6.2.1 序列式容器（Sequence Container）

STL 内部预先定义好了以下序列式容器：

- Array（其 class 名为array）
- Vector
- Deque
- List（singly/doubly linked）

以下讨论从 vector 开始，因为 array 是 TR1 新引入的，进入 C++ 标准库的时间比较短，而且它有一些特殊属性，与其他 STL 容器不共通。

Vector

Vector 将其元素置于一个 dynamic array 中管理。它允许随机访问，也就是说，你可以利用索引直接访问任何一个元素。在 array 尾部附加元素或移除元素都很快速，[3] 但是在 array 的中段或起始段安插元素就比较费时，因为安插点之后的所有元素都必须移动，以保持原本的相对次序。

以下例子针对整数类型定义了一个 vector，插入 6 个元素，然后打印所有元素：

```
// stl/vector1.cpp

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

int main()
{
    vector<int> coll;      // vector container for integer elements

    // append elements with values 1 to 6
    for (int i=1; i<=6; ++i) {
        coll.push_back(i);
    }

    // print all elements followed by a space
    for (int i=0; i<coll.size(); ++i) {
        cout << coll[i] << ' ';
    }
    cout << endl;
}
```

[3] 严格说来，元素追加动作是一种“摊提的，有折旧成本的（amortized）”高速，单一附加动作可能是缓慢的，因为 vector 可能需要重新分配内存，并将现有元素拷贝到新位置。不过这种事情不常发生，所以总体看来这个操作十分迅速。见 2.2 节第 10 页关于复杂度的讨论。

Vector 的头文件通过以下语句包含进来：

```
#include <vector>
```

以下声明式建立了一个“元素类型为 int”的 vector：

```
vector<int> coll;
```

由于没有任何初始化参数，所以 default 构造函数将它构造为一个空集合。push_back() 为容器附加新元素：

```
coll.push_back(i);
```

所有序列式容器都提供这个成员函数，因为尾附一个元素永远是可能的，而且效率相当高。size() 成员函数返回容器的元素个数：

```
for (int i=0; i<coll.size(); ++i) {
    ...
}
```

所有容器类都提供这个函数，唯一例外是 singly linked list（class forward_list）。
你可以通过 subscript（下标）操作符 []，访问 vector 内的某个元素：

```
cout << coll[i] << ' ';
```

元素被写至标准输出装置，所以整个程序的输出如下：

```
1 2 3 4 5 6
```

Deque

所谓 *deque*（发音类似“check”[4]），是“double-ended queue”的缩写。它是一个 dynamic array，可以向两端发展，因此不论在尾部或头部安插元素都十分迅速。在中间部分安插元素则比较费时，因为必须移动其他元素。

以下例子声明了一个元素为浮点数的 deque，并在容器头部安插 1.1 至 6.6 共 6 个元素，最后打印出所有元素。

```
// stl/deque1.cpp

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

int main()
{
    deque<float> coll;      // deque container for floating-point elements
```

[4] 有时候“deque”听起来颇类似“hack”，不过这纯属巧合 :-)

```
    // insert elements from 1.1 to 6.6 each at the front
    for (int i=1; i<=6; ++i) {
        coll.push_front(i*1.1);        // insert at the front
    }

    // print all elements followed by a space
    for (int i=0; i<coll.size(); ++i) {
        cout << coll[i] << ' ';
    }
    cout << endl;
}
```

本例中，下面这一行将 deque 的头文件包含进来：

```
#include <deque>
```

下面的声明式则是产生一个空的浮点数集合：

```
deque<float> coll;
```

push_front() 函数用来安插元素：

```
coll.push_front(i*1.1);
```

push_front() 会将元素安插于集合前端。注意，这种安插方式造成的结果是，元素排放次序与安插次序恰好相反，因为每个元素都安插于上一个元素的前面。程序输出如下：

```
6.6 5.5 4.4 3.3 2.2 1.1
```

你也可以使用成员函数 push_back() 在 deque 尾端附加元素。Vector 并未提供 push_front()，因为其时间效率不佳（在 vector 头端安插一个元素，需要先移动全部元素）。一般而言，STL 容器只提供具备良好时间效率的成员函数，所谓“良好”通常意味着其复杂度为常量或对数，以免程序员调用性能很差的函数。

Array

一个 array[5]对象乃是在某个固定大小的 array （有时称为一个 static array 或 C array）内管理元素。因此，你不可以改变元素个数，只能改变元素值。你必须在建立时就指明其大小。Array 也允许随机访问，意思是你可以直接访问任何一个元素——只要你指定相应的索引。

