsam
  trait      Merkmal

smart:
    elegant
    raffiniert
    klug
raffiniert:
    clever
    smart
```

搜寻具有某特定实值（*values*）的元素

下面的例子展示如何使用全局的 `find_if()`算法来搜寻具有某特定 *value* 的元素:

```
// cont/mapfind.cpp

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

/* function object to check the value of a map element
*/
template <class K, class V>
class value_equals {
   private:
   V value;

   public:
   // constructor (initialize value to compare with)
```

```
    value_equals (const V& v)
      : value(v) {
    }

    // comparison
    bool operator() (pair<const K, V> elem) {
    return elem.second == value;
    }
};

int main()
{
    typedef map<float,float> FloatFloatMap;
    FloatFloatMap coll;
    FloatFloatMap::iterator pos;

    // fill container
    coll[1]=7;
    coll[2]=4;
    coll[3]=2;
    coll[4]=3;
    coll[5]=6;
    coll[6]=1;
    coll[7]=3;

    // search an element with key 3.0
    pos = coll.find(3.0); // logarithmic complexity
    if (pos != coll.end()) {
       cout << pos->first << ": "
            << pos->second << endl;
    }

    // search an element with value 3.0
    pos = find_if(coll.begin(),coll.end(), // linear complexity
                  value_equals<float,float>(3.0));
    if (pos != coll.end()) {
         cout << pos->first << ": "
              << pos->second << endl;
    }
}
```

程序输出如下：

```
3: 2
4: 3
```

6.6.6 综合实例：运用 Maps, Strings 并于执行期指定排序准则

这里再示范一个例子。此例针对高级程序员而非 STL 初学者。你可以把它视为展现 STL 威力与障碍的一个范例。更明确地说，这个例子展现了以下技巧：

- 如何使用 maps
- 如何撰写和使用仿函数（functor，或名 function object）
- 如何在执行期定义排序准则
- 如何在“不在乎大小写”的情况下比较字符串（strings）

```
// cont/mapcmp.cpp

#include <iostream>
#include <iomanip>
#include <map>
#include <string>
#include <algorithm>
using namespace std;

/* function object to compare strings
* - allows you to set the comparison criterion at runtime
* - allows you to compare case insensitive
*/
class RuntimeStringCmp {
  public:
    // constants for the comparison criterion
    enum cmp_mode {normal, nocase};

  private:
    // actual comparison mode
    const cmp_mode mode;

    // auxiliary function to compare case insensitive
    static bool nocase_compare (char c1, char c2)
    {
       return toupper(c1) < toupper(c2);
    }
```

```
  public:
    // constructor: initializes the comparison criterion
    RuntimeStringCmp (cmp_mode m=normal) : mode(m) {
    }

    // the comparison
    bool operator() (const string& s1, const string& s2) const {
        if (mode == normal) {
            return s1 < s2;
        }
        else {
            return lexicographical_compare(s1.begin(), s1.end(),
                                           s2.begin(), s2.end(),
                                           nocase_compare);
        }
    }
};

/* container type:
* - map with
* -string keys
* -string values
* - the special comparison object type
*/
typedef map<string, string, RuntimeStringCmp> StringStringMap;

// function that fills and prints such containers
void fillAndPrint(StringStringMap& coll);

int main()
{
    // create a container with the default comparison criterion
    StringStringMap coll1;
    fillAndPrint(coll1);
```

```
    // create an object for case-insensitive comparisons
    RuntimeStringCmp ignorecase(RuntimeStringCmp::nocase);

    // create a container with the case-insensitive
    // comparisons criterion
    StringStringMap coll2(ignorecase);
    fillAndPrint(coll2);
}

void fillAndPrint(StringStringMap& coll)
{
    // fill insert elements in random order
    coll["Deutschland"] = "Germany";
    coll["deutsch"] = "German";
    coll["Haken"] = "snag";
    coll["arbeiten"] = "work";
    coll["Hund"] = "dog";
    coll["gehen"] = "go";
    coll["Unternehmen"] = "enterprise";
    coll["unternehmen"] = "undertake";
    coll["gehen"] = "walk";
    coll["Bestatter"] = "undertaker";

    // print elements
    StringStringMap::iterator pos;
    cout.setf(ios::left, ios::adjustfield);
    for (pos=coll.begin(); pos!=coll.end(); ++pos) {
        cout << setw(15) << pos->first.c_str() << " "
             << pos->second << endl;
    }
    cout << endl;
}
```

main()构造出两个容器，并对它们调用 fillAndPrint()。这个函数以相同的元素值填充上述两个容器，然后打印其内容。两个容器的排序准则不同：

1. coll1 使用一个型别为 RuntimeStringCmp 的缺省仿函数。这个仿函数以元素的 operator< 来执行比较操作。
2. coll2 使用一个型别为 RuntimeStringCmp 的仿函数，并以 nocase 为初值。nocase 会令这个仿函数以“大小写无关”模式来完成字符串的比较和排序。

程序输出如下：

```
Bestatter    undertaker
Deutschland  Germany
Haken        snag
Hund         dog
Unternehmen  enterprise
arbeiten     work
deutsch      German
gehen        walk
unternehmen  undertake

arbeiten     work
Bestatter    undertaker
deutsch      German
Deutschland  Germany
gehen        walk
Haken        snag
Hund         dog
Unternehmen  undertake
```

第一部分打印第一个容器的内容，该容器以 operator< 进行排序。首先输出所有键值为大写的字符串，然后是键值为小写的字符串。

第二部分以“大小写无关”模式打印所有字符串，次序和第一部分不同。请注意，第二部分少列了一个元素，因为在大小写无关的情况下"Unternehmen"和"unternehmen" 被视为两个相同字符串[29]，而我们使用的 map 并不接纳重复元素。很不幸，打印结果乱七八糟。原本“*value* 应为 "enterprise"“的那个 *key*（是个德文字），其 *value* 却变成 "undertake"。看来这里应该使用 multimap。没错，multimap 的确是用来表现字典的一个典型容器。

[29] 德语中的所有名词，第一个字母皆大写。动词全部小写。

6.7 其它 STL 容器

STL 是个框架，除了提供标准容器，它也允许你使用其它数据结构作为容器。你可以使用字符串或数组作为 STL 容器，也可以自行撰写特殊容器以满足特殊需求。如果你自行撰写容器，仍可从诸如排序、合并等算法中受益。这样的框架正是“开放性封闭（*Open-Closed*）”原则的极佳范例[30]：允许扩展，谢绝修改。

下面是使你的容器“STL 化”的三种不同方法：

1. **The invasive approach**[31]（侵入性作法）
   直接提供 STL 容器所需接口。特别是诸如 `begin()`和 `end()`之类的常用函数。这种作法需以某种特定方式编写容器，所以是侵入性的。
2. **The noninvasive approach**（非侵入性作法）
   由你撰写或提供特殊迭代器，作为算法和特殊容器间的界面。此一作法是非侵入性的，它所需要的只是“遍历容器所有元素”的能力——这是任何容器都能以某种形式展现的能力。
3. **The wrapper approach**（包装法）
   将上述两种方法加以组合，我们可以写一个外套类别（wrapper class）来包装任何数据结构，并显示出与 STL 容器相似的接口。

本节首先将 string 视为标准容器来讨论，当做侵入性作法的一个例子，然后再以非侵入性作法讨论重要的标准容器：array。当然你也可以使用包装法来存取 array 的数据。本节最后概略讨论了一个目前尚未被涵盖于标准规格中的容器：**hash table**。

任何 STL 容器都应该能够以不同的配置器（allocator）加以参数化。C++ 标准程序库提供了一些特殊函数和类别，帮助你撰写配置器并对付尚未初始化的内存。详见 15.2 节, p728。

6.7.1 Strings 可被视为一种 STL 容器

C++ 标准程序库的 `string` 类别，乃是“以侵入性作法编写 STL 容器”的一个好例子（关于 `string` 类别的详尽讨论，请见第 11 章）。Strings 可被视为以字符（characters）为元素的一种容器；字符构成序列，你可以在序列上来回移动遍历。因此，标准的 `string` 类别提供了 STL 容器接口。Strings 也提供成员函数 `begin()` 和 `end()`，返回随机存取迭代器，可用来遍历整个 string。同时，为了支持迭代器

---

[30] 我从 Robert C. Martin 那儿头一次听说这个名称，他是从 Bertrand Meyer 那儿听来的。

[31] 有时也说成 **intrusive** 和 **nonintrusive**

和迭代器配接器（iterator adapters），strings 也提供了一些操作函数，例如 push_back() 用以支持 back inserters。

从 STL 角度来思考，string 的处理有点不寻常，因为我们通常将 string 当做一个对象来处理（我们可以传递、复制或设定 string）。但如果要对单个字符进行处理，采用 STL 算法将大有益处。例如可以采用 istream 迭代器读取字符，或转换 string 内的字符（譬如转成大写或小写）。此外，透过 STL 算法，可以对 string 采取特殊的比较规则——标准 string 接口并不提供这种能力。

p497, 11.2.13 节是 string 完整章节的一部分，在那里我详细讨论了 string 的 STL 相关特性，并给出一些实例。

6.7.2 Arrays 可被视为一种 STL 容器

我们也可以把数组当成 STL 容器来使用，但 array 并不是类别，所以不提供 begin() 和 end() 等成员函数，也不允许存在任何成员函数。在这里，我们只能采用非侵入性作法或包装法。

直接运用数组

采取非侵入性作法很简单，你只需要一个对象，它能够透过 STL 迭代器接口，遍历数组的所有元素。事实上这样的对象早就恭候多时了，它就是普通指针。STL 设计之初就决定让迭代器拥有和普通指针相同的接口，于是你可以将普通指针当成迭代器来使用。这又一次展示了纯粹抽象的泛化概念：“行为类似迭代器”的任何东西就是一种迭代器。事实上指针正是一个随机存取迭代器（参见 p255, 7.2.5 节）。以下例子示范如何以 array 作为 STL 容器：

