

9 函数

函数是构成C语言程序的基本单位，可以说C语言的全部工作都是由各种各样的函数完成的。由于采用了函数模块式的结构，C语言易于实现结构化程序设计。使得程序的层次结构清晰，便于程序的编写、阅读和调试。

9.0 概述

函数的概念来自数学，C语言中的函数与数学中的函数本质上是相同的：给定若干个输出，可以得到一个确定的输出。但C语言在函数形式和功能上更为灵活。通常，在C语言中，函数被设计用来完成某一项专门的任务。

C语言源程序是由函数组成的。实用程序往往由多个函数组成。在设计一个复杂的程序时，往往会把整个程序划分为若干个功能较为单一的模块，然后分别予以实现，最后再把所有模块装配起来。这种程序设计中分而治之的策略，称为模块化程序设计方法。

```
# include<stdio.h>
void PrintStars();           // 函数原型声明
void PrintMsg(char msg[]);   // 函数原型声明

int main() {
    PrintStars();             // 函数调用
    PrintMsg("Hello,world!"); // 函数调用
    PrintStars();             // 函数调用
    /* 函数运行结果:
    *****
    Hello,world!
    *****
    */
    return 0;
}

void PrintStars() {           // 函数定义
    // 函数功能：输出一行星号
    printf("*****\n");
}

void PrintMsg(char msg[]) {   // 函数定义
    // 函数功能：输出一条信息
    puts(msg);
}
```

上面的例子中一共出现了5个函数，下面予以说明：

- 一个源程序文件由若干函数组成。编译器以文件为单位进行编译，而不是以函数为单位进行编译。无论源程序中含有多少个函数，最后都只生成一个目标文件。
- 一个完整的函数由函数头部和函数体组成。函数头部包括函数名、函数参数和函数返回值，函数体中就是函数的实现代码。函数的实现也被称为函数定义。
- C程序的执行从 `main()` 函数开始，调用执行其他函数后流程回到 `main()` 函数，正常情况下，会在这里结束整个函数的运行。如果希望C程序是一个可单独执行的程序，则必须有且只有一个 `main()` 函数。
- `printf()` 等函数属于系统定义好的库函数，存储在头文件 `stdio.h` 中，只需要用 `#include` 语句导入即可。

- 用户定义函数可以调用库函数，也可以调用在此函数前声明过的函数，即允许函数嵌套调用。

9.1 函数声明

在使用变量前，需要先对变量进行声明，这样做主要是告诉编译器，某个变量存在，以及让编译器为这个变量安排合适的内存空间，对其内部存储的数据做出正确的解释。对函数的使用也类似。在调用函数时，编译器需要知道该函数是否存在，而且需要检查调用者的参数是否符合被调用函数的要求。如果这样，那么被调用的函数就要写在调用函数的语句之前。但有时很难做到。所以，C 语言中采用了另一种机制——函数原型来解决这一问题。

函数原型又称为函数声明，是和变量声明类似的语法，它只有函数名、返回值和函数参数，而没有实现代码。与之对应的，是函数的定义。在 C 语言中，变量的声明和定义是一个意思，而函数的声明和定义则是截然不同的两回事。

函数原型告诉编译器：函数的名字、能够接收的参数个数、参数类型、参数的顺序以及返回值的类型。编译器通过函数原型来检查函数来检查函数调用是否合法。函数原型并不复杂，上面的例子中包含这样两行代码：

```
void PrintStars():           // 函数原型声明
void PrintMsg(char msg[]);   // 函数原型声明
```

这就是标准的函数原型声明。以第二个为例，它告诉编译器，名为 `PrintMsg` 的函数没有返回值，有一个参数，是字符型数组。若调用者的参数不符合这一要求，编译器将报错。

一般来说，函数原型的声明形式如下：

```
返回类型 函数名([参数列表]);
```

在 C 语言的发展中，关于函数原型的声明形式经历了多次改变，现代 C 语言编译器为了保持兼容性，对这些改变都予以承认，导致原型的声明形式比较混乱，现一一列举如下：

1. 如果函数的使用在后，函数的定义在前，则函数原型的声明可以省略。这样带来的问题是所有的函数定义都要写在 `main()` 函数之前，导致阅读程序困难；而且对于某些递归的调用，将无法实现定义在调用之前。
2. 函数声明的位置，一般是放在一个源文件中所有函数定义之前。但也允许将声明放在函数里面，如同对变量的声明一样，比如：

```
int main()
{
    void PrintMsg(char msg[]);
    PrintMsg("Hello, world!");
    return 0;
}
```

