第三章 字符串、向量和数组

命名空间

- 作用域操作符 `::`：编译器因从操作符左侧名字所示的作用域中寻找右侧那个名字

```
::a      // 默认没显示 是从程序全局变量中找
std::cin
```

- 每个名字都要需要独立的using声明

```
using std::cin;
using std::cout;
using std::endl;
```

> 可以用简化的方法，声明该程序下所函数都使用某一特定命名空间
>
> ```
> using namespace std;     // 所有标准库函数都在std命名空间内，所以可这样声明
> ```

- 头文件不应包括using声明

标准库（STL)——string

需要包含头文件 `<string>`

表示一个字符的序列

定义和初始化

```
string s1;              // 默认初始化，空串

string s2(s1);          // 相当于s2=s1;
string s2 = s1;

string s3("value");     // s3="value";
char t[] = "value"; string s3(t);
string s3(t,t+5);       //左边右开

string s4(n,'c');       // 将s4初始化为由连续n个字符c组成的串
string s5(s3.begin()+1,s3.end());       // 左闭右开

```

可以利用初始化方法得到倒序的string

```
string a="abc";
string b(a.rbegin(),a.rend());    //b="cba"
```

- 拷贝初始化：使用等号初始化一个变量；
- 直接初始化：不使用等号

```
string s(3,'n');        // 直接初始化
string s = string(3,'n');   // 拷贝初始化，
```

基本操作

类既能定义通过函数名调用的操作，也能定义 `<<、+` 等各种运算符在该类对象上的新含义（符号重载）

```
os<<s;      // s写到输入流中
is>>s;      // 从is中读取字符串赋给s，返回左侧对象（ 作为结果
getline(is,s);      // 从is中读取一行赋给s，可以读取一行中空白，直到换行符为止，返回流函数作为结果

s.empty();
s.size();
s[n];
s1 + s2;
s1 = s2;
s1 == s2; s1 != s2;
>,>=,<,<=       // 利用字符在字典中的顺序 按顺序进行比较，大小写敏感
```

- 在从输入流中读操作时，会自动忽略开头的空白（空格符，换行符、制表符等），从第一个真正的字符开始，知道遇到下一个空白为止
- `string::size_type` 类型：实际上s.size()返回的是该类型，是一个**无符号类型**的值而且能够存下任何string对象的大小
  - 用下标访问string，其实会自动将整型的下标转换为无符号类型

> 标准库类型都定义集中配套的类型，体现类标准库类型与机器无关的特性

- 字符字面值和string对象相加
  - 字符字面值自动转换为string对象

> 要确保'+'的两侧运算对象至少有一个string，char类型才会自动转换

  - **切记，字符串字面值与string是不同的类型**
- 处理单个字符：`<cctype>头文件` 中的函数

其中函数	描述
isalnum(c)	字母或数字
isalpha(c)	字母
isdigit(c)	数字
islower(c)	小写
isupper(c)	大写
tolower(c)	将大写字母输出为小写（但是不改变原来字母）
toupper(c)	将小写字符输出为大写（但是不改变原来字母）

> **C++11** 基于范围的for语句

```
for (declaration: expression)        // expression是一个对象，用于表示一个序列，declaration 负责定义一个变量，该变量用于访问序列中的 基础元素
statement                            // 每次迭代，declaration部分的变量都会被初始化为expression部分的下一个变量
```

- expression 要支持迭代器访问的方式才可用此类循环

成员函数

```
str.insert()
```

标准库（STL)——vector

- 需要头文件 `<vector>`
- 表示对象的集合，其中所有的对象的类型都相同

> 这里的对象可以是很多种类型，可以是基本内置类型，也可以是自定义类型（那自然也就可是是STL库中定义的类型），但是不能是对象

- **实例化**：编译器根据模版创建类或函数的过程。

  对于 **类模版**，我们要通过一些格外信息来指定模版到底实例化什么样的类。提供信息的方式总是这样，在模版名字后面跟一对尖括号，在括号内放信息

  > string 是一个类模版吗?
  >
  > 不是，string是basic_string的实例化，其实string s，实际上是 `basic_string<char> s` 这样的

