Chapter 3 字符串、向量和数组

3.1 命名空间的using声明

1. 作用域操作符（::）：编译器应从操作符左侧的名字所示的作用域中寻找右侧那个名字。因此，std::cin的意思就是要使用命名空间std中的名字cin。
2. 使用using就可以声明就无需专门的前缀::也能使用所需的名字。
3. 头文件一般不应包含using声明，因为头文件的内容会拷贝到所有引用它的文件中去，由此可能产生始料未及的名字冲突。

3.2 标准库类型string

1. 作为标准库的一部分，string定义在命名空间std中。所以当我们直接使用string的时候，需要包含

```
#include <string>
using std::string; // 注意不要忽略后面这个分号
```

2. string初始化的方式

string s1	默认初始化，s1是空串
string s2(s1)	s2是s1的副本
string s2 = s1	等价于上一条，s2是s1的副本
string s3("value")	s3是字面值“value”的副本，除了字面值最后那个空字符外
string s3 = "value"	等价于上一条，s3是字面值“value”的副本
string s4(n, 'c')	把s4初始化为连续n个字符

3. 拷贝初始化与直接初始化：
    - 拷贝初始化：使用‘=’来初始化
    - 不使用‘=’则是直接初始化

```
string s5 = "haha"; // 拷贝初始化
string s6("haha"); // 直接初始化
string s7(10, 'c'); // 直接初始化，s7内容是ccccccccccc
```

4. 需要注意的string操作

s.size()	返回s中的字符数

s1+s2	返回s1和s2连接后的结果
s1==s2	如果s1和s2中所含的字符完全一样，则他们相等，string对象对相等性判断对字母的大小写敏感
s2!=s2	同上
<,>=,<=,>	利用字符在字典中的顺序进行比较

5. getline函数的参数是一个输入流和一个string对象（getline(cin， str)），函数从给定的输入流中读取内容，直到遇到换行符位置（**注意换行符也被读进来了**）然后把所读的内容存入到那个string对象中去（**注意此时不存换行符**，实际上换行符被丢弃掉了，string对象中不包含该换行符）。如果getline一开始就遇到了换行符，那么它将存入string一个空串。

6. string的成员函数size()返回的并不是一个int或unsigned，而是string::size_type类型的值。size_type是一种无符号类型的值而且能足够存放下任何string对象的大小。所有用于存放string类的size函数返回值的变量，都应该是string::size_type类型的。

7. c++11新标准中auto或者decltype可以推断size_type的类型：

```
auto len = line.size() // len的类型是string::size_type
```

8. 由于size函数返回的是一个无符号整型数，所以此时如果混用带符号的类型数可能会产生意想不到的后果。

```
int n = -20;
if(s.size() < n){ // 此处n会被转成unsigned类型的求模数
    // code_block
}
```

9. 两个string对象之间的比较：

    - 如果两个string对象长度不同，且较短string对象的每个字符都与长对象的字符相同，则会返回长度比较的结果
    - 如果两个string对象在某些位置上的字符不同，那么就直接返回第一对不同字符之间的比较结果

10. 当把string对象和字符字面值及字符串字面值混用在一条语句中使用时，必须确保每个加法运算符（+）的两侧的运算对象至少有一个是string

```
string s4 = s1 + ", "; // 正确：把一个string对象和一个字面值相加
string s5 = "Hello" + ", "; // 错误，两个运算对象都不是string
string s6 = s1 + "," + "world";

string s7 = "Hello" + ", " + s2; // 错误，不能把字面值直接相加
// string7的编译过程如下：
string s7 = ("Hello" + ", ") + s2;
```

```
// 然而("Hello" + ", ")这个过程编译器无法做到，所以是错的
```

