C语言入门（*CS100*）Part Three

> *-- By Yiming Li*

> 本文章的内容参考至*Stephen Prata*的《C Primer Plus》

> 本内容适用于*CS*专业学生

函数

- 函数就是程序中独立的功能

函数的基本语法

在C语言中，函数的定义和使用是程序设计中的重要部分。以下是关于如何定义以及使用函数的详细说明：

- **函数定义**：
  - 函数定义指定了函数的名称、返回类型、参数列表以及函数体。
  - 语法格式如下：

```
返回类型 函数名(参数类型1 参数名1, 参数类型2 参数名2, ...)
{
    // 函数体
    // 包含实现功能的代码块
    // 可能包含return语句
}
```

  - 示例：定义一个简单的函数，用于计算两个整数的和。

```
int add(int a, int b) {
    return a + b;
}
```

- **函数声明（原型）**：
  - **如果函数定义出现在调用点之后，那么需要提前声明函数**，即提供函数原型。
  - 语法格式为：

```
返回类型 函数名(参数类型1, 参数类型2, ...);
```

  - 继续上面的例子，在使用`add`函数之前可以这样声明：

```
int add(int, int);
```

- **什么时候需要声明函数?**

  在C语言中，是否需要声明函数取决于函数定义的位置以及程序的结构。以下是具体的情况说明：

  - **不需要声明函数的情况**：
    - 当函数定义出现在所有调用该函数的代码之前时，可以不进行函数声明。
    - 例如：

```c
// 先定义函数
int add(int a, int b) {
    return a + b;
}

int main() {
    int sum = add(3, 5); // 直接调用，无需提前声明
    return 0;
}
```

  - **需要声明函数的情况**：
    - 如果函数定义出现在调用点之后，则必须在调用之前声明函数原型。
    - 这种情况常见于将函数定义放在源文件末尾或分散在不同文件中的情形。
    - 例如：

```c
// 提前声明函数原型
int add(int, int);

int main() {
    int sum = add(3, 5); // 调用已声明但未定义的函数
    return 0;
}

// 定义函数
int add(int a, int b) {
    return a + b;
}
```

  - **推荐做法**：
    - 即使函数定义在调用之前，也建议在源文件顶部或头文件中声明函数原型。这样做有助于提高代码的可读性和维护性，并且可以在编译时进行类型检查。
    - 通常会在`.h`头文件中声明函数原型，在对应的`.c`文件中实现函数。

  总结来说，当函数定义位于所有调用它的代码之前时，可以省略函数声明；否则，为了确保编译器能够正确解析函数调用，应该先声明函数原型。

- `void`**关键字的使用** 在C语言中如果没有返回值那么就使用`void`关键字表示返回值类型。

同样的如果不需要接受外部的数据则函数的定义时候`返回值 函数名()/(void)`这是语法格式

- **函数调用**:
  - 当需要执行某个函数时，可以通过函数名加上实际参数来调用它。
  - 例如，调用上面定义的`add`函数:

```
int sum = add(3, 5); // 调用add函数，并将结果赋值给sum变量
```

  函数调用中的参数叫做实参，实参可以是常量，变量或者是表达式。之间使用逗号隔开

  只有函数完成定义或者是函数声明之后才能调用函数.

- **注意事项**:
  - 函数内部可以有0个或多个`return`语句，但是一旦遇到`return`就会立即结束函数并返回指定的值给调用者。
  - 参数传递分为按值传递和按地址传递（通过指针），这会影响函数内部对传入数据的操作是否会影响到原始数据。

变量的作用域

- 变量的作用范围分为两种：1.局部变量2.全局变量

1. 局部变量是在函数内部或者是在`{}`内定义的，其作用域仅限于函数内部或者是`{}`块内，离开函数或块后在使用这种变量是非法的。

```
int fun1(int a)
{
    int b,c;
    ...
}
int main()
{
    int a,b,c;
    ...
}
```

   - 在这一段代码中`fun1`中的`a,b,c`三个变量是局部变量，且作用域在整个`fun1`内定义的
   - 同样的`int main()`中的`a,b,c`三个变量也是局部变量是不能够在别的函数中使用的
   - 正因为两个作用的范围不同所以不会出现重名报错，也不可以相互替代。
2. 全局变量是定义函数体外部的变量：作用域是从"定义处"到文件结束

