

第 1 章



STL 介绍

本章将会说明标准模板库(Standard Template Library, STL)背后的基本思想, 让你对 STL 的各种对象如何相互支持有一个全局的把握。你将看到关于本章内容的更深入示例和讨论。本章将介绍以下内容:

- 什么是 STL
- 如何定义和使用模板
- 什么是容器
- 什么是迭代器以及如何使用它们
- 智能指针的重要性以及它们在容器中的用法
- 什么是算法, 如何运用它们
- 数值库提供了哪些支持
- 什么是函数对象
- 如何定义和使用 lambda 表达式

除了介绍 STL 背后的一些基本思想, 本章将会对一些你需要熟悉的 C++ 语言特性做一个简短的回顾, 因为在后续章节中会频繁地使用它们。如果已经熟悉某些章节的主题, 可以略过它们。

1.1 基本思想

STL 是一个功能强大且可扩展的工具集，用来组织和处理数据。为了最大限度地满足各种类型数据的需求，这个工具集全部由模板定义。虽然断定你肯定比较熟悉如何定义和使用类模板、函数模板，但本书还是会在下一节为你介绍一些它们的要领。STL 可以被划分为 4 个概念库：

- **容器库(Containers Library)**定义了管理和存储数据的容器。这个库的模板被定义在以下几个头文件中：array、vector、stack、queue、deque、list、forward_list、set、unordered_set、map 以及 unordered_map。
- **迭代器库(Iterators Library)**定义了迭代器，迭代器是类似于指针的对象，通常被用于引用容器类的对象序列。这个库被定义在单个的头文件 iterator 中。
- **算法库(Algorithms Library)**定义了一些使用比较广泛的通用算法，可以运用到容器中的一组元素上。这个库的模板被定义在 algorithm 头文件中。
- **数值库(Numerics Library)**定义了一些通用的数学函数和一些对容器元素进行数值处理的操作。这个库也包含一些用于随机数生成的高级函数。这个库的模板被定义在 complex、cmath、valarray、numeric、random、ratio 以及 cfloat 这些头文件中。其中 cmath 已经有一些年头了，由于它包含了很多的数学函数，因此在 C++ 11 中得到了扩展。

只需要几行简洁明了的 STL 代码，很多复杂困难的任务都可以轻松地完成。例如，以下代码可以从标准输入流中读取任意数目的浮点数，然后计算并输出它们的平均值：

```
std::vector<double> values;
std::cout << "Enter values separated by one or more spaces. Enter Ctrl+Z
to end:\n ";
values.insert(std::begin(values), std::istream_iterator<double>(std::cin),
std::istream_iterator<double>());
std::cout << "The average is "
<< (std::accumulate(std::begin(values), std::end(values),
0.0)/values.size())
<< std::endl;
```

只有 4 行代码，虽然每行都很长，但是我们没有使用循环去输入，因为这些工作 STL 已帮我们完成。可以很容易地将这段代码修改为从文件中获取数据，然后实现上面同样的任务。STL 强大、广泛的适用性，使它已经成为任何 C++程序员必备的工具箱。所有的 STL 名称都被定义在 std 命名空间中，所以不会在代码中显式地使用 std 来限定 STL 名称。当然，本书会在任何必要的地方使用 std 来限定。

1.2 模板

模板是一组函数或类的参数实现。编译器能够在需要使用函数或类模板时，用模板生

成一个具体的函数或类的定义，也可以定义参数化类型的模板。因此模板并不是可执行代码，而是用于生成代码的蓝图或配方。在程序中，如果一个模板从来没有被使用过，那么它将会被编译器忽略，不会生成可执行代码。一个没有被使用的模板可能会包含一些编程错误，并且包含这个模板的程序仍然可以编译和运行。当模板用于生成代码时，也就是当它被编译时，模板中的错误才会被编译器标识出来。

从模板生成的函数或类的定义是模板的实例或实例化。模板的参数值通常是数据类型，所以一个函数或类的定义可以通过 `int` 类型的参数值生成，也可以通过 `string` 类型的参数值生成。形参的参数值不一定是数据类型(非类型模板参数)；一个具体的形参可能是 `int` 类型，那么它就需要一个 `int` 类型的实参。下面是一个非常简单的函数模板示例：

```
template <typename T> T& larger(T& a, T& b)
{
    return a > b ? a : b;
}
```

这是一个可以比较两个参数值大小且返回较大参数值的函数的模板，使用这个模板的唯一限制是参数值的类型必须是可以用 `>` 比较的。类型参数 `T` 决定了具体创建什么样的模板实例。当使用 `larger()` 时，编译器可以从提供的参数中推断出具体的参数类型，即使是隐式提供。例如：

```
std::string first {"To be or not to be"};
std::string second {"That is the question."};
std::cout << larger(first, second) << std::endl;
```

上述代码需要包含 `string` 头文件。编译器可以从输入参数推断出 `T` 为 `string` 类型。如果想指定 `T` 为 `string` 类型，可以编写语句 `larger<std::string>(first,second)`。当函数的两个输入参数类型不同时，需要明确指出模板类型参数。例如，如果编写语句 `larger(2,3.5)`，编译器在推断 `T` 的类型时，就会产生多义性——`T` 可以是 `int`，也可以是 `double`，这样使用就会产生错误消息。如果将 `larger(2,3.5)` 写为 `larger<double>(2,3.5)`，就可以消除这个问题。

以下是一个模板类示例：

```
template <typename T> class Array
{
private:
    T* elements;                // Array of type T
    size_t count;              // Number of array elements

public:
    explicit Array(size_t arraySize);    // Constructor
    Array(const Array& other);           // Copy Constructor
    Array(Array&& other);                // Move Constructor
    virtual ~Array();                   // Destructor
    T& operator[](size_t index);         // Subscript operator
    const T& operator[](size_t index) const; // Subscript operator-const
                                         // arrays
    Array& operator=(const Array& rhs);  // Assignment operator
    Array& operator=(Array&& rhs);       // Move assignment operator
```

```
    size_t size() { return count; }           // Accessor for count
};
```

`size_t` 类型别名定义在 `cstdint` 头文件中，表示无符号整型。上面的代码是元素类型为 `T` 的数组模板。如果愿意，可以这样隐式地定义数组 `Array<T>`。在模板定义的外面——外部成员函数的定义中，必须这样写 `Array<T>`。赋值操作符允许将一个 `Array<T>` 对象赋值给另一个对象，这个操作在普通数组中是无法完成的。如果想禁用这个功能，仍然需要声明 `operator=()` 为模板的成员函数。如果不这样做，编译器会自动为模板实例生成一个默认的 `public` 类型的赋值操作函数。为了阻止这种情况发生，应该按照如下方式进行定义：

```
Array& operator=(const Array& rhs)=delete; // No assignment operator
```

一般来说，如果想定义任何拷贝构造函数或移动构造函数、拷贝赋值运算符或移动赋值运算符以及析构函数，都需要定义这 5 个类成员，或者用 `deleted` 指定你不想要的成员。

■ **注意：** 实现了移动构造函数和移动赋值运算符的类都有移动语义。

`size()` 成员函数在类模板中默认以内联的方式实现，所以不再需要外部定义。类模板成员函数的外部定义是放在头文件中的模板，通常和类模板在一个头文件中。一个成员函数甚至可以不依赖类型参数 `T`，如果 `size()` 在类模板中没有定义，那么 `size()` 将需要一个模板定义。定义了成员函数的模板的类型参数列表必须和类模板的类型参数列表相同，下面介绍如何定义构造函数：

```
template <typename T>           // This is a function template with
                                // parameter T
Array<T>::Array(size_t arraySize) try : elements {new T[arraySize]}, count
    {arraySize}
{}
catch(const std::exception& e)
{
    std::cerr << "Memory allocation failure in Array constructor." << std::endl;
    rethrow e;
}
```

为元素分配内存的操作可能会抛出一个异常，所以构造函数是一个 `try` 代码块，这能捕获到异常，处理必须被重新抛出的异常——如果不重新抛出那些 `catch` 语句块中的异常，那么它会被重新抛出。在构造函数名称的限定中，模板类型参数 `T` 是必要的，因为它将函数模板的定义和类模板关联起来了。需要注意的是，对于成员函数，不必使用 `typename` 这个关键字，它仅用于模板的参数列表中。

当然，可以以内联的方式为类模板的成员函数指定一个外部模板。例如，下面的代码将展示 `Array` 模板的拷贝构造函数可能的定义方式：

```
template <typename T>
inline Array<T>::Array(const Array& other)
try : elements {new T[other.count]}, count {other.count}
{
}
```

```

    for (size_t i {}; i < count; ++i)
        elements[i] = other.elements[i];
}
catch (std::bad_alloc&)
{
    std::cerr << "memory allocation failed for Array object copy." << std::endl;
}

```

这就假定 T 类型适用于赋值运算符，在使用它之前，你不会看到模板的代码，也不会意识到赋值运算符上的依赖。这说明除了先前提到的那些动态分配内存的类的其他成员，定义一个赋值运算符是多么重要。

■ **注意：**当指定一个模板的参数时，`class` 和 `typename` 关键字是可以互换使用的。因此，当定义一个模板时，可以写作 `template<typename T>` 或 `template<class T>`。因为 T 不必是类类型，所以本书更喜欢选择使用 `typename`，因为当 T 参数既可以是基本类型，也可以是类类型时，这种表达看起来更直观。

编译器因为一个特定类型对象的定义，实例化了一个类模板。这里有一个示例：

```
Array<int> data {40};
```

每个类模板类型参数都需要一个实参，除非有默认实参。当这条语句被编译时，发生了三件事：因为 `Array<int>` 类的定义被创建，所以确定了参数类型；因为必须调用构造函数去生成一个对象，所以生成了构造函数的定义；析构函数也被生成用来销毁对象。这就是编译器创建和释放一个数据对象所做的全部事情，也是这个时候从模板生成的唯一代码。通过用 `int` 替换模板中的 T，生成了类的定义，这非常巧妙。编译器只编译程序使用的成员函数，所以不需要得到模板参数被替换的完整类。基于数据对象被定义的规则，可以这样定义类：

```

class Array<int>
{
private:
    int* elements;
    size_t count;

public:
    explicit Array(size_t arraySize);
    virtual ~Array();
};

