

第 4 章 复合类型、string 和 vector

2018 年 8 月 26 日

目录

- ① 引用
- ② 指针
- ③ 数组
- ④ 指针和数组
- ⑤ string 类型
- ⑥ vector 类型
- ⑦ 枚举类型

4 复合类型、string 和 vector

学习目标

- 理解指针和引用的工作机理；
- 掌握指针、引用和数组的使用方法；
- 理解指针与数组的关系，能够运用指针访问数组元素；
- 学会运用数组、string 和 vector 解决实际问题。



什么是复合类型？

4.1 引用

复合类型

复合类型 (`compound type`) 是指基于其它类型定义的类型，包括指针、引用、数组、函数、类、联合体和枚举类型等。

引用

- **引用**^a (`reference`) 指为已经创建的对象**重新起一个名字**
- 编译器只是将别名绑定到所引用的对象上，**不会把对象的内容复制给引用**
- 引用是函数间共享局部对象的重要途径，对于**提高程序的效率**有重要作用

^aC++11 引入了右值引用 (`rvalue reference`)，为了沿用传统的说法，如无提示，本书中的引用都是指左值引用 (`lvalue reference`)。

4.1 引用

引用语法格式：

```
int counter = 0;
int &refCnt = counter;    //refCnt 引用 counter 对象的内容
int &refCnt2;             //错误：定义引用时必须和一个对象绑定
```

`refCnt` 是 `counter` 的别名，可以通过 `refCnt` 对 `counter` 所在的内存空间的内容进行读写操作：

```
refCnt = 2;    //修改了 counter 所在的内存空间的内容
int i = refCnt; //通过引用读取 counter 对象的内容，并初始化对象 i
```

4.1 引用

定义引用时，除了需要初始化外，还需要注意以下几点：

NO.1

定义多个引用时，每个引用必须用 `&` 标明：

```
int i = 0;
int &r1 = i, j = 0, &r2 = r1;
//r1 和 r2 都是 i 的引用，而 j
是 int 类型
```

NO.2

只能引用同类型的对象：

```
double d = 0;
int &r3 = d; //错误：r3 只能引
用 int 类型对象
```

NO.3

引用的对象必须是非 `const` 左值：

```
int i = 0; const int ci = 0;
int &r4 = 100, &r5 = i+1, &r6 =
ci; //错误：只能引用非 const 左
值
```

4.1 引用



建议：

在书写上，建议把引用符号与对象名放在一起，而不是把类型名和引用符号放在一起，这样有助于改进程序的可读性。这种书写方式也适用于指针的定义。

4.1 引用

——引用 `const` 对象

引用 `const` 对象语法格式：

```
const int ci = 0;
const int &r1 = ci; //r1 引用 const 对象 ci
r1 = 1;           //错误：相当于修改 const 对象 ci 的值
```

对于一个 `const` 对象

- 无法通过其引用来修改该对象
- 只能对该引用所绑定的对象进行读操作，不能执行写操作
- 可用任何类型兼容的对象来初始化 `const` 引用

例如：

```
int i = 0;
const int &r1 = i;      //正确：使用左值对象初始化
const int &r2 = 1;      //正确：使用字面值常量初始化
const int &r3 = i+1;    //正确：使用表达式 i+1 的结果初始化
const int &r4 = 3.14;   //正确：使用 double 类型数据初始化
```


4.1 引用

——auto 和引用

auto 能正确的推导出引用吗？

- 下面代码中的 **r** 是什么类型？

```
int i = 0, &ri = i;  
auto r = ri;
```

- 下面代码中的 **cr** 是什么类型？

```
const int ci=0;  
auto &cr = ci;
```

4.1 引用

——auto 和引用

auto 能正确的推导出引用吗？

- 下面代码中的 **r** 是什么类型？

```
int i = 0, &ri = i;
```

```
auto r = ri; //r 是 int 类型而不是 int 类型引用，auto 被推导为  
int
```

- 下面代码中的 **cr** 是什么类型？

```
const int ci=0;
```

```
auto &cr = ci;
```

4.1 引用

——auto 和引用

auto 能正确的推导出引用吗？

- 下面代码中的 **r** 是什么类型？

```
int i = 0, &ri = i;
```

```
auto r = ri; //r 是 int 类型而不是 int 类型引用，auto 被推导为  
int
```

- 如果希望定义一个整型引用，需要显式指出引用类型

```
auto &r = i; //r 是 int 类型引用
```

- 下面代码中的 **cr** 是什么类型？

```
const int ci=0;
```

```
auto &cr = ci;
```

4.1 引用

——auto 和引用

auto 能正确的推导出引用吗？

- 下面代码中的 **r** 是什么类型？

```
int i = 0, &ri = i;
```

```
auto r = ri; //r 是 int 类型而不是 int 类型引用，auto 被推导为  
int
```

- 如果希望定义一个整型引用，需要显式指出引用类型

```
auto &r = i; //r 是 int 类型引用
```

- 下面代码中的 **cr** 是什么类型？

```
const int ci=0;
```

```
auto &cr = ci; //cr 是 const int 类型引用，auto 被推导为 const  
int
```

4.1 引用

——decltype 和引用

decltype 和引用

- `decltype` 能够根据表达式的类型来定义对象
- 如果表达式是一个对象，`decltype` 会推导出对象的类型
- 如果表达式是一个引用，`decltype` 也会推导出引用类型：

```
int i = 0, &r1=i;
decltype (r1) r2 = i;    //r2 为 int 引用
decltype (r1+0) r2;      //r2 为 int 类型
```

注意：对象名加上圆括号推导出引用

如果把对象加上一个圆括号，那么 `decltype` 会推导出引用类型，例如：

```
int i = 0;
decltype ((i)) r2;      //错误：r2 为 int 引用，必须初始化
```

4.1 引用

——右值引用

右值引用 (rvalue reference)

- C++11 非常重要的新特性
- 程序员可以操纵右值对象，尤其是临时对象
- 可以通过右值引用获取即将消亡的右值对象的资源

通过 && 定义右值引用，语法格式为：

```
int i = 0;  
int &&rr1 = i+1;    //正确：rr1 为右值引用，绑定到一个临时对象  
int &&rr2 = i;      //错误：rr2 为右值引用，不能绑定到左值对象
```

以下代码会出现什么情况？为什么？

```
int &&rr3 = rr1;
```

4.1 引用

——右值引用

右值引用 (rvalue reference)

