> 课题:
> 周四：C语言基础04-函数指针
> 指针函数与函数指针
> 函数指针传值，
> 函数执行过程与函数回调
> 函数参数详解

1 指针

指针自身类型

> **你只要把指针声明语句里的指针名字去掉，剩下的部分就是这个指针的类型。**

```
int * ptr;//指针的类型是int *
```

指针所指向的类型

> 当你通过指针来访问指针所指向的内存区时，指针所指向的类型决定了编译器将把那片内存区里的内容当做什么来看待。
> 从语法上看，**你只须把指针声明语句中的指针名字和名字左边的指针声明符*去掉**，剩下的就是指针所指
> 向的类型

```
int * ptr; //指针所指向的类型是int
```

二维数组在概念上是二维的，有行和列，但在内存中所有的数组元素都是连续排列的，它们之间没有“缝隙”。以下面的二维数组 a 为例：

int a[3][4] = { {0, 1, 2, 3}, {4, 5, 6, 7}, {8, 9, 10, 11} };

从概念上理解，a 的分布像一个矩阵：

```
0   1   2   3
4   5   6   7
8   9  10  11
```

但在内存中，a 的分布是一维线性的，整个数组占用一块连续的内存：

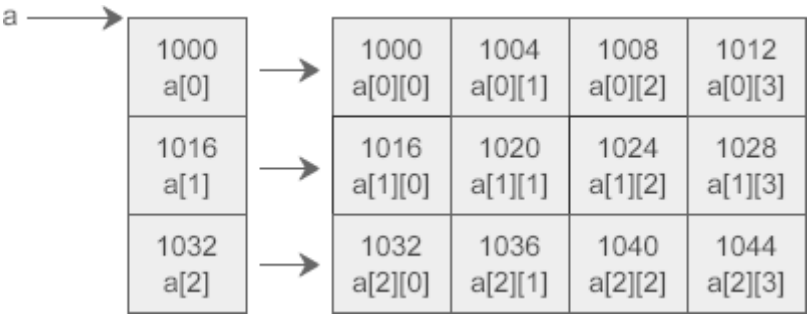
0	1	2	3	4	5	6	7	8	9	10	11

1.1 指针数组内存分布

C语言中的二维数组是按行排列的，也就是先存放 a[0] 行，再存放 a[1] 行，最后存放 a[2] 行；每行中的 4 个元素也是依次存放。数组 a 为 int 类型，每个元素占用 4 个字节，整个数组共占用 4×(3×4) = 48 个字节。

C语言允许把一个二维数组分解成多个一维数组来处理。对于数组 a，它可以分解成三个一维数组，即 a[0]、a[1]、a[2]。每一个一维数组又包含了 4 个元素，例如 a[0] 包含 a[0][0]、a[0][1]、a[0][2]、a[0][3]。

假设数组 a 中第 0 个元素的地址为 1000，那么每个一维数组的首地址如下图所示：



为了更好的理解指针和二维数组的关系，我们先来定义一个指向 a 的指针变量 p：

int (*p)[4] = a;

括号中的 `*` 表明 p 是一个指针，它指向一个数组，数组的类型为 `int [4]`，这正是 a 所包含的每个一维数组的类型。

`[ ]` 的优先级高于 `*`，`( )` 是必须要加的，如果赤裸裸地写作 `int *p[4]`，那么应该理解为 `int *(p[4])`，p 就成了一个指针数组，而不是二维数组指针，这在《C语言指针数组》中已经讲到。

对指针进行加法（减法）运算时，它前进（后退）的步长与它指向的数据类型有关，p 指向的数据类型是 `int [4]`，那么 `p+1` 就前进 4×4 = 16 个字节，`p-1` 就后退 16 个字节，这正好是数组 a 所包含的每个一维数组的长度。也就是说，`p+1` 会使得指针指向二维数组的下一行，`p-1` 会使得指针指向数组的上一行。

数组名 a 在表达式中也会被转换为和 p 等价的指针！

下面我们就来探索一下如何使用指针 p 来访问二维数组中的每个元素。按照上面的定义：
1) `p` 指向数组 a 的开头，也即第 0 行；`p+1` 前进一行，指向第 1 行。

2) `*(p+1)` 表示取地址上的数据，也就是整个第 1 行数据。注意是一行数据，是多个数据，不是第 1 行中的第 0 个元素，下面的运行结果有力地证明了这一点：

```
#include <stdio.h>
int main(){
    int a[3][4] = { {0, 1, 2, 3}, {4, 5, 6, 7}, {8, 9, 10, 11} };
    int (*p)[4] = a;
    printf("%d\n", sizeof(*(p+1)));
    return 0;
}
```