11. 为了与c兼容，c++语言中的**字符串字面值不是标准库类型的string对象**。而是字符串数组。

12. cctype头文件中的函数：

isalnum(c)	当c是字母或数字时为真
isalpha(c)	当c是字母时为真
iscntrl(c)	当c是控制字符时为真
isdigit(c)	当c是数字时为真
isgraph(c)	当c不是空格但可以打印时为真
islower(c)	当c是小写字母时为真
isprint(c)	当c是可打印字符时为真（即c是空格或c具有可视形式）
ispunct(c)	当c是标点符号时为真
isspace(c)	当c为空白时为真
isupper(c)	当c为大写字母时为真
isxdigit(c)	当c为十六进制数字时为真
tolower(c)	转换成小写字母
toupper(c)	转换成大写字母

13. cctype是c语言中的ctype.h标准库，只不过cctype命名规范上符合c++对标准库的文件名的要求（不包含.h）。所以去掉.h同时在开头加上c，表示这个库是一个属于c语言标准的头文件。

14. range for语句：其中expression是一个对象，用于表示一个序列。declaration负责定义一个变量，该变量将被用于访问序列中的基础元素。每次迭代，declaration部分的变量会被初始化为expression部分的一个元素值。

```
/*
 * for (declaration: expression){
 *    statement;
 * }
 */
string str("Hello World");
for(auto c: str)
    cout << c << endl;
```

15. 一个更复杂的例子：

```
string str("Hello World!!!!");
```

```
decltype (str.size()) count = 0;
for(auto c: str){
    if(ispunct(c)){
        count++;
    }
}
cout << count << "punctuation characters in " << str << endl; // count is 4
```

16. 如果想要改变string对象中字符的值，则**必须把循环变量定义成引用类型**。

```
string str("Hello World!!!!");
for(auto &s: str){
    s = toupper(s);
}
cout << str << endl;
```

17. string的下标的值被称作“下标”或“索引”。任何表达式只要它的值是一个整型就能作为索引。所有带符号的类型都会被转换成string::size_type表达的无符号类型。切记，每次**使用下标访问string时都需要判断那个位置是否为空**（isempty()），否则结果将是未定义的。

18. 使用下标进行迭代：

```
for(decltype(s.size()) index = 0; index != s.size()
                        && !isspace(s[index]); index++){
    s[index] = toupper(s[index]);
}
```

19. 应该总是将下标设置为string::size_type，因为此类型是无符号数，可以确保下标不会小于0。

3.3 标准库类型vector

1. vector是一个类模板。根据模板创建类或函数的过程称为实例化（instantiation）。vecotr能容纳绝大多数类型的对象作为其元素，但是因为引用不是对象，**所以不存在包含引用的vector**。

2. 在早期的c++标准中，vector的定义必须在外层右尖括号和其元素之间添加一个空格。某些编译器仍然需要用这种方式来声明处理。

```
vector<vector<int> > vs;
```

3. 初始化vector的对象的集中方式：

vector v1	v1是一个空vector
vecotr v2(v1)	v2包含所有v1元素的副本

vector v3(n, val)	包含了n个重复的val
vector v4(n)	包含了n个重复地执行了值初始化的对象
vector v5{a, b, c...}	列表初始化，v5包含了a b c几个元素并被赋予相应的初始值
vector v6 = {a, b, c...}	列表初始化，等价于上句

4. 初始化需要注意的三种情况：
    - 使用拷贝初始化时（=）只能提供一个初始值（string str = "Hello World"）
    - 如果提供的是一个类内初始值，则只能使用拷贝初始化或使用花括号的形式初始化
    - 如果提供的是初始元素值的列表，则只能把初始值都放在花括号里进行列表初始化，而不能放在圆括号里

```
vector<string> v1{"a", "b", "c"}; // 列表初始化
vector<string> v2("a", "b", "c"); // 错误
```

5. 值初始化（value-initialized）：只提供vector对象容纳的元素数量而略去初始值，此时库会自动创建一个值初始化的元素初值，并把它赋给容器中的所有元素。如果vector对象的元素是内置类型，比如int，则元素初始值自动设为0。如果元素是某种类类型，比如string，则元素由类默认初始化。

```
vector<int> vec(10, -1) // 创建一个包含10个初始值为-1的vector
vector<int> ivec(10); // 10个都初始化为0的元素
vector<string> svec(10); // 10个空的string元素
```