```
// cont/array1.cpp

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

int main()
{
    int coll[] = { 5, 6, 2, 4, 1, 3 };

    // square all elements
    transform (coll, coll+6,          // first source
               coll,                  // second source
               coll,                  // destination
               multiplies<int>());    // operation
```

```
    // sort beginning with the second element
    sort (coll+1, coll+6);

    // print all elements
    copy (coll, coll+6,
          ostream_iterator<int>(cout," "));
    cout << endl;
}
```

千万注意，一定要正确传递数组尾部位置，这里是 coll+6。记住，一定要确保区间尾端是最后元素的下一个位置。

程序输出如下：

```
25 1 4 9 16 36
```

p382 和 p421 还有一些例子。

一个数组外包装

Bjarne Stroustrup 的《*The C++ Programming Language*》第三版中，介绍了一个很有用的数组包装类别，性能不输一般的数组，而且更安全。这是“使用者自行定义 STL 容器”的一个好例子。该容器所使用的，就是包装法：在数组之外包装一层常用的容器界面。

Class carray（这是 "C array" 或 "constant size array" 的缩写）定义如下[32]：

```
// cont/carray.hpp

#include <cstddef>

template<class T, std::size_t thesize>
class carray {
   private:
   T v[thesize]; // fixed-size array of elements of type T

   public:
   // type definitions
   typedef T     value_type;
   typedef T*    iterator;
```

32 原始例子名为 c_array，定义于 Bjarne Stroustrup 的《*The C++ Programming Language*》第三版 17.5.4 节。这里我做了一些改动。

```
    typedef const T*        const_iterator;
    typedef T&              reference;
    typedef const T&        const_reference;
    typedef std::size_t     size_type;
    typedef std::ptrdiff_t  difference_type;

    // iterator support
    iterator begin() { return v; }
    const_iterator begin() const { return v; }
    iterator end() { return v + thesize; }
    const_iterator end() const { return v+thesize; }

    // direct element access
    reference operator[](std::size_t i) { return v[i]; }
    const_reference operator[](std::size_t i) const { return v[i]; }

    // size is constant
    size_type size() const { return thesize; }
    size_type max_size() const { return thesize; }

    // conversion to ordinary array
    T* as_array() { return v; }
};
```

下面是 carray 的一个运用实例：

```
// cont/carray1.cpp

#include <algorithm>
#include <functional>
#include "carray.hpp"
#include "print.hpp"
using namespace std;

int main()
{
    carray<int,10> a;

    for (unsigned i=0; i<a.size(); ++i) {
        a[i] = i+1;
    }
```

```
    PRINT_ELEMENTS(a);

    reverse(a.begin(),a.end());
    PRINT_ELEMENTS(a);

    transform(a.begin(),a.end(),      // source
              a.begin(),              // destination
              negate<int>());         // operation
    PRINT_ELEMENTS(a);
}
```

如你所见，你可以使用一般容器接口（begin(), end(), operator[]）来直接操作这个容器。这么一来你也就可以使用那些需要调用 begin()和 end()的各项操作了，例如某些 STL 算法，以及 p118 所介绍的辅助函数 PRINT_ELEMENTS()。

程序输出如下：

```
1 2 3 4 5 6 7 8 9 10
10 9 8 7 6 5 4 3 2 1
-10 -9 -8 -7 -6 -5 -4 -3 -2 -1
```

（译注：关于这个 class，更细致的实作手法请参考 http://www.boost.org/ 的 Boost 程序库）

6.7.3 Hash Tables

有一个数据结构可用于群集（collection）身上，非常重要，却未包含于 C++ 标准程序库内，那就是 hash table。最初的 STL 并未涵盖 hash table，然而确实曾有提案要求，将 hash table 并入标准规格。但是标准委员会觉得这份提议来得太晚，没有采纳。（我们必须在某个时间点中止引入新功能，开始关注细节，否则工作永无止境）

不过，C++ 社群早已经有了数种可用的 hash table 实作版本。一般而言程序库会提供四种 hash table：hash_set, hash_multiset, hash_map, hash_multimap。和其它关联式容器一样，"multi" 版允许元素重复，"map" 版的元素是个 *key/value* pair。Bjarne Stroustrup 在其《*The C++ Programming Language*》第三版 17.6 节中曾经实作一个 hash_map 容器作为示范，并进行详细讨论。关于 hash table 的具体实现，可参见 STLport（http://www.stlport.org）。当然，由于 hash table 尚未正规化，所以不同的实作版本可能在细节上有所不同。（译注：如果你对 hash table 的运用和设计原理感兴趣，请参考《STL 源码剖析》by 侯捷，碁峰 2002，5.7 节~5.11 节。该处讨论的是 SGI STL 实作版本，涵括底层 hash table 和外显接口 hash_set, hash_multiset, hash_map, hash_multimap）

6.8 动手实现 Reference 语义

通常，STL 容器提供的是“value 语义”而非“reference 语义”，后者在内部构造了元素副本，任何操作返回的也是这些副本。p135, 5.10.2 节讨论了这种作法的优劣，并分析了产生后果。总之，要在 STL 容器中用到“reference 语义”（不论是因为元素的复制代价太大，或因为需要在不同群集中共享同一个元素），就要采用智能型指针，避免可能的错误。这里有一个解决办法：对指针所指的对象采用 **reference counting**（参考计数）智能型指针[33]。

```
// cont/countptr.hpp

#ifndef COUNTED_PTR_HPP
#define COUNTED_PTR_HPP

/* class for counted reference semantics
* - deletes the object to which it refers when the last CountedPtr
* that refers to it is destroyed
*/
template <class T>
class CountedPtr {
  private:
    T* ptr;       // pointer to the value
    long* count;  // shared number of owners

  public:
    // initialize pointer with existing pointer
    // - requires that the pointer p is a return value of new
    explicit CountedPtr (T* p=0)
      : ptr(p), count(new long(1)) {
    }