运行结果：

```
16
```

> `*(p+1)+1` 表示第 1 行第 1 个元素的地址。如何理解呢？

`*(p+1)` 单独使用时表示的是第 1 行数据，放在表达式中会被转换为第 1 行数据的首地址，也就是第 1 行第 0 个元素的地址，因为使用整行数据没有实际的含义，编译器遇到这种情况都会转换为指向该行第 0 个元素的指针；就像一维数组的名字，在定义时或者和 sizeof、& 一起使用时才表示整个数组，出现在表达式中就会被转换为指向数组第 0 个元素的指针。

4) `*(*(p+1)+1)` 表示第 1 行第 1 个元素的值。很明显，增加一个 * 表示取地址上的数据。

根据上面的结论，可以很容易推出以下的等价关系：

```
a+i == p+i
a[i] == p[i] == *(a+i) == *(p+i)
a[i][j] == p[i][j] == *(a[i]+j) == *(p[i]+j) == *(*(a+i)+j) == *(*(p+i)+j)
```

【实例】使用指针遍历二维数组。

```
    for(int i=0; i<3; i++){
        for(int j=0; j<4; j++)
        printf("value :%d\n",*(*(p+i)+j));
    }
```

运行结果：

```
 0   1   2   3
 4   5   6   7
 8   9  10  11
```

1.2 指针数组和二维数组指针的区别

指针数组和二维数组指针在定义时非常相似，只是括号的位置不同：

```
int *(p1[5]);  //指针数组，可以去掉括号直接写作 int *p1[5];int (*p2)[5];  //二维数组指针，不能去掉括号
```

指针数组和二维数组指针有着本质上的区别：

指针数组是一个数组，只是每个元素保存的都是指针，

以上面的 p1 为例，在32位环境下它占用 4×5 = 20 个字节的内存。

二维数组指针是一个指针，它指向一个二维数组，以上面的 p2 为例，它占用 4 个字节的内存。

> 1. 内存上不一样
> 2. 步长不一样
> 3. 指针类型也不一样
> 4. 指向内容也不一样

2 函数定义

2.1 什么是函数指针

如果在程序中定义了一个函数，那么在编译时系统就会为这个函数代码分配一段存储空间，这段存储空间的首地址称为这个函数的地址。而且函数名表示的就是这个地址。既然是地址我们就可以定义一个指针变量来存放，这个指针变量就叫作函数指针变量，**简称函数指针。**

那么这个指针变量怎么定义呢？虽然同样是指向一个地址，但指向函数的指针变量同我们之前讲的指向变量的指针变量的定义方式是不同的。

例如：

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

> **这个语句就定义了一个指向函数的指针变量 p。首先它是一个指针变量，所以要有一个“ * ”，即（ * p）；**

其次前面的 int 表示这个指针变量可以指向**返回值**类型为 **int 型的函数**；

后面括号中的两个 int 表示这个指针变量可以指向有**两个参数且都是 int 型的函数。**

所以合起来这个语句的意思就是：定义了一个指针变量 p，该指针变量可以指向返回值类型为 int 型，且有两个整型参数的函数。

> p 的类型为 int(*)(int，int)。

所以函数指针的定义方式为：

函数返回值类型 (* 指针变量名) (函数参数列表);

> “函数返回值类型”表示该指针变量可以指向具有什么返回值类型的函数；
>
> “函数参数列表”表示该指针变量可以指向具有什么参数列表的函数。
>
> 这个参数列表中只需要写函数的参数类型即可。

我们看到，函数指针的定义就是将“函数声明”中的“函数名”改成“（* 指针变量名）”。

但是这里需要注意的是：“（ * 指针变量名）**”两端的括号不能省略**，

括号改变了运算符的优先级。如果省略了括号，就不是定义函数指针而是一个函数声明了，即声明了一个返回值类型为指针型的函数。

> 那么怎么判断一个指针变量是指向变量的指针变量还是指向函数的指针变量呢?

首先看变量名前面有没有“ * ”，如果有“*”说明是指针变量；

其次看变量名的后面有没有带有形参类型的圆括号，

> 如果有就是指向函数的指针变量，即函数指针，
>
> 如果没有就是指向变量的指针变量。

最后需要注意的是，指向函数的指针变量没有 ++ 和 -- 运算。

2.2如何用函数指针调用函数

给大家举一个例子：

```
int Func(int x);   /*声明一个函数*/
int (*p) (int x);  /*定义一个函数指针*/
p = Func;          /*将Func函数的首地址赋给指针变量p*/
```

赋值时函数 Func 不带括号，也不带参数。由于函数名 Func 代表函数的首地址，因此经过赋值以后，指针变量 p 就指向函数 Func() 代码的首地址了。

下面来写一个程序，看了这个程序你们就明白函数指针怎么使用了：

```
# include <stdio.h>
int Max(int, int);  //函数声明
int main(void)
{
    int(*p)(int, int);  //定义一个函数指针
    int a, b, c;
    p = Max;  //把函数Max赋给指针变量p, 使p指向Max函数
```

```
    printf("please enter a and b:");
    scanf("%d%d", &a, &b);
    c = (*p)(a, b);  //通过函数指针调用Max函数
    printf("a = %d\nb = %d\nmax = %d\n", a, b, c);
    return 0;
}
int Max(int x, int y)  //定义Max函数
{
    int z;
    if (x > y)
    {
        z = x;
    }
    else
    {
        z = y;
    }
    return z;
}
```