- 如果用户在定义的时候不显示给出变量的初始值，全局变量默认值为0

```
int a,b;//  全局变量
void fun1()
{
    ...
}
double x,y;//全局变量（注意只能够在fun2,main中使用，不能够在fun1中使用）
int fun2()
{
    ...
}
int main()
{
    ...
}
```

注意事项

1. 函数不调用就不执行
2. 函数名不能够重复
3. 函数与函数之间是平级关系不能够嵌套定义，但是可以嵌套使用
4. 自定义函数写在`main`函数的下面，但是需要在上方声明
5. `return`下面的代码无效
6. 函数的返回类型如果写了`void`，表示没有返回值，如果写了`return`后面不能够加上具体的数值，仅仅表示结束函数

C语言中的常见函数

C语言中的函数有很多，但是他们会根据不同的功能进行分类，分别分在了不同的头文件中 `<math.h>`, `<stdio.h>`,`<stdlib.h>`,`<string.h>`

头文件名称	包括的函数举例
<math.h>	pow(),sqrt(),fabs(),(l/ll)abs(),cell(),floor()
<stdio.h>	printf(),scanf()
<stdlib.h>	malloc(),free(),calloc(),rand()
<string.h>	strlen(),strcmp(),strlwr(),strupr(),strcat(),strcpy()
<time.h>	time()

需要查找： 查找头文件 或者问ai

- 举例常见函数使用`time()`

```
#include <stdio.h>
#include <time.h>
int main()
{
```

```
    long long res = time(NULL);
    printf("lld\n",res);
    return 0;
    // 表示从1970年1月1日0: 0: 0开始过了res秒之后的时间点
}
```

- 随机数的获取（伪随机数：是通过数学中的线性同余公式进行计算出来的）

```
// 通过线性同余方程实现的内容
int num0 = 0;
int num1 = (31 * num0 + 13) % 100;
```

这以上的内容其实就是线性同余方程的内容，但是在C语言中不需要你自己实现，有对应的函数`rand()`函数

- 所以在C语言中计算随机数就只有两个步骤：

1. `srand()`函数设置种子
2. `rand()`函数获取随机数
3. 这两个函数都被包括在`<stdlib.h>`*standard library*标准库中

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    // 1. 设置种子
    // 初始值，因为没一个随机数都是通过一个数字来生成的
    srand(1); // 注意一定要放在for循环的外部，因为这是一个伪随机数
    for (int i = 0; i < 10; i++)
    {
        int res = rand();
        printf("%d\n", res);
    }
    return 0;
}
```

- 随机数的弊端：

1. 种子不变随机数的结果是固定的
2. 随机数的范围

- 解决办法

1. 使用刚才的时间作为种子
2. 使用百分号取余来限制范围

```
#include <stdio.h>
#include <stdlib.h>
```

```c
#include <time.h>
int main()
{
    srand(time(NULL));
    for (int i = 0; i < 10; i++)
    {
        int res = rand();
        printf("%d\n", res % 10);
    }
    return 0;
}
```

综合小练习：猜数字小游戏

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int random(void);
int main()
{
    int num = random();
    printf("Please enter a number between 0 and 100:");
    int guess, times = 0;
    scanf("%d", &guess);
    while (guess != num)
    {
        guess > num ? printf("the number you enter need to be smaller\n") :
printf("the number you enter need to be bigger\n");
        scanf("%d", &guess);
        times++;
    }
    printf("Welldown!! the randit number is %d \nthe number you guess is %d\nYou
have used %d times to guess!!", num, guess, times+1);
}
int random(void)
{
    srand(time(NULL));
    int res = rand() % 100;
    return res;
}
```

函数练习：使用ASCII编码来完成大小写转换

```c
#include <stdio.h>
char upper(char ch);

int main()
{
    char ch;
```

```
    scanf("%c", &ch);
    ch = upper(ch);
    printf("%c\n", ch);
    return 0;
}
char upper(char ch)
{
    return ch - ('a' - 'A');
}
```

数组

- 数组是一种容器，可以用来储存同种数据类型的多个值

1. 数组怎么定义
2. 数组初始化
3. 元素访问
4. 数组遍历
5. 内存中的数组
6. 数组的常见问题
7. 二维数组

数组的定义

`数据类型 数组名[长度]`例如`int arr[40]`

- 数组的特点

1. 数组是一个连续的空间
2. **一旦定义长度不能够发生改变**

```
#include <stdio.h>
int main()
{
    // 定义数组储存全班80人的年龄
    int arr1[80];// 可以储存int,short类型
    // 定义数组储存全班50人的身高
    double arr2[50];
    // 定义数组储存全什么每件衣服的价格
    double arr3[5];
    return 0;
}
```