    // copy pointer (one more owner)
    CountedPtr (const CountedPtr<T>& p) throw()
      : ptr(p.ptr), count(p.count) {
        ++*count;
    }
```

[33] 感谢 Greg Colvin 和 Beman Dawes 对这个 class 的实作内容所给予的回应。

```
    // destructor (delete value if this was the last owner)
    ~CountedPtr () throw() {
        dispose();
    }

    // assignment (unshare old and share new value)
    CountedPtr<T>& operator= (const CountedPtr<T>& p) throw() {
        if (this != &p) {
            dispose();
            ptr = p.ptr;
            count = p.count;
            ++*count;
        }
        return *this;
    }

    // access the value to which the pointer refers
    T& operator*() const throw() {
        return *ptr;
    }
    T* operator->() const throw() {
        return ptr;
    }

  private:
    void dispose() {
        if (--*count == 0) {
            delete count;
            delete ptr;
        }
    }

};

#endif /*COUNTED_PTR_HPP*/
```

这个 class 有点类似标准规格书所提供的 auto_ptr class（参见 p38, 4.2 节）。用来初始化智能型指针值，应当是 operator new 的返回值。但是和 auto_ptr 不同的是，这种智能型指针一旦被复制，原指针和新的副本指针都是有效的。只有当指向同一对象的最后一个智能型指针被摧毁，其所指对象才会被删除。

你可以改善这个 class，例如你可以为它实现自动型别转换，或是提供“将拥有权由智能型指针交给调用端”的能力。

以下程序说明如何使用这个 class：

```
// cont/refsem1.cpp

#include <iostream>
#include <list>
#include <deque>
#include <algorithm>
#include "countptr.hpp"
using namespace std;

void printCountedPtr (CountedPtr<int> elem)
{
    cout << *elem << ' ';
}

int main()
{
    // array of integers (to share in different containers)
    static int values[] = { 3, 5, 9, 1, 6, 4 };

    // two different collections
    typedef CountedPtr<int> IntPtr;
    deque<IntPtr> coll1;
    list<IntPtr> coll2;

    /* insert shared objects into the collections
    * - same order in coll1  coll1
    * - reverse order in coll2 coll2
    */
    for (int i=0; i<sizeof(values)/sizeof(values[0]); ++i) {
         IntPtr ptr(new int(values[i]));
         coll1.push_back(ptr);
         coll2.push_front(ptr);
    }
```

```
    // print contents of both collections
    for_each (coll1.begin(), coll1.end(),
              printCountedPtr);
    cout << endl;
    for_each (coll2.begin(), coll2.end(),
              printCountedPtr);
    cout << endl << endl;

    /* modify values at different places
    * - square third value in coll1
    * - negate first value in coll1
    * - set first value in coll2 to 0
    */
    *coll1[2] *= *coll1[2];
    (**coll1.begin()) *= -1;
    (**coll2.begin()) = 0;

    // print contents of both collections again
    for_each (coll1.begin(), coll1.end(),
              printCountedPtr);
    cout << endl;
    for_each (coll2.begin(), coll2.end(),
              printCountedPtr);
    cout << endl;
}
```

程序输出如下：

```
3 5 9 1 6 4
4 6 1 9 5 3

-3 5 81 1 6 0
0 6 1 81 5 -3
```

注意，如果你调用一个辅助函数，而它在某处保存了群集（collection）内的某个元素（一个 IntPtr），那么即使群集被销毁，或其元素全被删除，那个智能型指针所指的元素依然有效。

关于其它智能型指针类别，请参考 http://www.boost.org/ 的 Boost 程序库，该程序库是 C++ 标准程序库的扩充（在那儿 CountedPtr<> 名为 shared_ptr<>）。

6.9 各种容器的运用时机

C++ 标准程序库提供了各具特长的不同容器。现在的问题是：该如何选择最佳的容器类别？表 6.33 作了一番概述，但其中有些描述可能不一定实际。例如，如果你需要处理的元素数量很少，可以忽略复杂度，因为线性算法通常对元素本身的处理过程比较快，这种情况下，“线性复杂度搭配快速的元素处理”要比“对数复杂度搭配缓慢的元素处理”来得划算。

以下规则作为表 6.33 的补充，可能对你有所帮助：

- 缺省情况下应该使用 vector。vector 的内部结构最简单，并允许随机存取，所以数据的存取十分方便灵活，数据的处理也够快。
- 如果经常要在序列头部和尾部安插和移除元素，应该采用 deque。如果你希望元素被移除时，容器能够自动缩减内存，那么你也应该采用 deque。此外，由于 vectors 通常采用一个内存区块来存放元素，而 deque 采用多个区块，所以后者可内含更多元素。
- 如果需要经常在容器的中段执行元素的安插、移除和移动，可考虑使用 list。List 提供特殊的成员函数，可以在常数时间内将元素从 A 容器转移到 B 容器。但由于 list 不支持随机存取，所以如果只知道 list 的头部却要造访 list 的中段元素，性能会大打折扣。

  和所有“以节点为基础”的容器相似，只要元素还是容器的一部分，list 就不会令指向那些元素的迭代器失效。vectors 则不然，一旦超过其容量，它的所有 iterators、pointers、references 都会失效；执行安插或移除操作时，也会令一部分 iterators、pointers、references 失效。至于 deque，当它的大小改变，所有 iterators、pointers、references 都会失效。
- 如果你要的容器是这种性质：“每次操作若不成功，便无效用”（并以此态度来处理异常），那么你应该选用 list（但是不保证其 assignment 操作符和 `sort()`；而且如果元素比较过程中会抛出异常，那就不要调用 `merge()`, `remove()`, `remove_if()`, `unique()`，参见 p172），或是采用关联式容器（但是不保证多元素安插动作，而且如果比较准则（comparision criterion）的复制/赋值动作都可能抛出异常，那么也不保证 `swap()`）。STL 的异常处理通论请见 5.11.2 节，p139。6.10.10 节，p249 提供了一个表，列举出“异常发生时提供特别保障”的所有容器操作函数。
- 如果你经常需要根据某个准则来搜寻元素，那么应当使用“以该排序准则对元素进行排序”的 set 或 multiset。记住，理论上，面对 1,000 个元素的排序，对数复杂度比线性复杂度好 10 倍。这也正是二叉树拿手好戏的发挥时机。

表 6.33 STL 容器能力一览表

	Vector	Deque	List	Set	Multiset	Map	Multimap
典型内部结构	dynamic array	array of arrays	doubly linked list	binary tree	binary tree	binary tree	binary tree
元素	value	value	value	value	value	*key*/*value* pair	*key*/*value* pair
元素可重复	是	是	是	否	是	对 *key* 而言否	是
可随机存取	是	是	否	否	否	对 *key* 而言是	否
迭代器类型	随机存取	随机存取	双向	双向 元素被视为常数	双向 元素被视为常数	双向 *key* 被视为常数	双向 *key* 被视为常数
元素搜寻速度	慢	慢	非常慢	快	快	对 *key* 而言快	对 *key* 而言快
快速安插移除	尾端	头尾两端	任何位置	—	—	—	—
安插移除导致除效 iterators, pointers, references	重新分配时	总是如此	绝不会	绝不会	绝不会	绝不会	绝不会
释放被移除元素之内存	绝不会	有时会	总是如此	总是如此	总是如此	总是如此	总是如此
允许保留内存	是	否	—	—	—	—	—
交易安全若失败不带来任何影响	尾端 push/pop 时	头尾两端 push/pop 时	任何时候除了排序和赋值	任何时候除了多元素安插	任何时候除了多元素安插	任何时候除了多元素安插	任何时候除了多元素安插

就搜寻速度而言，hash table 通常比二叉树还要快 5~10 倍。所以如果有 hash table 可用，就算它尚未标准化，也应该考虑使用。但是 hash table 的元素并未排序，所以如果元素必须排序，它就用不上了。由于 hash table 不是 C++ 标准程序库的一员，如果你要保证可移植性，就必须拥有其源码。

- 如想处理 *key/value* pair，请采用 map 或 multimap（可以的话请采用 hash table）。
- 如果需要关联式数组，应采用 map。
- 如果需要字典结构，应采用 multimap。

有一个问题比较棘手：如何根据两种不同的排序准则对元素进行排序？例如存放元素时，你希望采用客户提供的排序准则，搜寻元素时，希望使用另一个排序准则。这和数据库的情况相同，你需要在数种不同的排序准则下进行快速存取。这时候你可能需要两个 sets 或 maps，各自拥有不同的排序准则，但共享相同的元素。注意，数个群集共享相同的元素，乃是一项特殊技术，6.8 节，p222 对此有所阐述。

关联式容器拥有自动排序能力，并不意味它们在排序方面的执行效率更高。事实上由于关联式容器每安插一个新元素，都要进行一次排序，所以速度反而不及序列式容器经常采用的手法：先安插所有元素，然后调用 9.2.2 节，p328 介绍的排序算法进行一次完全排序。

下面两个简单的程序分别使用不同的容器，从标准输入读取字符串，进行排序，然后打印所有元素（去掉重复字符串）：

1. 运用 set：