```

可以看到，只有构造函数和析构函数这两个成员函数。编译器不会创建任何和生成对象实例无关的东西，也不会包含程序中没有用到的模板代码。

可以为模板定义别名，当使用 STL 时，这一点非常有用。这里有一个为模板定义别名的示例：

```
template<typename T> using ptr = std::shared_ptr<T>;
```

这个模板定义 `ptr<T>` 作为智能指针模板类型 `std::shared_ptr<T>` 的别名，这样的话就可

以在代码中用 `ptr<std::string>` 代替 `std::shared_ptr<std::string>`，这显然更简洁，更容易阅读。下面使用 `using`，代码会更加简化：

```
using std::string;
```

现在就可以在代码中用 `ptr<string>` 代替 `std::shared_ptr<std::string>`，模板的类型别名能使代码更容易阅读，也更方便编写。

1.3 容器

容器是 STL 各种功能的基础，因为 STL 的其他部分和容器都有关联。容器是一个以特定方式存储和组织其他对象的对象。当使用容器时，不可避免地会使用 `iterators` 去访问其数据，所以你需要对 `iterators` 有一个较好的理解。STL 提供了以下几种类型的容器：

- **序列容器(Sequence Containers)**以线性组织的方式存储对象，和数组类似，但是不需要连续的存储空间。可以通过调用成员函数或者通过一个迭代器以线性方式访问对象。在一些地方，可以通过索引下标(`[]`)的方式来访问。
- **关联容器(Associative Containers)**存储了一些和键关联的对象。可以通过一个相关的键从关联容器中获取对应的值。也可以通过迭代器从关联容器中得到对象。
- **容器适配器(Container Adapters)**是提供了替换机制的适配类模板，可以用来访问基础的序列容器或关联容器。

有一点需要特别注意，除非对象是右值的——临时对象——一种有移动语义的类型，否则所有的 STL 容器存储的都是对象的副本。STL 要求移动构造函数和赋值运算符必须被指定为 `noexcept`，这就意味着它们不会抛出异常。如果在容器中添加了一个不具有移动特性的对象，然后修改了原始对象，那么原始对象和容器中的对象将会不同。然而，当获取一个对象时，得到的是一个容器中对象的引用，所以可以修改这个对象的值。对象的副本由拷贝构造函数获得。对于一些对象来说，拷贝可能会产生大量的开销，这样的话，最好在容器中存储这个对象的指针。如果这个对象的移动语义被实现，也可以存储移动对象。

■ **警告：**不要在以基类为元素的容器中存放派生类，这将会导致派生类对象的切片。如果想在容器中访问派生类，则需要使用多态。也就是说，在存储基类指针或基类智能指针的容器中存储派生类指针。

容器在堆上存放对象，并且自动管理它们所占用的内存。存储类型 `T` 的容器的空间分配由 `allocator` 管理，`allocator` 的类型由模板的参数指定。默认的参数类型是 `std::allocator<T>`，一个这种类型的 `allocator` 为对应类型的对象分配堆空间，这使你有了自己提供 `allocator` 的可能性。因为性能，你可能选择自己做这些事，但是这种情况很少。大多数时候，默认的 `allocator` 性能是不错的。如何定义一个 `allocator` 是一个高级话题，在本书中并不想做深入讨论，所以当最后一个模板的参数表示 `allocator` 时，省略了它，使用默认值。`std::vector<typename T, typename Allocator>` 模板有一个默认的 `allocator`——`std::allocator<T>`，所以可以简写为 `std::vector<typename T>`。指出这一点，主要是想让你知

道可以自己提供一个 `allocator`。

一个 `T` 类型的对象如果要存放在容器中，需要满足一些特定的要求，这些要求最终取决于你对这些元素执行的操作。一个容器通常会复制、移动、交换其元素。这有一个基本满足在容器中存放 `T` 类型对象的示例：

```
class T
{
public:
    T();                // default constructor
    T(const T& t);      // Copy constructor
    ~T();              // Destructor
    T& operator=(const T& t); // Assignment operator
};
```

考虑到在大多数情况下，编译器已经默认实现了上述几种成员函数，这也是大多数类需要实现的成员函数。注意到我们并没有为 `T` 定义 `operator<()` 函数，但是如果在关联容器(例如 `map`、`set`)中使用对象的话，对象需要定义 `operator<()`。如果容器的元素没有提供小于运算符的话，像 `sort()` 和 `merge()` 这类算法就不能运用到这些元素上。

■ **注意：**如果对象类型不符合所用容器的一些要求，或者你误用了容器模板，就可能会得到一些编译器对于 STL 内部头文件的编译错误。当这种事情发生时，不要急于提交 STL 中的错误，而是小心检查使用了 STL 的代码。

1.4 迭代器

迭代器是一个行为类似于指针的模板类对象。只要迭代器 `iter` 指向一个有效对象，就可以通过使用 `*iter` 解引用的方式来获取一个对象的引用。如果 `iter` 指向一个可以访问成员类对象，类成员就可以通过 `iter->member` 来使用。所以，使用迭代器就像使用指针。

当要以某种方式处理容器的元素时，可以通过使用迭代器来访问它们，尤其是要使用 STL 算法时，迭代器将算法和不同类型容器的元素联系了起来。迭代器将算法和数据源分离出来，算法不需要知道数据来源于容器。迭代器是一个定义在 `iterator` 头文件中的模板类型实例，这个头文件被所有定义了容器的头文件所包含。

通常会使用一对迭代器来定义一段元素，这些元素可能是容器中的对象，或是标准数组中的元素，或是字符串中的字符，总之可以是任意支持迭代器的对象的元素。一段元素是一个通过起始迭代器指向第一个元素，通过结束迭代器指向最后一个元素的元素序列。甚至当这个元素序列是这个容器中的元素的子集时，第二个迭代器仍然指向这个序列的最后一个元素而不是这段元素的最后一个元素。结束迭代器表示一个容器中的所有元素，不会指向具体某个元素，所以不能解引用。STL 中的迭代器提供一个标准的机制用于识别一段元素。一段元素的规格是独立于元素来源的，因此一个给定的算法可以运用到来自于任何容器的一段元素上，只要迭代器能够满足这个算法的一些要求。稍后会介绍更多不同迭代器的一些特性。

一旦明白迭代器是如何工作的，就很容易定义自己的模板函数来处理那些数据序列，它们由迭代器作为参数指定，然后函数模板实例就可以被运用到来自任何数据源的一段元素上。这段代码做的工作其实就像从数组中读取数据。你将在后面看到一个这样操作的示例。

1.4.1 获取迭代器

可以通过调用容器对象的 `begin()` 和 `end()` 函数来获取容器的迭代器；返回的这两个迭代器分别指向第一个和最后一个元素。`end()` 返回的迭代器并没有指向一个有效的元素，所以既不能解引用，也不能递增它。字符串类就有这样的函数，所以也可以这样去获取它们的迭代器。还可以将容器作为参数，通过调用全局函数 `std::begin()` 和 `std::end()` 来获取迭代器，这和使用容器的 `begin()` 和 `end()` 来获取迭代器是一样的，这两个全局函数被定义在 `iterator` 头文件的模板中。全局函数 `begin()` 和 `end()` 也可以用普通的数组或字符串对象作为参数，因此这是一种通用的获取迭代器的方式。

迭代器允许以一种逐步自增步进的方式从一个元素移到另一个元素，就像图 1-1 演示的那样。图 1-1 中的容器表明，它可能是一个字符串对象或数组，也可能是一个 STL 容器。通过将自增的起始迭代器和结束迭代器进行比较来判断是否访问到最后一个元素。你还能对迭代器进行一些其他操作，不过这些操作取决于迭代器的类型，反过来也取决于使用的容器的类型。有两个全局函数 `cbegin()` 和 `cend()` 可以返回数组、容器或字符串对象的常量迭代器。记住，常量迭代器指向的是一个常量，但是仍然可以修改这个迭代器本身的值。本章稍后的部分会介绍一些返回其他类型迭代器的全局函数。

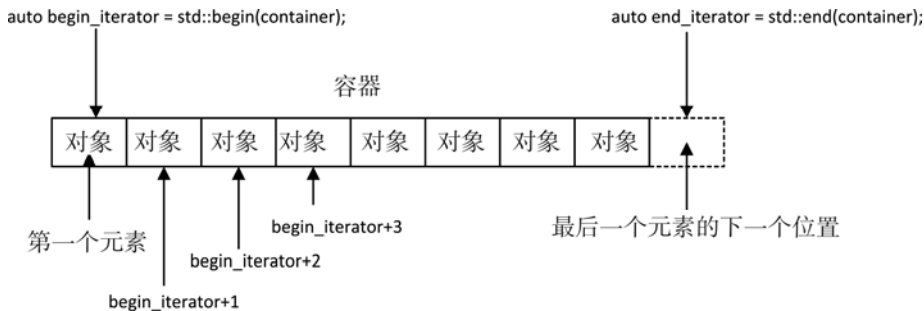


图 1-1 迭代器的操作

1.4.2 迭代器的类别

所有类型的迭代器都必须有一个拷贝构造函数、一个析构函数以及一个拷贝赋值运算符。迭代器指向的对象必须是可交换的(`swappable`)；下一节将进一步解释这意味着什么。5 种类型的迭代器分别反映了它们不同层次的功能。不同的算法要求具有不同功能的迭代器，以此来确认算法能做哪些操作。不同类别的迭代器并不是一种新类型的迭代器模板。通过确定迭代器模板的参数，可以确认一个迭代器类型支持的类别。稍后会在本章进行更深入的解释。