数组的初始化

定义数组的时候第一次给数组赋值：`数组类型 数组名[长度] = {数据值，数据值····}`

- 几点注意：

1. 数据值的个数就是数组的长度如果没有定义数组的长度，但是给出了固定的数据值的个数那么就不需要补全长度
2. 长度没有省略的情况之下：数据值的个数要**小于等于**给定长度（`int arr [5] = {1,2,3}`）虽然没有用完所有的内容，但是后面空余的部分是存在内容的，这就是**默认值**：`int`:0;`double`:0.0;`char`:`\0`;字符串:`NULL`。
3. 如果给多了：**程序就会报错**

```
int arr1[5] = {17,18,19,20,21};
int arr2[] = {17,18,19,20,21};
```

这两种定义都是允许的，因为在没有确定的数组个数的时候，程序会自动将后面已知的几个内容作为数组的长度储存。

元素访问

1. 获取某个数据`int num = arr[5];`
2. 修改某个数据`arr[5] = 10;`

数组遍历

```
for (int i =0 ;i <=4 ;i++)
{
    printf("%d\n",arr[i]);
}
```

内存中的数据

- 内存：软件运行时，用来临时储存数据的

1. 将数据保存到内存中
2. 从内存中的对应位置把数据取出来

- 怎么放，怎么找？————内存地址 内存地址就是操作系统对每一个字节的编号，快速管理内存空间
- 如何编号的呢？————内存地址的规则
  - 32位操作系统，内存按照32位的的二进制表示：`0000 0000 0000 0000 0000 0000 0000 0000`-`1111 1111 1111 1111 1111 1111 1111 1111`;因此一共可以储存4亿多个字节(4096MB~4G)
  - 64位操作系统，内存按照上述的表示，一共可以支持(17179TB)的空间。但是对于一个64位的二进制而言明显太长了，因此我们会将它每四个为一组转换成为16进制：`0000 0000 0000 0000 0001 0000 1111 1010 0000 0000 0010 0000 0000 0010 0000 1000 = 0 0 0 0 1 0 F A 0 0 0 2 0 0 2 0 0`方便阅读。
- 例如`int`类型在内存中占位四个字节：一般记录第一个字节的地址:首地址,在程序中我们常见的使用`&`来获取首地址
- 例子:打印某个内存地址

```c
#include <stdio.h>
int main()
{
    // 获取变量的内存地址
    int a = 10, b = 20;
    printf("%p\n", &a);
    printf("%p\n", &b);
    return 0;
}
```

```
000000042D1FF8CC
000000042D1FF8C8
```

- 数组的内存储存： 和之前的一样我们关注的地址也是第一个索引的元素所对应的首地址，通过首地址+偏移量=arr[1]来获取某个元素的地址

```c
int arr[] = {1,2,3};
printf("%p",&arr);//获取的是首地址（第一个元素，第一个各自的内存地址）
printf("%p",&arr[0]); // 获取的是第一个字符的首地址
printf("%p",&arr[1]);
printf("%p",&arr[2]);
```

- 几个问题

1. 通过变量的首地址，就可以确定变量中储存的数据了（错误的：通过首地址我们只能确定第一个字节的内容，但是我们并不知道后面字节的内容，同时也要根据不同的数据类型所占用的字节数量不同来确定到底有多少个字节。**因此在我们获得数据的时候要确定首地址+数据类型**）
2. 为什么数组的索引时从0开始的（数组的索引的0表示的是偏移量是0，后面的会根据偏移量的不同存在不同的索引）
3. 数组的长度是如何计算的呢？（注意：数组的长度和数组所占字节长度不同；$数组长度 = 总长度/数据类型占用的字节个数$）

数组的深拷贝

- 使用指针的自增来进行拷贝

```c
#include <stdio.h>

int main()
{
    int source[] = {2, 5, 8, 9, 13, 46, 55, 94, 102, 144, 561, 1024};
    int len = sizeof(source) / sizeof(source[0]);
    int target[len]; // 目标数组

    // 使用指针逐个复制
```

```
    int *src_ptr = source;
    int *tgt_ptr = target;
    for (int i = 0; i < len; i++)
    {
        *tgt_ptr++ = *src_ptr++;
    }

    // 打印目标数组以验证
    printf("Source array: ");
    for (int i = 0; i < len; i++)
    {
        printf("%d ", source[i]);
    }
    printf("\n");

    printf("Target array: ");
    for (int i = 0; i < len; i++)
    {
        printf("%d ", target[i]);
    }
    printf("\n");

    return 0;
}
```