- C++11 非常重要的新特性
- 程序员可以操纵右值对象，尤其是临时对象
- 可以通过右值引用获取即将消亡的右值对象的资源

通过 && 定义右值引用，语法格式为：

```
int i = 0;  
int &&rr1 = i+1;    //正确：rr1 为右值引用，绑定到一个临时对象  
int &&rr2 = i;      //错误：rr2 为右值引用，不能绑定到左值对象
```

以下代码会出现什么情况？为什么？

```
int &&rr3 = rr1;  
编译器报错：rr1 为左值，rr3 不能绑定到左值对象
```

4.1 引用

——右值引用

将左值显式转换成右值

```
int &&rr3 = std::move(rr1);    //将 rr1 转换成右值
```

- 为什么要将左值转换成右值？

通用引用

当右值引用声明 `&&` 与类型推导结合在一起，它将变得非常灵活。它既可以与左值绑定也可以与右值绑定，此时它变成一种通用引用类型（universal reference）：

```
int i = 0;  
auto &&rr1 = 10;    //rr1 为右值引用  
auto &&rr2 = i;     //rr2 为左值引用
```


4.1 引用

——右值引用

将左值显式转换成右值

```
int &&rr3 = std::move(rr1);    //将 rr1 转换成右值
```

- 为什么要将左值转换成右值？

因为，有时候有些左值对象可以像右值一样使用，它们具有“临时性”，当我们确定这个左值永远不会被访问时，就可以使用 `move` 转换。例如在函数体内部创建的左值局部对象（包括形参），这类对象在调用函数时创建，在函数返回时便消亡。

通用引用

当右值引用声明 `&&` 与类型推导结合在一起，它将变得非常灵活。它既可以与左值绑定也可以与右值绑定，此时它变成一种通用引用类型（universal reference）：

```
int i = 0;
auto &&rr1 = 10;    //rr1 为右值引用
auto &&rr2 = i;     //rr2 为左值引用
```

4.2 指针

访问数据的方式

- 直接访问：
 - 对象名：本质上是数据所在的内存空间的地址映射
- 间接访问：
 - 引用：通过引用访问已经存在的对象的内容，效果上与使用原对象名对数据的读写相同
 - 指针（`pointer`）：把数据的内存地址存放到专门存放地址的对象中，通过地址对象对数据进行访问

4.2 指针

——指针的定义

指针语法格式：

```
int i = 100;
```

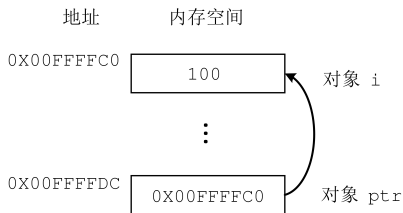
```
int *ptr = &i; //用 i 的地址初始化
```

通过取址符（&）获取一个对象的地址，把其存放到一个指针对象

上述代码实现的功能：

定义一个指向 `int` 类型对象的指针对象 `ptr`，`ptr` 中存放的是 `i` 的地址，指向 `i`。

示意图如右图所示



4.2 指针

——指针的定义

解引用操作符 (*):

如果要访问 `i` 的内容, 通过解引用操作符 (*) 来实现:

```
cout << *ptr << endl; //读操作, 读取对象 i 的内容, 输出 100
*ptr = 10;             //写操作, 修改对象 i 的内容, i 的值变为 10
```

提示: 巧读符号

如果 * 或 & 紧跟类型说明, 则定义的对象为指针或引用; 如果 * 或 & 出现在表达式中, 则为解引用或取址符, 例如:

```
int i = 0;
int *ptr = &i;  /* 紧随 int, 故 ptr 为指针; & 在表达式中, 故为取址符
int &ref = *ptr; //& 紧随 int, 故 ref 为引用; * 在表达式中, 故为解引用
```

4.2 指针

——指针的定义

定义指针对象需注意：

- 指针的类型必须和所指向的对象的类型一致，void 指针和基类指针除外。

```
int i = 10;
```

```
double *ptr = &i;    //错误: ptr 和 i 的类型不匹配
```

- 和引用类似，定义多个相同类型的指针对象，每个对象名前面都要加 *。

```
int i, *ptr1, *ptr2; //i 为 int 类型, ptr1 和 ptr2 为指针对象
```

- 若无具体的指向对象，则需用 nullptr 来初始化。在语句块内部定义但未初始化的指针，存放的是一个随机值。

```
{  
int *ptr1 = nullptr;    //ptr1 为空指针，没有指向任何对象  
int *ptr2;              //ptr2 为野指针，有潜在危险  
}
```

4.2 指针

——指针的定义

野指针

C++11 引入了新的关键字 `nullptr`，用来初始化空指针。若指针对象未显式初始化，则默认值可能是随机值（与定义的位置有关），这样的指针称为野指针（wild pointer）。

使用野指针是非常危险的，可能造成程序崩溃或更严重的后果。

提示：使用 `nullptr`

在 C++11 之前的标准中，使用 `NULL`^a 或 `0` 代表空指针，例如：

```
int *p = 0;    //p 为空指针
```

在新标准下，建议使用 `nullptr`。有些情况下可以避免一些使用上的不便，也不会引起理解上的困难（常量 `0` 是 `int` 类型不是 `int*` 类型）。

^a在 C++ 中，`NULL` 是 `0` 的宏定义

4.2 指针

——改变指向

改变指针指向

```
int i = 10, j = 100;  
int *ptr1 = &i, *ptr2 = &j;    //ptr1 指向 i, ptr2 指向 j  
ptr1 = ptr2;    //改变 ptr1 的指向, 使其指向 j, 与 ptr1 = &j; 等价  
ptr1 = nullptr; //改变 ptr1 的指向, ptr1 变成空指针
```

上面第三条语句将 `ptr2` 的内容赋值给 `ptr1`, 即将 `j` 的地址赋给 `ptr1`, 因此两个指针对象均指向对象 `j`。上面最后一条语句将 `ptr1` 置空, 不指向任何对象。

4.2 指针

——const 和指针

const 和指针

用 `const` 修饰符修饰指针对象，使其成为一个指向 `const` 对象的指针 (pointer-to-const)，表明不能通过该指针修改所指向对象的值

例如：

```
const int ci = 10, cj = 1;
const int *ptrc = &ci; //ptrc 指向常量 ci
```

- 不能通过 `ptrc` 修改 `ci` 的值：
`*ptrc = 0;` //错误：不能修改 `ptrc` 指向的常量 `ci`
- `ptrc` 指向可被修改，甚至使其指向一个非 `const` 对象
`ptrc = &cj;` //指向另外一个常量
`int i = 0;`
`ptrc = &i;` //还可以指向一个非 `const` 对象
但是，依然不能通过 `ptrc` 修改 `i` 的值，尽管它的指向是一个非 `const` 对象。