  > "类模板"和"模版类"的区分：
  >
  > - 类模版是一个是使用模版的类声明；
  > - 模版类是使用类模版的类的定义

定义和初始化

```
vector<T> v1;        // 默认初始化
vector<T> v2(v1);    // v2中包含v1所有元素的副本
vector<T> v2 = v1;
vector<T> v3(n,val);
vector<T> v4(n);     // v4中包含了n个重复地执行 值初始化 的对象
vector<T> v5{a,b,c...};      // 列表初始化
vector<T> v5={a,b,c...};
```

> - 对于 `v2 = v1`，会使得v2和v1的内容一样，不用考虑长度的问题
> - 值初始化，即只提供vector对象容纳的元素数量而不用显示说明初始化，这时会基于对象的类型进行对应值初始化。
>
>   内置类型初始化为0，类类型有默认构造函数初始化。
>
>   **所以，如果对象的类不支持默认初始化，该方法就无效**

再论括号和花括号初始化

```
vector<int> a{10};      // a有1个元素，该元素值为10
vector<int> a(10);      // a有10个元素，这些元素都进行值初始化
```

- 如果用的是圆括号，可以说提供的值是用来构造 `vector` 对象的；

  如果用的花括号，可以理解为列表初始化，会尽可能将花括号内这值直接当成元素的初始值的列表来进行处理

```
vector<string> v{"hi"};
vector<string> v("10");      // 错误，不能用字符字面值构造vector对象
vector<string> v{10};        // v构造了10个默认初始化的对象
vector<string> v{10,"hi"};
```

- 如果初始化时使用了花括号但提供的值又不能用来列表初始化，就要考虑是否用该对象来构造

  确认无法执行列表初始化后，编译器会尝试用默认值初始化对象

  所以，这也是为什们用*c++11*花括号初始化有更好的表现

基本操作

```
v.push_back(基本元素);          // 添加元素到末尾
v.empty();          // 判断是否为空
v.size();           // 返回v中元素个数
v[n];               // 返回v中第n个位置上元素的引用

v1 == v2;
<,<=,>,>=       // 按基本元素的比较规则，顺序比较
```

- **如果循环体内部有向 `vector` 对象添加元素的语句，则不能使用范围for循环**，但是可以使用基于变量的循环、

  > **原因**

- 只有当基本元素可以比较时，vector才可比较
- **不能用下标的形式添加元素**

  > 下标只能去访问那些已经存在的元素，而不能添加元素

迭代器介绍(第九章扩展)

- **所有STL容器**都可使用迭代器，但是只**有少数几种才同时支持下标操作**

  > 严格来说，string对象并不属于容器类型，但其支持许多和容器类似的操作

- 迭代器提供了对对象的间接访问，指向了某一元素的位置
- 有效的迭代器：指向某个元素 or 指向容器尾元素的下一位置（其他情况均为无效）

迭代器的使用

```
// 推荐使用方式，不用管具体的迭代器类型
auto b = v.begin(), e = v.end();
```

- end()返回的迭代器称为尾后迭代器。如果容器为空，`begin()==end()`，两者都是尾后跌倒器；

  > 可以认为，这样设置的意义是为了一致性，即数组的访问是左闭右开[0, size())，所以迭代器也是左边右开

```
vector<string> v{"hi"};
auto it = v.begin();
cout<<(*it).size();
cout<<it->size();
```

- C++定义类 **箭头运算符**，把解引用和成员访问的两个操作结合在一起

迭代器类型

- `iterator` `const_iterator`：常量迭代器类型，能读取不能修改它所指的元素值
- `a.begin() or a.end()`返回的迭代器类型由对象是否为常量决定

  ```
  vector<int> v;
  constr vector<int> cv;
  auto it1 = v.begin();
  auto it2 = cv.begin();      // 返回常量迭代器
  ```