```
// cont/sortset.cpp

#include <iostream>
#include <string>
#include <algorithm>
#include <set>
using namespace std;

int main()
{
    /* create a string set
    * - initialized by all words from standard input
    */
    set<string> coll((istream_iterator<string>(cin)),
                     (istream_iterator<string>()));

    // print all elements
    copy (coll.begin(), coll.end(),
          ostream_iterator<string>(cout, "\n"));
}
```

2. 运用 vector：

```
// cont/sortvec.cpp

#include <iostream>
#include <string>
#include <algorithm>
#include <vector>
using namespace std;

int main()
{
    /* create a string vector
    * - initialized by all words from standard input
    */
    vector<string> coll((istream_iterator<string>(cin)),
                        (istream_iterator<string>()));

    // sort elements
    sort (coll.begin(), coll.end());

    // print all elements ignoring subsequent duplicates
    unique_copy (coll.begin(), coll.end(),
                 ostream_iterator<string>(cout, "\n"));
}
```

我在我的系统上使用大约 150,000 个字符串来测试这两个程序，我发现 vectors 版本快 10% 左右。如果使用 `reserve()`，vectors 版本还可以再快将近 5%。如果允许重复元素（改用 multiset 取代 set，调用 `copy()` 取代 `unique_copy()`），则情况发生剧烈变化：vectors 版本领先超过 40%！这些比较虽然不具代表性，但至少证实了一点：对各种不同的元素处理方法多加尝试是值得的。

现实中预测哪种容器最好，往往相当困难。STL 的一大优点就是你可以轻而易举地尝试各种版本。主要工作——各种数据结构和算法——已经就位，你只需依照对自己最有利的方式将它们组合运用就行了。

6.10 细说容器内的型别和成员

本节讨论各种 STL 容器，阐述 STL 容器所支持的一切操作函数。型别和成员一律按功能分组。针对每一种型别定义和操作，本节描述其标记式（signature）、行为、支持者(容器)。本节涉及的容器包括 vectors、deques、lists、sets、multisets、maps、multimaps 和 strings。后续数节中 *container* 指的是“支持该成员”的某容器型别。

6.10.1 容器内的型别

container::**value_type**

- 元素型别。
- 用于 sets 和 multisets 时是常数。
- 用于 maps 和 multimaps 时是 `pair <const` ***key-type***, ***value-type***`>`。
- 在 vectors, deques, lists, sets, multisets, maps, multimaps, strings 之中都有定义。

container::**reference**

- 元素的引用型别（reference type）。
- 典型定义：*container*::value_type&。
- 在 `vector<bool>` 中其实是个辅助类别（参见 p158）。
- 在 vectors, deques, lists, sets, multisets, maps, multimaps, strings 中都有定义。

container::**const_reference**

- 常数元素的引用型别（reference type）。
- 典型定义：const *container*::value_type&。
- 在 `vector<bool>` 中是 bool。
- 在 vectors, deques, lists, sets, multisets, maps, multimaps, strings 中都有定义。

container::**iterator**

- 迭代器型别。
- 在 vectors, deques, lists, sets, multisets, maps, multimaps, strings 中都有定义。

container::**const_iterator**

- 常数迭代器的型别。
- 在 vectors, deques, lists, sets, multisets, maps, multimaps, strings 中都有定义。

container::**reverse_iterator**

- 反向迭代器型别。
- 在 vectors, deques, lists, sets, multisets, maps, multimaps 中都有定义。

container::**const_reverse_iterator**

- 常数反向迭代器的型别。
- 在 vectors, deques, lists, sets, multisets, maps, multimaps, strings 中都有定义。

container::**size_type**

- 无正负号整数型别，用以定义容器大小。
- 在 vectors, deques, lists, sets, multisets, maps, multimaps, strings 中都有定义。

container::**difference_type**

- 有正负号整数型别，用以定义距离。
- 在 vectors, deques, lists, sets, multisets, maps, multimaps, strings 中都有定义。

container::**key_type**

- 用以定义关联式容器的元素内的 *key* 型别。
- 用于 sets 和 multisets 时，相当于 value_type。
- 在 sets, multisets, maps, multimaps 中都有定义。

container::**mapped_type**

- 用以定义关联式容器的元素内的 *value* 型别。
- 在 maps 和 multimaps 中都有定义。

container::**key_compare**

- 关联式容器内的“比较准则”的型别。
- 在 sets, multisets, maps, multimaps 中都有定义。

container::**value_compare**

- 用于整个元素之“比较准则”的型别。
- 用于 sets 和 multisets 时，相当于 key_compare。
- 在 maps 和 multimaps 中，它是“比较准则”的辅助类别，仅比较两元素的 *key*。
- 在 sets, multisets, map, multimap 中都有定义。

container::**allocator_type**

- 配置器型别。
- 在 vectors, deques, lists, sets, multisets, maps, multimaps, strings 中都有定义。

6.10.2 生成（Create）、复制（Copy）、销毁（Destroy）

STL 容器支持下列构造函数和析构函数，并且大多数构造函数允许将配置器作为一个附加参数传递（参见第 6.10.9 节, p246）。

container::*container* ()

- default 构造函数
- 产生一个新的空容器
- vectors, deques, lists, sets, multisets, maps, multimaps, strings 都支持

```
explicit container::container (const CompFunc& op)
```

- 以 op 为排序准则，产生一个空容器（参见 p191 和 p213 实例）。
- 排序准则必须定义一个 **strict weak ordering**（参见 p176）。
- sets, multisets, maps, multimaps 支持。

```
explicit container::container (const container& c)
```

- copy 构造函数。
- 产生既有容器的一个副本。
- 针对 c 中的每一个元素调用 copy 构造函数。
- vectors, deques, lists, sets, multisets, maps, multimaps, strings 都支持。

```
explicit container::container (size_type num)
```

- 产生一个容器，可含 *num* 个元素。
- 元素由其 default 构造函数创建。
- vectors, deques, lists 都支持。

```
container::container (size_type num, const T& value)
```

- 产生一个容器，可含 *num* 个元素。
- 所有元素都是 *value* 的副本。
- T 是元素型别。
- 对于 strings，*value* 并非 pass by reference。
- vectors、deques、lists 和 strings 都支持。

```
container::container (InputIterator beg, InputIterator end)
```

- 产生容器，并以区间 [beg;end) 内的所有元素为初值。
- 此函数为一个 member template（参见 p11）。因此只要源区间的元素型别可转换为容器元素型别，此函数即可派上用场。
- vectors、deques、lists、sets、multisets、maps、multimaps、strings 都支持。

```
container::container (InputIterator beg, InputIterator end,
                      const CompFunc& op)
```

- 产生一个排序准则为 *op* 的容器，并以区间 [beg;end) 内的所有元素进行初始化。
- 此函数为一个 member template（参见 p11）。因此只要源区间的元素型别可转换为容器元素型别，此函数即可派上用场。
- 排序准则必须定义一个 **strict weak ordering**（参见 p176）。
- sets、multisets、maps 和 multimaps 都支持。

