第14讲：深入理解指针(4)

目录

正文开始

1. 字符指针变量

在指针的类型中我们知道有一种指针类型为字符指针 `char*`；

一般使用:

代码块

```
int main()
{
    char ch = 'w';
    char *pc = &ch;
    *pc = 'w';
    return 0;
}
```

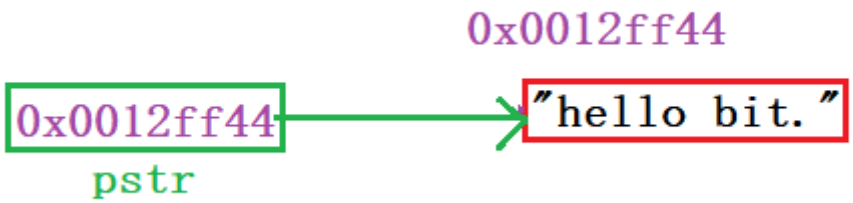
还有一种使用方式如下：

代码块

```
int main()
{
    const char* pstr = "hello bit.";//这里是把一个字符串放到pstr指针变量里了吗?
    printf("%s\n", pstr);
```

```
    return 0;
}
```

代码 `const char* pstr = "hello bit.";` 特别容易让同学以为是把字符串 `hello bit` 放到字符指针 `pstr` 里了，但是本质是把字符串 `hello bit.` 首字符的地址放到了pstr中。



上面代码的意思是把一个常量字符串的首字符 `h` 的地址存放到指针变量 `pstr` 中。

《剑指offer》中收录了一道和字符串相关的笔试题，我们一起来学习一下：

代码块

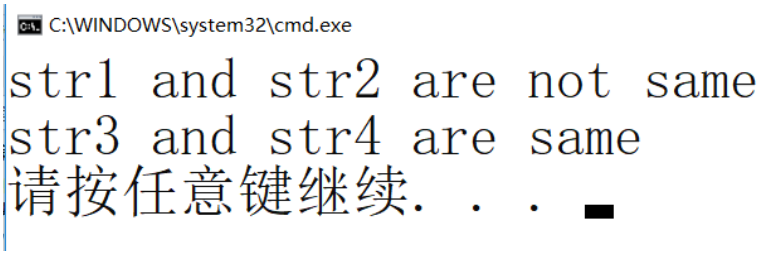
```c
#include <stdio.h>

int main()
{
    char str1[] = "hello bit.";
    char str2[] = "hello bit.";
    const char *str3 = "hello bit.";
    const char *str4 = "hello bit.";

    if(str1 ==str2)
        printf("str1 and str2 are same\n");
    else
        printf("str1 and str2 are not same\n");

    if(str3 ==str4)
        printf("str3 and str4 are same\n");
    else
        printf("str3 and str4 are not same\n");

    return 0;
}
```



这里str3和str4指向的是一个同一个常量字符串。C/C++会把常量字符串存储到单独的一个内存区域，当几个指针指向同一个字符串的时候，他们实际会指向同一块内存。但是用相同的常量字符串去初始化不同的数组的时候就会开辟出不同的内存块。所以str1和str2不同，str3和str4相同。

2. 数组指针变量

2.1 数组指针变量是什么?

之前我们学习了指针数组，指针数组是一种数组，数组中存放的是地址（指针）。

数组指针变量是指针变量？还是数组?

答案是：指针变量。

我们已经熟悉：

- **整形指针变量**：`int * pint;` 存放的是整形变量的地址，能够指向整形数据的指针。
- **浮点型指针变量**：`float * pf;` 存放浮点型变量的地址，能够指向浮点型数据的指针。

那数组指针变量应该是：存放的应该是数组的地址，能够指向数组的指针变量。

下面代码哪个是数组指针变量?

代码块

```
int * p1[10];
int (*p2)[10];
```

思考一下：p1, p2分别是什么?