4.2 指针

——const 和指针

const 和指针

- 一个普通指针不能指向 const 对象：

```
int *ptr = &ci;           //错误: ptr 不能指向常量
```

- ptrc 指向可被修改，甚至使其指向一个非 const 对象

```
ptrc = &cj;               //指向另外一个常量
```

```
int i = 0;
```

```
ptrc = &i;                //还可以指向一个非 const 对象
```

但是，依然不能通过 ptrc 修改 i 的值，尽管它的指向是一个非 const 对象。

const 指针

const 指针的指向不允许被改变。const 指针语法格式：

```
int j = 0, i = 0;
```

```
int *const cptr = &i; //定义时初始化, cptr 只能指向对象 i
```

```
cptr = &j;           //错误: 不能改变 cptr 的指向
```

```
*cptr = 10;         //正确: 可以通过 *cptr 修改其指向的对象 i 的值
```

4.2 指针

——const 和指针

修饰 const 对象的 const 指针

```
const int *const cptrc = &ci; // cptrc 是一个指向常量 ci 的常量指针
```

第一个 `const` 修饰符表明 `cptrc` 为一个指向 `const` 对象的指针，第二个 `const` 修饰符表明 `cptrc` 不能改变指向。因此，上面定义的指针对象 `cptrc` 只能指向 `ci`，而且也不能通过 `cptrc` 修改 `ci` 的值。

4.2 指针

——类型推导和指针

auto 可自动推导出指针类型

如果表达式的值是地址值，auto 可以[自动推导出指针类型](#)：

```
int i = 0;
```

```
const int ci=10;
```

```
auto p = &i; //p 被推导为 int * 类型
```

```
auto pc = &ci; //pc 被推导为 const int * 类型，ci 的 const 属性被保留
```

p 和 pc 被分别推导为指向 int 类型和 const int 类型的指针，分别指向 i 和 ci。

4.2 指针

——类型推导和指针

注意：

定义对象时，符号`&`和`*`从属于对象名，并不是类型名的一部分，`auto`只是一个“占位符”，例如：

```
auto &ref = i, *ptr = &i;      //auto 被推导为 int
auto &ref2 = i, ptr2 = &i;     //错误：auto 的推导类型不一致
```

- 第一条语句：`auto` 被推导为 `int`，因此，`ref` 和 `ptr` 分别为与 `int` 类型相绑定的引用和指针
- 第二条语句：对于 `ref2`，`auto` 被推导为 `int` 类型，但是对于 `ptr2`，`auto` 被推导为 `int *` 类型，`auto` 的推导类型不一致

在同一条语句中定义多个对象时，它们的类型必须一致，否则出现编译错误。

4.2 指针

——类型推导和指针

利用 decltype 进行指针类型推导

```
int i = 0, *ptr = &i;
decltype (ptr) ptr2;           //ptr2 为 int *
decltype (*ptr) refi = i;      //正确: ref i 为 int &, 必须初始化
decltype (*ptr+0) j;           //正确: j 为 int 类型
```

在第三条语句中, 表达式 `*ptr` 是个解引用操作, 可以对 `ptr` 所指向的对象进行写操作, 因此 `decltype` 推导出来的是引用类型。如果想要推导出 `int` 类型, 可以利用表达式 `*ptr+0` 进行类型推导, 如第四条语句。

4.2 指针

——void 指针

void 指针

- 一类特殊的指针，它能够指向任何类型的对象
- 只简单地将对象的地址存储起来，对于对象的类型并不感兴趣

例如: `double x = 0;`

`int i = 0;`

`void *p = &x;` //正确: 可以存放 `double` 类型对象的地址

`p = &i;` //正确: 也可以存放 `int` 类型对象的地址

注意:

不能把 `void` 指针随意赋给一个普通指针，必须确保它们指向相同类型的对象，而且需要进行类型转换

例如:

`double x = 0, *ptrd = &x;`

`void *ptr = &x;`

`ptrd = static_cast<double *>(ptr);` //需要强制类型转换

4.2 指针

——多级指针

二级指针

将一个指针对象的地址存放到另一个指针对象中即构成**多级指针**(二级指针)。格式如下：

```
int i = 1, *ptr = &i;  
int **pptr = &ptr;    //用指针对象 ptr 的地址初始化 pptr  
可用三种方式访问对象 i:  
cout << i << '\t' << *ptr << '\t' << **pptr << endl;
```

三级指针

```
int ***ppptr = &pptr;  
cout << ***ppptr << endl;    //输出 1
```

4.2 指针

——引用和指针

引用和指针的区别：

- **定义引用时必须初始化**，定义指针时不需要初始化；
- **不存在空引用**。引用必须与有效的内存单元关联，指针可以为 `nullptr`；
- **赋值行为不同**。对引用赋值修改与其相绑定的对象的值，对指针赋值改变其指向的对象，例如：

```
int i = 0, j = 1, &r = i, *p = &i;  
r = 4;           //修改与 r 绑定的对象 i 的值  
p = &j;          //修改指针 p 的值，使其指向 j
```

总体来说，指针更加灵活，具有在运行期间获得并操纵地址的能力，但缺少安全性。与指针相比，引用的特点正好相反：缺乏灵活性但更安全。

4.2 指针

——引用和指针

建议：能用引用的地方不要用指针

指针的灵活性导致其在使用过程中**并不安全**，引用虽然牺牲了一定灵活性，但得到了安全保障，而且能实现指针的大部分功能。因此，除非走投无路，否则不要使用指针。

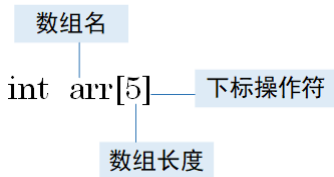
4.3 数组

——数组的定义和初始化

数组

数组是由有限个同类型元素组成的**有序集合**，所有元素顺序存放在一段连续的内存空间中。如下定义一个存储 5 个整型元素的数组：

```
int arr[5];
```



地址	内存空间	
0X0000FFB0	1	arr[0]
0X0000FFB4	2	arr[1]
0X0000FFB8	3	arr[2]
0X0000FFBC	4	arr[3]
0X0000FFC0	5	arr[4]

