第3章 字符串、向量和数组

3.1 命名空间的 `using` 声明

1. 位于头文件的代码，一般不应该使用 `using` 声明。

3.2 标准库类型 `string`

3.2.1 定义和初始化 `string` 对象

1. 拷贝初始化 *copy initialization*
2. 直接初始化 *direct initialization*

3.2.2 `string` 对象上的操作

1. `string` 的 `size()` 返回的是一个 `string::size_type` 类型的值，如果表达式中已经有了 `size()` 函数据就不要再使用 `int` 了，这样可以避免混用 `int` 和 `unsigned` 可能带来的问题。
2. 当把 `string` 对象和字符字面值及字符串字面值混在一条语句中使用时，必须确保每个加法运算符（+）的两侧运算对象至少有一个是 `string`：

```
string s1 = "hello" + ".";          // 错误

string s2 = "world";
string s3 = "hello" + "," + s2;     // 错误
```

3. 为了兼容 C，C++ 中的字符串字面值并不是标准库类型 `string` 的对象。

3.2.3 处理 `string` 对象中的字符

1. C++ 标准库兼容了 C 语言的标准库。C 语言的头文件形式是 *name*.h, C++ 则将这些文件命名为 *cname*。
2. 在名为 *cname* 的头文件中定义的名字从属于命名空间 `std`，而定义在名为 .h 的头文件中的则不然。

3.3 标准库类型 `vector`

1. 可以将模板看作为编译器生成类或函数编写的一份说明。
2. 编译器根据模板创建类或函数的过程叫做 *实例化 instantiation*。
3. 因为引用不是对象，所以不存在包含引用的 `vector` 。

3.3.1 定义和初始化 `vector` 对象

1. 列表初始化 `vector` 过程会尽可能地把花括号内的值当成是元素初始值的列表来处理，只有当无法执行列表初始化时才会考虑其他初始化方式。确定无法执行列表初始化后，编译器会尝试用默认值初始化 `vector` 对象。

```
vector<int>     v1{10};          // v1有一个元素，该元素为10
vector<string>  v2{10};          // v2有10个默认初始化的元素。

vector<int>     v3{10, 1};       // v3有2个元素，值分别是10和1
vector<string>  v4{10, "hi"};    // v4有10个值为 "hi" 的元素
```

3.3.2 向 `vector` 对象中添加元素

1. 在定义 `vector` 对象的时候先设定其大小，可能性能还会降低。除非所有的元素的值都是一样时效率才会高。

3.3.3 其他 `vector` 操作

1. 要使用 `size_type` ，需首先指定它是由哪种类型定义的。

```
vector<int>::size_type  // 正确
vector::size_type       // 错误
```

3.4 迭代器介绍

3.4.1 使用迭代器

1. 尾后迭代器 *off-the-end iterator*
2. 但凡使用范围 `for` 循环或者迭代器的循环体，都不要向容器中添加元素。 *（因为容器被添加元素后其内存地址可能会变）*。

3.4.2 迭代器运算

1. 指向同一个容器的两个迭代器相减得到迭代器之间的距离，类型为 `difference_type` 的带符号整型数。

3.5 数组

3.5.1 定义和初始化内置数组

1. 数组的元素应为对象，因此不存在引用的数组。
2. 字符串字面值初始化字符数组时，要注意结尾处还有一个空字符。
3. 不能将数组的内容拷贝给其他数组作为其初始值，也不能用数组为其他数组赋值。
4. 想要理解数组声明的含义，最好的办法是从数组的名字开始按照由内向外的顺序阅读。

```
int arr[10];                // 含有10个整数的数组
int *ptrs[10];              // 含有10个整型指针的数组
int (*p_array)[10] = &arr;  // 指向一个含有10个整数的数组
int (&arr_ref)[10] = arr;   // 引用一个含有10个整数的数组
int *(&array)[10]  = ptrs;  // array是数组的引用，该数组含有10个整型指针
```

3.5.2 访问数组元素

1. 使用数组下标的时候，通常将其定义为 `size_t` 类型。
2. `size_t` 类型是一种机器相关的无符号类型，它被设计得足够大以便能表示内存中任意对象的大小。

3.5.3 指针和数组

1. 当使用 `decltype` 关键字时，数组名字转换为指针的操作不会发生。
2. 数组不是类类型。
3. 两个指向同一个数组的数组指针相减的结果类型是一种定义在 *cstddef* 头文件中的带符号的机器相关的标准库类型： `ptrdiff_t` 。
4. 标准库类型限定使用的下标必须是无符号类型，而内置的下标运算无此要求可以为负数（为负时相当于当前位置往后移）。

3.5.4 C 风格字符串

1. 对于大多数应用来说，使用标准库 `string` 要比使用 C 风格字符串更安全、更高效。

3.5.5 与旧代码的接口

```
// string转字符数组
string s;
char *str = s;                                  // 错误
const char *str = s.c_str();                    // 正确

// 数组转vector
int int_arr[] = {0, 1, 2, 3, 4, 5};
vector<int> ivec(begin(int_arr), end(int_arr)); // 正确
vector<int> subVec(int_arr + 1, int_arr + 4);   // 正确
```

3.6 多维数组

1. 对于二维数组来说常把第一个维度称为行，第二个维度称为列。
2. 要使用范围for语句处理多维数组，除了最内层的循环外，其他所有循环的控制变量都应该是引用类型（不用引用的话，会被隐式转换为指针）。