数组指针变量

代码块

```
int (*p)[10];
```

解释：`p` 先和 `*` 结合，说明 `p` 是一个指针变量，然后指针指向的是一个大小为10个整型的数组。所以 `p` 是一个指针，指向一个数组，叫 **数组指针**。

这里要注意：`[]` 的优先级要高于 `*` 号的，所以必须加上 `()` 来保证 `p` 先和 `*` 结合。

2.2 数组指针变量怎么初始化

数组指针变量是用来存放数组地址的，那怎么获得数组的地址呢？就是我们之前学习的 `&数组名` 。

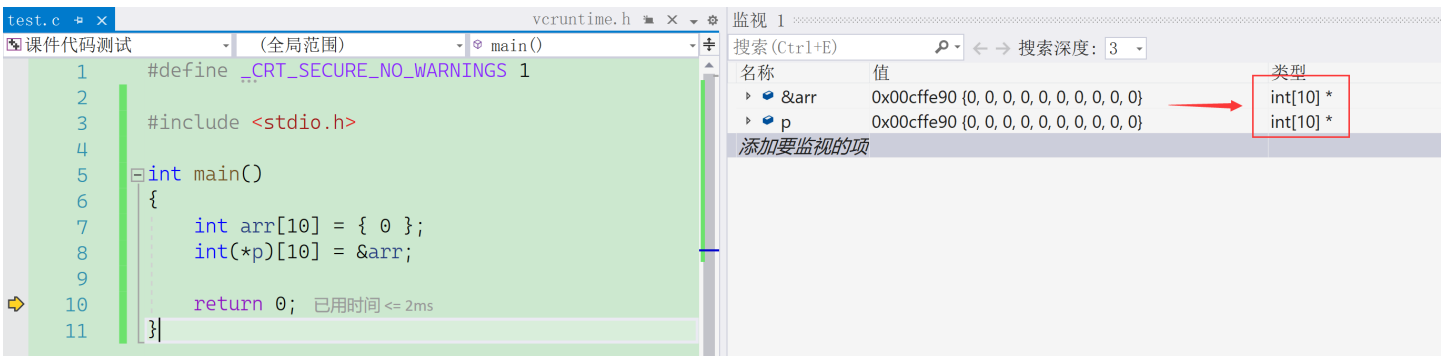
代码块

```
int arr[10] = {0};
&arr;//得到的就是数组的地址
```

如果要存放个数组的地址，就得存放在**数组指针变量**中，如下：

代码块

```
int(*p)[10] = &arr;
```



我们调试也能看到 `&arr` 和 `p` 的类型是完全一致的。

数组指针类型解析：

代码块

```
int  (* p)  [10] = &arr;
 |      |      |
 |      |      |
 |      |      p指向数组的元素个数
 |      p是数组指针变量名
 p指向的数组的元素类型
```

3. 二维数组传参的本质

有了数组指针的理解，我们就能够讲一下二维数组传参的本质了。

过去我们有一个二维数组的需要传参给一个函数的时候，我们是这样写的：

代码块

```
#include <stdio.h>

void test(int a[3][5], int r, int c)
{
    int i = 0;
    int j = 0;
    for(i = 0; i < r; i++)
    {
        for(j = 0; j < c; j++)
        {
            printf("%d ", a[i][j]);
        }
        printf("\n");
    }
}

int main()
{
    int arr[3][5] = {{1,2,3,4,5}, {2,3,4,5,6},{3,4,5,6,7}};
    test(arr, 3, 5);
    return 0;
}
```

这里实参是二维数组，形参也写成二维数组的形式，那还有什么其他的写法吗？

首先我们再次理解一下二维数组，二维数组其实可以看做是每个元素是一维数组的数组，也就是二维数组的每个元素是一个一维数组。那么二维数组的首元素就是第一行，是个一维数组。

如下图：

	0	1	2	3	4
0	1	2	3	4	5
1	2	3	4	5	6
2	3	4	5	6	7

arr数组

所以，根据数组名是数组首元素的地址这个规则，二维数组的数组名表示的就是第一行的地址，是一维数组的地址。根据上面的例子，第一行的一维数组的类型就是 `int [5]` ，所以第一行的地址的类型就是数组指针类型 `int(*)[5]` 。那就意味着**二维数组传参本质上也是传递了地址，传递的是第一行这个一维数组的地址**，那么形参也是可以写成指针形式的。如下：