4.3 数组

——数组的定义和初始化

数组长度

数组长度必须为大于 0 的整型常量表达式：

```
unsigned cnt = 10;
```

```
int arr[cnt];           //错误，cnt 不是常量表达式
```

```
constexpr int sz = 10; //常量表达式
```

```
int arr1[sz];           //正确：存放 10 个整型数据的数组
```

```
float arrf[10.];        //错误：数组长度必须是整型
```

4.3 数组

——数组的定义和初始化

数组的初始化

- 定义数组时如果没有显式初始化，则采用默认的方式初始化^a。通常采用列表初始化的方式来显式初始化数组元素：

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

- 显式初始化部分数组元素：

```
int arr[5] = {1, 2, 3};    //等价于 arr[5] = {1, 2, 3, 0, 0}
```

- 编译器可以根据列表中提供的元素的个数，推断数组的长度：

```
int arr[] = {1, 2, 3, 4, 5};    //数组 arr 的长度为 5
```

^a如果定义的数组是局部对象，则元素取未定义的值。

4.3 数组

——数组的定义和初始化

字符数组

- 由于数组的特殊性，可以采用字符串字面值来初始化，例如：

```
char name[] = "Lisha";    //自动添加字符串结束符'\0'
```

- 这种方式等价于：

```
char name[] = {'L', 'i', 's', 'h', 'a', '\0'};
```

注意，上面的语句是初始化操作，不是赋值操作。不能将字符串常量赋值给一个数组

例如：

```
name = "Lisha";    //错误：数组不允许赋值操作
```

警告：数组中的数据不能整体操作

不能用一个数组初始化另外一个数组，也不能用一个数组赋值给另外一个数组

```
char n1[] = "Lisha";
```

```
char n2[] = n1;    //错误：不能用数组初始化数组
```

```
n2 = n1;    //错误：数组不能执行赋值操作
```

4.3 数组

——数组的定义和初始化

复杂数组的定义

复杂数组是指数组元素的类型是指针或引用（指向）其它数组：

```
int arr[5];           //定义一个含有 5 个 int 类型元素的数组
int *arrp[5];         //定义一个含有 5 个 int* 类型元素的数组，每个元素
                        //都是指针
int (*parr)[5] = &arr; //定义一个指向含有 5 个 int 类型元素的数组的
                        //指针
int (&rarr)[5] = arr;   //定义 arr 的一个引用
```

定义一个指向 arrp 的指针或引用：

```
int *(*parrp)[5] = &arrp;
int *(&rarrp)[5] = arrp;
parrp 和 rarrp 分别为指向指针数组 arrp 的指针和引用。
```

4.3 数组

——访问数组元素

通过下表操作符 `[]` 访问数组元素：

```
int arr[5] = {1, 2, 3, 4, 5} ;  
arr[0] = 10;      //写操作：修改第一个元素的值  
cout << arr[0] << endl;    //读操作：读取第一个元素，输出结果为：10
```

警告：C++ 不检查下标索引值是否有效

如果访问数组元素的下标值小于 0 或超过最大上限，C++ 不会提示语法错误，但程序会出现运行错误，因此程序员必须要保证下标的有效性。

4.3 数组

——访问数组元素

范围 for (range for) 语句

语法格式如下：

```
for(decl : expr){  
    statement;  
}
```

范围 for (range for) 语句

- `expr` 必须是一个对象序列，比如数组、容器（`vector`）或字符串（`string`）
- `decl` 是与序列中数据元素类型相同的对象，通常用 `auto` 来推导数据元素的类型

4.3 数组

——访问数组元素

示例：

```
int arr[5]={1,2,3,4,5};    //定义并初始化一个含有 5 个整型数的数组
for(auto i: arr){          //i 为 arr 中当前元素的副本
    cout << i << endl;    //打印输出当前获取的整数
}
```



思考：

上述 range for 语句可以对对象的内容进行修改吗？

4.3 数组

——访问数组元素

利用 range for 对数组元素进行写操作

需将 decl 声明为引用

```
for(auto &i: arr){    //i 为 arr 中当前元素的引用  
    i = 0;           //写操作：每一个元素设置为 0  
}
```

4.3 数组

——访问数组元素

例 4.1：

计算一个班级 30 名学生的数学科目的平均成绩和标准差。学生成绩随机生成。

提示：利用标准库中的随机函数生成 30 个学生的成绩，并存到数组中，再根据公式计算平均分和方差。

4.3 数组

——访问数组元素

例 4.1 :

```
#include <cstdlib>
#include <iostream>
using namespace std;
int main() {
    srand(0); // 使用固定种子，每次运行得到一样的结果，对于程序的调试是很重要的
    constexpr int sz = 30;
    int score[sz]; // 定义一个数组，存放30个学生的成绩
    int mean = 0; // 存放平均分数，初始值必须为0
    for (auto &i:score) // 使用范围for语句访问，注意引用&不能丢
        i = 50 + rand() % 51; // 成绩随机分布在50到100之间
        mean += i; // 累加每一个学生成绩到mean里面
    }
    mean /= sz; // 计算平均成绩
    double dev = 0;
    for (int i = 0; i < sz; ++i)
        dev += pow(score[i] - mean, 2); // 函数pow(x,a)计算x^a
    }
    dev = sqrt(dev / sz);
    cout << "平均成绩: " << mean << " 标准差: " << dev << endl;
    return 0;
}
```

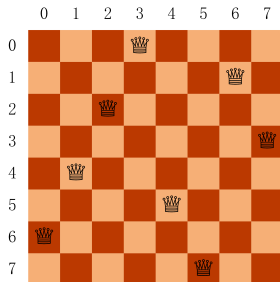
4.3 数组

——访问数组元素

例 4.2 :

在 8×8 的国际象棋棋盘上摆放八个皇后，使其不能相互攻击，即任意两个皇后不得处在同一行、同一列或者同一对角斜线上。下图所示是一种符合条件的摆放方案。本题计算出一种方案即可。

提示：本题可用回溯法（**backtracking**）求解。回溯法基本思想是采用深度优先搜索（**depth-first search**）策略，从当前局面往前走，能进则进，不能进则退回来，换一条路再试，也称为试探法。