```
container::~container ()
```

- 析构函数。
- 移除所有元素，并释放内存。
- 对每个元素调用其析构函数。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支持。

6.10.3 非变动性操作（Nonmodifying Operations）

大小相关操作（Size Operations）

```
size_type container::size () const
```

- 返回现有元素的数目。
- 欲检查容器是否为空，应使用 `empty()`，因为 `empty()` 可能更快。
- vectors、deques、lists、sets、multisets、maps、multimasp 和 strings 都支持。

```
bool container::empty () const
```

- 检验容器是否为空，并返回检查结果。
- 相当于 `container::size()==0`，但是可能更快（尤其对 lists 而言）。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支持。

```
size_type container::max_size () const
```

- 返回容器可包含的最大元素个数。
- 这是一个技术层次的数值，可能取决于容器的内存模型。尤其 vectors 通常使用一个内存区段（segment），所以 vector 的这个值往往小于其它容器。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支持。

容量操作（Capacity Operations）

```
size_type container::capacity () const
```

- 返回重分配内存之前所能容纳的最多元素个数。
- vectors 和 strings 都支持。

```
void container::reserve (size_type num)
```

- 在内部保留若干内存，至少能够容纳 num 个元素。
- 如果 *num* 小于实际容量，对 vectors 无效，对 strings 则是一个非绑定的缩减请求（nonbinding shrink request）。

- vectors 的容量如何缩小，请见 p149 例子。
- 每次重新分配都会耗用相当时间，并造成所有 references、pointers、iterators 失效。因此 `reserve()` 可以提高速度，保持 references、pointers、iterators 的有效性。详见 p149。
- vectors 和 strings 都支持。

元素间的比较（Comparison Operations）

```
bool comparison (const container& c1, const container& c2)
```

- 返回两个同型容器的比较结果。
- *comparison* 可以是下面之一：

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

- 如果两个容器拥有相同数量的元素，且元素顺序相同，而且所有相应元素两两相比之结果为 true，我们便说这两个容器相等。
- 要检验 A 容器是否小于 B 容器，需使用“字典顺序”来比较。关于“字典顺序”，请见 p360 `lexicographical_compare()` 的描述。
- vectors、deques、lists、sets、multisets、maps、multimaps, strings 都支持。

关联式容器特有的非变动性操作

这里所介绍的成员函数都是对应于 p338, 9.5 节和 p397, 9.9 节所讨论的 STL 算法的特殊实作版本。这些函数利用了关联式容器的元素已序性，提供更好的性能。例如在 1,000 个元素中进行搜寻，所需的比较平均不超过 10 次（参见 p21, 2.3 节）。

```
size_type container::count (const T& value) const
```

- 返回与 *value* 相等的元素个数。
- 这是 p338 所讨论的 `count()` 算法的特殊版本。
- `T` 是被排序值的型别
  - — 在 sets 和 multisets 中，`T` 是元素型别。
  - — 在 maps 和 multimaps 中，`T` 是 *key* 的型别。
- 复杂度：线性。
- sets、multisets、maps 和 multimaps 都支持。

```
iterator container::find (const T& value)
const_iterator container::find (const T& value) const
```

- 返回“实值等于 *value*”的第一个元素位置。
- 如果找不到元素就返回 `end()`。
- 这是 p341 所讨论的 `find()`算法的特殊版本。
- `T` 是被排序值的型别：
  - 在 sets 和 multisets 中，`T` 是元素型别。
  - 在 maps 和 multimaps 中，`T` 是 *key* 的型别。
- 复杂度：对数。
- sets、multisets、maps 和 multimaps 都支持。

```
iterator container::lower_bound (const T& value)
const_iterator container::lower_bound (const T& value) const
```

- 返回一个迭代器，指向“根据排序准则，可安插 *value* 副本的第一个位置”。
- 返回之迭代器指向“实值大于等于 *value* 的第一个元素”（有可能是 `end()`）。
- 如果找不到就返回 `end()`。
- 这是 p413 所讨论的 `lower_bound()`算法的特殊版本。
- `T` 是被排序值的型别：
  - 在 sets 和 multisets 中，`T` 是元素型别。
  - 在 map 和 multimap 中，`T` 是 *key* 的型别。
- 复杂度：对数。
- sets、multisets、maps 和 multimaps 都支持。

```
iterator container::upper_bound (const T& value)
const_iterator container::upper_bound (const T& value) const
```

- 返回一个迭代器，指向“根据排序准则，可安插 *value* 副本的最后一个位置”。
- 返回之迭代器指向“实值大于 *value* 的第一个元素”（有可能是 `end()`）。
- 如果找不到就返回 `end()`。
- 这是 p413 所讨论的 `upper_bound()`算法的特殊版本。
- `T` 是被排序值的型别：
  - 在 sets 和 multisets 中，`T` 是元素型别。
  - 在 map 和 multimap 中，`T` 是 *key* 的型别。
- 复杂度：对数。
- sets、multisets、maps 和 multimaps 都支持。

```
pair<iterator,iterator> container::equal_range (const T& value)
pair<const_iterator,const_iterator>
                        container::equal_range (const T& value) const
```

- 返回一个区间（一对迭代器），指向“根据排序准则，可安插 value 副本的第一个位置和最后一个位置”。
- 返回一个区间，其内的元素实值皆等于 value。
- 相当于：
  `make_pair(lower_bound(value),upper_bound(value))`
- 这是 p415 所讨论的 equal_range() 算法的特殊版本。
- T 是被排序值的型别：
  — 在 sets 和 multisets 中，T 是元素型别。
  — 在 map 和 multimap 中，T 是 *key* 的型别。
- 复杂度：对数。
- sets、multisets、maps 和 multimaps 都支持。

```
key_compare container::key_comp ()
```

- 返回一个“比较准则”。
- sets、multisets、maps 和 multimaps 都支持。

```
value_compare container::value_comp ()
```

- 返回一个作为比较准则的对象。
- 在 sets 和 multisets 中，它相当于 key_comp()。
- 在 maps 和 multimaps 中，它是一个辅助类别，用来比较两元素的 *key*。
- sets、multisets、maps 和 multimaps 都支持。

6.10.4 赋值（Assignments）

```
container& container::operator = (const container& c)
```

- 将 c 的所有元素赋值给现有容器，亦即以 c 的元素替换所有现有元素。
- 这个操作符会针对被覆盖的元素调用其 assignment 操作符，针对被附加的元素调用其 copy 构造函数，针对被移除的元素调用其析构函数。
- vectors、deques、lists、sets、multisets、maps 和 multimaps 都支持。

```
void container::assign (size_type num, const T& value)
```

- 将 *num* 个 *value* 赋值给现有容器，亦即以 *num* 个 *value* 副本替换掉所有现有元素。
- T 必须是元素型别。
- sets、multisets、maps 和 multimaps 都支持。

```
void container::assign (InputIterator beg, InputIterator end)
```

- 将区间[beg;end)内的所有元素赋值给现有容器，亦即以[beg;end)内的元素副本替换掉所有现有元素。
- 此函数为一个 member template（参见 p11）。因此只要源区间的元素型别可转换为容器元素型别，此函数即可派上用场。
- vectors、deques、lists 和 strings 都支持。

```
void container::swap (container& c)
```

- 和 c 交换内容。
- 两个容器互换：
  - 元素
  - 排序准则（如果有的话）。
- 此函数拥有常数复杂度。如果不再需要容器中的老旧元素，则应使用本函数来取代赋值动作（参见 p147, 6.1.2 节）。
- 对于关联式容器，只要“比较准则”进行复制或赋值时不抛出异常，本函数就不抛出异常。对于其它所有容器，此函数一律不抛出异常。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支持。

