第9章 顺序容器

9.1 顺序容器（sequential container）概述

1. `vector`： 可变大小数组， `deque`： 双端队列， `list`： 双向链表， `forward_list`： 单向链表， `array`： 固定大小数组， `string`： 专门用于保存字符的可变大小容器。
2. 与内置数组相比， `array` 是一种更安全、更容易使用的数组类型。
3. `forward_list` 没有 `size` 操作，因为保存和计算其大小就会比手写链表多出额外的开销。

9.2 容器库概览

1. 反向容器的额外成员（不支持 `forward_list` ）

```
reverse_iterator        // 按逆序寻址元素的迭代器
const_reverse_iterator  // 不能修改元素的逆序迭代器
c.rbegin(), c.rend()    // 返回指向c的尾元素和首元素之前位置的迭代器
c.crbegin(), c.crend()  // 返回 const_reverse_iterator
```

9.2.1 迭代器

1. `forward_list` 迭代器不支持递减运算符（--）。

9.2.2 容器类型成员

1. 通过 *类型别名* 我们可以在不了解容器中元素类型的情况下使用它。

9.2.4 容器定义和初始化

1. 当传递迭代器参数来拷贝一个范围时，就不要求新容器和原容器的类型是相同的了，只要能将要拷贝的元素转换为要初始化的容器的元素类型即可。
2. 只有顺序容器的构造函数才接受大小参数，关联容器并不支持。
3. 我们不能对内置数组欵写进行拷贝或者对象赋值操作，但是 `array` 并无此限制。 `array` 要求初始值的类型必须与要创建的容器大小和类型均相同。

9.2.5 赋值和 swap

1. 由于右边运算对象的大小可能与左边运算对象的大小不同，因此 array 类型不支持 assign，也不允许用花括号包围的值列表进行赋值。
2. 除 array 外， swap 交换两个容器内容的操作元素本身并未交换，只是交换了两个容器的内部数据结构。
3. 与其他容器不同， swap 两个 array 会真正的交换它们的元素。因此，指针、引用和迭代器所绑定的元素保持不变，但是元素值已经与另一个 array 中对应元素的值进行了交换。*(指向的内存地址没变，内存里的内容调换了)*。

```
// 例如：假定iter在swap之前指向svec1[0]的元素a
array<string, 1> svec1 = { "a" }, svec2 = { "b" };
auto iter = svec1.begin();
swap(svec1, svec2);

// 那么在swap之后它还是指向svec1[0]的元素，但是元素值变成了 b
iter == svec1.begin();  // true
&iter == "b";           // true
```

4. 除 string 外，指向容器的迭代器、引用和指针在 swap 操作之后都不会失效。它们仍指向 swap 操作之前所指的那些元素 *(指向的内存地址调换了，内存里的内容没换)*。

```
// 例如：假定iter在swap之前指向svec1[0]的元素a
vector<string> svec1 = { "a" }, svec2 = { "b" };
auto iter = svec1.begin();
swap(svec1, svec2);

// 那么在swap之后它指向svec2[0]的元素，且元素值还是a。
iter == svec2.begin();  // true
&iter == "a";           // true
```

5. 与其他容器不同，对一个 string 调用 swap 会导致迭代器、引用和指针失效。
6. 非成员版本的 swap 在泛型编程中是非常重要的。统一使用非成员版本的 swap 是一个好习惯。

9.2.7 关系运算符

1. 只有当其元素类型定义了相应的比较运算符时，才能使用关系运算符来比较两个容器。

9.3 顺序容器操作

9.3.1 向顺序容器添加元素

1. forward_list 有自己专有版本的 insert 和 emplace；且不支持 push_back 和 emplace_back。
2. vector 和 string 不支持 push_front 和 emplace_front。
3. insert 函数将元素插入到迭代器所指定的位置之前。
4. emplace 函数在容器中直接构造元素。传递给 emplace 函数的参数必须与元素类型的构造函数想匹配。

9.3.2 访问元素

```
vector<int> c = { 1 };
auto& v1 = c.back();      // 获得指向最后一个元素的引用
auto v2 = c.back();       // 获得最后一个元素的拷贝
```

9.3.3 删除元素

1. 删除 deque 中除首尾位置之外的任何元素都会使所有迭代器、引用和指针失效。
2. 指向 vector 或 string 中删除点之后位置的迭代器、引用和指针都会失效。
3. pop_front 和 pop_back 操作返回 void。

