

第6讲：函数

目录

1. 函数的概念
 2. 库函数
 3. 自定义函数
 4. 形参和实参
 5. return语句
 6. 数组做函数参数
 7. 嵌套调用和链式访问
 8. 函数的声明和定义
-

正文开始

1. 函数的概念

数学中我们其实就见过函数的概念，比如：一次函数 $y = kx + b$ ， k 和 b 都是常数，给一个任意的 x ，就得到一个 y 值。

其实在C语言也引入**函数（function）**的概念，有些翻译为：**子程序**，子程序这种翻译更加准确一些。**C语言中的函数就是一个完成某项特定的任务的一小段代码**。这段代码是有特殊的写法和调用方法的。

C语言的程序其实是由无数个小的函数组合而成的，也可以说：一个大的计算任务可以分解成若干个较小的函数（对应较小的任务）完成。同时一个函数如果能完成某项特定任务的话，这个函数也是可以复用的，提升了开发软件的效率。

在C语言中我们一般会见到两类函数：

- 库函数
- 自定义函数

2. 库函数

2.1 标准库和头文件

C语言标准中规定了C语言的各种语法规则，C语言并不提供库函数；C语言的国际标准ANSI C规定了一些常用的函数的标准，被称为标准库，那不同的编译器厂商根据ANSI提供的C语言标准就给出了一系列函数的实现。这些函数就被称为**库函数**。

我们前面内容中学到的 `printf`、`scanf` 都是库函数，库函数也是函数，不过这些函数已经是现成的，我们只要学会就能直接使用了。有了库函数，一些常见的功能就不需要程序员自己实现了，一定程度提升了效率；同时库函数的质量和执行效率上都更有保证。

各种编译器的标准库中提供了一系列的库函数，这些库函数根据功能的划分，都在不同的头文件中进行了声明。

库函数相关头文件：<https://zh.cppreference.com/w/c/header>

有数学相关的，有字符串相关的，有日期相关的等，每一个头文件中都包含了，相关的函数和类型等信息，库函数的学习不用着急一次性全部学会，慢慢学习，各个击破就行。

2.2 库函数的使用方法

库函数的学习和查看工具很多，比如：

C/C++官方的链接：<https://zh.cppreference.com/w/c/header>

中文版：<https://cppreference.cn/w/>

cplusplus.com：<https://legacycplusplus.com/reference/clibrary/>

举例：`sqrt`

代码块

```
1 double sqrt (double x);
2 //sqrt 是函数名
3 //x 是函数的参数，表示调用sqrt函数需要传递一个double类型的值
4 //double 是返回值类型 - 表示函数计算的结果是double类型的值
```

2.2.1 功能

Compute square root 计算平方根

Returns the *square root* of *x*. (返回平方根)

2.2.2 头文件包含

库函数是在标准库中对应的头文件中声明的，所以库函数的使用，务必包含对应的头文件，不包含是可能会出现一些问题的。

2.2.3 实践

代码块

```
1 #include <stdio.h>
2 #include <math.h>
3
4 int main()
5 {
6     double d = 16.0;
7     double r = sqrt(d);
8     printf("%lf\n", r);
9     return 0;
10 }
```

运行结果：

```
Microsoft Visual Studio 调试控制台
4. 000000
D:\code\2022\test\课件代码测试\Debug\课件代码测试.exe (进程 69144) 已退出，代码为 0。
按任意键关闭此窗口. . .
```

