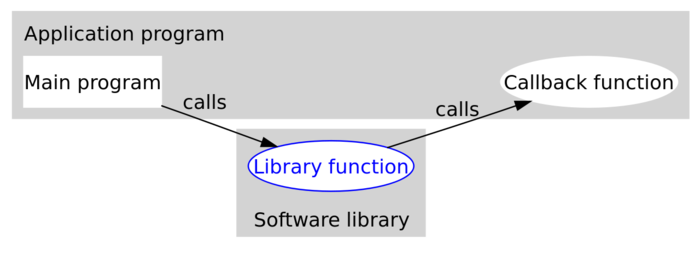
<https://segmentfault.com/a/1190000008293902>

## 什么是回调函数

我们先来看看百度百科是如何定义回调函数的：

回调函数就是一个通过函数指针调用的函数。如果你把函数的指针（地址）作为参数传递给另一个函数，当这个指针被用来调用其所指向的函数时，我们就说这是回调函数。回调函数不是由该函数的实现方直接调用，而是在特定的事件或条件发生时由另外的一方调用的，用于对该事件或条件进行响应。

这段话比较长，也比较绕口。下面我通过一幅图来说明什么是回调：



假设我们要使用一个排序函数来对数组进行排序，那么在主程序(Main program)中，我们先通过库，选择一个库排序函数(Library function)。但排序算法有很多，有冒泡排序，选择排序，快速排序，归并排序。同时，我们也可能需要对特殊的对象进行排序，比如特定的结构体等。库函数会根据我们的需要选择一种排序算法，然后调用实现该算法的函数来完成排序工作。这个被调用的排序函数就是回调函数(Callback function)。

结合这幅图和上面对回调函数的解释，我们可以发现，要实现回调函数，最关键的一点就是要将函数的指针传递给一个函数(上图中是库函数)，然后这个函数就可以通过这个指针来调用回调函数了。注意，回调函数并不是C语言特有的，几乎任何语言都有回调函数。在C语言中，我们通过使用函数指针来实现回调函数。那函数指针是什么？不着急，下面我们就先来看看什么是函数指针。

## 什么是函数指针

函数指针也是一种指针，只是它指向的不是整型，字符型而是函数。在C中，每个函数在编译后都是存储在内存中，并且每个函数都有一个入口地址，根据这个地址，我们便可以访问并使用这个函数。函数指针就是通过指向这个函数的入口，从而调用这个函数。

## 函数指针的使用

### 函数指针的定义

函数指针虽然也是指针，但它的定义方式却和其他指针看上去很不一样，我们来看看它是如何定义的：

/\* 方法1 \*/

**void** (\*p\_func)(**int**, **int**, **float**) = NULL;

/\* 方法2 \*/

**typedef** **void** (\*tp\_func)(**int**, **int**, **float**);

tp\_func p\_func = NULL;

这两种方式都是定义了一个指向返回值为 **void** 类型，参数为 **(int, int, float)** 的函数指针。第二种方法是为了让函数指针更容易理解，尤其是在复杂的环境下；而对于一般的函数指针，直接用第一种方法就行了。  
如果之前没见过函数指针，可能会觉得函数指针的定义比较怪，为什么不是 **void ()(int, int, float) \*p\_func** 而是 **void (\*p\_func)(int, int, float)** 这种形式？这个问题我也不知道，也没必要纠结，花点时间理解下它与普通指针的区别，实在不行就先记住它的形式。

### 函数指针的赋值

在定义完函数指针后，我们就需要给它赋值了我们有两种方式对函数指针进行赋值：

**void** (\*p\_func)(**int**, **int**, **float**) = NULL;

p\_func = &func1;

p\_func = func2;

上面两种方法都是合法的，对于第二种方法，编译器会隐式地将 **func\_2** 由 **void ()(int, int, float)** 类型转换成 **void (\*)(int, int, float)** 类型，因此，这两种方法都行。想要了解更详细的说明，可以看看下面这个stackoverflow的[链接](http://stackoverflow.com/questions/6893285/why-do-function-pointer-definitions-work-with-any-number-of-ampersands-or-as)。

### 使用函数指针调用函数

因为函数指针也是指针，因此可以使用常规的带 **\*** 的方法来调用函数。和函数指针的赋值一样，我们也可以使用两种方法：

/\* 方法1 \*/

int val1 = p\_func(1,2,3.0);

/\* 方法2 \*/

int val2 = (\*p\_func)(1,2,3.0);

方法1和我们平时直接调用函数是一样的，方法2则是用了 **\*** 对函数指针取值，从而实现对函数的调用。

### 将函数指针作为参数传给函数

函数指针和普通指针一样，我们可以将它作为函数的参数传递给函数，下面我们看看如何实现函数指针的传参：

/\* func3 将函数指针 p\_func 作为其形参 \*/

**void** **func3**(**int** a, **int** b, **float** c, **void** (\*p\_func)(**int**, **int**, **float**))

{

(\*p\_func)(a, b, c);

}

/\* func4 调用函数func3 \*/

**void** **func4**()

{

func3(1, 2, 3.0, func\_1);

/\* 或者 func3(1, 2, 3.0, &func\_1); \*/

}

### 函数指针作为函数返回类型

有了上面的基础，要写出返回类型为函数指针的函数应该不难了，下面这个例子就是返回类型为函数指针的函数：

**void** (\* func5(**int**, **int**, **float** ))(**int**, **int**)

{

...