从一个容器能够获取什么类型的迭代器，取决于这个容器的类型。算法可以通过传入

的迭代器参数的类别，来判断迭代器具有什么样的功能。一个算法可以两种方式使用迭代器参数的类别：第一种，为了满足操作，它会确立需要满足的最小功能要求；第二种，如果超出了迭代器的最小要求，算法使用扩展的功能可以更高效地执行运算。当然，算法只能运用到那些能提供具有要求功能的迭代器的容器中的元素上。

下面我们从简单到复杂，依次列出不同类别的迭代器：

1) **输入迭代器(input iterators)**提供对对象的只读访问。如果 `itex` 是一个输入迭代器，它必须支持表达式 `*iter` 以引用它所指向的值。输入迭代器只能单个使用，这也就意味着迭代器自增后，只能通过使用新的迭代器来访问上一个它所指向的元素。每次想访问一个序列时，就需要创建一个新的迭代器。可以对输入迭代器做这些操作：`++iter` 或 `iter++`，`iter1==iter2` 和 `iter1!=iter2`。需要注意的是，`*iter` 并没有减量运算，可以在输入迭代器上使用 `iter->member` 表达式。

2) **输出迭代器(output iterators)**提供对对象的只写访问。如果 `iter` 是一个输出迭代器，那么它允许我们赋一个新的值，因此支持表达式 `*iter=new_value`。输出迭代器只能单个使用。每次想写一个序列时，需要创建一个新的迭代器。可以对输出迭代器做这些操作：`++iter` 或 `iter++`，以及 `*iter`。需要注意的是，`*iter` 并没有减量操作，输出迭代器是只写的，所以不能使用 `iter->member` 这样的表达式。

3) **正向迭代器(forward iterators)**结合了输入和输出迭代器的功能，并且可以使用多次，因此可以随意多次用它进行读或写操作。进行什么样的操作取决于何时需求正向迭代器。`replace()`算法搜索一个序列，替换所有需要正向迭代能力的元素。

4) **双向迭代器(bidirectional iterators)**具有和正向迭代器同样的功能，但允许进行前向和后向遍历。因此除了自增来移到另一个元素，也可以使用前缀或后缀的自减运算(`--iter` 和 `iter--`)来移到前一个元素。

5) **随机访问迭代器(random access iterators)**提供了和双向迭代器同样的功能，但是能够支持对元素的随机访问。除了支持双向迭代器的一些操作之外，也支持一些其他操作：

- 通过一个整数进行自增或自减：`iter+n`、`iter-n`、`iter+=n`、`iter-=n`。
- 通过一个整数进行索引访问：`iter[n]`，等同于 `*(iter+n)`。
- 两个迭代器之间的差异：`iter1-iter2`，得到两个迭代器之间元素的个数。
- 迭代器的比较：`iter1<iter2`、`iter1>iter2`、`iter1≤iter2`、`iter1≥iter2`。

对一个序列进行排序，需要用随机访问迭代器指定这个序列。可以对随机访问迭代器使用下标运算。例如一个随机访问迭代器 `first`，表达式 `first[3]`就等同于 `*(first+3)`，所以可以访问第4个元素。一般来说，使用 `iter[n]`这种表达式，`n`是一个相对于 `iter` 的偏移量，我们能够通过它得到一个距离 `iter` `n` 个距离的元素的引用。需要注意的是，在下标中使用的索引值并没有被检查，所以无法阻止索引越界。

每种类别的迭代器都由一个称作迭代器标签类(iterator tag class)的空类来指定，`iterator tag class` 通常作为迭代器模板的参数使用。`iterator tag class` 的唯一目的是指定一个特定类型的迭代器能够做什么，所以它们可以作为迭代器模板的参数使用。标准的 `iterator tag class` 如下所示：

```
input_iterator_tag
output_iterator_tag
```

```

forward_iterator_tag 派生于 input_iterator_tag
bidirectional_iterator_tag 派生于 forward_iterator_tag
random_access_iterator_tag 派生于 bidirectional_iterator_tag

```

以上这些类的继承结构反映了迭代器类别的累加性质。当一个迭代器模板的实例被创建时，第一个模板类型的参数是一个 `iterator tag class`，这将决定迭代器具有什么样的功能。在第 2 章将演示如何定义自己的迭代器，之后将介绍如何指定它的类别。

如果一个算法要求特定类别的迭代器，你就不能使用一个低级别的迭代器，而是需要使用一个高级别的迭代器。正向迭代器双向迭代器，随机访问迭代器可以是可变的或不可变的，这取决于解引用一个迭代器是否能够获取到一个引用或 `const` 引用。显然，不能对一个解引用的迭代器使用左赋值。

迭代器的特性取决于它来自容器的类型。例如，`vector` 和 `deque` 提供了随机访问迭代器，也就表明这些容器中的元素都可以随机访问。输入迭代器、输出迭代器、正向迭代器通常用于指定算法的参数，表明它们满足算法所要求的最小条件。本书稍后将在关于对容器中元素使用算法的示例的基础上，更深入地解释迭代器。在一个实际使用场景下，会更容易理解。同时，下面这个简单的示例演示了迭代器对一个数组进行的操作：

```

// Ex1_01.cpp
// Using iterators
#include <numeric>           // For accumulate() - sums a range of elements
#include <iostream>          // For standard streams
#include <iterator>          // For iterators and begin() and end()

int main()
{
    double data[] {2.5, 4.5, 6.5, 5.5, 8.5};
    std::cout << "The array contains:\n";
    for (auto iter = std::begin(data); iter != std::end(data); ++iter)
        std::cout << *iter << " ";
    auto total = std::accumulate(std::begin(data), std::end(data), 0.0);
    std::cout << "\nThe sum of the array elements is " << total << std::endl;
}

```

可以看到，我们以数组作为全局函数 `begin()`和 `end()`的参数，得到了数组元素的迭代器。我们用这个迭代器循环列出了元素的值，用表达式`*iter`来解引用一个迭代器，以获取引用对象的值。当然，可以在循环中自增 `iter`，如下所示：

```

for (auto iter = std::begin(data); iter != std::end(data);)
    std::cout << *iter++ << " ";

```

直接包含 `iterator` 头文件是多余的，因为在容器的头文件中都包含了 `iterator`。我们包含了头文件 `numeric`，这个头文件定义了 `accumulate()`函数的模板。通过两个参数指定元素的范围——一个指向第一个元素，另一个指向最后一个元素，这样 `accumulate()`就可以计算出这些元素的和，第三个元素是元素和的初始值。`accumulate()`函数适用于任何支持加法运算的序列，同样也适用于任何定义了 `operator+`函数的类对象。

■ 注意：当对容器使用 `accumulate()` 时，你会发现另一个版本的函数模板，它允许你指定一个双目运算来替代默认的`+`。

1.4.3 流迭代器

可以使用流迭代器，在流和可以通过迭代器访问的数据源之间传输文本模式的数据。因为 STL 算法可以通过指定一对迭代器来输入序列，所以对于任何可以通过输入流访问的数据源，都可以将算法运用到它们的元素上。这就意味着，例如算法，可以像运用到容器中的对象上一样，运用到流中的对象上。算法可以运用到任何地方，只要能提供一个合适的迭代器。稍后，将解释如何使一个迭代器变得适用。同样地，也可以使用一个输出流迭代器，将一系列的元素传送到一个输出流。标准流迭代器有一个作为基类的迭代器模板。

可以创建流迭代器对象，用来处理指定类型的流对象数据。数据的类型就是迭代器模板的参数类型，流对象是构造函数的参数。一个 `istream_iterator<T>` 就是一个输入流迭代器，它可以从一个 `istream` 读取 `T` 类型的对象。`istream` 可以是一个文件流，也可以是一个标准的输入流 `cin`。我们通过使用 `>>` 运算符来读取对象。所以读取的对象要支持这个运算符。

`istream_iterator<T>` 无参构造函数可以创建一个结束迭代器来判断流是否结束。显然，流迭代器并不能用来传输混合类型的数据。`istream_iterator` 对象默认会忽略空格，可以通过使用 `std::noskipws` 操作底层的输入流来覆盖这个特性。一个 `istream_iterator` 只能使用一次，如果想再次从流中读取对象，需要使用一个新的 `istream_iterator` 对象。

输出流迭代器(`ostream_iterator`)弥补了输入流迭代器(`istream_iterator`)的不足，它为对象提供了一次向输出流输出的功能。使用 `<<` 运算符创建对象。当创建一个 `ostream_iterator` 对象时，可以选择字符串分隔符，用来分隔每个输出对象。

这里有一个使用输入流迭代器的示例：

```
// Ex1_02.cpp
// Using stream iterators
#include <numeric>           // For accumulate() - sums a range of elements
#include <iostream>          // For standard streams
#include <iterator>          // For istream_iterator

int main()
{
    std::cout << "Enter numeric values separated by spaces and enter Ctrl+Z
        to end:" << std::endl;

    std::cout << "\nThe sum of the values you entered is "
        << std::accumulate(std::istream_iterator<double>(std::cin),
            std::istream_iterator<double>(), 0.0)
        << std::endl;
}
```

这段代码通过 `accumulate()` 函数，计算出 `cin` 的输入流迭代器输入的一些数据的和。我们可以输入任意个数的数据。第二个参数是一个流结束迭代器，当出现流结束的条件时(就像文件流 EOF 标识)，用来和第一个参数指定的迭代器进行匹配；当从键盘输入 `Ctrl+Z` 时，

会导致这样的事情发生。

1.4.4 迭代器适配器

迭代器适配器是一个类模板，它为标准迭代器提供了一些特殊的行为，使它们能够从迭代器模板得到派生。适配器类模板定义了三种不同的迭代器：反向迭代器(reverse iterators)、插入迭代器(insert iterators)和移动迭代器(move iterators)。它们是通过下面几个模板类定义的：reverse_iterator、insert_iterator 和 move_iterator。