- 在这个地方我们注意到`*tgt_ptr++ = *src_ptr++;`语句的使用，这句话表示的意思是，先进行`*tat_ptr = *src_ptr;`，然后再运行`tgt_ptr++,src_ptr++`。这两个语句这么运行的理由是`*`解引用符的运算时的优先级要高于`++`。

数组中的常见问题

1. 数组作为函数的形参

```
#include <stdio.h>
void printarr(int arr[]);
int main()
{
    int arr[] = {1, 2, 3, 4, 5};
    printf("%p\n", &arr);
    printarr(arr);
    return 0;
}
void printarr(int arr[])
{
    printf("%p\n", arr);
}
```

- 输出`000000955A7FFB80\n000000955A7FFB80`**可见数组作为函数的参数传递时候传递的是数组的首地址。**定义的地方`arr`表示的是完整的数组，函数中的`arr`只是一个变量，用来记录数组的首地址。
- 几个小知识

1. %p表示打印指针（地址）
2. &表示获取地址，第二次因为在数组作为参数传递的时候退化成为了地址，所以不需要&
3. sizeof()函数的返回值是size_t打印这个类型使用的是%zu,同时size_t这个类型也可以隐式转换为int

```
#include <stdio.h>
void printarr(int arr[]);
int main()
{
    int arr[] = {1, 2, 3, 4, 5};
    printf("%zu\n", sizeof(arr));
    printarr(arr);
    return 0;
}
void printarr(int arr[])
{
    printf("%zu\n", sizeof(arr));
}
```

- 可以发现第一个sizeof(arr) = 20第二个sizeof(arr) = 8指针（地址）的长度就是八位（64个二进制数每四个一组）
- 总结：**如果要在函数中对数组进行遍历的话一定要将数组的长度传递过去**

数组的算法题

```
// 寻找最大值
#include <stdio.h>
int find_max(int arr[], int arr_len);
int main()
{
    int arr[] = {33, 5, 22, 55, 44};
    int arr_len = sizeof(arr) / sizeof(arr[0]);
    printf("%d\n", find_max(arr, arr_len));
    return 0;
}
int find_max(int arr[], int arr_len)
{
    int max = arr[0];
    for (int i = 1; i <  arr_len; i++)
    {
        max = max >= arr[i] ? max : arr[i];
    }
    return max;
}
```

```
// 随机生成1~100的十个数，并且计算总数平均数以及低于平均数的个数
#include <stdio.h>
#include <time.h>
```

```c
#include <stdlib.h>
int main()
{
    int arr[10], sum = 0, ans = 0;
    double avr;
    srand(time(NULL));
    for (int i = 0; i < 10; i++)
    {
        arr[i] = rand() % 100 + 1;
        sum += arr[i];
        printf("arr[i] = %d\n", arr[i]);
    }
    avr = sum / 10.0;
    for (int j = 0; j < 10; j++)
    {
        arr[j] < avr ? ans++ : ans;
    }
    printf("sum = %d\navr = %.2f\nans = %d", sum, avr, ans);
    return 0;
}
```

```c
// 数据反转,也称为冒泡排序
#include <stdio.h>
int main()
{
    int arr[5] = {0, 0, 0, 0, 0};
    for (int i = 0; i < 5; i++)
    {
        scanf("%d", &arr[i]);
    }
    for (int i = 0; i < 5; i++)
    {
        int temp = arr[i], j = 4 - i;
        arr[i] = arr[j];
        arr[j] = temp;
        if (i == j)
        {
            break;
        }
    }
    for (int i = 0; i < 5; i++)
    {
        printf("%d\n", arr[i]);
    }
    return 0;
}
```

```c
//打乱数组中的数据:方法先定义一个数组然后将每一个元素都和随机位置的元素进行交换位置
#include <stdio.h>
```

```c
#include <stdlib.h>
#include <time.h>
int main()
{
    int arr[] = {1, 2, 3, 4, 5};
    int len = sizeof(arr) / sizeof(int);
    srand(time(NULL));
    for (int i = 0; i < len; i++)
    {
        int index = rand() % len;
        int temp = arr[i];
        arr[i] = arr[index];
        arr[index] = temp;
    }
    for (int i = 0; i < len; i++)
    {
        printf("%d\n", arr[i]);
    }
}
/*
    小思考：这个代码的结果会不会出现重复的情况?
    答案：不会，虽然index有可能出现重复的情况，但是每一次进行的操作都是交换而不是与index有关的赋值
    所以不会对于最后的结果导致重复的出现
*/
```

