

指针

本章重点

1. 指针是什么
2. 指针和指针类型
3. 野指针
4. 指针运算
5. 指针和数组
6. 二级指针
7. 指针数组

正文开始@比特科技

指针是什么？

在计算机科学中，**指针**（Pointer）是编程语言中的一个对象，利用地址，它的值直接指向（points to）存在电脑存储器中另一个地方的值。由于通过地址能找到所需的变量单元，可以说，地址指向该变量单元。因此，将地址形象化的称为“指针”。意思是通过它能找到以它为地址的**内存单元**。

那我们就可以这样理解：

内存

内存	
一个字节	0xFFFFFFFF
一个字节	0xFFFFFFFFE
一个字节	

一个字节	0x00000002
一个字节	0x00000001
一个字节	0x00000000

指针

指针是个变量，存放内存单元的地址（编号）。

那对应到代码：

```
#include <stdio.h>
int main()
{
    int a = 10; //在内存中开辟一块空间
    int *p = &a; //这里我们对变量a，取出它的地址，可以使用&操作符。
                //将a的地址存放在p变量中，p就是一个之指针变量。
    return 0;
}
```

总结：指针就是变量，用来存放地址的变量。（存放在指针中的值都被当成地址处理）。

那这里的问题是：

- 一个小的单元到底是多大？（1个字节）
- 如何编址？

经过仔细的计算和权衡我们发现一个字节给一个对应的地址是比较合适的。

对于32位的机器，假设有32根地址线，那么假设每根地址线在寻址的是产生一个电信号正电/负电（1或者0）

那么32根地址线产生的地址就会是：

```
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000001
...
11111111 11111111 11111111 11111111
```

这里就有2的32次方个地址。

每个地址标识一个字节，那我们就可以给（ $2^{32}\text{Byte} == 2^{32}/1024\text{KB} == 2^{32}/1024/1024\text{MB} == 2^{32}/1024/1024/1024\text{GB} == 4\text{GB}$ ）4G的空闲进行编址。

同样的方法，那64位机器，如果给64根地址线，那能编址多大空间，自己计算。

这里我们就明白：

- 在32位的机器上，地址是32个0或者1组成二进制序列，那地址就得用4个字节的空间来存储，所以指针变量的大小就应该是4个字节。
- 那如果在64位机器上，如果有64个地址线，那一个指针变量的大小是8个字节，才能存放一个地址。

总结：

- 指针是用来存放地址的，地址是唯一标示一块地址空间的。
- 指针的大小在32位平台是4个字节，在64位平台是8个字节。

指针和指针类型

这里我们在讨论一下：指针的类型 我们都知道，变量有不同的类型，整形，浮点型等。那指针有没有类型呢？准确的说：有的。

当有这样的代码：

```
int num = 10;
p = &num;
```

要将&num (num的地址) 保存到p中，我们知道p就是一个指针变量，那它的类型是怎样的呢？我们给指针变量相应的类型。

```
char *pc = NULL;
int *pi = NULL;
short *ps = NULL;
long *pl = NULL;
float *pf = NULL;
double *pd = NULL;
```

这里可以看到，指针的定义方式是：type + *。其实：char* 类型的指针是为了存放 char 类型变量的地址。short* 类型的指针是为了存放 short 类型变量的地址。int* 类型的指针是为了存放 int 类型变量的地址。

那指针类型的意义是什么？

指针+-整数

```
#include <stdio.h>
//演示实例
int main()
{
    int n = 10;
    char *pc = (char*)&n;
    int *pi = &n;

    printf("%p\n", &n);
    printf("%p\n", pc);
    printf("%p\n", pc+1);
    printf("%p\n", pi);
    printf("%p\n", pi+1);
    return 0;
}
```

总结：指针的类型决定了指针向前或者向后走一步有多大（距离）。

指针的解引用

```
//演示实例
#include <stdio.h>

int main()
{
    int n = 0x11223344;
    char *pc = (char *)&n;
    int *pi = &n;
    *pc = 0;    //重点在调试的过程中观察内存的变化。
    *pi = 0;    //重点在调试的过程中观察内存的变化。
    return 0;
}
```

总结：指针的类型决定了，对指针解引用的时候有多大的权限（能操作几个字节）。比如：`char*` 的指针解引用就只能访问一个字节，而 `int*` 的指针的解引用就能访问四个字节。

野指针

概念：野指针就是指针指向的位置是不可知的（随机的、不正确的、没有明确限制的）

野指针成因

1. 指针未初始化

```
#include <stdio.h>
int main()
{
    int *p; //局部变量指针未初始化，默认为随机值
    *p = 20;
    return 0;
}
```

2. 指针越界访问

```
#include <stdio.h>
int main()
{
    int arr[10] = {0};
    int *p = arr;
    int i = 0;
    for(i=0; i<=11; i++)
    {
        //当指针指向的范围超出数组arr的范围时，p就是野指针
        *(p++) = i;
    }
    return 0;
}
```