6. 有些类型要求必须明确地提供初始值，那么就无法默认初始化该类的对象。另外，**如果只提供了元素的数量而没有设定初始值，只能使用直接初始化**：

```
vector<int> vi = 10; // 错误，必须使用直接初始化的形式指定向量大小
vector<int> vi(10); // 正确
```

7. 初始化时花括号和圆括号的区分：
    - 如果用的是圆括号，可以说提供的值是用来构造vector的对象的。
    - 如果用的是花括号，可以表述称我们想列表初始化该vector对象。初始化过程会尽可能的把花括号内的值当成元素初始值的列表来处理，只有在无法执行列表初始化时才会考虑其他初始化方式。
    - 如果初始化时使用了花括号的形式但是提供的值又不能用来列表初始化，就要考虑用这样的值来构造vector对象。

```
vector<int> v1(10); // v1有十个元素，每个都是0
vector<int> v2{10}; // v2有一个元素，为10
vector<int> v3(10, 1); // v3有十个重复的元素1
```

```
vector<int> v4{10, 1}; // v4有两个元素分别是10， 1
vector<string> v5{"hi"}; // v5有一个元素是 hi
vector<string> v6("hi"); // 错误，不能用字符串字面值构建vector对象
vector<string> v7{10}; // 10是int型，不能用来初始化值，
                       // 所以v7有十个默认初始化为空的字符串
vector<string> v8{10, "hi"}; // v8有10个值为"hi"的元素
```

8. 从标准输入中读取单词，将其作为vector对象的元素存储

```
string word;
vector<string> text; // 空vector对象
while (cin >> word){
    text.push_back(word); // 把word添加到text后面
}
```

9. 在定义vector对象的时候设定大小没有必要，如果设置大小可能使得性能反而更差

10. **如果循环体内部包含有向vector对象添加元素的语句，则不能使用范围for循环**。（也就是说，for语句体内不应该改变其所遍历序列的大小）

11. vector支持的操作

v.empty()	如果v不含有任何元素，返回真；否则返回假
v.size()	返回v中元素的个数
v.push_back(t)	向v的尾端天假一个值为t的元素
v[n]	返回第n个位置上的元素
v1 = v2	用v2中元素的拷贝替换v1的元素
v1 = {a, b, c...}	用列表中元素的拷贝替换v1的元素
v1 == v2	v1和v2相等当且仅当他们的元素数量相同且对应位置的元素值都相同
v1 != v2	同上
<, <=, >, >=	以字典顺序进行比较

12. 访问以及改变vector中的值

```
vector<int> v{1, 2, 3, 4, 5};
for(auto &i : v){
    i *= i;
}
for(auto i : v){
    cout << i << endl;
}
```

13. empty和size两个成员函数返回的是vector::size_type，注意不要忽略此处的::以及区分string::size_type

14. 两个vector对象相等，则必须个数、成员位置、成员的值都必须同时相等。比较大小时，如果较小vector的所有元素和较大者完全相同，那么返回是size大小的比较结果，如果元素值不同，那么比较第一个不同元素之间的大小。（由此可见vector里包含的元素必须支持相等性的判断和关系运算符等 操作）

15. vector的一个实例：

```
// 以10分为一个分数段统计成绩的数量：0-9,10-19，。。。。，90-99,100
vector<unsigned> scores(11, 0);
unsigned grade;
while(cin >>grade){
    if (grade <= 100){
        ++score[grade/10];
    }
}
```

16. **vector不支持以下标的形式增加元素**

```
vector<int> ivec;
for(decltype(ivec.size())) ix = 0; ix != 10; ++ix){
    ivec[ix] = ix; // 严重错误，ivec不包含任何元素
}
```

