第11讲：深入理解指针(1)

目录：

正文开始

1. 内存和地址

1.1 内存

在讲内存和地址之前，我们想有个生活中的案例：

假设有一栋宿舍楼，把你放在楼里，楼上有100个房间，但是房间没有编号，你的一个朋友来找你玩，如果想找到你，就得挨个房子去找，这样效率很低，但是我们如果根据楼层和楼层的房间的情况，给每个房间编上号，如：

```
一楼：101，102，103...
二楼：201，202，203...
...
```

有了**房间号**，如果你的朋友得到房间号，就可以快速的找房间，找到你。

生活中，每个房间有了房间号，就能提高效率，能快速的找到房间。

如果把上面的例子对照到计算机中，又是怎么样呢？

我们知道计算机上CPU（中央处理器）在处理数据的时候，需要的数据是在内存中读取的，处理后的数据也会放回内存中，那我们买电脑的时候，电脑上内存是 8GB/16GB/32GB 等，那这些内存空间如何高效的管理呢？

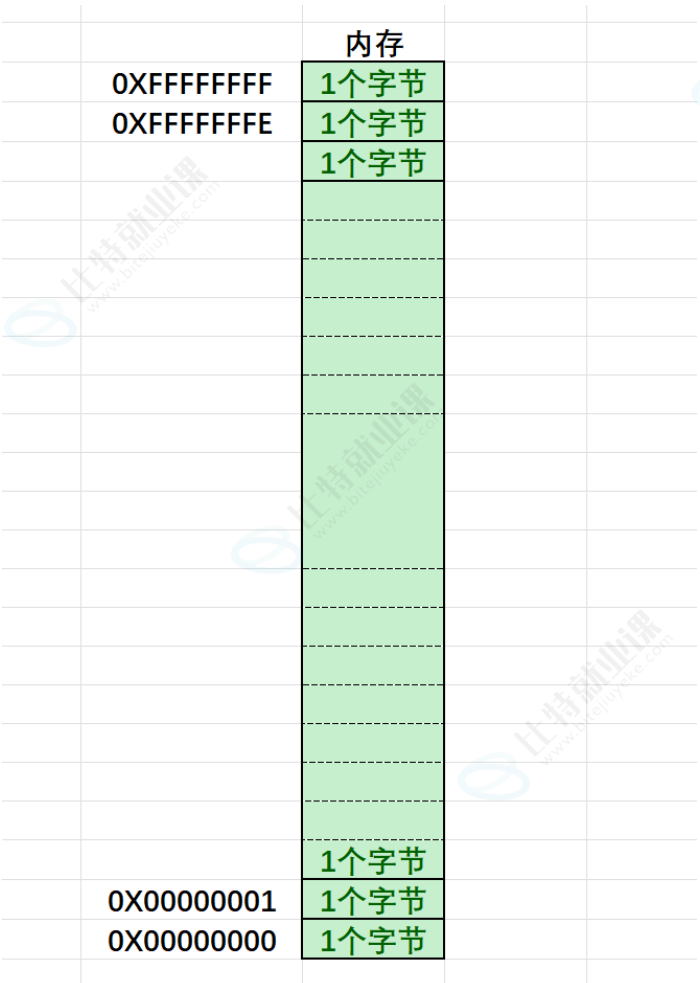
其实也是把内存划分为一个个的内存单元，每个内存单元的大小取1个字节。

计算机中常见的单位（补充）：

一个比特位可以存储一个2进制的位1或者0

```
bit - 比特位
Byte - 字节
KB
MB
GB
TB
PB
```

```
1Byte = 8bit
1KB = 1024Byte
1MB = 1024KB
1GB = 1024MB
1TB = 1024GB
1PB = 1024TB
```