9.3.4 特殊的 forward_list 操作

1. 在一个单向链表中没有简单的方法来获取一个元素的前驱。出于这个原因，在一个 forward_list 中添加和删除元素的操作是通过改变给定元素之后的元素来完成的。于是定义了 insert_after、emplace_after 和 erase_after 操作与普通顺序容器里的 insert、emplace 和 erase 操作相对应。
2. before_begin() 返回指向链表首元素之前不存在的元素的迭代器。首前迭代器（off_the_beginning iterator）不能解引用。

9.3.5 改变容器大小

1. 对 vector、string 或 deque 进行 resize 可能导致迭代器、指针和引用失效。

9.3.6 容器操作可能使迭代器失效

1. 对于 deque，如果在首尾位置添加元素，迭代器会失效，但指向存在的元素的引用和指针不会失效。

2. 当我们添加/删除 `vector` 或 `string` 的元素后，或在 `deque` 中首元素之外任何位置添加/删除元素后，原来 `end` 返回的迭代器总是会失效，所以不要缓存 `end` 返回的迭代器。

9.4 `vector` 对象是如何增长的

1. 调用 `reserve` 不会减少容器占用的内存空间。调用 `resize` 只改变容器中元素的数目，而不是容器的容量，也不能减少容器预留的内存空间。
2. 调用 `shrink_to_fit` 来要求 `deque` 、 `vector` 或 `string` 退回不需要的内存空间。但是*不能保证调用该函数后一定会退回多余的内存空间*。

9.5 额外的 `string` 操作

9.5.1 构造 `string` 的其他方法

1. 当我们从一个 `const char*` 创建 `string` 时，如果未传递计数值且数组也未以空字符结尾，或者给定计数值大于数组大小，则构造函数的行为是未定义的。
2. `s.substr(pos, n)` 返回一个 `string` ，包含 `s` 中从 `pos` 开始的 `n` 个字符的拷贝。

9.5.2 改变 `string` 的其他方法

1. `assgin` 和 `append` 函数无须指定要替换 `string` 中哪个部分： `assgin` 总是替换所有内容， `append` 总是将新字符追加到末尾。
2. `replace` 函数提供了两种指定删除元素范围的方式：可以通过一个位置和一个长度来指定范围，也可以通过一个迭代器范围来指定。
3. `insert` 函数允许两种方式指定插入点：用一个下标或一个迭代器。元素都会插入到给定下标（或迭代器）之前的位置。

9.5.3 `string` 搜索操作

1. 如果搜索失败，返回一个名为 `string::npos` 的 `static` 成员，标准库将 `npos` 定义为一个 `const string::size_type` 类型。
2. 搜索是 *大小写敏感* 的。

9.5.5 数值转换

1. 如果 `string` 不能转换为一个数值，这些函数抛出一个 `invalid_argument` 异常，如果转换得到的数值无法用任何类型来表示，则抛出一个 `out_of_range` 异常。

9.6 容器适配器（adaptor）

1. 适配器是标准库中的一个通用概念。*容器*、*迭代器和函数* 都有适配器，本质上一个适配器是一种机制，能使某种事物的行为看起来像另外一种事物一样。
2. 除了顺序容器外，标准库还定义了三个顺序容器适配器： `stack` 、 `queue` 和 `priority_queue` 。

```
// 每种容器适配器都定义两个构造函数：默认构造函数创建一个空对象；接受一个容器的构造函数拷贝该容器来初始化
deque<int> deq;
stack<int> stk;          // 默认构造函数
stack<int> stk2(deq);    // 拷贝容器内容

// 我们可以在创建一个适配器时将一个命名的顺序容器作为第二个类型参数来重载默认容器类型。
vector<int> vc;
stack<int, vector<int>> vc_stk;
stack<int, vector<int>> vc_stk2(vc);
```

3. 所有适配器都要求容器具有添加和删除以及访问尾元素的能力，因此适配器不能构造在 `array` 和 `forward_list` 之上。
4. 可以使用除 `array` 和 `forward_list` 之外的任何容器类型来构造 `stack` 。
5. `queue` 可以构造于 `list` 或 `deque` 之上，但是不能基于 `vector` 构造。
6. `priority_queue` 可以构造于 `vector` 或 `deque` 之上，但是不能基于 `list` 构造。
7. `priority_queue` 允许我们为队列中的元素建立优先级。新加入的元素会排在所有优先级比它低的已有元素之前。