2.2.4 库函数文档的一般格式

1. 函数原型
2. 函数功能介绍
3. 参数和返回类型说明
4. 代码举例
5. 代码输出
6. 相关知识链接

3. 自定义函数

了解了库函数，我们的关注度应该聚焦在自定义函数上，自定义函数其实更加重要，也能给程序员写代码更多的创造性。

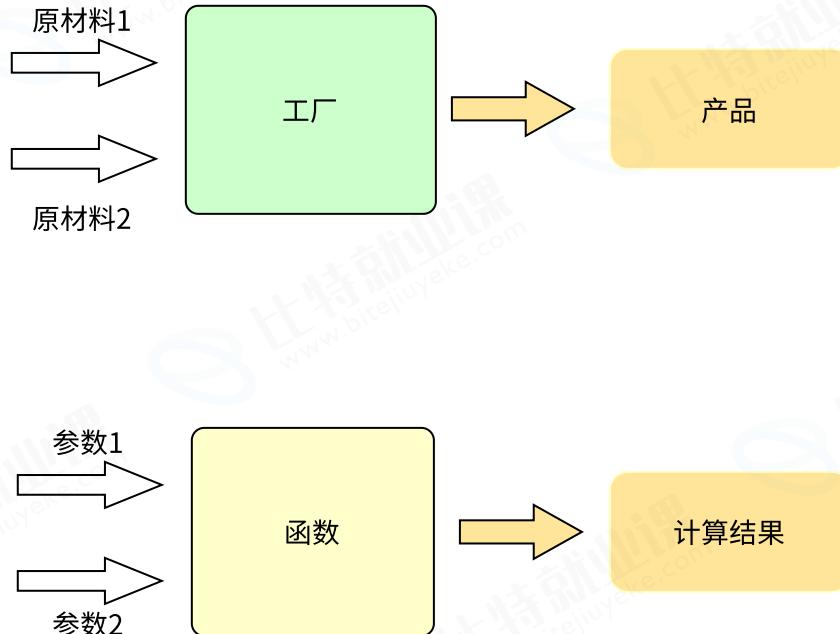
3.1 函数的语法形式

其实自定义函数和库函数是一样的，形式如下：

代码块

```
1 ret_type fun_name(形式参数)
2 {
3
4 }
```

- `ret_type` 是函数返回类型
- `fun_name` 是函数名
- 括号中放的是形式参数
- {}括起来的是**函数体**



我们可以把函数想象成小型的一个加工厂，工厂得输入原材料，经过工厂加工才能生产出产品，那函数也是一样的，函数一般会输入一些值（可以是0个，也可以是多个），经过函数内的计算，得出结果。

- `ret_type` 是用来表示函数计算结果的类型，有时候返回类型可以是 `void`，表示什么都不返回
- `fun_name` 是为了方便使用函数；就像人的名字一样，有了名字方便称呼，函数有了名字方便调用，所以函数名尽量要根据函数的功能起的有意义。
- 函数的参数就相当于，工厂中送进去的原材料，函数的参数也可以是 `void`，明确表示函数没有参数。如果有参数，要交代清楚参数的类型和名字，以及参数个数。
- {}括起来的部分被称为**函数体**，函数体就是完成计算的过程。

3.2 函数的举例

举个例子：

写一个加法函数，完成2个整型变量的加法操作。

代码块

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int a = 0;
6     int b = 0;
7     //输入
8     scanf("%d %d", &a, &b);
9     //调用加法函数，完成a和b的相加
10    //求和的结果放在r中
11    //to do
12
13    //输出
14    printf("%d\n", r);
15    return 0;
16 }
```

我们根据要完成的功能，给函数取名：Add，函数Add需要接收2个整型类型的参数，函数计算的结果也是整型。

所以我们根据上述的分析写出函数：

代码块

```
1 #include <stdio.h>
2 int Add(int x, int y)
3 {
4     int z = 0;
5     z = x + y;
6     return z;
7 }
8
9 int main()
10 {
11     int a = 0;
12     int b = 0;
13     //输入
14     scanf("%d %d", &a, &b);
15     //调用加法函数，完成a和b的相加
16     //求和的结果放在r中
17     int r = Add(a, b);
18     //输出
19     printf("%d\n", r);
```

```
20     return 0;
21 }
```

Add函数也可以简化为：

代码块

```
1 int Add(int x, int y)
2 {
3     return x + y;
4 }
```

函数的参数部分需要交代清楚：参数个数，每个参数的类型是啥，形参的名字叫啥。

上面只是一个例子，未来我们是根据实际需要来设计函数，函数名、参数、返回类型都是可以灵活变化的。

4. 形参和实参

在函数使用的过程中，把函数的参数分为，实参和形参。

再看看我们前面写的代码：

代码块