如果将声明写在函数里，则只有包含此声明的函数能够调用被声明的函数，其他使用者还需另外再声明。

3. 关于函数参数，也有三种方式。第一种是将参数类型和参数名称完整列出。第二种是只列出参数类型，比如：

```
void PrintMsg(char []);
```

这种形式虽然可以让编译器检测参数类型，但不利于理解参数的实际意义，在某些情况下还可能出错。第三种就是完全省略参数表，比如：

```
void PrintMsg();
```

这样也可以通过编译，但编译器无法通过声明来检测调用者的参数是否正确，没有达到声明的目的，还容易让人误以为没有参数。如果确实没有参数，可以写成这种形式：

```
void PrintMsg(void);
```

一般推荐写全。

在开发复杂程序时，通常会声明很多函数原型，这时一般会把所有函数原型同意写到若干个头文件中，利用 `#include` 语句包含进源程序即可。

9.2 函数定义

所谓函数定义就是指对函数的实现。一个完整的函数包括函数头和函数体两部分，函数的代码就是写在函数体中。函数定义的一般形式如下：

```
返回值类型 函数名([形式参数表])
{
    [变量声明]
    [可执行语句]
}
```

```
void PrintStr()           // 函数头部
{                          // 函数开始
    char str[] = "Hello, world!"; // 变量声明
    puts(str);            // 执行语句
} // 函数结束
```

这个例子比较简单，它不是一个完整的程序，不能运行。

1. 函数头应与前面函数的原型声明一致。函数名也是用户自定义标识符，要合法。
2. 函数可以有返回值。如果有，应该在函数名前标明返回值类型；如果没有，应标明 `void`。（如果不标，默认是 `int`，但为了区分，最好都标注出来）
3. 参数表中可以有零个或多个参数，称为形式参数，简称形参。如果没有参数，圆括号不能省。
4. 函数体以一对花括号来定界。

9.3 函数返回值

在上面的例子中出现过一个 `PrintStr()` 函数，它只在屏幕上输出一行星号，然后就返回了。尽管可以看到输出，但无法了解函数的运行情况，也不知道运行是否成功，结果如何。

这个函数无法传值给调用者，因为缺乏两个要素：一是返回值类型，二是 `return` 语句。

返回值类型可以是基本类型，也可以是其他类型，其次，返回值的输出还需要一个关键字：`return`。该关键字本身就表明了含义，一旦程序执行到此处——无论后面是否还有语句，就立刻返回。至于返回到哪里，就要看函数是被谁调用，这条语句会返回到最初的调用点。

`return` 语句有两种形式：

```
return ;
```

```
return 表达式;
```

前者不会将值传回调用者，后者会将表达式的值计算出来，传递给调用者（这里的表达式可以是任何合法的表达式）。

一个函数可以有多个 `return` 语句，但在函数的一次运行中，最多只可能有一个 `return` 语句被执行。

9.4 函数调用

函数调用的一般形式如下：

```
函数名([实参列表]);
```

函数调用中的实参必须与函数定义时的形式参数一一对应。

函数调用可以赋值给变量，也可以直接当做变量进行运算或者作为其他语句的参数。

9.5 形式参数与实际参数

在定义函数时，写在函数名后面括号中的变量就称为形式参数，简称形参；在调用函数时传入的表达式称为实际参数，简称实参。

注意：

- 形式参数只能是变量，不能是表达式，也不能是常量，但可以用 `const` 来修饰。
- 实际参数可以是任何表达式，只要表达式的结果兼容声明的形参类型。

9.6 局部变量与全局变量

能够使用变量的范围称为该变量的作用域。

一般，局部变量定义在一个函数开始的部分，即所有执行语句之前。且局部变量只能在声明的函数的内部使用，在外部则无法使用。同时在外部定义和声明使用局部变量时，也无需考虑其他函数是否也使用了同名的变量。