4.3 数组

——访问数组元素

例 4.2 :

```
#include <iostream>
using namespace std;
int main() {
    constexpr int sz = 8;
    int que[sz] = { 0 }; // 每一行皇后都从第0列开始摆放
    int i = 0; // 从第0行开始摆放
    while (i >= 0)
        int k = 0;
        while (k < i) // 检查前面所有皇后是否和第i行皇后冲突
            if (que[k] != que[i] && (abs(que[i] - que[k]) != abs(i - k)))
                ++k; // 第k行和第i行皇后没有冲突
            else
                break; // 第k行和第i行皇后产生冲突，退出，转到第15行
        }
    if (k < i) { // 检测到冲突
        ++que[i]; // 处理冲突：移动第i行皇后到当前位置的下一列
        while (que[i] == sz) // 当前行所有尝试都失败，需要回溯
            que[i] = 0; // 重置当前行皇后位置
            --i; // 回溯到上一行
            if (i < 0)
                break; // 如果回溯到第0行之前，结束运行，转到第24行
            ++que[i]; // 前一行皇后后移一列
        }
    continue; // 重新检测是否与前面已安排皇后冲突，转到第7行
```

4.3 数组

——访问数组元素

例 4.2 :

```
else { // 没有检测到冲突，安排下一行皇后
    ++i; // 移动到下一行
    if (i < sz)
        continue; // 安排下一行皇后，已安排在第0列，转到第7行
    cout << "找到一个方案："; // 否则找到一个方案并输出
    for (k = 0; k < sz; ++k)
        cout << que[k];
    cout << endl;
    break; // 结束运行，转到第37行
}
return 0;
}
```

4.3 数组

——多维数组

多维数组指的是数组中的元素类型为数组类型。

二维数组

```
int a2d[3][5];
```

阅读方法：a2d 是一个数组，有 3 个元素，再往右阅读（[] 为左结合性），每个元素（a2d[i], i=0,1,2）类型为一个存放 5 个整型数的一维数组，即包含数组的数组，a2d 存放 15 个整数

三维数组

```
int a3d[2][3][5];
```

阅读方法：a3d 是个数组，有 2 个元素，每个元素（a3d[i], i=0,1）是一个二维数组（类型与 a2d 一样），依此类推，直到阅读完最后一维。按照这个方法可以定义更高维的数组。

无论有多少维数，数组元素都存放在一段连续的内存空间。一维数组可以对应数学中的向量。二维数组可对应矩阵，因此二维数组的第一维常称为行，第二维称为列。

4.3 数组

——多维数组

多维数组初始化

- 用列表方式初始化多维数组

如：

```
int a2d[3][5] = {  
    {0, 1, 2, 1, 4},  
    {7, 5, 4, 5, 7},  
    {0, 8, 5, 2, 9}};
```

- 内嵌的花括号可以省略：

```
int a2d[3][5] = {0, 1, 2, 1, 4, 7, 5, 4, 5, 7, 0, 8, 5, 2, 9};
```

- 显式初始化部分数组元素：

```
int a2d[3][5] = {0, 1, 2};
```

- 显式初始化每个一维数组中的第一个元素：

```
int a2d[3][5] = {{0}, {1}, {2}};
```

- 通过列表元素让编译器自动推断第一维长度：

```
int a2d[][5] = {0, 1, 2, 1, 4, 7, 5, 4, 5, 7, 0, 8};
```

或者：

```
int a2d[][5] = {{0}, {1}, {2}};
```

4.3 数组

——多维数组

访问多维数组元素

用范围 `for` 语句处理多维数组时，除了最内层的循环外，其它各层循环中**必须使用引用**，例如：































```
int a2d[3][5];  
for (auto &row : a2d)    //row 被推导为 int(&)[5] 类型  
    for (auto col : row) //正确: row 为一维数组的引用
```

4.3 数组

——多维数组

例 4.3 :

打印扫雷游戏的地图。在一个 $n \times n$ 网格化的地图上，随机分布一些地雷，要求在每个没有设置地雷的网格内标记出其相邻区域内地雷的数目，每个网格相邻区域只包括同一行和同一列紧邻的 4 个网格。

	0	1	2	3	4	5	6	7
0		2	0	0	2		2	0
1			1	1		3		2
2	2		1	1	2	2		
3	1	1	1					2
4		2	2		3			2
5					1	3		
6		3	2		2			2
7			1	1	0	2		1

4.3 数组

——访问数组元素

例 4.3 :

```
#include <cstdlib>
#include <iostream>
#include <ctime>
using namespace std;
int main() {
    srand(time(0));
    constexpr int sz = 8;
    char map[sz][sz];
    for (auto &row : map)           // 每个元素的引用
        for (auto &col : row)       // 内嵌数组中每个元素的引用
            int num = rand() % 100;
            if (num <= 40)           // 以0.4的概率设置每个方格的地雷
                col = '*';
            else
                col = '0';           // 没有地雷的方格初始化为字符0
    }
```

4.3 数组

——访问数组元素

例 4.3 :

```
for (int i = 0; i < sz; ++i) {
    for (int j = 0; j < sz; ++j) {
        if (map[i][j] != '*')
            continue;           // 跳过地雷的方格
        if (i + 1 < sz && map[i + 1][j] != '*') map[i + 1][j] += 1;
        if (i - 1 >= 0 && map[i - 1][j] != '*') map[i - 1][j] += 1;
        if (j + 1 < sz && map[i][j + 1] != '*') map[i][j + 1] += 1;
        if (j - 1 >= 0 && map[i][j - 1] != '*') map[i][j - 1] += 1;
    }
}

for (int i = 0; i < sz; ++i) {
    for (int j = 0; j < sz; ++j) {
        cout << map[i][j] << " ";
    }
    cout << endl;
}

return 0;
}
```

4.4 指针和数组

——指针指向数组

指针和数组的关系非常紧密

- 数组名被转换成数组第一个元素的地址

```
int arr[] = {1, 2, 3, 4, 5};  
int *p = arr;           //arr 被转换成 arr[0] 的地址  
第二句等价于:  
int *p = &arr[0];
```

- 利用auto进行类型推导，得到的是一个指针

```
auto pa = arr;           //pa 为 int * 类型，显然是一个指针  
cout << *pa;            //输出 arr[0] 的值 1
```

- 利用decltype定义新数组时，数组名arr不会转换为指针

```
decltype (arr) ar2;      //ar2 为存放 5 个整型数的一维数组
```

- 可用指针指向多维数组

```
int a2d[3][5];  
int (*p2d)[5] = a2d;    //指向 a2d 的第一个元素
```

上面的指针定义中，圆括号不能省略

4.4 指针和数组

——指针指向数组

数组名和指针对象的关系

一个数组名可理解为一个 `const` 指针，但两者并不完全等价
如：

```
int * const p = &arr[0];    //arr 可以理解为 const 指针 p
cout << sizeof(arr) << " " << sizeof(p);
```

