

C 语言教程讲义

Potato

2024 年 7 月 23 日

第六章

本文件编译于 2024 年 7 月 23 日

6 函数

6.1 函数的定义

想象一下，我们有一个需求，需要输入三个整型 a 、 b 和 $c(c \neq 0)$ ，计算这样一个数字

$$x = \frac{a^2 + b^2}{c^2} + \frac{a(a^2 - c^2)}{b^2 + c}$$

我们的程序会写成这样

```
1  #include <stdio.h>
2
3  int main() {
4      int a;
5      int b;
6      int c;
7      scanf("%d %d %d", &a, &b, &c);
8
9      if (c == 0) {
10         printf("c 不能为 0!\n");
11     } else {
12         double x = (double)(a * a + b * b) / (c * c) + (double)(a * (a * a - c * c)) / (b * b + c);
13         printf("%lf\n", x);
14     }
15
16     return 0;
17 }
```

这里算 x 的式子看着就让人头大，也许我们可以提前把三个数的平方算出来，把程序改成这样

```

1  #include <stdio.h>
2
3  int main() {
4      int a;
5      int b;
6      int c;
7      scanf("%d %d %d", &a, &b, &c);
8
9      int a2 = a * a;
10     int b2 = b * b;
11     int c2 = c * c;
12
13     if (c == 0) {
14         printf("c不能为0!\n");
15     } else {
16         double x = (double)(a2 + b2) / c2 + (double)(a * (a2 - c2)) / (b2 + c);
17         printf("%lf\n", x);
18     }
19
20     return 0;
21 }

```

这样代码确实简洁了很多，但是我们发现，对三个数进行平方运算的代码是一样的，我们如果能把这段代码提取出来，使得代码可以被复用，我们的程序就会简单很多。这就像是数学中的函数一样，我们给定一个输入，它给出一个输出。c 语言也提供了函数，便于我们复用代码。我们在定义变量的时候，有变量的声明语法和赋值语法，函数也是类似的，有函数原型和函数的实现。

我们在数学中学习函数时，知道函数有名字（比如正弦函数），函数传入一个值，并返回一个值（比如我们向正弦函数传入 $\pi/4$ ，返回的就是 $\sqrt{2}/2$ ）。有些函数可能传入两个或多个值（比如 $f(a, b, c) = a + b + c$ ），但是总是只返回一个值（比如 $f(1, 2, 3)$ 返回 6）。

c 语言中的函数也是类似的，一个函数，有函数名，参数列表和返回值。所谓函数名，就是函数的名字，对函数的取名要尽可能简明扼要地显示出函数的作用，比如起名 Square 就要优于起名为 S，函数使用大驼峰命名，就像命名常量一样；参数列表，就是传入函数的变量，每个参数之间用逗号连接，如我们要传入两个整型，就可以写成 int a, int b；至于返回值，就是函数计算后得到的值。由此，我们得到了定义函数原型的语法：返回值 函数名 (参数列表)；

代码风格提示：函数使用大驼峰命名，即 CapitalizeTheFirstLetterOfEachWord

比如，我们想要实现这个函数

$$\text{Square}(x) = x^2, x \in \mathbb{Z}$$

就可以这样写它的函数原型

```

1  int Square(int x);

```

这行代码意味着我们的函数名为 Square，一看就知道是用于计算平方的。它的参数列表只有一个整型 x，返回值也是整型。

注意，函数原型要写在 main 函数之前。有了函数原型，就要写函数的具体实现了，其语法是将函数原型照抄一遍，但是将最后的分号改成花括号，将程序主体写在花括号内，比如

```
1  int Square(int x) {  
2      return x * x;  
3  }
```

函数实现写在哪儿都可以，只要写在函数原型之后即可。这里的 return 就是用于返回返回值的。函数执行到 return 就结束了，因此一切写到 return 后的语句都是无效的。当我们调用函数 Square 时，里面的程序就会像 main 函数一样从上到下依次执行。

接下来就要调用函数了，调用函数的语法是函数名 (参数)。也就是说，我们可以这样调用 Square。

```
1  Square(a);
```

那么，我们怎么获得函数的返回值呢？我们定义一个变量，用它来接收即可，比如

```
1  #include <stdio.h>  
2  
3  int Square(int x);  
4  
5  int Square(int x) {  
6      return x * x;  
7  }  
8  
9  int main() {  
10     int a = 2;  
11     int b = Square(a);  
12  
13     printf("a的平方是%d\n", b);  
14  
15     return 0;  
16 }
```

运行程序，输出结果是 4。我们明明是对 a 进行计算，为什么函数里使用的是 x 呢？观察 main 函数内的代码，我们发现，我们将 a 作为参数传给了 Square，因此 Square 内的 x 就是 a。

就像是变量的声明和赋值可以放在一起一样，函数原型和函数实现也可以写在一起，像是上面的代码，我们可以改写成

```
1  #include <stdio.h>  
2  
3  int Square(int x) {  
4      return x * x;  
5  }  
6
```

```

7   int main() {
8       int a = 2;
9       int b = Square(a);
10
11      printf("a的平方是%d\n", b);
12
13      return 0;
14  }