```
1 #include <stdio.h>
2 int Add(int x, int y)
3 {
4     int z = 0;
5     z = x + y;
6     return z;
7 }
8
9 int main()
10 {
11     int a = 0;
12     int b = 0;
13     //输入
14     scanf("%d %d", &a, &b);
15     //调用加法函数，完成a和b的相加
16     //求和的结果放在r中
17     int r = Add(a, b);
18     //输出
19     printf("%d\n", r);
20 }
```

4.1 实参

在上面代码中，第2~7行是 Add 函数的定义，有了函数后，再第17行调用Add函数的。

我们把第17行调用Add函数时，传递给函数的参数a和b，称为**实际参数**，简称**实参**。

实际参数就是真实传递给函数的参数。

4.2 形参

在上面代码中，第2行定义函数的时候，在函数名 Add 后的括号中写的 x 和 y，称为**形式参数**，简称**形参**。

为什么叫形式参数呢？实际上，如果只是定义了 Add 函数，而不去调用的话，Add 函数的参数 x 和 y 只是形式上存在的，不会向内存申请空间，不会真实存在的，所以叫形式参数。形式参数只有在函数被调用的过程中为了存放实参传递过来的值，才向内存申请空间，这个过程就是**形参的实例化**。

4.3 实参和形参的关系

虽然我们提到了实参是传递给形参的，他们之间是有联系的，但是形参和实参各自是独立的内存空间。

这个现象是可以通过调试来观察的。请看下面的代码和调试演示：

代码块

```

1 #include <stdio.h>
2 int Add(int x, int y)
3 {
4     int z = 0;
5     z = x + y;
6     return z;
7 }
8
9 int main()
10 {
11     int a = 0;
12     int b = 0;
13     //输入
14     scanf("%d %d", &a, &b);
15     //调用加法函数，完成a和b的相加
16     //求和的结果放在r中
17     int r = Add(a, b);
18     //输出

```

```
19     printf("%d\n", r);
20
21 }
```

名称	值
a	3
b	5
&a	0x0053fb00 {3}
&b	0x0053faf4 {5}
x	3
y	5
&x	0x0053fa10 {3}
&y	0x0053fa14 {5}

我们在调试的时候可以观察到，`x`和`y`确实得到了`a`和`b`的值，但是`x`和`y`的地址和`a`和`b`的地址是不一样的，所以我们可以理解为形参是实参的一份临时拷贝。

5. return 语句

在函数的设计中，函数中经常会出现`return`语句，这里讲一下`return`语句使用的注意事项。

- `return`后边可以是一个数值，也可以是一个表达式，如果是表达式则先执行表达式，再返回表达式的结果。
- `return`后边也可以什么都没有，直接写 `return;` 这种写法适合函数返回类型是`void`的情况。
- `return`语句执行后，函数就彻底返回，后边的代码不再执行。
- `return`返回的值和函数返回类型不一致，系统会自动将返回的值隐式转换为函数的返回类型。
- 如果函数中存在`if`等分支的语句，则要保证每种情况下都有`return`返回，否则会出现编译错误。
- 函数的返回类型如果不写，编译器会默认函数的返回类型是`int`。
- 函数写了返回类型，但是函数中没有使用`return`返回值，那么函数的返回值是未知的。

6. 数组做函数参数

在使用函数解决问题的时候，难免会将数组作为参数传递给函数，在函数内部对数组进行操作。

比如：写一个函数将一个整型数组的内容，全部置为-1，再写一个函数打印数组的内容。

简单思考一下，基本的形式应该是这样的：

代码块

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int arr[] = {1,2,3,4,5,6,7,8,9,10};
6     set_arr(); //设置数组内容为-1
7     print_arr(); //打印数组内容
8     return 0;
9 }
```

这里的set_arr函数要能够对数组内容进行设置，就得把数组作为参数传递给函数，同时函数内部在设置数组每个元素的时候，也得遍历数组，需要知道数组的元素个数。所以我们需要给set_arr传递2个参数，一个是数组，另外一个是数组的元素个数。仔细分析print_arr也是一样的，只有拿到了数组和元素个数，才能遍历打印数组的每个元素。

