指向函数的指针 ------ 函数指针（function pointer）

函数具有可赋值给指针的物理内存地址，一个函数的函数名就是一个指针，它指向函数的代码。一个函数的地址是该函数的进入点，也是调用函数的地址。函数的调用可以通过函数名，也可以通过指向函数的指针来调用。函数指针还允许将函数作为变元传递给其他函数。

不带括号和变量列表的函数名，这可以表示函数的地址，正如不带下标的数组名可以表示数组的首地址。

定义形式：

类型 （*指针变量名）（参数列表）；

例如：

int (*p)(int i,int j);

p 是一个指针，它指向一个函数，该函数有 2 个整形参数，返回类型为 int。p 首先和*结合，表明 p 是一个指针。然后再与（）结合，表明它指向的是一个函数。指向函数的指针也称为函数指针。

```
#include <stdio.h>

#define   GET_MAX       0
#define   GET_MIN       1

int get_max(int i,int j)
{
      return i>j?i:j;
}

int get_min(int i,int j)
{
      return i>j?j:i;
}

int compare(int i,int j,int flag)
```

```
{
	int ret;

	//这里定义了一个函数指针，就可以根据传入的 flag，灵活地决定其是指向求大数或求小数的函数
	//便于方便灵活地调用各类函数
	int (*p)(int,int);

	if(flag == GET_MAX)
		p = get_max;
	else
		p = get_min;

	ret = p(i,j);

	return ret;
}

int main()
{
	int i = 5,j = 10,ret;

	ret = compare(i,j,GET_MAX);
	printf("The MAX is %d\n",ret);

	ret = compare(i,j,GET_MIN);
	printf("The MIN is %d\n",ret);

	return 0 ;
}
```

C 语言-函数指针（Function Pointer）及进阶

Ostkaka 、 关注

2016.07.30 20:37* 字数 2558 阅读 2218 评论 6 喜欢 30

前言

初学 C 语言的童鞋，通常在学完函数和指针的知识后，已经是萌萌哒，学习到了函数指针（请注意不是函数和指针），更是整个人都不好了，这篇文章的目的，就是帮助我的童鞋们理解函数指针。

函数指针概述

首先我们需要回顾一下函数的作用：完成某一特定功能的代码块。
再来回忆一下指针的作用：一种特殊的变量，用来保存地址值，某类型的指针指向某类型的地址。
下面定义了一个求两个数最大值的函数：

```
int maxValue (int a, int b) {

    return a > b ? a : b;

}
```

而这段代码编译后生成的 CPU 指令存储在代码区，而这段代码其实是可以获取其地址的，而其地址就是函数名，我们可以使用指针存储这个函数的地址——函数指针。
函数指针其实就是一种特殊的指针——指向一个函数的指针。在很多高级语言中，它的思想是很重要的，尤其是它的“回调函数”，所以理解它是很有必要的。

函数指针定义与使用

任何变量定义都包含三部分: 变量类型 + 变量名 = 初值，那么定义一个函数指针，首先我们需要知道要定义一个什么样的函数指针（指针类型），那么问题来了，函数的类型又是什么呢？我们继续分析这段代码：

```
int maxValue (int a, int b) {

    return a > b ? a : b;

}
```

这个函数的类型是有两个整型参数，返回值是个整型。对应的函数指针类型：

```
int (*) (int a, int b);
```

对应的函数指针定义：

```
int (*p)(int x, int  y);
```

参数名可以去掉，并且通常都是去掉的。这样指针 p 就可以保存函数类型为两个整型参数，返回值是整型的函数地址了。

```
int (*p)(int, int);
```

通过函数指针调用函数：

```
int (*p)(int, int) = NULL;

p = maxValue;

p(20, 45);
```

回调函数

上述内容是函数指针的基础用法，然而我们可以看得出来，直接使用函数 maxValue 岂不是更方便？没错，其实函数指针更重要的意义在于函数回调，而上述内容只是一个铺垫。
举个例子:
现在我们有这样一个需求：实现一个函数，将一个整形数组中比 50 大的打印在控制台，我们可能这样实现：

```
void compareNumberFunction(int *numberArray, int count, int compareNumber) {

    for (int i = 0; i < count; i++) {

        if (*(numberArray + i) > compareNumber) {

            printf("%d\n", *(numberArray + i));

        }

    }

}

int main() {



    int numberArray[5] = {15, 34, 44, 56, 64};

    int compareNumber = 50;

    compareNumberFunction(numberArray, 5, compareNumber);



    return 0;

}
```

这样实现是没有问题的，然而现在我们又有这样一个需求：实现一个函数，将一个整形数组中比 50 小的打印在控制台。”What the fuck!”对于提需求者，你可能此时的心情是这样：