```
void swap (container& c1, container& c2)
```

- 相当于 c1.swap(c2)（参见稍早前的描述）。
- 对于关联式容器，只要“比较准则”进行复制或赋值时，不抛出异常，本函数就不抛出异常。对于其它所有容器，此函数一律不抛出异常。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支持。

6.10.5 直接元素存取

```
reference container::at (size_type idx)
const_reference container::at (size_type idx) const
```

- 二者都返回索引 *idx* 所代表的元素（第一个元素的索引为 0）。
- 如果传入一个无效索引（< 0 或 >= size()），会导致 out_of_range 异常。
- 后续的修改或内存重新分配，可能会导致返回的 reference 无效。
- 如果调用者保证索引有效，那么最好使用速度更快的 operator[]。
- vectors、deques 和 strings 都支持。

```
reference container::operator[] (size_type idx)
const_reference container::operator[] (size_type idx) const
```

- 二者都返回索引 *idx* 所代表的元素（第一个元素的索引为 0）。
- 如果传入一个无效索引（< 0 或 >= size()），会导致未定义的行为。所以调用者必须确保索引有效，否则应该使用 at()。
- (1) 修改 strings 或 (2) 内存重新分配，可能会导致 non-const strings 返回的 reference 失效（详见 p487）。
- vectors、deques 和 strings 都支持。

```
T& map::operator[] (const key_type& key)
```

- 关联式数组的 operator[]。
- 在 map 中，会返回 *key* 所对应的 *value*。
- 注意：如果不存在“键值为 *key*”的元素，则本操作会自动生成一个新元素，其初值由 *value* 型别的 default 构造函数给定。所以不存在所谓的无效索引。例如：

```
map<int,string> coll;
coll[77] = "hello";  // insert key 77 with value "hello"
cout << coll[42];    // Oops, inserts key 42 with value "" and
                     // prints the value
```

  详见 p205, 6.6.3 节。

- T 是元素的 *value* 型别。
- 相当于：

```
(*((insert(make_pair(x,T()))).first)).second
```

- 只有 map 支持此一操作。

```
reference container::front ()
const_reference container::front () const
```

- 都返回第一个元素（第一个元素的索引为 0）。
- 调用者必须确保容器内有元素（size()>0），否则会导致未定义的行为。
- vectors、deques 和 lists 都支持。

```
reference container::back ()
const_reference container::back () const
```

- 都返回最后一个元素（索引为 size()-1）。
- 调用者必须确保容器内拥有元素（size()>0）；否则会导致未定义的行为。
- vectors、deques 和 lists 都支持。

6.10.6 “可返回迭代器”的各项操作

本节各个成员函数都会返回迭代器，凭借这些迭代器你可以遍历容器中的所有元素。表 6.34 列出各种容器所提供的迭代器类型（参见 p251, 7.2 节）。

表 6.34 各种容器提供的迭代器类型

容器	迭代器类型（iterator category）
Vector	随机存取
Deque	随机存取
List	双向
Set	双向，元素为常量
Multiset	双向，元素为常量
Map	双向，*key* 为常量
Multimap	双向，*key* 为常量
String	随机存取

```
iterator container::begin ()
const_iterator container::begin () const
```

- 返回一个迭代器，指向容器起始处（第一元素的位置）。
- 如果容器为空，则此动作相当于 `container::end()`。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支持。

```
iterator container::end ()
const_iterator container::end () const
```

- 返回一个迭代器，指向容器尾端（最后元素的下一位置）。
- 如果容器为空，则此动作相当于 `container::begin()`。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支持。

```
reverse_iterator container::rbegin ()
const_reverse_iterator container::rbegin () const
```

- 返回一个逆向迭代器，指向逆向迭代时遍历的第一个元素。
- 如果容器为空，则此动作相当于 `container::rend()`。
- 关于逆向迭代器，详见 p264, 7.4.1 节。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支持。

```
reverse_iterator container::rend ()
const_reverse_iterator container::rend () const
```

- 返回一个逆向迭代器，指向逆向迭代时遍历的最后一个元素的下一位置。
- 如果容器为空，则此操作相当于 *container*::rbegin()。
- 关于逆向迭代器，详见 p264, 7.4.1 节。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支持。

6.10.7 元素的安插（Inserting）和移除（Removing）

```
iterator container::insert (const T& value)
pair<iterator,bool> container::insert (const T& value)
```

- 安插一个 *value* 副本于关联式容器。
- 元素可重复者（multisets 和 multimap）采用第一形式。返回新元素的位置。
- 元素不可重复者（sets 和 map）采用第二形式。如果有“具备相同 *key*”的元素已经存在，导致无法安插，会返回现有元素的位置和一个 false。如果安插成功，返回新元素的位置和一个 true。
- T 是容器元素的型别，对 map 和 multimap 而言那是一个 *key*/*value* pair。
- 函数如果不成功，不带来任何影响。
- sets、multisets、maps、multimaps 和 strings 都支持。

```
iterator container::insert (iterator pos, const T& value)
```

- 在迭代器 *pos* 的位置上安插一个 value 副本。
- 返回新元素的位置。
- 对于关联式容器（sets、multisets、maps 和 multimaps），*pos* 只作为一个提示，指向安插时必要的搜寻操作的起始建议位置。如果 *value* 刚好可安插于 *pos* 之后，则此函数具有“分期摊还之常数时间”复杂度，否则具有对数复杂度。
- 如果容器是 sets 或 maps，并且已内含一个“实值等于 *value*（意即两者的 *key* 相等）”的元素，则此调用无效，并返回现有元素的位置。
- 对于 vectors 和 deques，这个操作可能导致指向其它元素的某些 iterators 和 references 无效。
- T 是容器元素的型别，在 maps 和 multimaps 中是一个 *key*/*value* pair。
- 对于 strings，*value* 并不采用 pass by reference。
- 对于 vectors 和 deques，如果元素的复制操作（copy 构造函数和 operator=）不抛出异常，则此函数一旦失败并不会带来任何影响。对于所有其它容器，函数一旦失败并不会带来任何影响。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支持。

```
void container::insert (iterator pos, size_type num, const T& value)
```

- 在迭代器 *pos* 的位置上安插 *num* 个 *value* 副本。
- 对于 vectors 和 deques，此操作可能导致指向其它元素的 iterators 和 references 失效
- T 是容器元素的型别，在 maps 和 multimaps 中是一个 *key/value* pair。
- 对于 strings，*value* 并不采用 pass by reference。
- 对于 vectors 和 deques，如果元素复制动作（copy 构造函数和 operator=）不抛出异常，则函数失败亦不会带来任何影响。对于 lists，函数若失败不会带来任何影响。
- vectors、deques、lists 和 strings 都支持。

```
void container::insert (InputIterator beg, InputIterator end)
```

- 将区间 [*beg*, *end*) 内所有元素的副本安插于关联式容器内。
- 此函数是个 member template（参见 p11），因此只要源区间的元素可转换为容器元素的型别，本函数就可派上用场。
- sets、multisets、maps、multimaps 和 strings 都支持。

```
void container::insert (iterator pos, InputIterator beg,
                                      InputIterator end)
```

- 将区间 [*beg*, *end*) 内所有元素的副本安插于迭代器 *pos* 所指的位置上。
- 此函数是个 member template（参见 p11），因此只要源区间的元素可转换为容器元素的型别，本函数就可派上用场。
- 对于 vectors 和 deques，此操作可能导致指向其它元素的 iterators 和 references 失效。
- 对于 lists，此函数若失败不会带来任何影响。
- vectors、deques、lists 和 strings 都支持。

```
void container::push_front (const T& value)
```

- 安插 *value* 的副本，使成为第一个元素。
- T 是容器元素的型别。
- 相当于 insert(begin(), *value*)。
- 对于 deques，此一操作会造成“指向其它元素”的 iterators 失效，而“指向其它元素”的 references 仍保持有效。
- 此函数若失败不会带来任何影响。
- deques 和 lists 都支持。