17. 通过下标访问不存在的元素的行为非常常见，而且会产生很严重的后果。所谓的缓冲区溢出（buffer overflow）所指的就是这类错误。确保下标合法的一种有效手段就是尽可能使用范围for语句。

3.4 迭代器

1. 严格来说，string对象不属于迭代器类型，但是string支持很多与容器类型类似的操作，也就是说，string也支持迭代器。

2. 和指针不一样的是，获取迭代器不是使用取地址符，有迭代器的类型同时拥有返回迭代器的成员。比如begin和end的成员。begin成员返回指向第一个元素的迭代器，end成员返回指向尾部元素下一个位置的迭代器，也就是说该迭代器指示的是一个本不存在的“尾后（off the end）”元素，此迭代器仅是一个下标而已。

3. **如果容器为空，begin和end返回的是同一个迭代器，都是尾后迭代器。**

```
auto b = v.begin(), e = v.end();
```

4. 迭代器运算符：

---

*iter	返回迭代器iter所指示的元素
iter->mem	解引用iter并获取该元素的名为mem的成员，等价于(*iter).mem
++iter	指示下一个元素
--iter	指示上一个元素
iter1 == iter2	如果两元素指示同一个元素或容器尾后迭代器，则相等，否则不等
iter1 != iter2	

5. 可以直接使用迭代器来获取它所指示的元素，**执行解引用的迭代器必须有合法并确实指示着某个元素**。

```
string s("some string");
if(s.begin() != s.end()){ //确保s非空
    auto it = s.begin();
    *it = toupper(*it);
}
```

6. **end返回的迭代器并不实际指示某个元素，所以不能对其进行递增或者解引用的操作。**
7. 在使用迭代器时，要尽可能更多使用"!="符号来代替"<"
8. 迭代器两种类型：iterator和const_iterator，const_iterator只能读取但却不能改变所指的元素值。**如果vector或者string对象是一个常量，那我们只能使用const_iterator**，如果该对象不是一个常量，那么这两种迭代器都可以使用
9. begin和end返回的具体类型由对象是否是常量决定，如果对象是常量那么begin和end返回的是const_iterator，如果对象不是常量，则返回的是iterator。**如果对象只需要读操作，最好使用const_iterator**。
10. 为了得到const_iterator类型的返回值，c++11标准引入了两个新函数，分别是cbegin和cend，不论对象是否是const，这两个函数返回的都是const_iterator

```
auto ite = v.cbegin() // it3是const_vector类型
```

11. **但凡是使用了迭代器的循环体，都不要向迭代器所属的容器添加元素**
12. 练习：使用迭代器将所有vector中的元素都变为原来的两倍

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

int main()
{
    vector<int> ve{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

```
    auto vi = ve.begin();
    for (; vi != ve.end(); vi++)
    {
        *vi = *vi * 2;
    }
    for (auto i : ve)
    {
        cout << i << " ";
    }
    cout << endl;
    return 0;
}
```

13. 两个迭代器之间相减的结果是他们之间的距离。参与运算的两个迭代器必须是指向的同一个容器中的元素或者尾元素的下一个位置。其类型是difference_type的带符号整型数。string和vector都定义了difference_type，因为这个距离可正可负，所以difference_type是带符号类型的。

14. 使用迭代器可以进行二分搜索：

```
// text必须有序
// beg和end是我们的搜索范围
auto beg = text.begin(), end = text.end();
auto mid = text.begin() + (end - beg) / 2; // 初始状态中间点
while(mid != end && *mid != sought){
    if(sought < *mid)
        end = mid; // 搜索点在前半部分
    else
        beg = mid + 1; // 搜索点在后半部分
    mid = beg + (end - beg) / 2;
}
```

3.5 数组

1. 数组也是存放类型相同的对象的容器，这些对象本身没有名字，需要通过其所在位置访问。与vector不同的是，数组的大小确定不变，不能随意向数组中增加元素。因为数组的大小固定，因此对某些特殊的应用来说程序运行时性能较好，但同时损失一些灵活性。