```

既然如此，为什么还要写一个函数原型呢？我们前面说，函数实现可以写在其它任何地方，也就是说，上面的代码可以写成

```

1   #include <stdio.h>
2
3   int Square(int x);
4
5   int main() {
6       int a = 2;
7       int b = Square(a);
8
9       printf("a的平方是%d\n", b);
10
11      return 0;
12  }
13
14  int Square(int x) {
15      return x * x;
16  }

```

如果我们有一堆函数，我们就可以只在 main 前写函数原型，将所有的函数实现写在后面，这样代码更加简洁。后面学习头文件时，我们会对这一点有更深刻的了解。

当然了，我们也可以在函数里写除了求值外的其它功能，比如将 Square 改成

```

1   int Square(int x) {
2       printf("计算%d的平方!\n", x);
3       return x * x;
4   }

```

如果我们的函数不需要返回值怎么办？比如我们写一个函数用来打印 n 次 Hello，函数体内肯定是这样的

```

1   for (int i = 0; i < n; ++i) {
2       printf("Hello\n");
3   }

```

这个函数不需要返回值，因此使用 void 写在返回值的地方。因此这个函数就变成了

```

1 void SayHello(int n) {
2     for (int i = 0; i < n; ++i) {
3         printf("Hello\n");
4     }
5 }

```

在调用它的时候，也就不需要一个变量要接收它的返回值了，我们可以直接这样写

```

1 #include <stdio.h>
2
3 void SayHello(int n) {
4     for (int i = 0; i < n; ++i) {
5         printf("Hello\n");
6     }
7 }
8
9 int main() {
10     SayHello(2);
11
12     return 0;
13 }

```

实际上，有返回值的函数也不必须要接收它的返回值，但是这样就浪费了它的返回值。

我们在写函数原型时写的参数列表叫做函数的形式参数，简称形参。在调用函数时传入的参数叫做函数的实际参数，简称实参。

一个函数可以没有返回值，也可以没有形参。比如

```

1 #include <stdio.h>
2
3 void SayHi() {
4     printf("Hi\n");
5 }
6
7 int main() {
8     SayHi();
9
10    return 0;
11 }

```

在一些比较古老的编译器中，我们需要使用 void 来显式地说明函数没有形参，比如将上面的函数改成

```

1 void SayHi(void) {
2     printf("Hi\n");
3 }

```

我们写一个函数，可能需要让它完成很多工作，因此我们推荐在函数原型前写多行注释要详细阐释函数的作用以及每个参数、返回值的作用。比如

```

1  /*
2  计算数字的平方
3  x 需要计算的底数
4  return 传入数的平方值
5  */
6  int Square(int x);

```

另外值得注意的是，函数就像变量一样，是不能重名的。比如这样是不允许的

```

1  int Square(int x);
2  int Square(int x); // 重复定义!

```

观察这种情况

```

1  int Square(int x);
2  int Square(int x, int y);

```

像这样，两个函数只是名称相同，但参数列表或返回值不同的情况叫做函数的重载。c 语言不支持函数重载，因此上面的代码也是错误的。但是很多计算机语言都是允许这样的，如 C++ 和 Java。

我们可以在函数里调用别的函数，比如

```

1  #include <stdio.h>
2
3  void A() {
4      printf("Hi\n");
5  }
6
7  void B() {
8      A();
9      printf("Hi Hi Hi\n");
10 }
11
12 int main() {
13     B();
14
15     return 0;
16 }

```

其实函数还可以调用自己，这种情况叫函数的递归。比如我们可以设计一段程序计算斐波那契数列，它的定义是

$$F(n) = \begin{cases} 1, & \text{if } n = 1, 2; \\ F(n-1) + F(n-2), & \text{if } n > 2; \end{cases}$$

因此可以这样递归地实现斐波那契数列的计算

```

1  int Fibonacci(int n) {
2      if (n == 1 || n == 2) {
3          return 1;
4      }
5      // 使用卫语句简化了函数!
6      return Fibonacci(n - 1) + Fibonacci(n - 2);
7  }

```

比如我们要计算 n 为 5 的时候的值，函数就会去计算 n 为 4 和 n 为 3 时的值；要想计算 n 为 4 时的值，函数就需要计算 n 为 3 和 n 为 2 的值；要想计算 n 为 3 的值，就需要 n 为 2 和 n 为 1 的值。也就是说，在使用这个函数的时候，需要一直调用这个函数直到不再需要调用自己。所以递归的函数往往运算比较慢，而且也不利于我们阅读。因此，除了个别极其利于使用递归的情况，我们一般很少使用递归。

6.2 main 函数

我们一直在 `main` 函数内写程序，它是程序入口，`main` 函数中的程序会有上到下执行，我们的一切程序都是由 `main` 函数组织起来的。`main` 函数写成

```

1  int main() {}