```
void container::push_back (const T& value)
```

- 安插 *value* 的副本，使成为最后一个元素。
- T 是容器元素的型别。
- 相当于 insert(end(), *value*)。

- 对于 vectors，如果造成内存重新分配，此操作会造成“指向其它元素”的 iterators 和 references 失效。
- 对于 deques，此一操作造成“指向其它元素”的 iterators 失效，而“指向（或说代表）其它元素”的 reference 始终有效。
- 此函数若失败不会带来任何影响。
- vectors、deques、lists 和 strings 都支持。

```
void list::remove (const T& value)
void list::remove_if (UnaryPredicate op)
```

- remove()会移除所有“实值等于 *value*”的元素。
- remove_if()会移除所有“使判断式 *op*(*elem*)结果为 true”的元素。
- 注意在函数调用过程中，*op* 不应改变状态。详见 p302, 8.14 节。
- 两者都会调用被移除元素的析构函数。
- 剩余元素的相对次序保持不变（stable）。
- 这是 p378 所讨论的 remove()算法的特殊版本。
- T 是容器元素的型别。
- 细节和范例见 p170。
- 只要元素的比较动作不抛出异常，此函数也不抛出异常。
- 只有 lists 支持这个成员函数。

```
size_type container::erase (const T& value)
```

- 从关联式容器中移除所有和 *value* 相等的元素。
- 返回被移除的元素个数。
- 调用被移除元素的析构函数。
- T 是已序（*sorted*）元素的型别。
  - — 在 sets 和 multisets 中，T 是元素型别。
  - — 在 map 和 multimap 中，T 是 *key* 的型别。
- 此函数不抛出异常。
- sets、multisets、maps 和 multimaps 都支持。

```
void container::erase (iterator pos)
iterator container::erase (iterator pos)
```

- 将迭代器 *pos* 所指位置上的元素移除。
- 序列式容器（vectors、deques、lists 和 strings）采用第二形式，返回后继元素的位置（或返回 end()）。
- 关联式容器（sets、multisets、maps 和 multimaps）采用第一形式，无返回值。
- 两者都调用被移除元素的析构函数。

- 注意，调用者必须确保迭代器 *pos* 有效。例如：
  `coll.erase(coll.end()); // ERROR → undefined behavior`
- 对于 vectors 和 deques，此操作可能造成“指向其它元素”的 iterators 和 references 无效。
- 对于 vectors 和 deques，只要元素复制操作（copy 构造函数和 operator=）不抛出异常，此函数就不抛出异常。对于其它容器，此函数不抛出异常。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支持。

`void container::erase (iterator beg, iterator end)`
`iterator container::erase (iterator beg, iterator end)`

- 移除区间 [*beg*, *end*) 内的所有元素。
- 序列式容器（vectors、deques、lists 和 strings）采用第二形式，返回被移除的最后一个元素的下一位置（或返回 end()）。
- 关联式容器（sets、multisets、maps 和 multimaps）采用第一形式，无返回值。
- 一如区间惯例，始于 *beg*（含）终于 *end*（不含）的所有元素都被移除。
- 调用被移除元素的析构函数。
- 调用者必须确保 *beg* 和 *end* 形成一个有效序列，并且该序列是容器的一部分。
- 对于 vectors 和 deques，此操作可能导致“指向其它元素”的 iterators 和 references 失效。
- 对于 vectors 和 deques，只要元素复制动作（copy 构造函数和 operator=）不抛出异常，此函数就不抛出异常。对于其它容器，此函数不抛出异常。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支持。

`void container::pop_front ()`

- 将容器的第一个元素移除。
- 相当于 *container*.erase(*container*.begin())。
- 注意：如果容器是空的，会导致未定义行为。因此，调用者必须确保容器至少有一个元素，也就是 size()>0。
- 此函数不抛出异常。
- deques 和 lists 都支持。

`void container::pop_back ()`

- 将容器的最后一个元素移除。
- 相当于 *container*.erase(--*container*.end())，前提是其中的表达式有效。在 vector 中此表达式不一定有效（参见 p258）。
- 注意，如果容器为空，会导致未定义行为。因此，调用者必须确保容器至少包

含一个元素，也就是 size()>0。

- 此函数不抛出异常。
- vectors、deques 和 lists 都支持。

```
void container::resize (size_type num)
void container::resize (size_type num, T value)
```

- 两者都将容器大小改为 *num*。
- 如果 size()原本就是 *num*，则两者皆不生效用。
- 如果 *num* 大于 size()，则在容器尾端产生并附加额外元素。第一形式透过 default 构造函数来构造新元素，第二形式则以 *value* 的副本作为新元素。
- 如果 *num* 小于 size()，则移除尾端元素，直到大小为 size()。每个被移除元素的析构函数都会被调用。
- 对于 vectors 和 deques，这些函数可能导致"指向其它元素"的 iterators 和 references 失效。
- 对于 vectors 和 deques，只要元素复制操作（copy 构造函数和 operator=）不抛出异常，这些函数就不抛出异常。对于 lists，函如果失败不会带来任何影响。
- vectors、deques、lists 和 strings 都支持。

```
void container::clear ()
```

- 移除所有元素（将容器清空）。
- 调用被移除元素的析构函数。
- 这一容器的所有 iterators 和 references 都将失效。
- 对于 vectors 和 deques，只要元素复制操作（copy 构造函数和 operator=）不抛出异常，此函数就不抛出异常。对于其它容器，此函数不抛出异常。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支持。

6.10.8 Lists 的特殊成员函数

```
void list::unique ()
void list::unique (BinaryPredicate op)
```

- 移除 lists 之内相邻而重复的元素，使每一个元素都不同于下一个元素。
- 第一形式会将所有"和前一元素相等"的元素移除。
- 第二形式的意义是：任何一个元素 *elem*，如果其前一元素是 *e*，而 *elem* 和 *e* 造成二元判断式 *op(elem, e)* 获得 true 值，那么就移除 *elem*[34]。换言之，这个判断式并非拿元素和其目前的前一紧临元素比较，而是拿元素和其未被移除的前一元素比较。

---

34 第二版的 unique()仅在支持 member templates 的系统中可用（参见 p11）。

- 注意，*op* 不应在函数调用过程中改变状态，详见 p302, 8.1.4 节。
- 被移除元素的析构函数会被调用。
- 这是 p381 unique()算法的 lists 特别版本。
- 如果“元素比较动作”中不抛出异常，则此函数亦不抛出异常。

```
void list::splice (iterator pos, list& source)
```

- 将 *source* 的所有元素移动到*this，并安插到迭代器 *pos* 所指位置。
- 调用之后，*source* 清空。
- 如果 *source* 和*this 相同，会导致未定义的行为。所以调用端必须确定 *source* 和*this 是不同的 lists。如果要移动同一个 lists 内的元素，应该使用稍后提及的其它 splice()形式。
- 调用者必须确定 *pos* 是*this 的一个有效位置；否则会导致未定义的行为。
- 本函数不抛出异常。

```
void list::splice (iterator pos, list& source, iterator sourcePos)
```

- 从 *source* list 中，将位于 *sourcePos* 位置上的元素移动至 *this，并安插于迭代器 *pos* 所指位置。
- *source* 和 *this 可以相同。这种情况下，元素将在 lists 内部被移动。
- 如果 *source* 和 *this 不是同一个 list，在此操作之后，其元素个数少 1。
- 调用者必须确保 *pos* 是 *this 的一个有效位置、*sourcePos* 是 *source* 的一个有效迭代器，而且 *sourcePos* 不是 source.end()；否则会导致未定义行为。
- 此函数不抛出异常。

```
void list::splice (iterator pos, list& source,
                   iterator sourceBeg, iterator sourceEnd)
```

- 从 *source* list 中，将位于[*sourceBeg*, *sourceEnd*)区间内的所有元素移动到 *this，并安插于迭代器 *pos* 所指位置。
- *source* 和 *this 可以相同。这种情况下，*pos* 不得为被移动序列的一部分，而元素将在 lists 内部移动。
- 如果 *source* 和 *this 不是同一个 list，在此操作之后，其元素个数将减少。
- 调用者必须确保 *pos* 是 *this 的一个有效位置、*sourceBeg* 和 *sourceEnd* 形成一个有效区间，该区间是 *source* 的一部分；否则会导致未定义的行为。
- 本函数不抛出异常。