- 数组的常见算法：

1. 基本查找
2. 二分查找

- 基本前提：数据一定要有序摆放（从小到大或从大到小）
- 核心逻辑：每次排除一半的查找范围

```c
// 代码实现
#include <stdio.h>

int main()
{
    int target = 149, times = 1;
    int arr[10000] = {0};
    for (int i = 0; i < 10000; i++)
    {
        arr[i] = i;
    }
    int len = sizeof(arr) / sizeof(int);
    int min = 0, max = len - 1, mid;

    printf("The %d times guess min = %d, max = %d, mid = %d\n", times, min, max, arr[mid]);
```

```
    while (min <= max)
    {
        mid = min + (max - min) / 2;
        printf("The %d times guess min = %d, max = %d, mid = %d\n", times, min,
max, arr[mid]);
        if (arr[mid] == target)
        {
            break;
        }
        else if (arr[mid] > target)
        {
            max = mid - 1;
        }
        else
        {
            min = mid + 1;
        }
        times++;
    }

    if (min <= max)
    {
        printf("Target %d found in %d times.\n", target, times);
    }
    else
    {
        printf("Target %d not found.\n", target);
    }

    return 0;
}
```

- 改进(前提是数据要均匀排布)：$$mid = min +\frac{key-arr[min]}{arr[max]-arr[min]}\times(max-min)$$例如`arr[] = {1,2,3,4,5,6,7,8,9,10}`而我要找的数字是3则会有$mid = 0 + \frac{3-1}{10-1}\times(9-0)$

3. 冒泡排序 冒泡排序（Bubble Sort）是一种简单的排序算法，它通过重复地遍历要排序的列表，依次比较相邻的两个元素，并根据需要交换它们的位置，直到整个列表有序。

- 基本思想 冒泡排序的核心思想是通过不断交换相邻的元素将较大的元素逐渐“浮”到列表的末尾，而较小的元素则“沉”到列表的开头。每进行一次完整的遍历，列表中最大的元素就会“冒泡”到它的最终位置。

- 算法步骤

    1. **比较相邻元素**：从列表的第一个元素开始，比较相邻的两个元素。
    2. **交换位置**：如果前一个元素大于后一个元素，则交换它们的位置。
    3. **重复遍历**：重复上述过程，直到整个列表有序。每次遍历后，最后一个未排序的元素就是当前最大的元素，无需再参与下一次比较。
    4. **终止条件**：如果一次遍历中没有发生任何交换，说明列表已经有序，算法可以提前终止。
    - 示例

    假设我们要对列表 `[5, 3, 8, 4, 6]` 进行冒泡排序：

1. **第一轮遍历**：
   - 比较 5 和 3，交换后列表变为 `[3, 5, 8, 4, 6]`
   - 比较 5 和 8，无需交换
   - 比较 8 和 4，交换后列表变为 `[3, 5, 4, 8, 6]`
   - 比较 8 和 6，交换后列表变为 `[3, 5, 4, 6, 8]`
   - 第一轮结束后，最大的元素 8 已经“冒泡”到列表末尾。
2. **第二轮遍历**：
   - 比较 3 和 5，无需交换
   - 比较 5 和 4，交换后列表变为 `[3, 4, 5, 6, 8]`
   - 比较 5 和 6，无需交换
   - 第二轮结束后，第二大的元素 6 已经“冒泡”到倒数第二的位置。
3. **第三轮遍历**：
   - 比较 3 和 4，无需交换
   - 比较 4 和 5，无需交换
   - 第三轮结束后，第三大的元素 5 已经“冒泡”到倒数第三的位置。
4. **第四轮遍历**：
   - 比较 3 和 4，无需交换
   - 第四轮结束后，第四大的元素 4 已经“冒泡”到倒数第四的位置。

此时，列表已经完全有序，结果为 `[3, 4, 5, 6, 8]`。

- 时间复杂度
  - **最佳情况**：当列表已经有序时，冒泡排序只需要进行一次遍历，时间复杂度为 **O(n)**。
  - **最坏情况和平均情况**：当列表完全逆序时，冒泡排序需要进行 **n(n-1)/2** 次比较和交换，时间复杂度为 **O(n^2)**。
- 空间复杂度