代码块

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int arr[] = {1,2,3,4,5,6,7,8,9,10};
6     int sz = sizeof(arr)/sizeof(arr[0]);
7     set_arr(arr, sz); //设置数组内容为-1
8     print_arr(arr, sz); //打印数组内容
9     return 0;
10 }
```

数组作为参数传递给了set_arr 和 print_arr 函数了，那这两个函数应该如何设计呢？

这里我们需要知道数组传参的几个重点知识：

- 函数的形式参数要和函数的实参数个数匹配
- 函数的实参是数组，形参也是可以写成数组形式的
- 形参如果是一维数组，数组大小可以省略不写
- 形参如果是二维数组，行可以省略，但是列不能省略
- 数组传参，形参是不会创建新的数组的
- 形参操作的数组和实参的数组是同一个数组

根据上述的信息，我们就可以实现这两个函数：

代码块

```
1 void set_arr(int arr[], int sz)
2 {
3     int i = 0;
4     for(i=0; i<sz; i++)
5     {
6         arr[i] = -1;
7     }
8 }
9
10 void print_arr(int arr[], int sz)
11 {
12     int i = 0;
13     for(i=0; i<sz; i++)
14     {
15         printf("%d ", arr[i]);
16     }
17     printf("\n");
18 }
```

7. 嵌套调用和链式访问

7.1 嵌套调用

嵌套调用就是函数之间的互相调用，每个函数就像一个乐高零件，正是因为多个乐高的零件互相无缝的配合才能搭建出精美的乐高玩具，也正是因为函数之间有效的互相调用，最后写出来了相对大型的程序。

假设我们计算某年某月有多少天？如果要函数实现，可以设计2个函数：

- `is_leap_year()`: 根据年份确定是否是闰年
- `get_days_of_month()`: 调用`is_leap_year`确定是否是闰年后，再根据月计算这个月的天数

代码块

```
1 int is_leap_year(int y)
2 {
3     if(((y%4==0)&&(y%100!=0))||((y%400==0)))
4         return 1;
5     else
6         return 0;
7 }
```

```

8
9 int get_days_of_month(int y, int m)
10 {
11     int days[] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
12     int day = days[m];
13     if (is_leap_year(y) && m == 2)
14         day += 1;
15
16     return day;
17 }
18
19 int main()
20 {
21     int y = 0;
22     int m = 0;
23     scanf("%d %d", &y, &m);
24     int d = get_days_of_month(y, m);
25     printf("%d\n", d);
26     return 0;
27 }
```

这一段代码，完成了一个独立的功能。代码中反应了不少的函数调用：

- `main` 函数调用 `scanf`、`printf`、`get_days_of_month`
- `get_days_of_month` 函数调用 `is_leap_year`

未来的稍微大一些代码都是函数之间的嵌套调用，但是函数是不能嵌套定义的。

7.2 链式访问

所谓链式访问就是将一个函数的返回值作为另外一个函数的参数，像链条一样将函数串起来就是函数的链式访问。

比如：

代码块

```

1 #include <stdio.h>
2
3 int main()
4 {
5     int len = strlen("abcdef");//1.strlen求一个字符串的长度
6     printf("%d\n", len);//2.打印长度
7     return 0;
8 }
```

前面的代码完成动作写了2条语句，把如果把strlen的返回值直接作为printf函数的参数呢？这样就是一个链式访问的例子了。

代码块

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("%d\n", strlen("abcdef")); //链式访问
6     return 0;
7 }
```

在看一个有趣的代码，下面代码执行的结果是什么呢？

代码块

```
1 #include <stdio.h>
2 int main()
3 {
4     printf("%d", printf("%d", printf("%d", 43)));
5     return 0;
6 }
```

这个代码的关键是明白 printf 函数的返回是啥？

代码块

```
1 int printf ( const char * format, ... );
```

Return Value

On success, the total number of characters written is returned.

If a writing error occurs, the error indicator (`ferror`) is set and a negative number is returned.