使用运算符 `sizeof` 测试的输出结果为：20 4，分别为一个含有 5 个整型元素的数组和一个指向整型类型的指针对象的大小

4.4 指针和数组

——利用指针访问数组

利用指针访问数组

对于如下数组和指针：

```
int arr[] = {1, 2, 3, 4, 5};  
int *p = arr;    //p 指向数组 arr
```

C++ 支持如下指针运算：

- **指针的移动**。int *p2 = p + 3; //返回 p 后面第 3 个元素的地址，即 &arr[3]

也可以利用自增或自减运算符移动指针 p 的位置：

```
int *p3 = p++; //p 后移一个位置，p3 指向 p 原来的位置 &arr[0]  
int *p4 = ++p; //p 继续后移一个位置，p4 和 p 指向同一个位置 &arr[2]
```

- **关系运算**。支持所有的关系运算符，返回两侧的指针所指向元素的前后位置关系，比如 p == p4, p4 > p3, p3 <= p2 等都为真。
- **指针相减**。两个指针相减的结果为所指向数组元素的位置距离，比如表达式 p4-p3 的结果为 2。

4.4 指针和数组

——利用指针访问数组

关于指针运算需注意：

- **指针**在运算的过程中**不能越界**。第一个元素到最后一个元素的下一个位置为有效位置，比如：
`p2 = &arr[0];` //正确：指向第一个元素，等价于 `p2 = arr;`
`p2 = &arr[5];` //正确：指向尾元素后面的一个位置，等价于 `p2 = arr + 5;`
虽然不存在元素 `arr[5]`，但可以计算该位置的地址。
- 仅当两个指针指向同一个数组时，它们之间的运算才有意义。

4.4 指针和数组

——利用指针访问数组

利用指针访问二维数组

比如：

```
int a2d[3][5];
```

```
int (*p2d)[5] = a2d;
```

将下标运算符作用于指针来访问数组元素： `p2d[1][1] = 1;`

利用 auto 简化代码

```
int a2d[3][5] = {{1}, {1}, {1}};
```

```
for(auto p = a2d; p < a2d + 3; ++p) //p 的类型为 int (*)[5]
```

```
    for(auto q = *p; q < *p + 5; ++q) //q 的类型为 int *
```

```
        cout << *q << " ";
```

```
    }
```

```
    cout << endl;
```

```
}
```

4.5 string 类型

string 类型是 C++ 标准库类型

支持变长的字符串和常用的字符串操作。

using 声明

如下声明来引入单个名字: `using std::cin;`

一劳永逸地引入 std 命名空间内所有的名字:

```
using namespace::std;
```

提示：合理使用 using 声明

一般来说，**using 声明不应放到头文件里面**。一个头文件常被多个代码文件包含，因此该头文件里面的 **using** 声明会被引用到所有包含该头文件的代码文件里。如果两个不同的文件使用了相同的名字，会引起命名冲突，有些时候甚至没有觉察到由于命名冲突而使用了错误的名字，因为程序好像是正确的。另外，程序员自己的命名不要和标准库的名字冲突，比如 `sort`、`list`、`string` 等。

string 类型

——定义和初始化 string 对象

定义 string 类型

`string` 是类类型，可以采用如下方法定义一个 `string` 类型对象：

```
string str1;           //默认初始化，定义一个空字符串
string str2(str1);     //等价于 string str2 = str1; str2 是 str1 的一个拷贝
string str3 = "Rosita"; //复制初始化
string str4("Rosita");  //直接初始化
string str5(5, 'R');    //直接初始化，str5 的内容为 RRRRR
```

4.5 string 类型

——string 类型常用操作

string 对象的输入和输出

```
string s;  
cin >> s; //遇到空白字符停止  
cout << s; //输出 s 的内容
```

利用 `getline` 函数读取空白字符：

```
getline(cin, s);
```

当输入 "hello C++ " 时（注意里面的空格），s 的内容为 "hello C++ "。

string 对象的大小

```
string s;  
cin >> s;  
cout << s.size() << endl; //输出 s 里面字符的个数，与 s.length() 等价  
if(!s.empty()) //如果 s 非空，则输出其内容  
    cout << s;
```

string 对象的关系运算

比较规则：如果两个 `string` 对象长度不一样，且较小的 `string` 对象和较大的对象前面的每个字符都一样，则较小的 `string` 对象小于较大的 `string` 对象；否则返回相同位置上第一对不同字符的比较结果，例如：

```
string s1 = "Hello C++";
```

```
string s2 = "Hello";
```

```
string s3 = "Hi";
```

依据上述规则，s1 大于 s2，s2 小于 s3。

string 对象的加法运算

```
string s1 = "Hello ", s2 = "C++";  
string s3 = s1 + s2;  
s1 += s2;  
string s4 = "Hello " + s2;
```

4.5 string 类型

——string 类型常用操作

利用下标运算和 at 函数访问单个字符

```
string s = "hello";  
s[1] = 'H';    //对第二个元素进行写操作  
cout << s.at(1) << endl;
```

下标运算和 `at` 函数都要求一个有效的位置值，最小值为 0，最大值为对象的长度-1。利用 `at` 成员函数访问是安全的，它会自动检查位置的合法性

C++11 还支持 `front` 和 `back` 操作访问第一个和最后一个字符，例如：

```
cout << s.front() << " " << s.back() << endl;    //打印输出 h o
```

4.5 string 类型

——string 类型常用操作

例 4.4 :

猜单词游戏，其中一个玩家给定一个单词，让另外一个玩家猜。规则如下：每次猜测单词里面可能的一个字母，并给定猜错的最大次数，比如 3 次。每次猜测要给出相关提示，包括猜测的字母是否正确、字母是否已经猜测过、剩余机会次数以及当前猜测的进度，未猜中的字母用符号 * 代替。如果在给定的最大猜错次数内正确猜出单词的所有字母，则挑战成功，否则失败。失败时给出单词的全部字母。

4.5 string 类型

——string 类型常用操作

例 4.4 :

```
#include <iostream>
#include <string>
using namespace std;
int main() {
    string target;
    cout << "请给出一个单词：";
    cin >> target;
    cout << string(100, '\n'); //输出100个换行，用来隐藏输入的单词
    int length = target.length();
    string attempt(length, '*'), badchars; //分别记录当前正确和错误的猜测
    int guesses = 5; //最大尝试次数
    cout << "单词已准备好，它有" << length << "个字母：" << attempt << endl;
    do{
        char letter;
        cout << "请猜测一个字母：";
        cin >> letter; //badchars或attempt中已有letters
        if (badchars.find(letter) != string::npos ||
            attempt.find(letter) != string::npos)
            cout << "已经猜过该字母，请重猜" << endl;
        continue; //string::npos 匹配失败标志位
    }
}
```