冒泡排序是原地排序算法，只需要常数级别的额外空间，空间复杂度为 **O(1)**。

- 优缺点
  - **优点**：
    - 实现简单，容易理解。
    - 原地排序，不需要额外的存储空间。
  - **缺点**：
    - 效率较低，时间复杂度为 O(n^2)，不适合处理大规模数据。

冒泡排序虽然效率不高，但由于其简单性，常用于教学或小规模数据的排序。在实际应用中，更高效的排序算法（如快速排序、归并排序等）更为常用。

```c
#include <stdio.h>

void bubbleSort(int arr[], int n) //在C中将一个数组传递进去就是将他的地址传进去了所以在内部的修改也是会导致外部的函数变动的（python的《那些年我们踩过的坑》）
{
    int i, j, temp;
    int swapped; // 用于优化的标志变量
    for (i = 0; i < n - 1; i++)
```

```
    { // 外层循环控制轮数
        swapped = 0; // 每轮开始时，假设数组已经有序
        for (j = 0; j < n - 1 - i; j++)
        { // 内层循环进行相邻元素比较
            if (arr[j] > arr[j + 1])
            { // 如果逆序
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                swapped = 1; // 发生了交换，说明数组可能仍然无序
            }
        }
        if (swapped == 0) // 如果这一轮没有发生交换，提前结束排序
            break;
    }
}

int main()
{
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr) / sizeof(arr[0]);
    bubbleSort(arr, n);
    printf("Sorted array: \n");
    for (int i = 0; i < n; i++)
    {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

枚举enum

- 在C语言中，枚举（Enumeration）是一种用户自定义的数据类型，用于定义一组具有固定值的常量。枚举类型可以提高代码的可读性和可维护性，因为它允许你为整数值赋予有意义的名称。

1. 枚举的定义

- 枚举类型通过关键字 enum 定义。其基本语法如下：

```
enum 枚举名 {
    枚举值1,
    枚举值2,
    枚举值3,
    ...
};
```

- 枚举名：是用户自定义的类型名称。
- 枚举值：是一组整型常量，每个枚举值都有一个唯一的名称。

2. 默认值

- 如果没有显式指定枚举值的整数值，C语言会自动为枚举值分配整数值。默认情况下，第一个枚举值的值为 0，后续的枚举值依次加 1。

```
enum Weekday {
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY,
    SUNDAY
};
```

- 在这个例子中：
  - MONDAY 的值为 0
  - TUESDAY 的值为 1
  - WEDNESDAY 的值为 2 ...
  - SUNDAY 的值为 6

3. 显式指定值

- 你可以为枚举值显式指定整数值。如果指定了某个值，后续的枚举值会从该值开始依次加 1。 示例：

```
enum Weekday {
    MONDAY = 1,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY,
    SUNDAY
};
```

- 在这个例子中：
  - MONDAY 的值为 1
  - TUESDAY 的值为 2
  - WEDNESDAY 的值为 3 ...
  - SUNDAY 的值为 7

4. 枚举变量的声明

- 定义枚举类型后，可以声明该类型的变量，并为其赋值。

```
#include <stdio.h>

enum Weekday {
    MONDAY,
```

```
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY,
    SUNDAY
};

int main() {
    enum Weekday today = WEDNESDAY;

    if (today == WEDNESDAY) {
        printf("Today is Wednesday.\n");
    }

    return 0;
}
```

5. 枚举的用途

- 枚举类型的主要用途是提高代码的可读性和可维护性。通过使用有意义的名称代替整数值，可以避免硬编码，使代码更容易理解和维护。

```
enum Color {
    RED,
    GREEN,
    BLUE
};

void printColor(enum Color c) {
    switch (c) {
        case RED:
            printf("Color is Red.\n");
            break;
        case GREEN:
            printf("Color is Green.\n");
            break;
        case BLUE:
            printf("Color is Blue.\n");
            break;
    }
}

int main() {
    enum Color myColor = GREEN;
    printColor(myColor);
    return 0;
}
```

6. 匿名枚举 你也可以定义匿名枚举，即不指定枚举名。匿名枚举的枚举值可以直接使用。 示例：

```c
enum {
    RED = 1,
    GREEN,
    BLUE
};

int main() {
    int color = GREEN;
    printf("Color value: %d\n", color);
    return 0;
}
```

7. 枚举的注意事项

- 枚举值本质上是整数，因此可以参与整数运算。