If a multibyte character encoding error occurs while writing wide characters, `errno` is set to `EILSEQ` and a negative number is returned.

printf函数返回的是打印在屏幕上的字符的个数。

上面的例子中，我们就第一个printf打印的是第二个printf的返回值，第二个printf打印的是第三个printf的返回值。

第三个printf打印43，在屏幕上打印2个字符，再返回2

第二个printf打印2，在屏幕上打印1个字符，再放回1

第一个printf打印1

所以屏幕上最终打印：4321

8. 函数的声明和定义

8.1 单个文件

一般我们在使用函数的时候，直接将函数写出来就使用了。

比如：我们要写一个函数判断一年是否是闰年。

代码块

```
1 #include <stido.h>
2
3 //判断一年是不是闰年
4 int is_leap_year(int y)
5 {
6     if(((y%4==0)&&(y%100!=0)) || (y%400==0))
7         return 1;
8     else
9         return 0;
10 }
11
12 int main()
13 {
14     int y = 0;
15     scanf("%d", &y);
16     int r = is_leap_year(y);
17     if(r == 1)
18         printf("闰年\n");
19     else
20         printf("非闰年\n");
21     return 0;
22 }
```

上面代码中橙色的部分是**函数的定义**，绿色的部分是**函数的调用**。

这种场景下是函数的定义在函数调用之前，没啥问题。

那如果我们将函数的定义放在函数的调用后边，如下：

代码块

```
1 #include <stido.h>
```

```

2
3 int main()
4 {
5     int y = 0;
6     scanf("%d", &y);
7     int r = is_leap_year(y);
8     if(r == 1)
9         printf("闰年\n");
10    else
11        printf("非闰年\n");
12    return 0;
13 }
14
15 //判断一年是不是闰年
16 int is_leap_year(int y)
17 {
18     if(((y%4==0)&&(y%100!=0)) || (y%400==0))
19         return 1;
20     else
21         return 0;
22 }

```

这个代码在VS2022上编译，会出现下面的警告信息：

输出

显示输出来源(S): 生成

已启动生成...

1>----- 已启动生成: 项目: 课件代码测试, 配置: Debug Win32 -----
1>test.c
1>D:\code\2022\test\课件代码测试\课件代码测试\test.c(9,25): warning C4013: “is_leap_year” 未定义; 假设外部返回 int
1>已完成生成项目“课件代码测试.vcxproj”的操作。
===== 生成: 成功 1 个, 失败 0 个, 最新 0 个, 跳过 0 个 =====

这是因为C语言编译器对源代码进行编译的时候，从第一行往下扫描的，当遇到第7行的is_leap_year函数调用的时候，并没有发现前面有is_leap_year的定义，就报出了上述的警告。

把怎么解决这个问题呢？就是函数调用之前先声明一下is_leap_year这个函数，声明函数只要交代清楚：函数名，函数的返回类型和函数的参数。

如：`int is_leap_year(int y);` 这就是**函数声明**，函数声明中参数只保留类型，省略掉名字也是可以的。

代码变成这样就能正常编译了。

代码块

```

1 #include <stdio.h>
2
3 int is_leap_year(int y); //函数声明

```

```

4
5 int main()
6 {
7     int y = 0;
8     scanf("%d", &y);
9     int r = is_leap_year(y);
10    if(r == 1)
11        printf("闰年\n");
12    else
13        printf("非闰年\n");
14    return 0;
15 }
16
17 //判断一年是不是闰年
18 int is_leap_year(int y)
19 {
20     if(((y%4==0)&&(y%100!=0)) || (y%400==0))
21         return 1;
22     else
23         return 0;
24 }
```

函数的调用一定要满足，先声明后使用；

函数的定义也是一种**特殊的声明**，所以如果函数定义放在调用之前也是可以的。

8.2 多个文件

一般在企业中我们写代码时候，代码可能比较多，不会将所有的代码都放在一个文件中；我们往往回根据程序的功能，将代码拆分放在多个文件中。

一般情况下，函数的声明、类型的声明放在头文件（.h）中，函数的实现是放在源文件（.c）文件中。

如下：

add.c

代码块

```

1 //函数的定义
2 int Add(int x, int y)
3 {
4     return x+y;
5 }
```

add.h

代码块

```
1 //函数的声明  
2 int Add(int x, int y);
```

test.c

代码块

```
1 #include <stdio.h>  
2 #include "add.h"  
3  
4 int main()  
5 {  
6     int a = 10;  
7     int b = 20;  
8     //函数调用  
9     int c = Add(a, b);  
10    printf("%d\n", c);  
11    return 0;  
12 }
```