1. 反向迭代器

反向迭代器的工作机制和标准迭代器的相反，可以创建双向或随机迭代器版本的反向迭代器。容器的成员函数 rbegin()和 rend(), 分别返回一个指向最后一个元素的反向迭代器，以及一个指向最开始元素的前一个位置的反向迭代器。同样名称的全局函数，作用和成员函数是一样的，如图 1-2 所示。

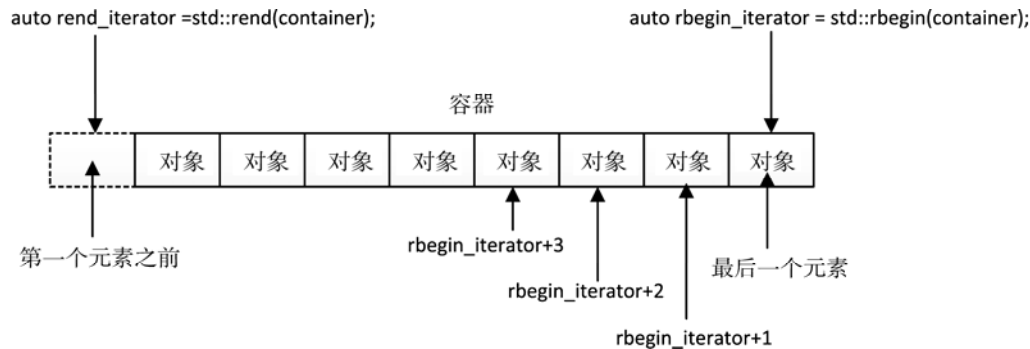


图 1-2 反向迭代器的操作

可以像使用标准随机访问迭代器一样，使用反向随机访问迭代器进行下标操作，效果是相反的。对于标准迭代器 `iter` 来说，表达式 `iter[n]` 代表在 `iter` 后面，和 `iter` 指向的元素有 `n` 个元素间隔的元素，所以等同于 `*(iter+n)`。对于反向迭代器 `riter` 来说，表达式 `riter[n]` 等同于 `*(riter+n)`，所以 `riter[n]` 代表在 `riter` 前面，和 `riter` 指向的元素有 `n` 个元素间隔的元素。

图 1-3 显示了容器的反向迭代器和标准迭代器的关系。可以看到，容器元素的反向迭代器相对于正常的迭代器来说，向左偏移了一些。每一个反向迭代器都包含了一个标准迭代器，里面的迭代器也有相似的移位，所以没有指向相同的元素。每一个反向迭代器对象都有一个成员函数 `base()`，可以返回一个基础的迭代器，因为它也是一个标准的迭代器，所以和反向迭代器的作用是相反的。反向迭代器 `riter` 的基础迭代器指向 `riter` 的下一个位置，如图 1-3 所示。有些容器的成员函数并不接受反向迭代器，当准备使用一个算法时，正好出现这种情况，已经使用了反向迭代器，可以调用 `base()` 去获取一个和反向迭代器对应的标准迭代器。显然，需要注意的事实是，基础迭代器总是指向反向迭代器指向的下一个位置，你会在下一节更多地了解这些内容。

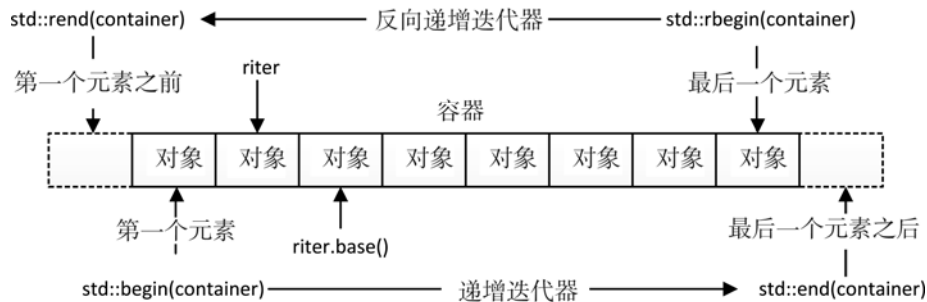


图 1-3 标准迭代器和反向迭代器是如何和容器关联的

2. 插入迭代器

尽管插入迭代器基于标准迭代器，但是它们的功能却有很大不同。一般的迭代器只能访问或改变序列中存在的元素。插入迭代器通常用于在容器的任何位置添加新的元素。插入迭代器不能被运用到标准数组和 `array<T,N>` 这样的容器上，因为它们的元素个数是固定的。

有三种插入迭代器：

- 后向插入迭代器(`back_insert_iterator`)通过调用成员函数 `push_back()` 将一个新的元素添加到容器的尾部。`vector`、`list`、`deque` 容器都有一个 `push_back()` 函数。如果容器没有定义 `push_back()` 函数，那么后向插入迭代器无法使用。以容器作为全局函数 `back_inserter()` 的参数传入可以得到一个后向插入迭代器对象。
- 前向插入迭代器(`front_insert_iterator`)通过调用成员函数 `push_front()` 将一个新的元素添加到容器的头部。`list`、`forward_list`、`deque` 容器都有一个 `push_front()` 函数。如果容器没有定义 `push_front()` 函数，那么前向插入迭代器无法使用。以容器作为全局函数 `front_inserter()` 的参数传入可以得到一个前向插入迭代器对象，显然这些容器必须是 `list`、`forward_list` 或 `deque` 容器类型。
- 可以使用插入迭代器(`insert_iterator`)向任何有 `insert()` 函数的容器中插入一个新的元素。因为在 `string` 类的头文件中定义了一个 `insert()` 函数，所以插入迭代器可以正常使用。以容器作为全局函数 `inserter()` 的第一个参数可以得到一个插入迭代器对象，第二个参数是一个指向容器中插入位置的迭代器。

插入迭代器一般被用做拷贝或生成一段元素的算法的参数。在后面的章节你会看到它们的具体应用。

3. 移动迭代器

移动迭代器是从普通的迭代器中创建的，这个迭代器指向一定范围内的元素。可以用移动迭代器将某个范围的类对象移动到目标范围，而不需要通过拷贝去移动。可以将移动迭代器作为输入迭代器，将所指向的对象转换为右值引用，这样就能移动对象而不是去拷贝。移动迭代器会使指向的一段数据元素处于一种不确定的状态，这样就不能再次使用它们了。例如，`make_move_iterator()` 被定义在 `iterator` 头文件中，因为可以将从 `begin()` 和 `end()` 得到的迭代器作为 `make_move_iterator()` 的参数来得到一个移动迭代器，所以可以通过将从 `begin()` 和 `end()` 获取的迭代器作为 `make_move_iterator()` 的参数来移动容器中的一段元素。在本书稍后部分，你将会看到使用移动迭代器的示例。

1.5 迭代器上的运算

迭代器的头文件定义了 4 个实现了迭代器运算的函数模板：

- `advance()`需要两个参数，第一个参数是迭代器 `iter`，第二个参数是整数值 `n`，将 `iter` 移动了 `n` 个位置。迭代器可以是任何具有输入迭代器功能的迭代器；如果迭代器是一个双向迭代器或随机访问迭代器，第二个参数也可以是负数。这个函数没有返回值。例如：

```
int data[] {1, 2, 3, 4, 5, 6};
auto iter = std::begin(data);
std::advance(iter, 3);
std::cout << "Fourth element is " << *iter << std::endl;
```

- `distance()`需要两个迭代器作为参数，可以返回两个迭代器之间的元素个数。例如：

```
int data[] {1, 2, 3, 4, 5, 6};
std::cout << "The number of elements in data is "
          << std::distance(std::begin(data), std::end(data))
          << std::endl;
```

- `next()`需要两个参数，第一个参数是迭代器 `iter`，第二个参数是整数值 `n`，`n` 的默认值是 1。通过这两个参数，`next()`可以得到 `iter` 正向偏移 `n` 之后所指向位置的一个迭代器。迭代器 `iter` 必须是具有正向迭代器能力的迭代器。例如：

```
int data[] {1, 2, 3, 4, 5, 6};
auto iter = std::begin(data);
auto fourth = std::next(iter, 3);
std::cout << "1st element is " << *iter << " and the 4th is " << *fourth <<
std::endl;
```

- `prev()`需要两个参数，第一个参数是 `iter`，第二个参数是整数值 `n`，`n` 的默认值为 1。通过这两个参数，`prev()`返回 `iter` 反向偏移 `n` 之后所指向位置的一个迭代器。迭代器 `iter` 必须是具有双向迭代器功能的迭代器。

```
int data[] {1, 2, 3, 4, 5, 6};
auto iter = std::end(data);
std::cout << "Fourth element is " << *std::prev(iter, 3) << std::endl;
```

显然这些函数都能通过使用随机访问迭代器来获取算术运算后的结果，但是其他类型的低级别迭代器就不能。这些函数能够简化代码。在其他不具有这种能力的迭代器上，如果想实现和 `advance()` 相同的功能，需要写一个循环。

1.6 智能指针

指针是 C++ 语言的一部分，通常叫作原生指针，因为这种类型的变量都指向一个地址。原生指针可以指向自动变量、静态变量或堆上生成的变量。智能指针是一个可以模仿原生

指针的模板类。在很多情况下，它们没有任何差别。但是，有些情况下，它们还是有两点差别：

- 智能指针只能用来保存堆上分配的内存的地址。
- 不能像对原生指针那样，对智能指针进行一些自增或自减这样的算术运算。

对于在自由存储区创建的对象，通常使用智能指针而不是原生指针。智能指针的巨大优势是，不用担心不小心释放了堆内存，因为当不需要时，一个对象的内存会被分配给一个智能指针。智能指针自动释放当对象不再需要时，这意味着消除了内存泄漏的可能性。

可以在容器中使用智能指针，当使用一个类对象时，这一点尤其有用。如果使用一个类的基类作为智能指针的类型参数，可以用它指向一个派生类对象，这保存了一个对象的指针而不是对象，因而能够保持这个对象的多态性。智能指针的模板定义在 `memory` 头文件中，必须在代码中包含这个头文件来使用它们。在 `std` 命名空间中定义了三种不同类型的智能指针模板：