其中，每个内存单元，相当于一个**学生宿舍**，一个字节空间里面能放**8个比特位**，就好比同学们住的八人间，每个人是一个比特位。

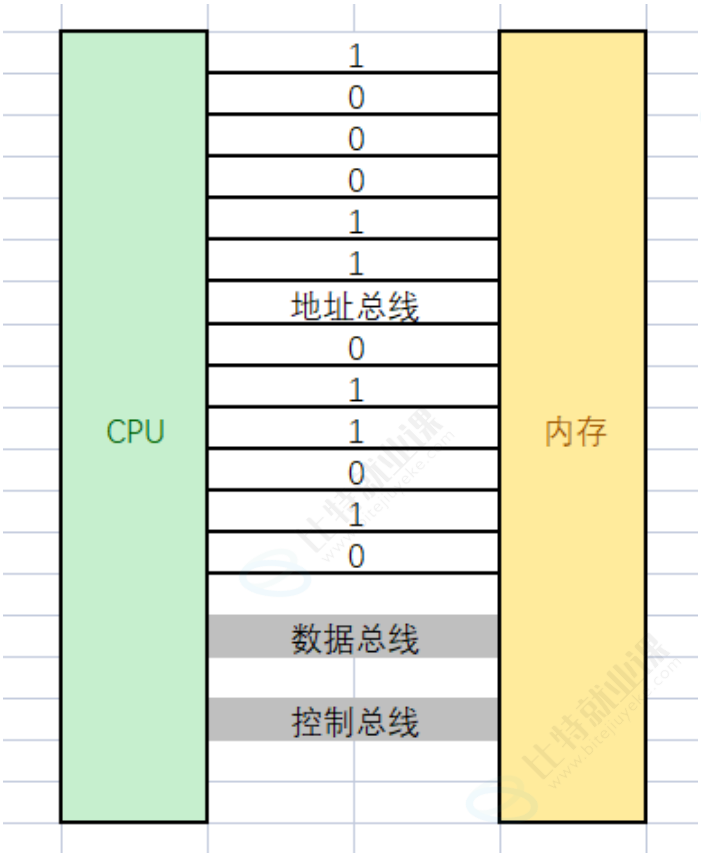
每个内存单元也都有一个编号（这个编号就相当于宿舍房间的门牌号），有了这个内存单元的编号，CPU就可以快速找到一个内存空间。

生活中我们把门牌号也叫地址，在计算机中我们把内存单元的编号也称为地址。C语言中给**地址**起了新的名字叫：指针。

所以我们可以理解为：

内存单元的编号 == 地址 == 指针

1.2 究竟该如何理解编址



CPU访问内存中的某个字节空间，必须知道这个字节空间在内存的什么位置，而因为内存中字节很多，所以需要给内存进行编址(就如同宿舍很多，需要给宿舍编号一样)。

计算机中的编址，并不是把每个字节的地址记录下来，而是通过硬件设计完成的。

钢琴、吉他 上面没有写上“剁、来、咪、发、唆、拉、西”这样的信息，但演奏者照样能够准确找到每一个琴弦的每一个位置，这是为何？因为制造商已经在乐器硬件层面上设计好了，并且所有的演奏者都知道。本质是一种约定出来的共识！

首先，必须理解，计算机内是有很多的硬件单元，而硬件单元是要互相协同工作的。所谓的协同，至少相互之间要能够进行数据传递。

但是硬件与硬件之间是互相独立的，那么如何通信呢？答案很简单，用**"线"**连起来。

而CPU和内存之间也是有大量的数据交互的，所以，两者必须也用线连起来。

不过，我们今天关心一组线，叫做**地址总线**。

硬件编址也是如此

我们可以简单理解，32位机器有32根地址总线，每根线只有两态，表示0,1【电脉冲有无】，那么一根线，就能表示2种含义，2根线就能表示4种含义，依次类推。32根地址线，就能表示2^32种含义，每一种含义都代表一个地址。