下面的例子定义出了一个 array，元素是 string：

[5] Class array<> 由 TR1 引入。

```
// stl/array1.cpp

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

int main()
{
    // array container of 5 string elements:
    array<string,5> coll = { "hello", "world" };

    // print each element with its index on a line
    for (int i=0; i<coll.size(); ++i) {
        cout << i << ": " << coll[i] << endl;
    }
}
```

下面这行将 array 的头文件包含进来：

```
#include <array>
```

以下声明式会建立一个 array，带有 5 个类型为 string 的元素：

```
array<string,5> coll
```

默认情况下这些元素都被元素的 default 构造函数初始化。这意味着，对基础类型而言，初值不明确（undefined）。

然而本程序用了一个初值列（initializer list，见 3.1.3 节第 15 页），这东西允许我们以一系列值将对象初始化于创建期。自 C++11 起这种初始化方法受到每一种容器的支持，所以当然我们也可以在 vector 和 deque 中使用它。既然如此，基础类型用的是 *zero initialization*，意味着基础类型保证被初始化为 0（见 3.2.1 节第 37 页）。

程序中借由 size() 和下标操作符 []，将所有元素联合其索引逐行写至标准输出装置。整个程序输出如下：

```
0: hello
1: world
2:
3:
4:
```

如你所见，这个程序输出 5 行，因为我们定义的 array 带有 5 个 string。根据初值列的设定，头 2 个元素被初始化为 "hello" 和 "world"，其余元素则拥有默认值，也就是空字符串。

注意，元素个数是 array 类型的一部分。因此 array<int,5> 和 array<int,10> 是两个不同的类型，你不能对此二者进行赋值或比较。

List

从历史角度看，我们只有一个 list class。然而自 C++11 开始，STL 竟提供了两个不同的 list 容器：class `list<>` 和 class `forward_list<>`。因此，*list* 可能表示其中某个 class，或者是个总体术语，代表上述两个 list class。然而就某种程度来说，forward list 只不过是受到更多限制的 list，现实中二者的差异并不怎么重要。因此当我使用术语 *list*，通常我指的是 class `list<>`，它的能力往往超越 class `forward_list<>`。如果特别需要指出 class `forward_list<>`，我会使用术语 *forward list*。所以本节讨论的是寻常的 list，是一开始就成为 STL 一部分的那个东西。

`list<>` 由双向链表（doubly linked list）实现而成。这意味着 list 内的每个元素都以一部分内存指示其前导元素和后继元素。

List 不提供随机访问，因此如果你要访问第 10 个元素，你必须沿着链表依次走过前 9 个元素。不过，移动至下一个元素或前一个元素的行为，可以在常量时间内完成。因此一般的元素访问动作会花费线性时间，因为平均距离和元素数量成比例。这比 vector 和 deque 提供的摊提式（amortized）常量时间，效率差很多。

List 的优势是：在任何位置上执行安插或删除动作都非常迅速，因为只需改变链接（link）就好。这表示在 list 中段处移动元素比在 vector 和 deque 快得多。

以下例子产生一个空 list，用以放置字符，然后将 'a' 至 'z' 的所有字符插入其中，利用循环每次打印并移除集合的第一个元素，从而打印出所有元素：

```
// stl/list1.cpp
#include <list>
#include <iostream>
using namespace std;

int main()
{
    list<char> coll;        // list container for character elements

    // append elements from 'a' to 'z'
    for (char c='a'; c<='z'; ++c) {
        coll.push_back(c);
    }

    // print all elements:
    // - use range-based for loop
    for (auto elem : coll) {
        cout << elem << ' ';
    }
    cout << endl;
}
```

就像先前的例子一样，头文件 <list> 内含 list 的声明。以下定义一个“元素类型为字符”的 list：

```
list<char> coll;
```

为了打印所有元素，我使用一个 range-based for 循环，这种循环自 C++11 之后提供，允许对每个元素执行指定的语句（见 3.1.4 节第 17 页）。List 并不提供作为“元素直接访问”之用的操作符 []。这是因为 list 并不提供随机访问，因此操作符 [] 会带来低下的效率。

在此循环中，当前正被处理的 coll 元素的类型被声明为 auto。因此 elem 的类型被自动推导为 char，因为 coll 是个 char 集合（auto 类型推导详见 3.1.2 节第 14 页）。另一种做法是明白声明 elem 的类型：

```
for (char elem : coll) {
    ...
}
```