2. **如果不清楚元素的确切个数，请使用vector**。

3. 数组维度说明了数组中元素的个数，因此必须大于0，数组元素的个数也属于数组类型的一部分，编译的时候维度应该是一致的，也就是说，维度必须是一个常量表达式：

```
unsigned cnt = 42; // 非常量
constexpr unsigned sz = 42; // 常量表达式

int arr[10]; // 含有10个整数的数组
int *ptr[sz]; // 含有42个整型指针的数组
string bad[cnt]; // 错误，cnt不是常量表达式
```

```
stirng strs[get_size()]; // 当get_size()是const_expr时正确，否则错误
```

4. 数组默认情况下元素被默认初始化，如果在函数内部定义了某种内置类型的数组，那么默认初始化会令数组含有未定义的值。

5. **定义数组时必须指定数组的类型，不允许使用auto关键字由初始值的列表推断类型，另外和vector一样，数组的元素应为对象，因此不存在存放引用的数组。**

6. 进行列表初始化则允许忽略数组的维度，编译器会根据初始值的个数来推测维度。如果指明维度，那么初始值的总量不应超出指定的大小，如果维度比提供的初始值数量大，则用提供的初始值来初始化靠前的元素，剩下的元素被初始化成默认值。

7. 字符数组的特殊性：使用**字符串字面值**来初始化字符数组时，一定要注意字符串字面值的结尾处还有一个空字符。

```
char a1[] = {'c', '+', '+'}; // 列表初始化，没有空字符
char a2[] = {'c', '+', '+', '\0'}; // 列表初始化，含有显示的空字符
char a3[] = "c++"; // 自动添加字符串结束的空字符
char a4[5] = "Hello"; // 错误，应当设置维度为6来存放额外的‘\0’
```

8. **不能将数组的内容拷贝给其他数组作为初始值，也不能用数组为其他数组赋值**

```
int a[] = {1, 2, 3};
int b[] = a; // 错误，不允许用一个数组初始化另一个数组
b = a; // 错误，不允许把一个数组直接赋值给另一个数组
```

9. 理解复杂的数组声明：最好是从数组的名字开始由内而外的顺序阅读

```
int *ptr[10]; // ptr是一个存放了10个int型指针的数组
int &ref[10] = a; // 错误，不存在引用的数组
int (*Parray)[10] = &arr; // 由内而外理解，Parray是一个指针，
                          // 指向一个存了10个int型整数的数组arr
int (&arrReff)[10] = arr; // 同样的，arrReff是一个引用，
                          // 引用了一个存了10个int型整数的数组arr
int *(&arry)[10] = ptr; // arry是一个引用，
                        // 引用了一个存了10个int型指针的数组
```

10. 使用数组的时候，编译器一般会把它转换成指针。当使用数组作为一个auto变量的初始值时，推断得到的类型是指针而非数组。

```
int a[] = {1, 2, 3, 4};
auto b(a); // auto推断b是一个整型指针，指向a的第一个元素，实际上

           // 编译器执行的是auto b(&a[0])
b = 42; // 错误，b是一个指针，不能直接赋值int型
```

11. **然而将上述代码的auto替换成decltype时，不会发生上述转换，而直接返回一个包含4个整数的数组。**

12. 指向数组元素的指针其实和vector或者string的迭代器类似，允许++或--来移动指针。c++11新标准引入了两个名为begin和end的函数，用于数组指针获取首元素和尾后元素，**这两个函数定义在iterator的头文件中。不过数组不是类类型，所以这两个函数不是成员函数，数组必须作为参数**：

```
int ia[] = {1, 2, 3, 4};
int *beg = begin(ia); // 指向ia首元素的指针
int *end = end(ia); // 指向ia尾元素的下一位置的指针
```