```

这说明它是一个返回值为整型的函数，我们之前一直在代码最后写

```

1  int main() {
2      return 0;
3  }

```

这就是 `main` 函数的返回值了。我们默认返回 0，因为这代表着程序正常执行。那么这个返回值在哪里能看到呢？我们执行完程序后，会发现终端最后有一行“退出代码 0”，这就是 `main` 函数的返回值。如果我们把 `main` 的返回值改成别的数字，那么这里的“退出代码”也会改变。

我们看到，`main` 函数就是一个没有形参，返回值为整型的函数。实际上 `main` 函数还可以写成这样

```

1  int main(int argc, char *argv[]) {}

```

这里的 `main` 接收两个参数，后面这个奇形怪状的 `argv` 暂且不需要知道是干什么的，我们在学习数组时会介绍它的作用。

为什么 `main` 函数可以有两种形式呢？不是说 `c` 语言不支持函数重载吗？实际上，`main` 函数是唯一一个特例，这与编译中的连接过程有关，我们就不多介绍了。

6.3 标准库函数

我们之前一直在使用 `printf` 和 `scanf`，我们发现，我们就是按照调用函数的函数名 (参数) 的语法使用它们的。实际上，这二者就是两个函数，它们是由编译器所连接的标准库实现的。

`c` 语言提供了很多库函数，可以使用它们完成一些最底层的任务。比如我们之后会学习到用于开辟内存空间的 `malloc`，复制字符串的 `strcpy`，复制内存的 `memcpy` 等。我们要想使用库函数，就需要引用对应的头文件。

其实我们已经引用过头文件了，就是在代码最开头写的

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

也即，引用标准库头文件的语法是#include <头文件名>。

我们以前使用过#define 来声明一个常量，这种写在井号后的语句叫做宏。在代码编译前会经由预处理器处理，它会把所有宏都替换成本文或者字面量。我们在使用#include 宏时，预处理器会将引用的头文件中的所有代码都替换到我们的代码中。头文件中的代码都是函数原型，函数的实现位于编译器连接的标准库的二进制文件中。

在学习项目结构时，我们会学习如何编写头文件并引用自定义的头文件。

我们在 visual studio 中，将光标放在函数上，右键，点击“转到定义”就可以转到函数的原型了。比如 printf 函数的原型是¹

```
1  _Check_return_opt_  
2  _CRT_STDIO_INLINE int __CRTDECL printf(  
3      _In_z_ _Printf_format_string_ char const* const _Format,  
4      ...)  
5  #if defined _NO_CRT_STDIO_INLINE  
6  ;  
7  #else  
8  {  
9      int _Result;  
10     va_list _ArgList;  
11     __crt_va_start(_ArgList, _Format);  
12     _Result = _vfprintf_l(stdout, _Format, NULL, _ArgList);  
13     __crt_va_end(_ArgList);  
14     return _Result;  
15 }  
16 #endif
```

这个函数原型包含了大量宏和我们没有学过的知识，因此非常复杂。我们在准备使用一个标准库函数时，如何快速获得它的用法呢？实际上，网站 cppreference² 提供了完整的 c 语言和 c++ 语言标准，我们可以查阅这个网站来获取标准库函数的信息。

还记得在最开始学习 c 语言，使用 scanf 时，visual studio 告诉我们“scanf 的返回值被省略”吗？我们现在可以解决这个问题了。

首先查看网站 cppreference，在“file input/output”下找到“scanf”，在“return value”下有这样一行话：“Number of receiving arguments successfully assigned (which may be zero in case a matching failure occurred before the first receiving argument was assigned)”。也即，返回值 0 意味着输入失败。因此，它的返回值实际上代表着函数执行成功与否。利用这一点，我们可以编写程序来处理错误输入

```
1  int stat= scanf("%d", &n);  
2  if (stat == 0) {
```

¹以 MSVC STL 作为标准

²<https://en.cppreference.com>


```
3     printf("输入有误!\n");
4 }
```

7 数组

我们有时需要储存很多变量。如果通过写很多个变量来实现这个功能，不仅难以对每个变量进行储存和读取，也有很强的限制性，因为若想改变变量的数量就需要重新编写代码。因此，c 语言提供了一种叫做数组的数据结构用来储存很多变量。

数组就像是一列火车，每一节车厢都装着一个变量。在我们需要获取某个变量的时候，只需要知道它位于第几号车厢即可。一样的，我们想要改变某个变量，只需要知道它的车厢号即可。一列火车有它的长度，我们的数组也有它的长度，并且一个数组只能装一种类型的数据。我们使用数组就可以解决上一段提到的问题了。

数组就是一种特殊的变量。它的声明语法是数据类型 数组名 [数组长度]。这里的数组长度指的是拥有的元素个数。比如我们要声明一个拥有 4 个元素的，装载整型变量的数组，就可以这样写：

```
1 int array[4];
```

如何赋值呢？对数组赋值一般叫做数组的初始化。如对于上面声明的数组，可以这样初始化：

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

注意，数组的声明和初始化要写在一起。