4.5 string 类型

——string 类型常用操作

例 4.4 :

```
auto loc = target.find(letter); //使用 auto 自动推导 loc 类型
if (loc == string::npos) {
    cout << "没有此字母!" << endl;
    --guesses; //允许错误次数 -1
    badchars += letter; //猜错的字母放到 badchars 里
}else {
    cout << "有这个字母，继续加油!" << endl;
    do { //把 attempt 里面相应的 * 用猜对的字母替换
        attempt[loc] = letter; //如果找到了，则下一次搜索从 loc+1 开始
        loc = target.find(letter, loc + 1);
    } while (loc != string::npos);
}
cout << "你猜测的单词:" << attempt << endl;
if (attempt != target)
    cout << "剩余" << guesses << "次猜错机会" << endl;
} while (guesses > 0 && attempt != target);
if (guesses > 0)
    cout << "成功了，恭喜你!" << endl;
else
    cout << "对不起，失败了，下次再挑战吧，单词是" << target << endl;
}
```

4.5 string 类型

——C 风格字符串

C 风格字符串

C 风格字符串不是一种类型，而是以空字符结尾 ('\0') 的字符数组，例如：
`char cstr[] = "Hello";`

表: 常用 C 风格字符串处理函数

<code>strlen(s)</code>	返回 <code>s</code> 的长度，不包含结束符 '\0'
<code>strcmp(s1,s2)</code>	字符串比较函数。和 C++ <code>string</code> 类型对象相比较的规则一样：如果 <code>s1==s2</code> ，返回 0；如果 <code>s1>s2</code> ，返回正值；如果 <code>s1<s2</code> ，返回负值
<code>strcpy(s1,s2)</code>	字符串复制函数。将字符串 <code>s2</code> 复制给 <code>s1</code> ，返回 <code>s1</code>
<code>strcat(s1,s2)</code>	字符串链接函数。将字符串 <code>s2</code> 附加到 <code>s1</code> 之后，返回 <code>s1</code>

4.5 string 类型

——C 风格字符串

利用指针处理 C 风格字符串

```
char *ps = cstr; //指向字符数组 cstr
char *ps2 = "C++";
cout << ps << ", " << ps2 << endl; //输出 Hello,C++
```

使用处理 C 风格字符串的函数时应注意：

- 每个操作对象必须以空字符 '\0' 结尾，否则会产生未定义的行为：

```
char cs[] = {'C', '+', '+'};
cout << strlen(cs) << endl; //错误：cs 没有以空字符结束
```

- 如果对操作对象进行修改，必须要有足够大的内存空间：

```
char small[] = "C++", big[] = "Programming";
cout << strcpy(small, big) << endl; //错误：small 内存空间不足
strcpy 函数要求第一个操作对象的内存空间至少应能容纳第二个对象的所有内容和一个空字符，strcat 函数也有这个要求，否则会产生严重的错误。
```

4.5 string 类型

——C 风格字符串

string 类对象使用 C 风格字符串处理函数

需要通过 `string` 类成员函数 `c_str` 来获取 `string` 对象存储的字符串的首地址，例如：

```
string str = "hello";
```

```
char carr[10];
```

```
strcpy(carr, str.c_str());
```

`string` 的成员函数 `c_str` 返回 `const char*` 类型的指针，确保其指向的对象不被修改。

建议：尽量选择 `string` 类而不用 C 风格字符串

尽管 C++ 全面支持 C 风格字符串，但 C 风格字符串在使用上既不方便又不安全，可能造成严重的安全隐患，因此不建议在程序中使用 C 风格字符串。

4.6 vector 类型

vector 类型

- vector 和数组都是有序元素的集合
- vector 支持变长操作，容量大小可根据需要动态调整
- vector 是一种容器类型，能够存放类型相同的元素
- 使用 vector 需要在程序中包含 vector 头文件：
`#include <vector>`

4.6 vector 类型

——定义和初始化 vector 对象

定义 vector 类型的方法

```
vector<int> v1;                //存放整数的空 vector
vector<int> v2 = {0,1,2};      //v2 有三个元素，值分别为 0、1 和 2
vector<int> v3(10);            //v3 可存放 10 个整数，值为默认值 0
vector<int> v4(10,1);          //v4 存放 10 个整数 1
vector<string> v5 = {"Hi","Lisha","Mandy","Rosita"};
vector<vector<int>> v6(10,v2);
```

与数组类似，vector 里面的元素也是顺序存放在连续的内存空间内

表: 定义和初始化 vector 对象常用方法

<code>vector<T> v1</code>	定义一个存放 T 类型元素的空对象 v1
<code>vector<T> v2(v1)</code>	复制 v1 里面所有元素到 v2
<code>vector<T> v3(n)</code>	指定初始元素为 n 个
<code>vector<T> v4(n,value)</code>	指定 n 个值为 value 的元素
<code>vector<T> v5={a,b,c,...}</code>	采用列表初始化,v5 的元素个数为列表里面值的个数
<code>vector<T> v6{a,b,c,...}</code>	等价于 <code>v5={a,b,c,...}</code>

4.6 vector 类型

——vector 类型常用操作

添加、删除元素

```
vector<int> vi;  
for(int i=0; i < 100; ++i)  
    vi.push_back(i); //依次添加  
100 个数: 0-99  
vi.push_back() 成员函数称为尾插,  
即从容器尾端添加元素  
vi.pop_back() 成员函数可从容器尾  
端移除一个元素  
vi.clear() 成员函数可移除容器所有  
元素
```

访问元素

可用下标运算符或 `at` 成员函数访问容器里的元素

```
cout << vi.at(1); //或者  cout  
<< vi[1];
```

`at` 函数会自动检查访问位置的合法性而下标运算符不会

4.6 vector 类型

——使用迭代器

为什么使用迭代器？

除 string 和 vector 外，C++ 标准库里面还有许多其它的容器类型不支持随机访问。为了统一，C++ 标准库提出了迭代器（iterator）。迭代器的行为类似于指针，支持对数据的间接访问，也支持在元素间移动。迭代器将在 11.1 节中详细介绍。

借助容器的成员函数获取元素迭代器

```
vector<int> vi = {0,1,2,3};  
auto itb = vi.begin(); // itb 指向 vi 的第一个元素  
auto ite = vi.end();   // ite 指向 vi 的尾后元素
```