运行结果：

```
解决方... | add.h | x86 | 本地 Windows 调试器 | (全局范围) | Add(int x, int y)  
头文件 | add.h | 未找到相关问题  
源文件 | add.c | 未找到相关问题  
资源文件 | test.c | 未找到相关问题  
//函数的声明  
int Add(int x, int y);  
//函数的定义  
int Add(int x, int y)  
{  
    return x + y;  
}  
#include <stdio.h>  
#include "add.h"  
int main()  
{  
    int a = 10;  
    int b = 20;  
    //函数调用  
    int c = Add(a, b);  
    printf("%d\n", c);  
    return 0;  
}
```

microsoft Visual Studio 调试控制台
30
D:\code\2022\test\课件代码测试\Debug\课件代码测试.exe
按任意键关闭此窗口. . .

有了函数声明和函数定义的理解，我们写代码就更加方便了。

8.3 static 和 extern

`static` 和 `extern` 都是C语言中的关键字。

`static` 是 **静态的** 的意思，可以用来：

- 修饰局部变量
- 修饰全局变量
- 修饰函数

`extern` 是用来**声明**外部符号的。

在讲解 `static` 和 `extern` 之前再讲一下：作用域和生命周期。

作用域 (scope) 是程序设计概念，通常来说，一段程序代码中所用到的名字并不总是有效（可用的，而限定这个名字的可用性的代码范围就是这个名字的作用域。

1. 局部变量的作用域是变量所在的局部范围。
2. 全局变量的作用域是整个工程（项目）。

生命周期指的是变量的创建(申请内存)到变量的销毁(收回内存)之间的一个时间段。

1. 局部变量的生命周期是：进入作用域变量创建，生命周期开始，出作用域生命周期结束。
2. 全局变量的生命周期是：整个程序的生命周期。

8.3.1 static 修饰局部变量

代码块

```
1 //代码1
2 #include <stdio.h>
3 void test()
4 {
5     int i = 0;
6     i++;
7     printf("%d ", i);
8 }
9 int main()
10 {
11     int i = 0;
12     for(i=0; i<5; i++)
13     {
14         test();
15     }
16     return 0;
```

代码块

```
1 //代码2
2 #include <stdio.h>
3 void test()
4 {
5     //static修饰局部变量
6     static int i = 0;
7     i++;
8     printf("%d ", i);
9 }
10 int main()
11 {
12     int i = 0;
13     for(i=0; i<5; i++)
14     {
15         test();
16     }
```

```
17 }
```

```
17 return 0;
```

```
18 }
```

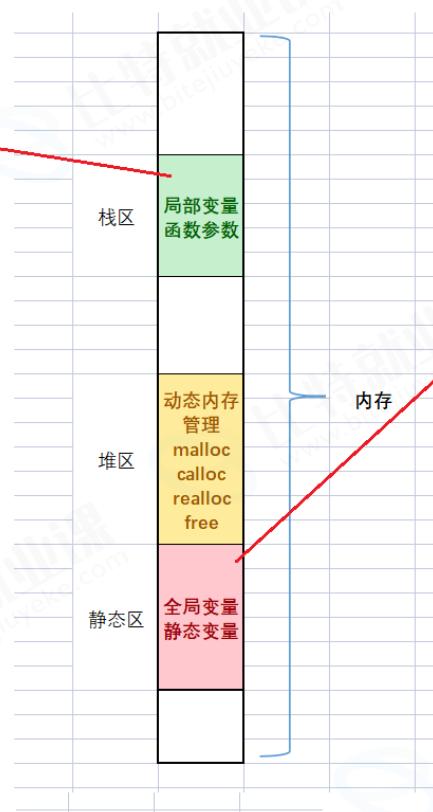
对比代码1和代码2的效果，理解 `static` 修饰局部变量的意义。

代码1的test函数中的局部变量i是每次进入test函数先创建变量（生命周期开始）并赋值为0，然后`++`，再打印，出函数的时候变量生命周期将要结束（释放内存）。