输出结果是:

```
please enter a and b:3 4
a = 3
b = 4
max = 4
```

3 函数指针数组

3.1 函数指针数组

语文要学好.
这三个词我们扩充一下

```
数组            int Array[N];
指针的数组      int *Array[N];
函数的指针的数组 int (*function_pointer)[N](int,int);
```

按照中文的习惯,函数指针数组应该就是函数的指针的数组的简写了.

从低级一点点进化到高级

3.2 数组

```
int a[2];
数组就是这样,没什么好说的了.就是有两个格子,里面存了两个int类型的数字.这两个格子被看成a.大小使用sizeof(a)来计算,就是两个int的大小.约定一个int大小是4个字节,那么sizeof(a)就是8个字节大小了.
```

3.3指针的数组（指针数组）

```
int *p[2];
难度加大了一点,但是我们冷静一下,还是可以理解的.
这还是一个数组,还是两个格子,每个格子里存的是指向int类型的指针.
每个指针的大小也约定是4个字节.那么sizeof(p);也就是8个字节了.
格子p1,格子p2
这两个格子分别指向int类型的数字. `p1->&a;p2->&b;`
指针->这个符号是指针专用的.只有指针才能使用->.
```

指向数组的指针（数组指针）

```
int (*p)[2];
难度加大了一点,但是我们冷静一下,还是可以理解的.（）的优先级高于[]
可以确定的是 他在内存中是一个单个指针，指针类型 和int *没有区别。而指向类型变成了数组，
这还是一个数组,还是两个格子,每个格子里存的是指向int类型
所以指向数组的指针，也成为数组指针
```

3.4 函数的指针的数组

```
按照道理里将应该写成  int (*func)(int a)[2];
但是应该写成          int (*func[N])(int, int);
开始变得有意思了.我们读一下这个东西.首先看哪里呢?
我们先不看哪里,我们先来把这一大串东西分解一下.
int  (*func[2]) (int a);好了,这么一看就明白了.
先读标识符,和他相邻的是指针符号,那么func是一个指针,指向的是一个函数，、、、
然后往右,是个数组,那么func应该指向一个数组.那这个数组里存的是什么样的东西呢?
存的是 参数是一个int,返回值是一个int类型的函数.意思是这两个格子里存的是两个函数的地址.
```

3.5 解释函数指针

我尝试说明一下这个类型是什么,为什么有这样的类型.

> 参数是一个int,返回值是一个int类型的函数 的类型

看到类型这两个字,你是怎么理解的?
在我看来,类型就是模型,就是模板,是一个蓝图,一个模具.
模具有什么特点?

> 我们有一个米老鼠的模具,使用橡皮泥塞进去,就能倒出来一个米老鼠了.
> 如果别人拿了一个不一样的米老鼠过来,你使用这个模具是不能卡进去的.
> 但是使用这个模具倒出来的米老鼠是正好可以卡进去的.

函数的类型就是描述这个函数长什么样子.

描述的要素有

> 返回值类型
>
> 参数个数
>
> 参数类型

要这三个方面一一匹配,我们就函数是同一个类型的.

比如 int (*func)(int a, int b); 这就是一个函数模板,**我们可以把这样的函数叫做米老鼠模板**.

如果不给这个模板起名字,我们就说形如这样的函数就叫做 返回值是一个int类型的指针,参数为int a, int b 的类型.
这么叫是不是太费劲了?现在我们就给它起名字叫米老鼠.
使用typedef来进行起名字.

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

好了,现在我们定义了一个新的函数指针类型叫milaoshu.

milaoshu类型是一个`指向 返回值是int 的有两个int参数的函数 的指针`的类型.

那我们就使用这个milaoshu类型进行声明吧.

```
milaoshu p = NULL;`
```

我们声明了一个milaoshu类型的指针变量p.
这个p目前是指向空的.
除了指向空,它还可以指向milaoshu的类型.
那就是p = func;
调用方式 p(a,b) 就等同于func(a,b);

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

typedef int (*milaoshu)(int a, int b);

int func(int a, int b){
    return a+b;
}
int main() {

    int a = 10;
    int b = 20;
    printf("a+b = %d\n", a+b);
    milaoshu p;  //如果不给这个类型起名叫米老鼠,那么在声明变量p的时候应该这样声明 int (*p)(int, int);
    p = func;
    int c = p(a ,b);
    printf("a+b = %d\n", c);
    return 0;
}
```