</img>
然而回到现实，这种需求是不可避免的，你可能想过复制粘贴，更改一下判断条件，然而作为开发者，我们要未雨绸缪，要考虑到将来可能添加更多类似的需求，那么你将会有大量的重复代码，使你的项目变得臃肿，所以这个时候我们需要冷静下来思考，其实这两个需求很多代码都是相同的，只要更改一下判断条件即可，而判断条件我们如何变得更加灵活呢？这时候我们就用到回调函数的知识了，我们可以定义一个函数，这个函数需要两个 int 型参数，函数内部实现代码是将两个整形数字做比较，将比较结果的 bool 值作为函数的返回值返回出来，以大于被比较数字的情况为例：

```
BOOL compareGreater(int number, int compareNumber) {

    return number > compareNumber;

}
```

同理，小于被比较的数字函数定义如下：

```
BOOL compareLess(int number, int compareNumber) {

    return number < compareNumber;

}
```

接下来，我们可以将这个函数作为 compareNumberFunction 的一个参数进行传递（没错，函数可以作为参数），那么我们就需要一个函数指针获取函数的地址，从而在 compareNumberFunction 内部进行对函数的调用，于是，compareNumberFunction 函数的定义变成了这样：

```
void compareNumberFunction(int *numberArray, int count, int compareNumber, BOOL
(*p)(int, int)) {
```

```
    for (int i = 0; i < count; i++) {
        if (p(*(numberArray + i), compareNumber)) {
            printf("%d\n", *(numberArray + i));
        }
    }
}
```

具体使用时代吗如下：

```
int main() {

    int numberArray[5] = {15, 34, 44, 56, 64};
    int compareNumber = 50;
    // 大于被比较数字情况：
    compareNumberFunction(numberArray, 5, compareNumber, compareGreater);
    // 小于被比较数字情况：
    compareNumberFunction(numberArray, 5, compareNumber, compareLess);

    return 0;
}
```

根据上述案例，我们可以得出结论：函数回调本质为函数指针作为函数参数，函数调用时传入函数地址，这使我们的代码变得更加灵活，可复用性更强。

动态排序

上面的案例如果你已经理解的话那么动态排序其实你已经懂了。首先我们应该理解动态这个词，我的理解就是不同时刻，不同场景，发生不同的事，这就是动态。话不多说，直接上案例。
需求： 有 30 个学生需要排序
按成绩排
按年龄排
…
这种无法预测的需求变更，就是我们上文说的动态场景，那么解决方案就是函数回调：

```
typedef struct student{

    char name[20];

    int age;

    float score;

}Student;



//比较两个学生的年龄

BOOL compareByAge(Student stu1, Student stu2) {

    return stu1.age > stu2.age ? YES : NO;

}

//比较两个学生的成绩

BOOL compareByScore(Student stu1, Student stu2) {
```

```
    return stu1.score > stu2.score ? YES : NO;
}
void sortStudents(Student *array, int n, BOOL(*p)(Student, Student)) {
    Student temp;
    int flag = 0;
    for (int i = 0; i < n - 1 && flag == 0; i++) {
        flag = 1;
        for (int j = 0; j < n - i - 1; j++) {
            if (p(array[j], array[j + 1])) {
                temp = array[j];
                array[j] = array[j + 1];
                array[j + 1] = temp;
                flag = 0;
            }
        }
    }
}
int main() {


    Student stu1 = {"小明", 19, 98};
    Student stu2 = {"小红", 20, 78};
    Student stu3 = {"小白", 21, 88};
```

```
    Student stuArray[3] = {stu1, stu2, stu3};

    sortStudents(stuArray, 3, compareByScore);



    return 0;

}
```

没错，动态排序就是这么简单！

函数指针作为函数返回值

没错，既然函数指针可以作为参数，自然也可以作为返回值。再接着上案例。
需求：定义一个函数，通过传入功能的名称获取到对应的函数。

功能名(name)	调用函数(function)
“max”	maxValue
“min"	minValue

整理一下发型，然后我们分析下需求，当前我们需要定义一个叫做 findFunction 的函数，这个函数传入一个字符串之后会返回一个 int (*)(int, int) 类型的函数指针，那么我们这个函数的声明是不是可以写成这样呢？

```
int (*)(int, int) findFunction(char *);
```

这看起来很符合我们的理解，然而，这并不正确，编译器无法识别两个完全并行的包含形参的括号(int, int)和(char *),真正的形式其实是这样：

```
int (*findFunction(char *))(int, int);
```

这种声明从外观上看更像是脸滚键盘出来的结果，现在让我们来逐步的分析一下这个声明的组成步骤：

1. findFunction 是一个标识符

- findFunction()是一个函数
- findFunction(char *)函数接受一个类型为 char *的参数
- *findFunction(char *)函数返回一个指针
- (*findFunction(char*))()这个指针指向一个函数
- (*findFunction(char*))(int, int)指针指向的函数接受两个整形参数
- int (*findFunction(char *))(int, int)指针指向的函数返回一个整形