代码2中，我们从输出结果来看，i的值有累加的效果，其实 test函数中的i创建好后，出函数的时候是不会销毁的，重新进入函数也就不会重新创建变量，直接上次累积的数值继续计算。

结论： `static`修饰局部变量改变了变量的生命周期，生命周期改变的本质是改变了变量的存储类型，本来一个局部变量是存储在内存的栈区的，但是被 `static` 修饰后存储到了静态区。存储在静态区的变量和全局变量是一样的，生命周期就和程序的生命周期一样了，只有程序结束，变量才销毁，内存才回收。但是作用域不变的。

```
1 //代码1
2 #include <stdio.h>
3 void test()
4 {
5     int i = 0;
6     i++;
7     printf("%d ", i);
8 }
9 int main()
10 {
11     int i = 0;
12     for(i=0; i<5; i++)
13     {
14         test();
15     }
16     return 0;
17 }
```



```
1 //代码2
2 #include <stdio.h>
3 void test()
4 {
5     //static修饰局部变量
6     static int i = 0;
7     i++;
8     printf("%d ", i);
9 }
10 int main()
11 {
12     int i = 0;
13     for(i=0; i<5; i++)
14     {
15         test();
16     }
17     return 0;
18 }
```

使用建议：未来一个变量出了函数后，我们还想保留值，等下次进入函数继续使用，就可以使用 `static` 修饰。

8.3.2 static 修饰全局变量

代码1

add.c

代码2

add.c

代码块

```
1 int g_val = 2018;
```

test.c

代码块

```
1 #include <stdio.h>
2 extern int g_val;
3 int main()
4 {
5     printf("%d\n", g_val);
6     return 0;
7 }
```

代码块

```
1 static int g_val = 2018;
```

test.c

代码块

```
1 #include <stdio.h>
2 extern int g_val;
3 int main()
4 {
5     printf("%d\n", g_val);
6     return 0;
7 }
```

extern 是用来声明外部符号的，如果一个全局的符号在A文件中定义的，在B文件中想使用，就可以使用 **extern** 进行声明，然后使用。

代码1正常，代码2在编译的时候会出现链接性错误。

结论：

一个全局变量被**static**修饰，使得这个全局变量只能在本源文件内使用，不能在其他源文件内使用。

本质原因是全局变量默认是具有外部链接属性的，在外部的文件中想使用，只要适当的声明就可以使用；但是全局变量被 **static** 修饰之后，外部链接属性就变成了内部链接属性，只能在自己所在的源文件内部使用了，其他源文件，即使声明了，也是无法正常使用的。

使用建议：如果一个全局变量，只想在所在的源文件内部使用，不想被其他文件发现，就可以使用 **static** 修饰。

8.3.3 static 修饰函数

代码1

add.c

代码块

```
1 int Add(int x, int y)
2 {
3     return x+y;
4 }
```

代码2

add.c

代码块

```
1 static int Add(int x, int y)
2 {
3     return x+y;
4 }
```

test.c

代码块

```
1 #include <stdio.h>
2 extern int Add(int x, int y);
3 int main()
4 {
5     printf("%d\n", Add(2, 3));
6     return 0;
7 }
```

test.c

代码块

```
1 #include <stdio.h>
2 extern int Add(int x, int y);
3 int main()
4 {
5     printf("%d\n", Add(2, 3));
6     return 0;
7 }
```

代码1是能够正常运行的，但是代码2就出现了链接错误。

其实 `static` 修饰函数和 `static` 修饰全局变量是一模一样的，一个函数在整个工程都可以使用，被 `static` 修饰后，只能在本文件内部使用，其他文件无法正常的链接使用了。

本质是因为函数默认是具有外部链接属性，具有外部链接属性，使得函数在整个工程中只要适当的声明就可以被使用。但是被 `static` 修饰后变成了内部链接属性，使得函数只能在自己所在源文件内部使用。

使用建议：一个函数只想在所在的源文件内部使用，不想被其他源文件使用，就可以使用 `static` 修饰。

完