代码块

```c
#include <stdio.h>

void test(int (*p)[5], int r, int c)
{
    int i = 0;
    int j = 0;
    for(i = 0; i < r; i++)
    {
        for(j = 0; j < c; j++)
        {
            printf("%d ", *(*(p+i)+j));
        }
        printf("\n");
    }
}

int main()
{
    int arr[3][5] = {{1,2,3,4,5}, {2,3,4,5,6},{3,4,5,6,7}};
    test(arr, 3, 5);
    return 0;
}
```

总结：二维数组传参，形参的部分可以写成数组，也可以写成指针形式。

4. 函数指针变量

4.1 函数指针变量的创建

什么是函数指针变量呢?

根据前面学习整型指针，数组指针的时候，我们的类比关系，我们不难得出结论：

函数指针变量应该是用来存放函数地址的，未来通过地址能够调用函数的。

那么函数是否有地址呢?

我们做个测试：

代码块

```
#include <stdio.h>
void test()
{
    printf("hehe\n");
}
int main()
{
    printf("test:  %p\n", test);
    printf("&test: %p\n", &test);
    return 0;
}
```

输出结果如下：

代码块

```
test:  005913CA
&test: 005913CA
```

确实打印出来了地址，所以函数是有地址的，函数名就是函数的地址，当然也可以通过 `&函数名` 的方式获得函数的地址。

如果我们要将函数的地址存放起来，就得创建函数指针变量咯，函数指针变量的写法其实和数组指针非常类似。如下：

代码块

```
void test()
{
    printf("hehe\n");
}

void (*pf1)() = &test;
void (*pf2)()= test;

int Add(int x, int y)
{
    return x+y;
}

int(*pf3)(int, int) = Add;
int(*pf3)(int x, int y) = &Add;//x和y写上或者省略都是可以的
```

函数指针类型解析：

```
代码块
1   int      (* pf3)   (int x, int y)
2    |          |      ------------
3    |          |             |
4    |          |             pf3指向函数的参数类型和个数的交代
5    |          函数指针变量名
6    pf3指向函数的返回类型
7
8    int (*) (int x, int y)  //pf3函数指针变量的类型
```

4.2 函数指针变量的使用

通过函数指针调用指针指向的函数。

```
代码块
1   #include <stdio.h>
2
3   int Add(int x, int y)
4   {
5       return x+y;
6   }
7
8   int main()
9   {
10      int(*pf3)(int, int) = Add;
11
12      printf("%d\n", (*pf3)(2, 3));
13      printf("%d\n", pf3(3, 5));
14      return 0;
15  }
```

输出结果：

```
代码块
1   5
2   8
```

4.3 两段有趣的代码

代码1

```
代码块
1   (*(void (*)())0)();
```

代码解析：

1. 上述代码其实是一次函数调用，调用的是0地址处的一个函数，这个函数没有参数，没有返回值。
2. 代码中的 `void (*)()` 是函数指针类型， `(void (*)())0` 类型放在括号中意思是强制类型转化，是将0这个整型值，强制类型转化成这种函数指针类型，也就是说0被当做函数的地址了。
3. `*(void (*)())0` ，前面加一个 `*` ，就是调用0地址处的这个函数，根据函数指针的类型能知道，这个函数没有参数，也没有返回值。

代码2

```
代码块
1   void ( *signal(int , void(*)(int)) )(int);
```

代码解析：

1. 上述代码是一次函数的声明。
2. 声明的函数名字叫 `signal` ，函数的参数有2个，第一个是int类型，第二个是函数指针类型: `void(*)(int)` 。 `signal` 函数的返回值类型也是函数指针类型 `void(*)(int)` 。

> 上述两段代码均出自：《C陷阱和缺陷》这本书

4.3.1 typedef 关键字

`typedef` 是用来类型重命名的，可以将复杂的类型，简单化。

比如，你觉得 `unsigned int` 写起来不方便，如果能写成 `uint` 就方便多了，那么我们可以使用：

```
代码块
1   typedef unsigned int uint;
2   //将unsigned int 重命名为uint
```