- `unique_ptr<T>` 对象就像一个指向类型 `T` 的指针，而且 `unique_ptr<T>` 是排他的，这意味着不可能有其他的 `unique_ptr<T>` 指向同一个地址。一个 `unique_ptr<T>` 对象完全拥有它所指向的内容。不能指定或复制一个 `unique_ptr<T>` 对象。可以通过使用一个定义在 `utility` 头文件中的 `std::move()` 函数来移出一个 `unique_ptr<T>` 对象存储的地址。在进行这个操作之后，之前的 `unique_ptr<T>` 变为无效。当需要独占一个对象的所有权时，可以使用 `unique_ptr<T>`。
- `shared_ptr<T>` 对象就像一个指向类型 `T` 的指针，和 `unique_ptr<T>` 不同的是，多个 `shared_ptr<T>` 可以指向同一个地址，因此 `shared_ptr<T>` 允许共享一个对象的所有权。引用计数保存了指向给定地址的 `shared_ptr<T>` 的数量，每当一个新的 `shared_ptr<T>` 指针指向一个特定堆地址时，引用计数就会加 1。当一个 `shared_ptr<T>` 被释放或者指向了新的地址时，引用计数就会减 1。当没有 `shared_ptr<T>` 指向这个地址时，引用计数将会变为 0，在堆上为这个对象分配的内存就会自动释放。所有指向同一个地址的 `shared_ptr<T>` 都能得到引用计数的值。
- `weak_ptr<T>` 可以从一个 `shared_ptr<T>` 对象创建，它们指向相同的地址。创建一个 `weak_ptr<T>` 不会增加 `shared_ptr<T>` 对象的引用计数，所以它会阻止指向的对象销毁。当最后一个 `shared_ptr<T>` 引用被释放或重新指向一个不同的地址时，它们所指向对象的内存将被释放，即使相关的 `weak_ptr<T>` 可能仍然存在。

使用 `weak_ptr<T>` 对象的主要原因是，我们可能在不经意间创建一个循环引用。循环引用从概念上说就是一个 `shared_ptr<T>` 对象 `pA`，指向了另一个 `shared_ptr<T>` 对象 `pB`，在这种情况下，两个对象都不能被释放。在实际中，循环引用的发生要比这个复杂。设计 `weak_ptr<T>` 就是为了避免循环引用的问题。通过使用 `weak_ptr<T>` 指向一个 `shared_ptr<T>` 所指向的对象，就可以避免循环引用，稍后会对此做一些解释。当最后一个 `shared_ptr<T>` 对象析构时，它所指向的对象也析构了，这个时候任何和 `shared_ptr<T>` 关联的 `weak_ptr<T>` 对象都指向一个无效的对象。

1.6.1 使用 `unique_ptr<T>` 指针

`unique_ptr<T>` 对象唯一指向了一个对象，所以它独享了这个对象的所有权。当一个 `unique_ptr<T>` 对象析构时，它所指向的对象也析构了。当不需要使用多个智能指针，想独享对象的所有权时，可以选择使用这种类型的智能指针。当一个对象被一个 `unique_ptr<T>` 指向时，也可以通过生成一个原生指针来访问对象。这里有一个使用构造函数生成一个 `unique_ptr<T>` 对象的示例：

```
std::unique_ptr<std::string> pname {new std::string {"Algernon"}};
```

在堆上生成的字符串对象被传入 `unique_ptr<string>` 构造函数。默认的构造函数会生成一个内部原生指针为空指针的 `unique_ptr<T>` 对象。

一种更好地生成 `unique_ptr<T>` 对象的方式是使用 `make_unique<T>()` 函数，这是一个定义在 `memory` 头文件中的函数模板：

```
auto pname = std::make_unique<std::string>("Algernon");
```

该函数通过将参数传递给类的构造函数，来生成一个堆上的字符串对象，并且返回一个指向这个对象的指针。可以按照 `T` 构造函数的要求，尽可能多地为 `make_unique<T>()` 函数提供参数。下面是一个示例：

```
auto pstr = std::make_unique<std::string>(6, '*');
```

这两个参数会被传入 `string` 构造函数中，生成一个包含 “*****” 的对象。

可以通过解引用的方式来使用一个对象，就像使用原生指针一样：

```
std::cout << *pname << std::endl; // Outputs Algernon
```

可以生成一个指向数组的 `unique_ptr<T>` 对象。例如：

```
size_t len{10};
std::unique_ptr<int[]> pnumbers {new int[len]};
```

这会生成一个 `unique_ptr` 对象，用来指向一个在空闲空间生成的具有 `len` 个元素的数组，同样也可以调用 `make_unique<T>()` 函数得到相同的结果：

```
auto pnumbers = std::make_unique<int[]>(len);
```

同样，也可以生成一个指针，用来指向一个在堆上生成的具有 `len` 个元素的数组。可以对 `unique_ptr` 变量使用索引来访问数组元素。这里展示了如何改变数组的值：

```
for(size_t i{} ; i < len ; ++i)
    pnumbers[i] = i*i;
```

这使数组元素的值等于它们所对应索引的平方值。当然，可以使用取下标的方式来输出数组的值：

```
for(size_t i{} ; i < len ; ++i)
    std::cout << pnumbers[i] << std::endl;
```

不能以传值的方式将一个 `unique_ptr<T>` 对象传入函数中，因为它们不支持拷贝，必须使用引用的方式。`unique_ptr<T>` 可以作为一个函数的返回值，因为它们不会被拷贝，但是它们必须以隐式移动运算的方式返回。

因为 `unique_ptr<T>` 不能被拷贝，所以只能通过移动或生成它们的方式，在容器中存放 `unique_ptr<T>` 对象。绝不会有两个 `unique_ptr<T>` 对象指向同一个对象，`shared_ptr<T>` 对象就没有这种特性，所以当需要多个指针指向一个对象时，或者当需要拷贝一个存放了智能指针的容器时，可以使用 `shared_ptr<T>`；否则，就要使用 `unique_ptr<T>`。在一个存放 `unique_ptr<T>` 类型元素的容器中，可能会需要一个指向可用对象的原生指针。下面的代码演示了如何从 `unique_ptr<T>` 获取一个原生指针：

```
auto unique_p = std::make_unique<std::string>(6, '*');
std::string pstr {unique_p.get()};
```

类的 `get()` 成员函数可以返回一个 `unique_ptr<T>` 所包含的原生指针。当需要访问一个对象时，指向它的智能指针被封装在一个类对象中。这时，就需要像上面那样做，这是一个典型的使用场景。因为 `unique_ptr<T>` 不能被拷贝，所以不能返回 `unique_ptr<T>`。

1. 重置 `unique_ptr<T>` 对象

当智能指针析构时，`unique_ptr<T>` 对象所指向的对象也会被析构。调用一个无参 `unique_ptr<T>` 对象的 `reset()` 函数可以析构它所指向的对象，`unique_ptr<T>` 对象中的原生指针将会被替换为空指针，这使你能够在任何时候析构智能指针所指向的对象。例如：

```
auto pname = std::make_unique<std::string>("Algernon");
...
pname.reset(); // Release memory for string object
```

也可以将一个新生成的 `T` 对象的地址值传给 `reset()` 函数。智能指针之前所指向的对象会被析构。然后它的地址值被替换为新对象的地址值：

```
pname.reset(new std::string{"Fred"});
```

`pname` 之前所指向的字符串对象的内存将会被释放，在内存中生成一个新的字符串对象——“Fred”，然后其地址被 `pname` 保存。

■ **警告：** 不要将其他 `unique_ptr<T>` 所指向的一个对象的地址值传给 `reset()`，或者去生成一个新的 `unique_ptr<T>` 对象，这种代码也许能通过编译，但是肯定会让程序崩溃。第一个 `unique_ptr<T>` 的析构会释放它所指向的对象的内存；第二个智能指针析构时，将会试图再次释放已经释放的内存。

可以调用智能指针的 `release()` 函数去释放一个 `unique_ptr<T>` 所指向的对象，这样就能够在不释放对象内存的情况下，将指向它的 `unique_ptr<T>` 内部的原生指针设为空指针。例如：

```
auto up_name = std::make_unique<std::string>("Algernon");
std::unique_ptr<std::string> up_new_name{up_name.release()};
```

`up_name`的成员函数 `release()`可以得到一个包含"Algernon"的字符串对象的原生指针,所以在执行完第二行代码时, `up_new_name` 会指向原始的那个字符串对象——"Algernon"。从 `unique` 指针转移到另一个指针时,也转移了所指向对象的所有关系。

可以通过交换两个 `unique_ptr<T>`指针的方式来交换两个对象:

```
auto pn1 = std::make_unique<std::string>("Jack");
auto pn2 = std::make_unique<std::string>("Jill");
pn1.swap(pn2);
```

在执行完第二行代码后, `pn1` 会指向字符串"Jill", `pn2` 指向字符串"Jack"。

3. 比较和检查 `unique_ptr<T>`对象

STL中非成员函数的模板函数定义了全套的比较运算符,用来比较两个 `unique_ptr<T>`对象或者比较一个 `unique_ptr<T>`对象和一个空指针(`nullptr`)。比较两个 `unique_ptr<T>`对象也就是比较它们两个的成员函数 `get()`返回的地址值,比较一个 `unique_ptr<T>`对象和空指针也就是将智能指针的成员函数 `get()`返回的地址和空指针作比较。

`unique_ptr<T>`可以隐式地转换为布尔值。如果一个对象包含一个空指针,将会被转换为 `false`; 否则转换为 `true`。这就意味着可以使用 `if` 语句来检查一个非空的 `unique_ptr<T>`对象:

```
auto up_name = std::make_unique<std::string>("Algernon");
std::unique_ptr<std::string> up_new{up_name.release()};
if(up_new) // true if not nullptr
    std::cout << "The name is " << *up_new << std::endl;
if(!up_name) // true if nullptr
    std::cout << "The unique pointer is nullptr" << std::endl;
```