利用解引用获取迭代器指向对象的内容

```
cout << *itb << endl; // 输出第一个元素值 0
```


4.6 vector 类型

——使用迭代器

指向 vector 类型的迭代器支持指针运算

```
for(auto it = vi.begin(); it != vi.end(); ++it){  
    *it *= 2; //每个元素乘 2  
    cout << *it << endl;  
}
```

在 `for` 循环的结束条件中，为了代码的通用性习惯上为迭代器选择 `!=` 运算，而不是 `<` 运算

4.6 vector 类型

——使用迭代器

使用成员选择运算符

当迭代器指向的元素类型为类类型时，可以用 `.` 或 `->` 运算符进行成员选择，例如：

```
vector<string> vs = {"Hi", "Lisha", "Mandy", "Rosita"};
for(auto it = vs.begin(); it != vs.end(); ++it ){
    cout << (*it).size() << endl; //选择 string 类成员函数 size
}
```

注意：迭代器外面的圆括号不可缺少，否则将表达完全不同的意思
如： `*it.size()`； //错误：迭代器 `it` 没有成员函数 `size`，相当于 `*(it.size())`；

使用 `->` 运算符简化表达

```
for(auto it = vs.begin(); it != vs.end(); ++it ){
    cout << it->size() << endl;
}
```



































4.6 vector 类型

——使用迭代器

例 4.5

为例 4.3 中的扫雷游戏地图中的每个方格编号，编号从 0 开始，按照从上到下、从左到右的顺序依次编号。例如，第 0 行第 0 列编号为 0，第 0 行第 1 列编号为 1，依次类推。如下图所示，其中雷区编号未显示。

提示：本题可用宽度优先搜索 (**breadth-first search**) 策略来求解。宽度优先搜索的基本思想是从初始状态点开始，找到下一步所有可能到达的状态点，然后再依次找到每一个点的下一个状态点，依次类推，直到找到问题的解。

	0	1	2	3	4	5	6	7
0					4	5	6	7
1	8		10			13		
2		17	18	19	20	21	22	
3	24		26			29		31
4		<u>33</u>					38	39
5		<u>41</u>		43	44	45		
6	<u>48</u>	<u>49</u>			52		54	
7		<u>57</u>		59				

4.6 vector 类型

——使用迭代器

例 4.5 :

```
#include <cstdlib>
#include <iostream>
#include <vector>
#include <iomanip> // 使用库函数 setw
using namespace std;
int main() {
    srand(0);
    constexpr int sz = 8;
    char map[sz][sz];
    for (auto &row : map)
        for (auto &col : row) // 以0.5的概率设置地雷，使用条件表达式简化代码
            col = rand() % 100 < 50 ? '*' : '0';
    // 打印地图，函数setw设置打印字符的宽度（见10.2.2节）
    for (int i = 0; i < sz; ++i) {
        for (int j = 0; j < sz; ++j) {
            if (map[i][j] == '*') cout << setw(3) << " * ";
            else cout << setw(3) << i*sz+j;
        }
        cout << endl;
    }
    int cell;
    cout << "请输入选择的方格编号[0-" << sz*sz - 1 << "]: ";
    cin >> cell;
```

4.6 vector 类型

——使用迭代器

例 4.5 :

```
if (map[cell/sz][cell%sz] == '*') {
    cout << "选择的是地雷" << endl;
}else // 容器nobomb存放未处理的方格编号，初始值为选择的方格编号
vector<int> result, nobomb(1,cell);
map[cell / sz][cell%sz] = '1';// 标记该方格已经遍历
do { // 取出nobomb中第一个待处理的方格编号，找到与cell相邻的方格
    cell = nobomb.front();
    //neibor存放与cell相邻的4个方格编号，如果没有对应方格编号标记为-1
    int neibor[]={(cell/sz>0)?cell-sz:-1,(cell%sz>0)?cell-1:-1,
        (cell/sz<sz-1)?cell+sz:-1,(cell%sz<sz-1)?cell+1:-1};
    for (auto &k : neibor) // 注意k!=-1必须放到逻辑与的左侧
        if (k != -1 && map[k/sz][k%sz] == '0'){
            nobomb.push_back(k); // 所有与cell相邻的无雷方格放到nobomb中
            map[k/sz][k%sz] = '1'; // 标记该方格已经遍历过
        }
    }
    result.push_back(cell); // 将处理完的方格编号cell放到result中
    nobomb.erase(nobomb.begin()); // 将cell从nobomb中移除
} while (!nobomb.empty());
for (auto i : result)
    cout << i << " ";
}

return 0;
```

4.7 枚举类型

——定义枚举类型

枚举类型

提供了一种简单的方式来使用和维护一组整数值

定义枚举类型

- 不限定作用域 (unscoped enumeration)

```
enum color{red, green, blue};
```

三个枚举成员的作用域与枚举类型本身的作用域相同

```
enum emotion{happy, calm, blue}; // 错误: 枚举成员 blue 已经定义过
```

- 限定作用域 (scoped enumeration)

```
enum class stoplight{red, green, yellow};
```

```
color c = red; // 正确, 可以访问 color 类型的枚举成员
```

```
stoplight a = red; // 错误: stoplight 类型的枚举成员 red 在此不可访问
```

```
stoplight b = stoplight::red; // 正确
```

4.7 枚举类型

——定义枚举类型

定义枚举类型

每个枚举成员都有一个常量整数值，默认值从 0 开始，依次加 1。也可以指定枚举成员的值：

```
enum class week{Sunday = 7, Monday = 1, Tuesday, Wednesday,  
Thursday, Friday, Saturday};
```

枚举类型 `week` 中成员 `Sunday` 值为 7，`Monday` 值为 1，其余成员值依次加 1。即 `Tuesday` 值为 2、`Wednesday` 值为 3，依次类推。

4.7 枚举类型

——使用枚举类型

使用枚举类型

编译器**不会**把一个整型值**自动转换为枚举类型**:

```
color c1 = 1; // 错误: 类型不匹配
```

要用强制类型转换将一个整型值转换为一个枚举常量:

```
color c2 = static_cast<color>(1);
```

用 switch 分支结构列举枚举成员

```
stoplight l= stoplight::red;
switch (l){
    case stoplight::red:
        cout << "stop!" << endl;
        break;
    case stoplight::green:
        cout << "pass carefully" << endl;
        break;
    case stoplight::yellow:
        cout << "slow down" << endl;
        break;
    default:
        cout << "light broken, call 122"
        << endl;
        break;
}
```