注意，elem 永远是当前正被处理的元素的一个拷贝（copy）。虽然你可以改动它，但其影响只限于“针对此元素而调用的语句”，coll 内部并没有任何东西被改动。如果你想改动传入的集合的元素，你必须将 elem 声明为一个非常量的 reference：

```
for (auto& elem : coll) {
    ...      // any modification of elem modifies the current element in coll
}
```

就像函数参数那样，通常你应该使用一个常量 reference 以避免发生一次 copy 操作。因此，下面的 function template 输出“被传入的容器内的所有元素”：

```
template <typename T>
void printElements (const T& coll)
{
    for (const auto& elem : coll) {
        std::cout << elem << std::endl;
    }
}
```

在 C++11 之前，你必须使用迭代器来访问所有元素。稍后才会介绍迭代器，你将在 6.3 节第 189 页发现一个相应的例子。

在 C++11 之前，打印所有元素的另一种做法（不使用迭代器）是逐一地“打印而后移除”第一元素，直到此 list 之中不再有任何元素：

```
// stl/list2.cpp

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

int main()
{
```

```
list<char> coll;          // list container for character elements

// append elements from 'a' to 'z'
for (char c='a'; c<='z'; ++c) {
    coll.push_back(c);
}

    // print all elements
    // - while there are elements
    // - print and remove the first element
    while (! coll.empty()) {
        cout << coll.front() << ' ';
        coll.pop_front();
    }
    cout << endl;
}
```

成员函数 empty() 的返回值告诉我们容器中是否还有元素。只要这个函数返回 false（也就是说，容器内还有元素），循环就继续进行：

```
while (! coll.empty()) {
    ...
}
```

循环之内，成员函数 front() 返回第一个元素：

```
cout << coll.front() << ' ';
```

pop_front() 函数删除第一个元素：

```
coll.pop_front();
```

注意，pop_front() 并不会返回被删除元素，所以你无法将上述两个语句合而为一。

程序的输出结果取决于所用的字集。如果是 ASCII 字集，输出如下：[6]

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

Forward List

自 C++11 之后，C++ 标准库提供了另一个 list 容器：forward list。forward_list<> 是一个由元素构成的单向（singly） linked list。就像寻常 list 那样，每个元素有自己一段内存，为了节省内存，它只指向下一元素。

[6] 如果是 ASCII 以外的字集，输出结果可能包含非字母字符，甚至可能什么都没有（如果 'z' 不大于 'a' 的话）。

因此，forward list 原则上就是一个受限的 list，不支持任何“后退移动”或“效率低下”的操作。基于这个原因，它不提供成员函数如 push_back() 乃至 size()。

现实中，这个限制比乍听之下甚至更尴尬棘手。问题之一是，你无法查找某个元素然后删除它，或是在它的前面安插另一个元素。因为，为了删除某个元素，你必须位于其前一元素的位置上，因为正是那个元素才能决定一个新的后继元素。也因此，forward list 对此提供了一个特殊成员函数，见 7.6.2 节第 305 页。

下面是 forward list 的一个小例子：

```
// stl/forwardlist1.cpp
#include <forward_list>
#include <iostream>
using namespace std;

int main()
{
    // create forward-list container for some prime numbers
    forward_list<long> coll = { 2, 3, 5, 7, 11, 13, 17 };

    // resize two times
    // - note: poor performance
    coll.resize(9);
    coll.resize(10,99);

    // print all elements:
    for (auto elem : coll) {
        cout << elem << ' ';
    }
    cout << endl;
}
```

一如以往，我们使用 forward list 的头文件 <forward_list> 定义一个类型为 forward_list 的集合，以长整数（long integer）为元素，并以若干质数为初值：

```
forward_list<long> coll = { 2, 3, 5, 7, 11, 13, 17 };
```

然后使用 resize() 改变元素个数。如果数量成长，你可以传递一个额外参数，指定新元素值。否则就使用默认值（对基础类型而言是 0）。注意，resize() 是一个昂贵的动作，它具备线性复杂度，因为为了到达尾端你必须一个一个元素地前进，走遍整个 list。不过这是一个几乎所有 sequence 容器都会提供的操作，就暂时忽略它那可能的低劣效率吧。只有 array 不提供 resize()，因为其大小固定不变。

像先前对待 list 那样，我们使用一个 range-based for 循环打印所有元素。输出如下：

```
2 3 5 7 11 13 17 0 0 99
```