  > **C++11** `v.cbegin()|cend()`：无论vector本身是声明类型，返回值都是常量迭代器

迭代器运算

- 所有类型的迭代器都支持递增运算；

  都能用 `== & !=` 对任意标准库类型的两个有效迭代器进行比较

- `string & vector` 的迭代器提供的类更多额外的运算符

```
iter +/- n;          // 移动 n*基本元素大小 的距离
iier += n;  iter -= n;
iter1 - iter2;       // 返回两个迭代器之间的距离（有符号类型）

>、>=、<、<=        // 位置前后比较，在前的小于在后的迭代器
```

> 这也是为什们如果for循环用迭代器，判断条件用!=。这样具有普适性。

数组

定义和初始化

- 数组的维度（即数组的大小）必须是一个常量表达式

> 如果不知道元素确切个数，推荐用 `vector`

- 默认情况下，数组元素被默认初始化
- 定义数组时，必须指定数组的类型。不能用 `auto` 来由初始化列表推断
- 字符数组的特殊性

```
char a1[] = {'C','+','+'};       // 末尾没有空字符
char a2[] = {'C','+','+','\0'}; // 显示添加空字符
char a3[] = "c++";       // 自动添加空字符
```

> 注意 a1是没有空字符的。这也是为什们，如果我们一个一个的读入字符，最后还要在末尾手动添加一个空字符

- 数组初始化不支持拷贝和赋值

```
int a[] ={2,3};
int b[] = a;     // 错误
b = a;           // 错误
```

> 数组能存放大多数类型的对象

```
int *ptrs[10];       // 指针数组，存放10个整型指针
int (*parray)[10];   // 数组指针，parray指向一个含有10个整数的数组
int &refs[10];       // 错误，不存在引用的数组（因为引用不是一个对象）
int (&arrRefs)[10]; // arrRef 引用一个含有10个整数的数组

int *(&array)[10] = ptrs;    // array 是一个数组的引用，该数组含有10个指针 &->[]->int *
```

理解数组声明的含义，最好的办法是从数组名字按照由内向外的顺序阅读

数组和指针

使用数组的时候编译器一般会把它转换成指针

> 所以`int a[10]`,a 其实就是指向该数组第一个元素的指针，a的类型实际上是一个指针，`int`只是指明了数组的基本元素类型
>
> `a[2]`下标访问，其实是利用指针访问

```
int ia[10] = {1,3};
auto ia2(ia);        // int* ia2;
decltype(ia) ia3 = {0,2};    // ia3是一个含有10个整数的数组
```

> **特殊情况：**
>
> **当使用`decltype`时，返回的类型仍然是该数组类型**

- **C++11** begin()、end()函数

```
int a[]={1,2,3};
int *beg = begin(a);
int *last = end(a);      // 尾后指针
```

  - `begin()、end()`并不是成员函数。
  - 包含在头文件中，是C++标准程序库中的一个头文件，定义了C++ STL标准中的一些迭代器模板类，这些类都是以std::iterator为基类派生出来的
- 指向不相关对象的指针不能进行比较

数组和引用

```
int &p[]=a;      //  错误 不合法，正常理解是叫做引用的数组，但是数组时不支持数组间的初始化的
int (&p)[3]=a    // 正确，但是要注意的是必须显示说明维度，这时p和a表示同一个
```

多维数组（难点）

严格的说，C++语言中没有多维数组。通常所说的多维数组其实是**数组的的数组**

- 多维数组首元素本身就是一个数组，所有数组名就是一个指向数组的指针

初始化

- 多维数组的每一行分别用花括号括起来

```
int ia[3][4]= {
    {0,1};
    {3,3};
    {2,2};
}
```

  - 内层的花括号不是必要的。因为只要知道列的值，就可以推断各自赋值的情况

遍历多维数组

- 多重循环
- 范围for语句

```
for(auto row : ia)          // 错误
    for(auto col : row)

--------------------------------
for(auto & row : ia)
    for(auto col : row)
```