3. 指针指向的空间释放

这里放在动态内存开辟的时候讲解，这里可以简单提示一下。

如何规避野指针

1. 指针初始化
2. 小心指针越界
3. 指针指向空间释放即使置NULL
4. 指针使用之前检查有效性

```
#include <stdio.h>
int main()
{
    int *p = NULL;
    //....
    int a = 10;
    p = &a;
    if(p != NULL)
    {
        *p = 20;
    }
    return 0;
}
```

指针运算

- 指针+- 整数
- 指针-指针
- 指针的关系运算

指针+-整数

```
#define N_VALUES 5
float values[N_VALUES];
float *vp;
//指针+-整数; 指针的关系运算
for (vp = &values[0]; vp < &values[N_VALUES];)
{
    *vp++ = 0;
}
```

指针-指针

```
int my_strlen(char *s)
{
    char *p = s;
    while(*p != '\0' )
        p++;
    return p-s;
}
```

指针的关系运算

```
for(vp = &values[N_VALUES]; vp > &values[0];)
{
    *--vp = 0;
}
```

代码简化, 这将代码修改如下：

```
for(vp = &values[N_VALUES-1]; vp >= &values[0];vp--)  
{  
    *vp = 0;  
}
```

实际在绝大部分的编译器上是可以顺利完成任务的，然而我们还是应该避免这样写，因为标准并不保证它可行。

标准规定：

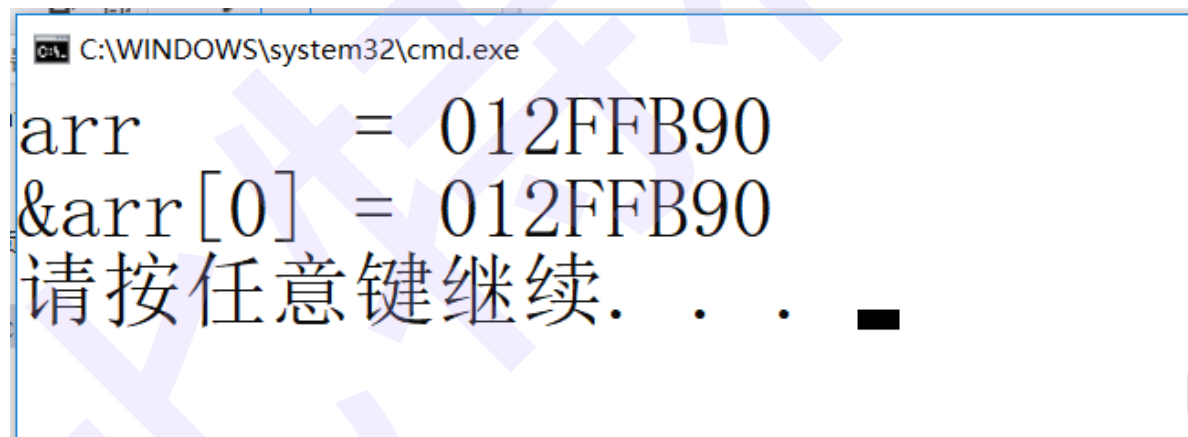
允许指向数组元素的指针与指向数组最后一个元素后面的那个内存位置的指针比较，但是不允许与指向第一个元素之前的那个内存位置的指针进行比较。

指针和数组

数组名是什么？我们看一个例子：

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

运行结果：



```
C:\WINDOWS\system32\cmd.exe  
arr = 012FFB90  
&arr[0] = 012FFB90  
请按任意键继续. . .
```

可见数组名和数组首元素的地址是一样的。

结论：数组名表示的是数组首元素的地址。

那么这样写代码是可行的：

```
int arr[10] = {1,2,3,4,5,6,7,8,9,0};  
int *p = arr; //p存放的是数组首元素的地址
```

既然可以把数组名当成地址存放到一个指针中，我们使用指针来访问一个就成为可能。

例如：

```
#include <stdio.h>

int main()
{
    int arr[] = {1,2,3,4,5,6,7,8,9,0};
    int *p = arr; //指针存放数组首元素的地址
    int sz = sizeof(arr)/sizeof(arr[0]);
    for(i=0; i<sz; i++)
    {
        printf("&arr[%d] = %p    <====> p+%d = %p\n", i, &arr[i], i, p+i);
    }
    return 0;
}
```

运行结果：

```
选择C:\WINDOWS\system32\cmd.exe
&arr[0] = 00F9FA90    <====> p+0 = 00F9FA90
&arr[1] = 00F9FA94    <====> p+1 = 00F9FA94
&arr[2] = 00F9FA98    <====> p+2 = 00F9FA98
&arr[3] = 00F9FA9C    <====> p+3 = 00F9FA9C
&arr[4] = 00F9FAA0    <====> p+4 = 00F9FAA0
&arr[5] = 00F9FAA4    <====> p+5 = 00F9FAA4
&arr[6] = 00F9FAA8    <====> p+6 = 00F9FAA8
&arr[7] = 00F9FAAC    <====> p+7 = 00F9FAAC
&arr[8] = 00F9FAB0    <====> p+8 = 00F9FAB0
&arr[9] = 00F9FAB4    <====> p+9 = 00F9FAB4
请按任意键继续. . .
```