13. **需要注意的是，尾后指针不能执行解引用和定增操作。**

14. 给一个指针加上或减去某个整数值，其结果仍然是指针，新指向的元素与原来的指针相比前进了（后退了）该整数值个位置。

15. 两个指针相减的结果为ptrdiff_t的标准库类型，和size_t一样，ptrdiff_t也是一种定义在cstddef头文件中的机器相关类型，因为差值可能是负值，所以ptrdiff_t是一种带符号类型。

16. 如果两个指针指向同一个数组的元素，或者指向该数组的为元素的下一位置，就能使用关系运算符对其进行比较。**如果指向不相关对象，则不能比较。**

```
int *b = arr, *e = arr + sz;
while(b < e){
    // code block
    b++;
}
```

17. **数组使用的是内置下标运算符，而vector或者string等标准库类型限定了使用的下标必须是无符号类型（大于等于0），而内置则无此要求。所以内置下标运算符可以处理负值（例如p[-2]），当然结果也必须指向原来指针所指同一数组中的元素（或是同一数组尾元素的下一位置）。**

18. 比较两个数组是否相等，需要先比较数组大小，再比较数组中每个元素是否相等，不能直接使用'='来判断数组名是否相等（因为数组名会被编译器转换成指向数组第一个元素的指针），而对于vector来说，直接使用'='来比较即可。

19. **在c++程序中最好不要使用c风格字符串。对于大多数情况来说，使用c++标准库的string要比使用c风格字符串更安全、更高效。**

20. 现代的c++程序应当尽量使用vector和迭代器，尽量避免使用内置数组和指针，应该尽量使用string，避免使用c风格的基于数组的字符串。

3.6 多维数组

1. 严格来说c++语言中没有多维数组，通常所说的多维数组其实是数组的数组。通常由两个维度来定义它：一个维度表示数组本身大小，另一个维度表示其元素（也是数组）大小。例如:int a[3][4] 表示的是

是一个大小为3的数组，每个元素是含有4个整数的数组。通常把第一个维度称作行，第二个维度称作列。

2. 多维数组的初始化：

```
int a[2][3] = {
    {1, 2, 3},
    {4, 5, 6}
}; // 内层的花括号并未是必须的，也可以直接写成{1, 2, 3, 4, 5, 6}

int b[2][3] = {
    {0},
    {1}
}; // 并非所有元素的值都必须包含在初始化列表之内，
   // 如果仅仅想初始化每一行第一个元素也是可以的
```

3. 多维数组的下标引用：

```
int arr[2][2][2] = 0;
a[1][2] = arr[1][2][3]; // 此处引用的是数组arr中某个具体的值
int (&row)[3] = a[1]; // row绑定到数组a的第二个3元素数组上
```

4. 可以使用范围for语句来遍历多维数组：

```
size_t cnt = 0;
for(auto &row: ia){
    for(auto &col: row){
        col = cnt;
        ++cnt;
    }
}
```

5. 注意，当使用范围for语句时，无论我们是否需要修改数组内的值，**我们必须将外层for循环的控制变量声明成引用类型，因为直接使用auto将会把该数组自动转换成指针**：

```
for(auto row: ia){
    for(auto colo: row){

    }
}// 该程序会报错，因为外部row被转换成指向第一个元素的指针（int*），
 // 而内部for中的row将无法使用范围for
```

6. 多维数组名指向的是第一个内层数组的指针。使用auto或者decltype以及标准库函数begin和end可以使数组的遍历变得简洁

```
// p指向多维数组ia中的第一个数组
for(auto p = begin(ia); p != end(ia); p++){
    // *p是一个含有若干整数的数组，q指向该数组的首元素
    for(auto q = begin(*p); q != end(*p); q++){
        // code block
    }
}
```

7. 使用类型别名来简化多维数组的指针

```
using int_array = int [4]; // 新标准下类型别名的声明
typedef int int_array[4]; // 等价的typedef声明

for(int_array *p = ia; p != ia + 2; ++p){
    for(int *q = *p; q != *p + 3; ++q){
        // code block
    }
}
```