> **要使用范围for语句处理多维数组，无论是否要改变数组内值，除最内层的循环外，其他所有的循环的控制变量都应是引用类型**
>
> > 对于`auto row : ia`，ia实际上是指针类型（编译器自动将这些数组形式的元素转换为指向该数组内首元素的指针），这样得到的row的类型就是`int *`,显然对于一个指针是不能用范围for循环；
> >
> > 当时如果加了引用，那么row 实际上就是代表的就是里面的数组，而不是指针了。

```
int ia[3][3]={1,2,4,5,5};

for(int *p : ia)          // p实际上是什么呢？？？？
    for(int p2 : p)      //错误

for(int (&p)[3] : ia)            // 全部正确，p实际上就是数组里面的一个数组
    for(int p2 : p)

```

难懂

指针和多维数组

当程序使用多维数组的名字时，也会自动转换为指向数组首元素的指针

- 因为多维数组实际上是数组的数组，所以多维数组名转换得到的指针实际上指向的是第一个内层数组的数组指针（即，**基本元素是数组首元素地址的数组指针**）

```
int ia[3][4];
int (*p)[4] = ia;    // p指向含有4个整数的数组，p[1]是一个地址值
```

- 利用`auto`来简化写法

```
for(auto p = begin(ia); p != end(ia); ++p)
    for(auto q = begin(*p); q != end(*p); ++q)
        ...
```

- 利用类型别名来简化

```
using int_array = int[4];        // 将4个整数组成的数组当成一个基本类型

int_array ia[3];     // 3个 4个整数组成的数组 的数组，相当于 int[4][3]
int_array *p = ;         // 指向 4个整数组成的数组 的指针； 所以每一个指针
```

C风格字符串

尽量用string、vector

尽量用迭代器而不是指针

与旧代码的接口

- 允许使用字符串字面值来初始化string对象

```
char c[] = "ab";
string s = c;
```

- string成员函数`str.c_str()`,返回一个C风格的字符串，即返回一个指针，指向一个以空字符结束的字符数组

```
char *str = s;      // 错误，不能用string对象初始化char*
char * str = s.c_str(); // 正确
```

- 允许使用数组初始化vector

```
int a[] = {1,2,3};
vector<int> vt = a;      // 错误;
vector<int> vt(a);       // 错误
vector<int> vt(a,a+2);       // 正确
```

C++ 标准库——扩展

C++ 标准库可以分为两部分：

- **标准函数库：** 这个库是由通用的、独立的、不属于任何类的函数组成的。函数库继承自 C 语言。
- **面向对象类库：** 这个库是类及其相关函数的集合。

C++ 标准库包含了所有的 C 标准库，为了支持类型安全，做了一定的添加和修改。

标准函数库

标准函数库分为以下几类：

- 输入/输出 I/O
- 字符串和字符处理
- 数学
- 时间、日期和本地化
- 动态分配
- 其他
- 宽字符函数

面向对象类库

标准的 C++ 面向对象类库定义了大量支持一些常见操作的类，比如输入/输出 I/O、字符串处理、数值处理。面向对象类库包含以下内容：

- 标准的 C++ I/O 类
- **string 类**
- 数值类
- **STL 容器类**
- STL 算法
- STL 函数对象
- **STL 迭代器**
- STL 分配器
- 本地化库
- **异常处理类**
- 杂项支持库

标准库（STL)扩展头文件——algorithm

Algorithm意为“算法”，是C++的标准模板库（STL）中的重要头文件之一，提供了大量**基于迭代器的非成员模板函数**

> 模版函数 和 函数模版 的区别?
>
> - 函数模板可以用来创建一个通用的函数，以支持多种不同的形参，避免重载函数的函数体重复设计。它的最大特点是把函数使用的数据类型作为参数。
> - 在使用函数模板时，要将这个形参实例化为确定的数据类型。将类型形参实例化的参数称为模板实参，用模板实参实例化的函数称为模板函数