现在我们的分析已经完成了，编译器可以通过了，现在程序员疯了，这对我们来说就像鲱鱼罐头一样难以下咽，那么我们是不是有更好的书写方式呢？（老司机友情提示：typedef）

最终代码演变成了这样：

```
// 重定义函数指针类型

typedef int (*FUNC)(int, int);



// 求最大值函数
```

```c
int maxValue(int a, int b) {
    return a > b ? a : b;
}

// 求最小值函数
int minValue(int a, int b) {
    return a < b ? a : b;
}
// findFunction 函数定义
FUNC findFunction(char *name) {
    if (0 == strcmp(name, "max")) {
        return maxValue;
    } else if (0 == strcmp(name, "min")) {
        return minValue;
    }

    printf("Function name error");
    return NULL;
}

int main() {
```

```
    int (*p)(int, int) = findFunction("max");

    printf("%d\n", p(3, 5));



    int (*p1)(int, int) = findFunction("min");

    printf("min = %d\n", p1(3, 5));



    return 0;

}
```

到了这里，函数指针的基础内容已经结束了，有的同学还有可能困惑，为什么我要以函数去获取函数呢，直接使用 maxValue 和 minValue 不就好了么，其实在以后的编程过程中，很有可能 maxValue 和 minValue 被封装了起来，类的外部是不能直接使用的，那么我们就需要这种方式，如果你学习了 Objective-C 你会发现，所有的方法调用的实现原理都是如此。

函数指针数组

现在我们应该清楚表达式“char * (*pf)(char * p)”定义的是一个函数指针 pf。既然 pf 是一个指针，那就可以储存在一个数组里。把上式修改一下：
char * (*pf[3])(char * p);
这是定义一个函数指针数组。它是一个数组，数组名为 pf，数组内存储了 3 个指向函数的指针。这些指针指向一些返回值类型为指向字符的指针、参数为一个指向字符的指针的函数。这念起来似乎有点拗口。不过不要紧，关键是你明白这是一个指针数组，是数组。

函数指针数组怎么使用呢？给一个非常简单的例子，只要真正掌握了使用方法，再复杂的问题都可以应对。如下：

```
char * fun1(char * p)

{
```

```c
    printf("%s\n",p);
    return p;
}
char * fun2(char * p)
{
    printf("%s\n",p);
    return p;
}
char * fun3(char * p)
{
    printf("%s\n",p);
    return p;
}
int main(){
    char * (*pf[3])(char * p);
    pf[0] = fun1; // 可以直接用函数名
    pf[1] = &fun2; // 可以用函数名加上取地址符
    pf[2] = &fun3;
    pf[0]("fun1");
    pf[0]("fun2");
    pf[0]("fun3");
    return 0;
```

```
}
```

是不是感觉上面的例子太简单，不够刺激？好，那就来点刺激的。

函数指针数组的指针

看着这个标题没发狂吧？函数指针就够一般初学者折腾了，函数指针数组就更加麻烦，现在的函数指针数组指针就更难理解了。

其实，没这么复杂。前面详细讨论过数组指针的问题，这里的函数指针数组指针不就是一个指针嘛。只不过这个指针指向一个数组，这个数组里面存的都是指向函数的指针。仅此而已。

下面就定义一个简单的函数指针数组指针：
char * (*(*pf)[3])(char * p);
注意，这里的 pf 和上面的 pf 就完全是两码事了。上一节的 pf 并非指针，而是一个数组名；这里的 pf 确实是实实在在的指针。这个指针指向一个包含了 3 个元素的数组；这个数字里面存的是指向函数的指针；这些指针指向一些返回值类型为指向字符的指针、参数为一个指向字符的指针的函数。这比面的函数指针数组更拗口。其实你不用管这么多，明白这是一个指针就 ok 了。其用法与前面讲的数组指针没有差别。下面列一个简单的例子：

```
char * fun1(char * p)

{

   printf("%s\n",p);

   return p;

}

char * fun2(char * p)

{

   printf("%s\n",p);

   return p;

}
```

```c
char * fun3(char * p)

{

   printf("%s\n",p);

   return p;

}

int main(){

   char * (*a[3])(char * p);

   char * (*(*pf)[3])(char * p);

   pf = &a;

   a[0] = fun1;

   a[1] = &fun2;

   a[2] = &fun3;

   pf[0][0]("fun1");

   pf[0][1]("fun2");

   pf[0][2]("fun3");

   return 0;

}
```

好了，到了这里 C 语言已经没有什么能阻挡你了。