当对一个 `unique_ptr()`指针对象调用 `reset()`或 `release()`时,需要先做这种检查,因为在解引用一个指针时,需要保证它是一个非空的 `unique_ptr<T>`指针。

1.6.2 使用 `shared_ptr<T>`指针

可以按如下方式定义一个 `shared_ptr<T>`对象:

```
std::shared_ptr<double> pdata {new double{999.0}};
```

可以通过解引用一个 `shared_ptr` 指针,使用它所指向的对象或者修改对象的值:

```
*pdata = 8888.0;
std::cout << *pdata << std::endl; // Outputs 8888.0
*pdata = 8889.0;
std::cout << *pdata << std::endl; // Outputs 8889.0
```

`pdata` 的定义过程,包含了一个 `double` 变量的堆内存的分配,以及另一个和智能指针相关的控制块的内存分配,这个控制块记录了智能指针的副本计数。分配堆内存是相当耗时的,可以通过使用一个定义在 `memory` 头文件中的 `make_shared<T>()`函数来得到一个

`shared_ptr<T>`对象，这个过程要有效率得多：

```
auto pdata = std::make_shared<double>(999.0); // Points to a double variable
```

尖括号中的数据类型指定了生成的变量的类型，函数名后圆括号中间的参数用来初始化生成的变量。

一般来说，`make_shared<T>()`函数的参数可以是任意数字，实际生成的数字取决于要生成的对象的类型。当需要使用 `make_shared<T>()`函数在堆上生成一个对象时，如果构造函数有要求的话，可以提供两个或更多个参数，这些参数用逗号隔开。`auto` 关键字可以使 `pdata` 的数据类型自动地从 `make_shared<T>()`返回的对象推导出来，所以事实上它是 `shared_ptr<double>`类型。尽管如此，当使用 `auto` 时，不要使用初始化列表，因为参数的类型会被推导为 `std::initializer_list`。

可以使用一个定义好的指针来初始化另一个 `shared_ptr<T>`指针：

```
std::shared_ptr<double> pdata2 {pdata};
```

`pdata2` 和 `pdata` 指向同一个变量，这会导致引用计数的增加。也可以将一个 `shared_ptr<T>`指针赋值给另一个 `shared_ptr<T>`指针：

```
std::shared_ptr<double> pdata{ new double{ 999.0 } };
std::shared_ptr<double> pdata2;      // Pointer contains nullptr
pdata2 = pdata;                     // Copy pointer - both point to the
                                   // same variable
std::cout << *pdata << std::endl;   // Outputs 999.0
```

当然，复制 `pdata` 会导致引用计数的增加。如果 `double` 变量使用的内存被释放，那么所有指向它的指针都需要被重置或析构。不能用 `shared_ptr<T>`指针来保存一个默认生成的数组，但是可以用来保存那些由你自己生成的 `array<T>`或 `vector<T>`容器对象。

■ **注意：**你可能会创建一个指向数组的 `shared_ptr<T>`指针，这需要为智能指针提供一个 `deleter` 函数去释放数组的堆内存，具体怎么做超出了本书的讨论范围。

`shared_ptr<T>`和 `unique_ptr<T>`的使用方式相似，通过使用 `shared_ptr<T>`的成员函数 `get()`，可以从 `shared_ptr<T>`得到一个原生指针。`pdata` 可以像先前的章节那样去定义。可以这样写：

```
auto pvalue = pdata.get(); // pvalue is type double* and points to 999.0
```

当需要使用一个原生指针时，可以像上面那样去使用。

■ **警告：**只能通过拷贝构造函数或赋值运算符去复制一个 `shared_ptr<T>`对象。通过一个由其他指针的 `get()`函数返回的原生指针，来生成一个 `shared_ptr<T>`指针。这可能会导致一些意想不到的问题，大多数情况也就意味着程序的崩溃。

1. 重置 `shared_ptr<T>`对象

如果将一个空指针赋给一个 `shared_ptr<T>`对象，那么它的地址值将会变为空，同样也

会使指针所指向对象的引用计数减 1。例如：

```
auto pname = std::make_shared<std::string>("Charles Dickens");
// Points to a string object
// ... lots of other stuff happening...
pname = nullptr; // Reset pname to nullptr
```

以"Charles Dickens"为初始值，生成一个在堆上存放的字符串对象，然后生成一个共享指针来保存这个地址值。最后，用空指针来替换 `pname` 所保存的值。当然，这时候任何其他保存了字符串对象地址的 `shared_ptr<T>` 对象仍然存在，因为引用计数还不等于 0。

通过调用 `shared_ptr<T>` 对象的无参 `reset()` 函数，可以得到同样的效果：

```
pname.reset(); // Reset to nullptr
```

也可以通过为 `reset()` 函数传入一个原生指针来改变共享指针指向的对象。例如：

```
pname.reset(new std::string{"Jane Austen"}); // pname points to new string
```

通过 `reset()` 的参数传入的地址对象，类型必须和之前存放在智能指针中的对象类型一致，或者必须可以隐式转换成之前的类型。

2. 比较和检查 `shared_ptr<T>` 对象

可以使用任意比较运算符来比较两个 `shared_ptr<T>` 保存的地址，或者将一个 `shared_ptr<T>` 保存的地址和空指针比较。最有用的比较运算符是 `=` 和 `!=`，这能够让我们知道两个指针是否指向同一个对象。给定两个 `shared_ptr<T>` 对象 `pA` 和 `pB`，它们指向同一类型，你能够像下面这样比较它们：

```
if((pA == pB) && (pA != nullptr))
    std::cout << " Both pointers point to the same object.\n";
```

这两个指针可能相等或者都是空指针，所以这样一种简单的比较并不能充分说明它们都指向同一个元素。像 `unique_ptr<T>` 指针对象和 `shared_ptr<T>` 指针对象，都可能被隐式转换成布尔值，所以不能像下面这样使用：

```
if(pA && (pA == pB))
    std::cout << " Both pointers point to the same object.\n";
```

也可以检查 `shared_ptr<T>` 对象是否有任何副本：

```
auto pname = std::make_shared<std::string>("Charles Dickens");
if(pname.unique())
    std::cout << there is only one..." << std::endl;
else
    std::cout << there is more than one..." << std::endl;
```

如果对象的实例数是 1，`unique()` 成员函数返回 `true`，否则返回 `false`。也可以知道当前有多个实例：

```
if(pname.unique())
```

```
std::cout << "there is only one..." << std::endl;
else
    std::cout << "there are " << pname.use_count() << " instances." << std::endl;
```

成员函数 `use_count()` 返回当前被调用对象的实例个数。如果 `shared_ptr<T>` 对象包含的为空指针，那么返回 0。

1.6.3 weak_ptr<T> 指针

`weak_ptr<T>` 对象只能从 `shared_ptr<T>` 对象创建。`weak_ptr<T>` 指针同样被当作类的成员变量去存储同一个类的其他实例对象。当这个类的对象是在堆上创建时，用它的 `shared_ptr<T>` 成员变量去指向另一个同类型的对象。在这种情况下，可能会生成一个循环引用，这使另一个同类型的类对象不能自动释放空间，这种情况可能并不常见，但却是有可能的，如图 1-4 所示。

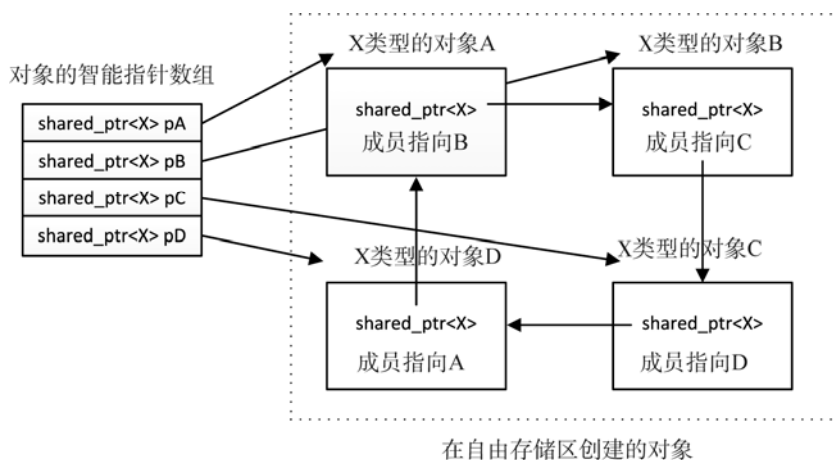


图 1-4 循环引用是如何阻止对象被删除的

删除图 1-4 所示数组中所有的智能指针或者把它们重置为空指针并不能释放它们所指向对象的内存，这是因为每个对象仍有一个 `shared_ptr<X>` 指针指向它们，而且也没有外部的指针能够访问到它们，所以它们无法被销毁。如果对象使用 `weak_ptr<X>` 作为成员变量来指向其他的对象，就可以解决这个问题。当外部数组中的指针被销毁或重置时，`weak_ptr<X>` 就不会阻止对象的销毁。

可以像下面这样去创建一个 `weak_ptr<T>` 对象：

```
auto pData = std::make_shared<X>();
// Create a shared pointer to an object of type X
std::weak_ptr<X> pwData {pData};
// Create a weak pointer from shared pointer
std::weak_ptr<X> pwData2 {pwData};
// Create a weak pointer from another
```

因此，可以从一个 `shared_ptr<T>` 或一个已有的 `weak_ptr<T>` 去创建一个 `weak_ptr<T>`。`weak_ptr<T>` 能做的事情有限，不能以解引用的方式去访问它所指向的对象。例如，可使用

`weak_ptr<T>`对象去做下面两件事：

- 可以判断它所指向的对象是否仍然存在，这也就意味着仍然有 `shared_ptr<T>` 对象指向它。
- 可以从一个 `weak_ptr<T>` 对象创建一个 `shared_ptr<T>` 对象。