在新的标准中，作用域不仅仅只用函数，`for`语句等的语句块也可以作为作用域，即在单一语句块中的局部变量在外部也无法使用。

而全局变量，就是定义在函数外部的变量，它的作用域是某个源文件，即每个函数都可以使用它。


C 语言中还规定，局部变量可以与全局变量同名，如果出现这种情况，函数只能访问局部变量，而全局变量会被屏蔽。

9.7 自动变量与静态变量

若从变量的生存时期（时间）来分类，则变量还可以分为自动存储变量和静态存储变量。

所谓静态存储变量，是指分配在静态存储空间上的变量；而自动存储变量则指分配在动态存储空间上的变量。

程序区用来存放程序代码，数据只能存放在静态存储区或者动态存储区中。其中，全局变量是存放在静态存储区中，程序一开始运行，就会为全局变量分配存储区，一直要到程序结束后，所占据的存储空间才会被系统收回。在整个程序执行过程中，它们占据固定存储单元，不会动态地分配和释放。

图 9 - 1 用户的存储空间

除了全局变量，前面提到的所有局部变量都是自动存储变量。对局部变量的定义：

```
数据类型 变量名;
```

它与下面的语句等价：

```
auto 数据类型 变量名;
```

也就是说，前面在定义局部变量时，实际上是省略了关键字 `auto`。也可以显示地加上它来修饰变量。唯一的例外就是形式参数，它们是局部变量，但不允许用 `auto` 来修饰。

对于自动变量，当程序执行到它们所在的函数中时，系统会为自动变量分配内存空间。如果该变量需要初始化，也会同时对其进行初始化；当函数执行完毕时，系统会收回自动变量所占的空间。这一切无需程序员干预。

与自动变量相对应的是静态变量，需要使用关键字 `static`，一般定义的形式如下：

```
static 数据类型 变量名;
```

例如：

```
void fun() {  
    static int sk;  
    static int sm = 0;  
}
```

`sk` 和 `sm` 都是静态变量。它们仍然是局部变量，作用域仍然是函数 `fun()`，在其他函数中也不可直接使用。

静态变量与自动变量的不同点在于：系统在第一次执行 `fun()` 的时候为 `sk` 和 `sm` 分配内存空间，并执行初始化操作。当函数执行完毕后，静态变量所占的空间不会被收回。当下一次再调用该函数时，变量不需要重新分配空间和初始化，它们仍然使用上次占据的空间，数据的值也没有发生变化。因此，静态变量是具有‘记忆’功能的。

另外，系统不会对自动变量做默认的初始化工作，也就是说，如果使用者不对自动变量做初始化，则自动变量中的数据是不可预测的。但系统会自动对静态变量做初始化工作，如果使用者不进行初始化，则静态变量中存储的数值为0。

由于静态变量必须存储在静态存储区中，而函数的形参总是存储在动态存储区（栈）里面，所以形参不可能声明成静态变量。

不仅局部变量可以声明成静态变量，全局变量也可以用 `static` 修饰。但是由于全局变量本身就已经是静态变量，所以用 `static` 修饰时，它的含义发生了变化，它表示这个全局变量只能在本文件中使用，不允许其他文件中的函数使用。

9.8 数组作为函数的参数

普通变量可以作为函数的参数。有时，传递的数据是存放在数组中的，（又或者是其他复合类型）所以也允许将数组作为参数传递。

首先，在函数定义时，需要将形参声明成数组的形式，当参数为一维数组时，形式如下：

```
返回类型 函数名(数组类型 数组名[]);
```

注意数组名后面的中括号，一般情况下定义普通数组时，中括号中需要一个整型数值常量，用来指定数组长度。但作为形参时，允许为空，也就是无需指定数组的大小。

在调用一个形参为数组的函数时，调用者通常会用一个数组作为实际参数。一般的调用形式如下：

```
函数名(数组名);
```

实参并非一定要是数组名，这不过是一种最普通的形式。以数组为形参的时候，所声明的数组本质上是指针；所以，实参只要求是指针就可以了，而数组名正是一种指针。

也正由于数组为参数的函数操作指针，即操作实际的数据地址，所以函数对形参的改变对实参也有效。

当参数为二维数组时，一般声明形式如下：

```
返回类型 函数名(数组类型 数组名[][列数]);
```

数组的行数不需要指定，即便指定了也无效，但列数值不可缺少。系统需要后面这个列值来计算某个元素在数组中的相对位置。与一维数组名作为形参一样，这里的二维数组本质上还是一个指针，系统不会为它分配存放数组元素的空间，它的行数也要由实参来决定。

调用时也需要一个二维数组名作为实参，通常形式如下：

函数名(二维数组名)

注意：二维数组作为实际参数时一定要保证列数相等，行数可以不等，但列数一定要相等，才可以作为参数输入。