所以 `p+i` 其实计算的是数组 `arr` 下标为 `i` 的地址。

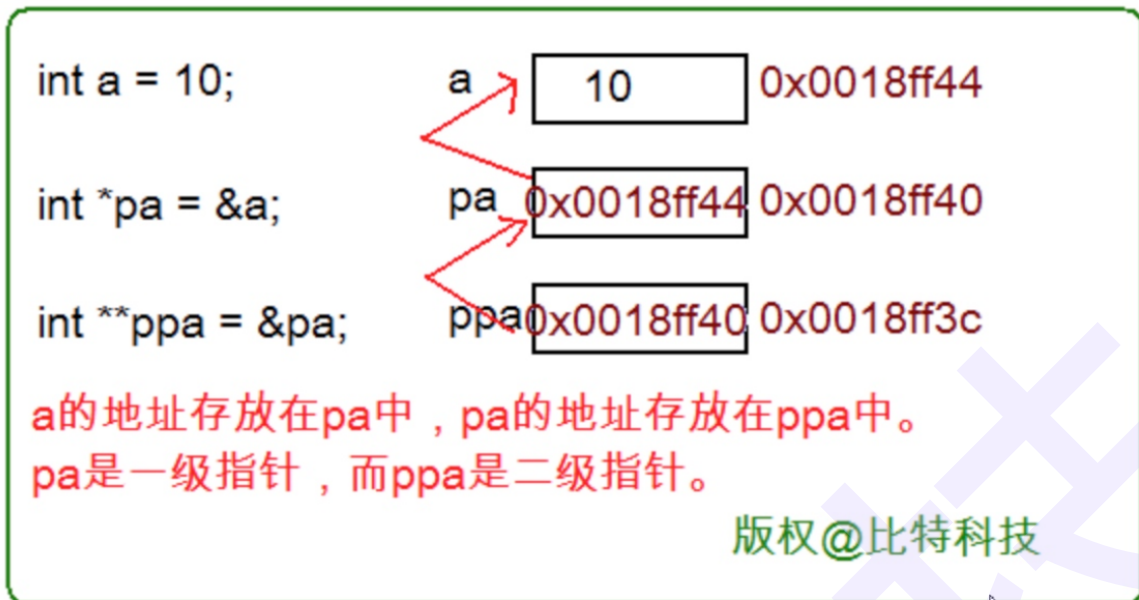
那我们就可以直接通过指针来访问数组。

如下：

```
int main()
{
    int arr[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };
    int *p = arr; //指针存放数组首元素的地址
    int sz = sizeof(arr) / sizeof(arr[0]);
    int i = 0;
    for (i = 0; i<sz; i++)
    {
        printf("%d ", *(p + i));
    }
    return 0;
}
```

二级指针

指针变量也是变量，是变量就有地址，那指针变量的地址存放在哪里？这就是二级指针。



对于二级指针的运算有：

- `*ppa` 通过对ppa中的地址进行解引用，这样找到的是 `pa`，`*ppa` 其实访问的就是 `pa`。

```
int b = 20;
*ppa = &b; // 等价于 pa = &b;
```

- `**ppa` 先通过 `*ppa` 找到 `pa`，然后对 `pa` 进行解引用操作：`*pa`，那找到的是 `a`。

```
**ppa = 30;
// 等价于 *pa = 30;
// 等价于 a = 30;
```

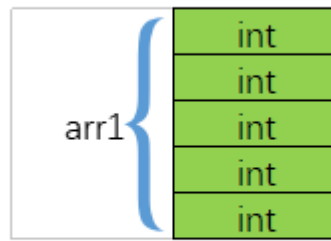
指针数组

指针数组是指针还是数组？

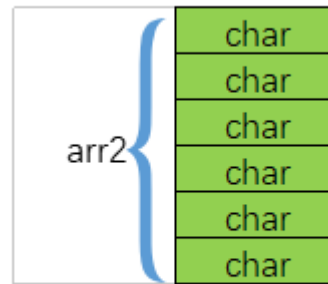
答案：是数组。是存放指针的数组。

数组我们已经知道整形数组，字符数组。

```
int arr1[5];
char arr2[6];
```

整形数组

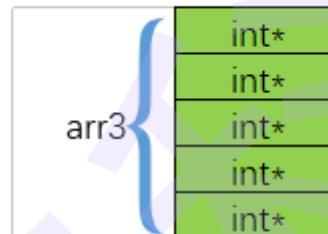


字符数组

那指针数组是怎样的？

```
int* arr3[5]; //是什么？
```

arr3是一个数组，有五个元素，每个元素是一个整形指针。



整形指针数组

本章完