这里演示了如何使用 `weak_ptr<T>` 指针来判断它所指向的对象是否还存在：

```
if(pwData.expired())
    std::cout << "Object no longer exists.\n";
```

如果对象不存在的话，`pwData` 对象的 `expired()` 函数会返回 `true`。可以像下面这样从一个 `weak_ptr<T>` 对象得到一个 `shared_ptr<T>` 对象：

```
std::shared_ptr<X> pNew {pwData.lock()};
```

如果 `pwData` 指向的对象仍然存在，`lock()` 函数会通过返回一个新的 `shared_ptr<X>` 来初始化 `pNew`，锁住对象。如果对象不存在，`lock()` 函数会返回一个具有空指针的 `shared_ptr<X>` 对象。可以在 `if` 语句中检查这个结果：

```
if(pNew)
    std::cout << "Shared pointer to object created.\n";
else
    std::cout << "Object no longer exists.\n";
```

如何使用 `weak_ptr<T>` 并不在本书的讨论范围中，所以不打算继续深入讨论 `weak_ptr<T>`。第 3 章会探讨存储智能指针的含义和优点。

1.7 算法

算法提供一些计算和分析的函数，主要运用于一些由一对迭代器指定的对象。起始迭代器指向第一个元素，结束迭代器指向最后一个元素的下一个位置。因为它们可以通过迭代器访问数据元素，所以算法并不关心数据在哪里。可以将算法运用到任何序列上，只要它们能提供特定的迭代器去访问，所以也能够将算法运用到一些容器的元素上，像字符串对象中的字符、标准数组元素、流、存放在类型容器中的序列，甚至任何元素，只要类支持迭代器。

算法是 STL 中最大的工具集。很多程序中都用到了它们，尽管它们的用途看起来可能比较专业。可以将它们分成三大类：

1) 非变化序列运算不会改变它们适用于任何方式的顺序。一个以指定值来匹配元素的算法不会改变元素的原始数据。数值运算(例如 `inner_product()` 和 `accumulate()`)处理元素时不会改变它们，就属于这一类运算。这种类型的算法包括 `find()`、`count()`、`mismatch()`、`search()`、`equal()`。

2) 可变序列运算会改变序列中元素的值。这种类型的算法包括 `swap()`、`copy()`、`transform()`、`replace()`、`remove()`、`reverse()`、`rotate()`、`fill()`、`shuffle()`。堆运算也属于这一类。

3) 很多实例中的排序、合并、关联运算，在使用时会改变序列的顺序。这种类型的算

法包括 `sort()`、`stable_sort()`、`binary_search()`、`merge()`、`min()`、`max()`。

当然，这里列出的每个种类的算法，是没有详尽的可用清单的；你在随后的章节中会了解到更多，以及如何运用它们。一些算法，例如 `transform()`，需要一个函数作为实参，运用到一段元素上。其他对元素重新排序的函数，经常提供一个判断去比较元素。下面让我们看一下把一个函数传给另一个函数作为参数的可能性。

1.8 将函数作为实参传入

函数的签名能不能作为另一个函数的实参使用是由这个函数的形参规格决定的。形参的规格取决于函数实参的类型。可以通过三种方式把一个函数作为实参传给另一个函数：

- 可以使用一个函数指针，这里就可以使用函数名作为参数值。这一点不会详述，因为你肯定已经熟悉了函数指针。下面是另外两种更好的方式：
- 可以传入一个函数对象作为实参。
- 可以使用一个 `lambda` 表达式作为实参。

在随后的章节中，你会看到很多充分运用了最后两种方式的示例，所以本书会提醒你一些这方面的细节，确保你不会对它们感到生疏。

1.8.1 函数对象

函数对象也被称作仿函数，这是一种重载了函数调用运算符 `operator()()` 的类对象。它们提供了一种比使用原生指针更加高效的、将函数作为实参传入另一个函数的方式。让我们来看一些简单的示例，假设像下面这样定义了 `Volume` 类。

```
class Volume
{
public:
    double operator()(double x, double y, double z) {return x*y*z; }
};
```

可以创建一个能像函数一样使用的 `Volume` 对象来计算容量：

```
Volume volume; // Create a functor
double room { volume(16, 12, 8.5) }; // Room volume in cubic feet
```

`room` 的初始化列表中的值，是通过调用 `volume` 对象的 `operator()()` 来获取的，所以这个表达式等同于 `volume.operator()(16,12,8.5)`。当一个函数接收一个函数对象作为实参时，它就可以像函数一样使用。当然，可以在类中定义不止一个版本的 `operator()()`，这允许一个对象以各种方式应用。假设我们定义了一个 `Box` 类，定义了一些成员变量来定义对象的长、宽、高，以及一些可以获取到这些值的成员函数。我们可以像下面这样扩展 `Volume` 类，使它容纳 `Box` 对象：

```
class Volume
{
```

```

public:
    double operator()(double x, double y, double z) {return x*y*z; }

    double operator()(const Box& box)
    { return box.getLength()*box.getWidth()*box.getHeight(); }
};

```

现在 Volume 对象能够计算 Box 对象的体积了：

```

Box box{1.0, 2.0, 3.0};
std::cout << "The volume of the box is " << volume(box) << std::endl;

```

为了允许将 Volume 对象作为实参传给函数，可以将函数的形参类型指定为 Volume&。STL 算法通常对参数使用一个更广义的规范，要求形参能够通过一个定义了这种类型的函数模板列表来表示函数。

1.8.2 lambda 表达式

一个 lambda 表达式定义了一个匿名函数。lambda 表达式定义的函数不同于一般函数的地方是，lambda 可以捕获它们作用域内的变量，然后使用它们。lambda 表达式通常用于 STL 算法。让我们看一个 lambda 表达式示例。假设你想赋予一个函数计算 double 类型值三次幂的功能。下面介绍如何使用 lambda 来实现：

```

[] (double value) { return value*value*value; }

```

这一对方括号称为 lambda 引入符。它们标识了 lambda 表达式的开始，不仅仅只有这里介绍的这种 lambda 引入符——方括号并不总是为空。lambda 引入符后面圆括号之间的是 lambda 的参数列表，这和一般函数的参数列表相似。在本例中，只有一个参数 value，当然也可以有更多的参数，用逗号隔开它们。可以为 lambda 表达式中的参数指定默认值。

lambda 表达式的主体部分在参数列表后的花括号中，这更像普通函数。lambda 的主体包含一条 return 语句，用来返回计算值。一般来说，lambda 主体可以包含任意数量的语句，需要注意的是，在上面的示例中并没有具体的返回类型，返回类型默认为返回值的类型。如果没有返回值，返回类型就是 void。可以使用尾返回类型语法(trailing return type syntax)来指定返回类型。上面的 lambda 表达式可以这样写：

```

[] (double value) -> double { return value*value*value; }

```

1. 对 lambda 表达式命名

尽管 lambda 表达式是匿名对象，但仍然可以用一个变量来保存它的地址。不需要知道它的类型，但编译器需要知道：

```

auto cube = [] (double value) { return value*value*value; };

```

auto 关键字告诉编译器从赋值运算符的右边来确定变量的类型，所以能够知道确定的类型来存放 lambda 表达式的地址。如果 lambda 表达式的引入符方括号中什么都没有，就可以这样做，有时候方括号中的内容会使你不能够以这种方式使用 auto。可以把 cube 当作

函数指针来使用，例如：

```
double x{2.5};
std::cout << x << " cubed is " << cube(x) << std::endl;
```

输入语句会输出 2.5 的立方值。

2. 将 lambda 表达式传给函数

通常，你并不知道一个 lambda 表达式的类型，因为没有通用的表达式类型。前面说过，可以把一个 lambda 表达式作为函数的实参，但这马上会产生一个疑问：如果以一个 lambda 表达式作为实参，那么如何确定形参的类型。这有很多的可能，简单的回答是定义一个函数的模板，它的类型参数是 lambda 表达式的类型。

编译器总是知道 lambda 表达式的类型，所以它们能实例化一个接受 lambda 表达式作为参数的函数模板。通过一个示例，我们很容易知道它们的工作原理。假设有一些 double 类型的值存放在容器中，想以任意方式转换它们；有时候想以它们的平方值代替它们，或者以它们的平方根，或者执行更复杂的转换——这取决于它们的值是否在某个特定的范围内。可以定义一个模板来转换一些由 lambda 表达式指定的元素。

下面看看这个模板的定义：

```
template <typename ForwardIter, typename F>
void change(ForwardIter first, ForwardIter last, F fun)
{
    for(auto iter = first; iter != last; ++iter)
        // For each element in the range...
        *iter = fun(*iter);           // ...apply the function to the object
}
```

形参 fun 接受任何适合的 lambda 表达式，也接受函数对象或普通的函数指针。你可能会好奇编译器是如何处理模板的，因为没有任何关于 fun 实现的信息，事实上是因为编译器根本没有处理它。编译器知道需要实例化一个模板时才会处理它。在上面模板的示例中，当使用它时，编译器知道关于这个 lambda 表达式的所有信息。这里有一个使用这个模板的示例：

```
int data[] {1, 2, 3, 4};
change(std::begin(data), std::end(data), [] (int value){ return
value*value; });
```

第二条语句会将 data 数组的元素全部替换为它们的平方值。

标准库的 functional 头文件定义了一个模板类型 std::function<>，这是对任意类型函数指针的封装，有给定的返回类型和形参类型。当然，这也包括 lambda 表达式。std::function 模板的类型实参是 Return_Type(Param_Types)的形式。Return_Type 是 lambda 表达式或被指向函数的返回值类型。Param_Types 是 lambda 表达式或被指向函数的参数类型的列表，它们都用逗号隔开。在前面章节中，表示 lambda 表达式的变量可以按如下方式定义：

```
std::function<double(double)> op { [] (double value) { return
value*value*value; } };
```

op 变量可以作为实参，传给任意接受函数实参且具有相同签名的函数。当然，也可以重新定义 op，让它们实现一些其他功能，只要它们具有相同的返回类型、形参个数和形参类型即可：

```
op = [] (double value) { return value*value; };
```