}

在这里， **func5** 以 **(int, int, float)** 为参数，其返回类型为 **void (\*)(int, int)** 。在C语言中，变量或者函数的声明也是一个大学问，想要了解更多关于声明的话题，可以参考我之前的文章 - [C专家编程》读书笔记(1-3章)](https://segmentfault.com/a/1190000008054683)。这本书的第三章花了整整一章的内容来讲解如何读懂C语言的声明。

### 函数指针数组

在开始讲解回调函数前，最后介绍一下函数指针数组。既然函数指针也是指针，那我们就可以用数组来存放函数指针。下面我们看一个函数指针数组的例子：

/\* 方法1 \*/

**void** (\*func\_array\_1[5])(**int**, **int**, **float**);

/\* 方法2 \*/

**typedef** **void** (\*p\_func\_array)(**int**, **int**, **float**);

p\_func\_array func\_array\_2[5];

上面两种方法都可以用来定义函数指针数组，它们定义了一个元素个数为5，类型是 **void (\*)(int, int, float)** 的函数指针数组。

## 回调函数

我们前面谈的都是函数指针，现在我们回到正题，来看看回调函数到底是怎样实现的。下面是一个四则运算的简单回调函数例子：

#include <stdio.h>

#include <stdlib.h>

/**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

\* 函数指针结构体

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\****\*\*\**\*/

**typedef** struct \_OP {

float (\*p\_add)(float, float);

float (\*p\_sub)(float, float);

float (\*p\_mul)(float, float);

float (\*p\_div)(float, float);

} OP;

/**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

\* 加减乘除函数

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\****\*\*\**\*/

float ADD(float a, float b)

{

**return** a + b;

}

float SUB(float a, float b)

{

**return** a - b;

}

float MUL(float a, float b)

{

**return** a \* b;

}

float DIV(float a, float b)

{

**return** a / b;

}

/**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

\* 初始化函数指针

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\****\*\*\**\*/

**void** init\_op(OP \*op)

{

op->p\_add = ADD;

op->p\_sub = SUB;

op->p\_mul = &MUL;

op->p\_div = &DIV;

}

/**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

\* 库函数

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\****\*\*\**\*/

float add\_sub\_mul\_div(float a, float b, float (\*op\_func)(float, float))

{

**return** (\*op\_func)(a, b);

}

int main(int argc, char \*argv[])

{

OP \*op = (OP \*)malloc(sizeof(OP));

init\_op(op);

/\* 直接使用函数指针调用函数 \*/

printf("ADD = %f, SUB = %f, MUL = %f, DIV = %f\n", (op->p\_add)(1.3, 2.2), (\*op->p\_sub)(1.3, 2.2),

(op->p\_mul)(1.3, 2.2), (\*op->p\_div)(1.3, 2.2));

/\* 调用回调函数 \*/

printf("ADD = %f, SUB = %f, MUL = %f, DIV = %f\n",

add\_sub\_mul\_div(1.3, 2.2, ADD),

add\_sub\_mul\_div(1.3, 2.2, SUB),

add\_sub\_mul\_div(1.3, 2.2, MUL),

add\_sub\_mul\_div(1.3, 2.2, DIV));

**return** 0;

}

这个例子有点长，我一步步地来讲解如何使用回调函数。

### 第一步

要完成加减乘除，我们需要定义四个函数分别实现加减乘除的运算功能，这几个函数就是:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 加减乘除函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**float** **ADD**(**float** a, **float** b)

{

**return** a + b;

}

**float** **SUB**(**float** a, **float** b)

{

**return** a - b;

}

**float** **MUL**(**float** a, **float** b)

{

**return** a \* b;

}

**float** **DIV**(**float** a, **float** b)

{

**return** a / b;

}

### 第二步

我们需要定义四个函数指针分别指向这四个函数：

/**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

\* 函数指针结构体

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\****\*\*\**\*/

**typedef** struct \_OP {

float (\*p\_add)(float, float);

float (\*p\_sub)(float, float);

float (\*p\_mul)(float, float);

float (\*p\_div)(float, float);

} OP;

/**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

\* 初始化函数指针

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\****\*\*\**\*/

**void** init\_op(OP \*op)

{

op->p\_add = ADD;

op->p\_sub = SUB;

op->p\_mul = &MUL;

op->p\_div = &DIV;

}

### 第三步

我们需要创建一个“库函数”，这个函数以函数指针为参数，通过它来调用不同的函数：

/**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

\* 库函数

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\****\*\*\**\*/

float add\_sub\_mul\_div(float a, float b, float (\*op\_func)(float, float))

{

**return** (\*op\_func)(a, b);

}

### 第四步

当这几部都完成后，我们就可以开始调用回调函数了：

/\* 调用回调函数 \*/

printf("ADD = %f, SUB = %f, MUL = %f, DIV = %f\n",

add\_sub\_mul\_div(1.3, 2.2, op->p\_add),

add\_sub\_mul\_div(1.3, 2.2, op->p\_sub),

add\_sub\_mul\_div(1.3, 2.2, MUL),

add\_sub\_mul\_div(1.3, 2.2, DIV));

简单的四部便可以实现回调函数。在这四步中，我们甚至可以省略第二步，直接将函数名传入“库函数”，比如上面的乘法和除法运算。回调函数的核心就是函数指针，只要搞懂了函数指针再学回调函数，那真是手到擒来了。

## 总结

本文主要讲了如何使用函数指针和回调函数。回调函数的核心就是函数指针，因此我花了大量篇幅讲解函数指针。对于回调函数的实现，我给出了一个例子，希望这个例子能给你帮助。回调函数很重要，如果连它都不会，C语言真不算入门了。当然了，即使会了它，也不要骄傲，因为C语言还有太多的东西需要我们去学习、实践。