```
void list::sort ()
void list::sort (CompFunc op)
```

- 对 lists 内的所有元素进行排序。
- 第一型式以 operator< 对 lists 中的所有元素进行排序。
- 第二型式透过如下的 *op* 操作来比较两元素，进而对 lists 中的所有元素排序[35]：*op(elem1,elem2)*。
- 实值相同的元素，其顺序保持不变（除非有异常被丢出）。
- 这是 p397 所讨论的 sort() 和 stable_sort()算法的“list 特殊版本”。

```
void list::merge (list& source)
void list::merge (list& source, CompFunc op)
```

- 将 lists *source* 内的所有元素并入*this。
- 调用后 *source* 变成空容器。
- 如果*this 和 *source* 在排序准则 operator< 或 op 之下已序（*sorted*），则新产生的 lists 也是已序。严格地说，标准规格书要求两个 lists 必须已序，但实际上对无序的 lists 进行合并也是可能的，不过使用前最好先确认一下。
- 第一形式采用 operator< 作为排序准则。
- 第二形式采用以下的 op 操作作为可有可无的排序准则，以此比较两个元素的大小[36]：op(*elem,sourceElem*)
- 这是 p416 所讨论的 merge()算法的 list 特殊版本。
- 只要元素的比较操作不抛出异常，此函数万一失败也不会造成任何影响。

```
void list::reverse ()
```

- 将 lists 中的元素颠倒次序。
- 这是 p386 所讨论的 reverse()算法的“list 特殊版本”。
- 本函数不抛出异常。

6.10.9 对配置器（Allocator）的支持

所有 STL 容器都能够与某个配置器对象（allocator object）所定义的某种特定内存模型（memory model）搭配合作（详见第 15 章）。本节讨论的是支持配置器的各个成员。标准容器要求：配置器（型别）的每一个实体都必须是可互换的（interchangeable），所以某一容器的空间，可透过另一同型容器释放之。因此，元素（及其储存空间）在同型的两个容器之间移动，并不会出现问题。

---

35 sort()的第二形式仅在支持 member templates 的系统中可用（参见 p11）

36 merge()的第二形式仅在支持 member templates 的系统中可用（参见 p11）

基本的配置器相关成员（Fundamental Allocator Members）

```
container::allocator_type
```

- 配置器型别。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支持。

```
allocator_type container::get_allocator () const
```

- 返回容器的内存模型（memory model）。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支持。

带有“可选择之配置器参数”的构造函数

```
explicit container::container (const Allocator& alloc)
```

- 产生一个新的空白容器，使用 *alloc* 作为内存模型（memory model）。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支持。

```
container::container (const CompFunc& op, const Allocator& alloc)
```

- 产生一个新的空白容器，使用 *alloc* 作为内存模型，并以 *op* 为排序准则。
- *op* 排序准则必须定义 **strict weak ordering**（参见 p176）。
- sets、multisets、maps 和 multimaps 都支持。

```
container::container (size_type num, const T& value,
                      const Allocator& alloc)
```

- 产生一个拥有 *num* 个元素的容器，使用 *alloc* 作为内存模型。
- 所生成的元素都是 *value* 的副本。
- T 是容器元素的型别。注意，对于 strings，*value* 采用 by value 的型式传递。
- vectors、deques、lists 和 strings 都支持。

```
container::container (InputIterator beg, InputIterator end,
                      const Allocator& alloc)
```

- 产生一个容器，以区间 [*beg*, *end*) 内的所有元素为初值，并使用 *alloc* 作为内存模型。
- 此函数是一个 member template（参见 p11）。所以只要源序列的元素能够转换为容器元素的型别，此函数就可执行。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支持。

```
container::container (InputIterator beg, InputIterator end,
                      const CompFunc& op, const Allocator& alloc)
```

- 产生一个以 *op* 为排序准则的容器，以区间 [*beg*, *end*) 中的所有元素为初值，并使用 *alloc* 作为内存模型。
- 本函数是一个 member template（参见 p11）。所以只要源序列的元素能够转换为容器元素的型别，本函数就可执行。
- 排序准则 *op* 必须定义 **strict weak ordering**（参见 p176）。
- sets、multisets、map 和 multimap 都支持。

6.10.10 综观 STL 容器的异常处理

p139, 5.11.2 节曾指出，不同的容器在异常发生时，给予不同程度的保证。通常 C++ 标准程序库在异常发生时并不会泄漏资源或破坏容器的恒常特性（invariants）。有些操作提供更强的保证（前提是其参数必须满足某些条件）：它们可以保证 **commit-or-rollback**（意思是“要么成功，要么不带来任何影响”），甚至可以保证绝不抛出异常。表 6.35 列出所有支持更严格保证的操作函数[37]。

对于 vectors、deques 和 lists 而言，`resize()` 也提供特别保证。其行为或许相当于 `erase()`，或许相当于 `insert()`，或许相当于什么也没做。

```
void container::resize (size_type num, T value = T())
{
    if (num > size()) {
        insert (end(), num-size(), value);
    }
    else if (num < size()) {
        erase (begin()+num, end());
    }
}
```

因此，它所提供的保证就是“`erase()` 和 `insert()` 两者所提供的保证”的组合（参见 p244）。

[37] 感谢 Greg Colvin 和 Dave Abrahams 提供这个表格。

表 6.35 “异常发生时给予特殊保证”的各个容器操作函数

操作	页次	保证
vector::push_back()	241	要么成功，要么无任何影响
vector::insert()	240	要么成功，要么无任何影响——前提是元素的复制/赋值操作不抛出异常
vector::pop_back()	243	不抛出异常
vector::erase()	242	不抛出异常——前提是元素的复制/赋值操作不抛出异常
vector::clear()	244	不抛出异常——前提是元素的复制/赋值操作不抛出异常
vector::swap()	237	不抛出异常
deque::push_back()	241	要么成功，要么无任何影响
deque::push_front()	241	要么成功，要么无任何影响
deque::insert()	240	要么成功，要不无任何影响——前提是元素的复制/赋值操作不抛出异常
deque::pop_back()	243	不抛出异常
deque::pop_front()	243	不抛出异常
deque::erase()	242	不抛出异常——前提是元素的复制/赋值操作不抛出异常
deque::clear()	244	不抛出异常——前提是元素的复制/赋值操作不抛出异常
deque::swap()	237	不抛出异常
list::push_back()	241	要么成功，要么无任何影响
list::push_front()	241	要么成功，要么无任何影响
list::insert()	240	要么成功，要么无任何影响
list::pop_back()	243	不抛出异常
list::pop_front()	243	不抛出异常
list::erase()	242	不抛出异常
list::clear()	244	不抛出异常
list::remove()	242	不抛出异常——前提是元素的比较操作不抛出异常
list::remove_if()	242	不抛出异常——前提是判断式 *predicate* 不抛出异常
list::unique()	244	不抛出异常——前提是元素的比较操作不抛出异常
list::splice()	245	不抛出异常
list::merge()	246	要么成功，要么无任何影响——前提是元素的比较操作不抛出异常
list::reverse()	246	不抛出异常
list::swap()	237	不抛出异常
[multi]set::insert()	240	要么成功，要么无任何影响——对单个元素而言
[multi]set::erase()	242	不抛出异常
[multi]set::clear()	244	不抛出异常
[multi]set::swap()	237	不抛出异常——前提是对“比较准则”执行复制/赋值操作时不抛出异常
[multi]map::insert()	240	要么成功，要么无任何影响——对单个元素而言
[multi]map::erase()	242	不抛出异常
[multi]map::clear()	244	不抛出异常
[multi]map::swap()	237	不抛出异常——前提是对“比较准则”执行复制/赋值操作时不抛出异常