现在，op 表示一个能够返回实参的平方值的函数。可以用 std::function 类型模板来指定任何可以调用的类型，包括任何 lambda 表达式或函数对象。

3. 捕获子句

lambda 引入符[]并不一定为空。它可以包含捕获子句，指定一些封闭范围内的变量，它们可以在 lambda 主体中使用。如果一个 lambda 表达式的方括号中没有任何参数，lambda 的主体部分就只能使用 lambda 内的一些局部变量。没有捕获子句的 lambda 表达式被称为无状态的 lambda 表达式，因为它们不能访问封闭范围内的任何东西。图 1-5 显示了 lambda 的语法结构。

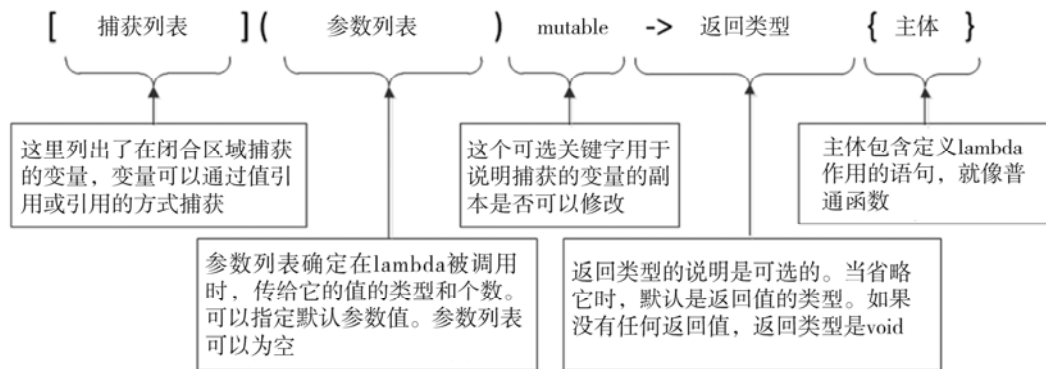


图 1-5 lambda 表达式的组成

默认的捕获子句捕获和 lambda 表达式定义同样范围内的所有变量。如果在方括号中放一个=，lambda 表达式的主体就能以值的形式访问所有封闭范围内的自动变量。也就是说，在 lambda 表达式里可以使用所有变量，但是不能改变它们的值。如果在包围了参数列表的括号后面添加了 mutable 关键字，就可以在 lambda 表达式中改变变量副本的值。使用 lambda 表达式时，仍然会保存着上一次执行时，从外部以值的方式捕获的变量的副本，所以副本是有效的静态值。

如果在方括号中添加了一个&，所有封闭范围内的变量都以引用的方式使用，所以它们的值能在 lambda 体中改变，这时候就不需要 mutable 关键字了。为了能够在 lambda 中使用外部变量，外部变量必须定义在 lambda 表达式之前。

不能用 auto 指定一个变量的类型，然后保存一个访问这个变量本身的 lambda 地址，这意味着正在用一个使用这个变量的 lambda 表达式去初始化这个变量。不能将 auto 用于任何使用了正在定义的变量的 lambda 表达式——auto 不能用于自引用。

可以捕获闭合范围内指定的变量。为了捕获那些想用值访问的指定变量，只需要在捕获子句中列出它们的名称。如果想以引用的方式捕获，需要在每一个名字的前面加上一个 `&` 前缀。捕获子句中两个以上的变量需要使用逗号隔开。可以在一个通过引用来捕获特定变量的捕获子句中，同时使用 `=`。捕获子句 `[=,&factor]`，允许我们以引用的方式使用 `factor`，以值引用的方式使用其他任何封闭范围内的变量。捕获子句 `[&,&factor]`，允许我们以值引用的方式捕获 `factor`，以引用的方式来捕获任何其他封闭范围内的变量。如果要修改 `factor` 副本的值，需要为它指定一个 `mutable` 关键字。

■ **警告：**以值引用的方式捕获封闭范围内的所有变量会增加很多开销。因为不管是否使用了它们，都为它们中的每一个创建了副本，只捕获那些需要使用的变量才是明智的。

`transform()` 算法将你提供的作为参数的函数运用到一定范围的元素上。`transform()` 的前两个参数是用来指定函数作用域的迭代器；第 3 个参数是一个指定结果存放起始位置的迭代器；第 4 个参数是运用到输入序列的函数。下面这个示例演示了如何和 `transform()` 算法一起使用函数、lambda 表达式和 `std::function` 模板类型：

```
// Ex1_03.cpp
// Passing functions to an algorithm
#include <iostream>           // For standard streams
#include <algorithm>          // For transform()
#include <iterator>           // For iterators
#include <functional>         // For function

class Root
{
public:
    double operator()(double x) { return std::sqrt(x); };
};

int main()
{
    double data[] { 1.5, 2.5, 3.5, 4.5, 5.5 };

    // Passing a function object
    Root root; // Function object
    std::cout << "Square roots are:" << std::endl;
    std::transform(std::begin(data), std::end(data),
        std::ostream_iterator<double>(std::cout, " "), root);

    // Using an lambda expression as an argument
    std::cout << "\n\nCubes are:" << std::endl;
    std::transform(std::begin(data), std::end(data),
        std::ostream_iterator<double>(std::cout, " "),
        [](double x){return x*x*x; });

    // Using a variable of type std::function<> as argument
    std::function<double(double)> op {[](double x){ return x*x; }};
    std::cout << "\n\nSquares are:" << std::endl;
    std::transform(std::begin(data), std::end(data),
        std::ostream_iterator<double>(std::cout, " "), op);
}
```

```

// Using a lambda expression that calls another lambda expression as
// argument
std::cout << "\n\n4th powers are:" << std::endl;
std::transform(std::begin(data), std::end(data),
               std::ostream_iterator<double>(std::cout, " "),
               [&op](double x){return op(x)*op(x); });

std::cout << std::endl;
}

```

输出结果如下:

```

Square roots are:
1.22474 1.58114 1.87083 2.12132 2.34521

Cubes are:
3.375 15.625 42.875 91.125 166.375

Squares are:
3.375 15.625 42.875 91.125 166.375

4th powers are:
11.3906 244.141 1838.27 8303.77 27680.6

```

如果理解了先前的章节，对于本例应该不会觉得很难。`transform()`算法处理的输入数据来自一个 `data` 数组。在每一个 `transform()` 调用中，前两个参数都是数组的开始和结束迭代器。输出目的地是由一个输出流迭代器指定的，它将数据写入标准输出流。`ostream_iterator` 构造函数的第二个参数是一个用来分隔每个写入值的分隔符。

第 1 个 `transform()` 调用传入一个 `Root` 对象作为最后一个参数。`Root` 类定义 `operator()` 成员函数来返回 `root` 参数的开方。第 2 个 `transform()` 调用表明可以写一个用来计算立方值的 `lambda` 表达式作为参数。第 3 个 `transform()` 调用表明 `std::function` 类型的模板也可以在这里使用。最后一个调用表明一个 `lambda` 表达式可以调用另一个 `lambda` 表达式。所以，当需要把函数作为参数传给算法时，可以使用这些技巧中的任意一个。

1.9 小结

这一章介绍了 STL 背后的一些基本思想。在本书后面的章节中，对于在本章介绍的 STL 的所有方面，都会做更深入的示范和解释。本章还概述了一些需要理解的 C++ 重要特性，因为它们是应用 STL 的基础，在随后的章节中会广泛地使用它们。本章所涉及的要點如下：

- STL 定义类模板，作为其他对象的容器。
- STL 定义迭代器——一种行为像指针的对象。通常用一对迭代器定义一段连续的元素。起始迭代器用于指向第一个元素，结束迭代器用于指向最后一个元素的下一个位置。

- 反向起始迭代器指向序列的最后一个元素，反向结束迭代器指向第一个元素的前一个位置。反向迭代器的工作意义和普通迭代器相反。
- `iterator` 头文件定义了一些全局函数，可以从容器、数组、任何支持迭代器的对象得到一些迭代器。全局函数 `begin()`、`cbegin()`、`end()`、`cend()` 都能够返回普通迭代器。函数 `rbegin()`、`crbegin()`、`rend()` | `crend()` 能够返回反向迭代器。
- 可以使用流迭代器转换流对象中给定类型的数据。
- STL 定义了一些实现了算法的函数模板，可以运用到由迭代器指定的一段元素上。
- 智能指针是一种表现有些像指针的对象。对象通常是在堆上分配地址。对象由智能指针管理，当没有智能指针指向对象时，对象会被自动销毁。智能指针不能自增或自减。
- 一个 `lambda` 表达式定义了一个匿名函数。`lambda` 表达式通常用来把函数作为参数传给 STL 算法。
- 能够用定义在 `functional` 头文件中的 `std::function<>` 模板类型，去指定任意种类的、有给定函数签名的可调用实体。

练习

这里有一些检验你是否记住了本章所讨论主题的练习。如果遇到困难，可以回顾前面的章节以寻找帮助。在这之后，如果还是无法解决的话，可以从 Apress 出版社的网站上下载解答(<http://www.apress.com/9781484200056>)，但这应该是最后的选择。

1. 编写一个程序，它定义了一个标准的数组：用你选择的一些词去初始化字符串对象，然后用迭代器一个一行地列出数组的内容。
2. 写一个程序，运用 `transform()` 算法，将前一个练习中数组的所有小写元音字母以 '*' 替换，并将结果一个一行地写入标准输出流。以 `lambda` 表达式的方式定义一个使用迭代器的函数来替换字符串中的元音字母。
3. 写一个程序，运用 `transform()` 算法，将第一个练习中的数组的字符串转换为大写，然后输出结果。用来转换字符串的函数，应该以 `lambda` 表达式的形式传给 `transform()`，这个 `lambda` 表达式会调用 `transform()`，来把 `std::toupper()` 库函数运用到字符串中的每一个字符上。