如果是指针类型，能否重命名呢？其实也是可以的，比如，将 `int*` 重命名为 `ptr_t` ,这样写：

代码块

```
typedef int* ptr_t;
```

但是对于数组指针和函数指针稍微有点区别：

比如我们有数组指针类型 `int(*)[5]` ,需要重命名为 `parr_t` ，那可以这样写：

代码块

```
typedef int(*parr_t)[5]; //新的类型名必须在*的右边
```

函数指针类型的重命名也是一样的，比如，将 `void(*)(int)` 类型重命名为 `pf_t` ,就可以这样写：

代码块

```
typedef void(*pfun_t)(int);//新的类型名必须在*的右边
```

那么要简化代码2，可以这样写：

代码块

```
typedef void(*pfun_t)(int);
pfun_t signal(int, pfun_t);
```

5. 函数指针数组

数组是一个存放相同类型数据的存储空间，我们已经学习了指针数组，

比如：

代码块

```
int * arr[10];
//数组的每个元素是int*
```

那要把函数的地址存到一个数组中，那这个数组就叫**函数指针数组**，那函数指针的数组如何定义呢？

代码块

```
int (*parr1[3])();
int *parr2[3]();
```

```
3   int (*)() parr3[3];
```

答案是：parr1

`parr1` 先和 `[]` 结合，说明 parr1是数组，数组的内容是什么呢?

是 `int (*)()` 类型的函数指针。

6. 转移表

函数指针数组的用途：**转移表**

举例：计算器的一般实现：

代码块

```
1   #include <stdio.h>
2   int add(int a, int b)
3   {
4       return a + b;
5   }
6   int sub(int a, int b)
7   {
8       return a - b;
9   }
10  int mul(int a, int b)
11  {
12      return a * b;
13  }
14  int div(int a, int b)
15  {
16      return a / b;
17  }
18  int main()
19  {
20      int x, y;
21      int input = 1;
22      int ret = 0;
23      do
24      {
25          printf("*************************\n");
26          printf("  1:add           2:sub  \n");
27          printf("  3:mul           4:div  \n");
28          printf("  0:exit                 \n");
29          printf("*************************\n");
30          printf("请选择: ");
31          scanf("%d", &input);
```

```
        switch (input)
        {
        case 1:
            printf("输入操作数: ");
            scanf("%d %d", &x, &y);
            ret = add(x, y);
            printf("ret = %d\n", ret);
            break;
        case 2:
            printf("输入操作数: ");
            scanf("%d %d", &x, &y);
            ret = sub(x, y);
            printf("ret = %d\n", ret);
            break;
        case 3:
            printf("输入操作数: ");
            scanf("%d %d", &x, &y);
            ret = mul(x, y);
            printf("ret = %d\n", ret);
            break;
        case 4:
            printf("输入操作数: ");
            scanf("%d %d", &x, &y);
            ret = div(x, y);
            printf("ret = %d\n", ret);
            break;
        case 0:
            printf("退出程序\n");
            break;
        default:
            printf("选择错误\n");
            break;
        }
    } while (input);

    return 0;
}
```

使用函数指针数组的实现：

代码块

```
#include <stdio.h>
int add(int a, int b)
{
    return a + b;
```

```c
}
int sub(int a, int b)
{
    return a - b;
}
int mul(int a, int b)
{
    return a*b;
}
int div(int a, int b)
{
    return a / b;
}
int main()
{
    int x, y;
    int input = 1;
    int ret = 0;
    int(*p[5])(int x, int y) = { 0, add, sub, mul, div }; //转移表
    do
    {
        printf("************************\n");
        printf("  1:add           2:sub  \n");
        printf("  3:mul           4:div  \n");
        printf("  0:exit                 \n");
        printf("************************\n");
        printf( "请选择: " );
        scanf("%d", &input);
        if ((input <= 4 && input >= 1))
        {
            printf( "输入操作数: " );
            scanf( "%d %d", &x, &y);
            ret = (*p[input])(x, y);
            printf( "ret = %d\n", ret);
        }
        else if(input == 0)
        {
            printf("退出计算器\n");
        }
        else
        {
            printf( "输入有误\n" );
        }
    }while (input);
    return 0;
}
```

完