地址信息被下达给内存，在内存上，就可以找到该地址对应的数据，将数据在通过数据总线传入CPU内寄存器。

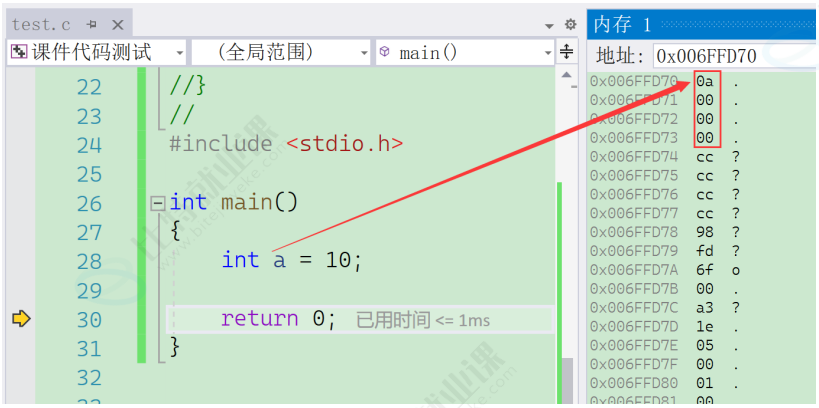
2. 指针变量和地址

2.1 取地址操作符（&）

理解了内存和地址的关系，我们再回到C语言，在C语言中创建变量其实就是向内存申请空间，比如：

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

```c
int main()
{
    int a = 10;
    return 0;
}
```



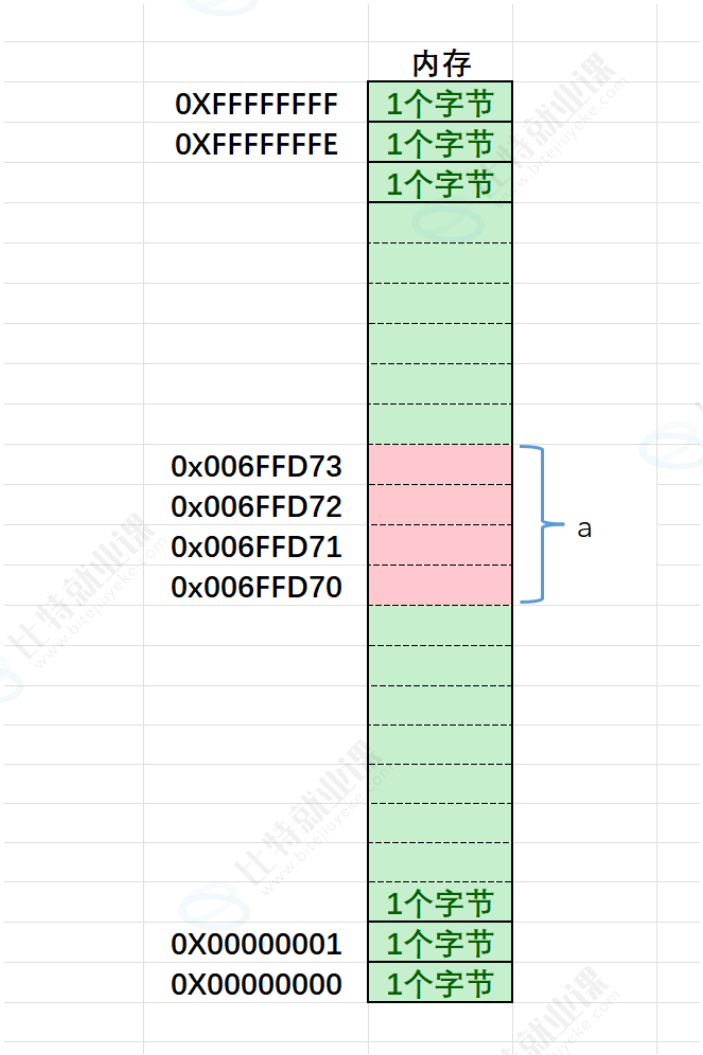
比如，上述的代码就是创建了整型变量a，内存中申请4个字节，用于存放整数10，其中每个字节都有地址，上图中4个字节的地址分别是：

```
0x006FFD70
0x006FFD71
0x006FFD72
0x006FFD73
```

那我们如何能得到a的地址呢?

这里就得学习一个操作符(&)-取地址操作符

```c
#include <stdio.h>
int main()
{
    int a = 10;
    &a;//取出a的地址
    printf("%p\n", &a);
    return 0;
}
```



变量在内存中的存储

按照我画图的例子，会打印处理：006FFD70

&a取出的是a所占4个字节中地址较小的字节的地址。

虽然整型变量占用4个字节，我们只要知道了第一个字节地址，顺藤摸瓜访问到4个字节的数据也是可行的。

2.2 指针变量和解引用操作符（*）

2.2.1 指针变量

那我们通过取地址操作符(&)拿到的地址是一个数值，比如：`0x006FFD70`，这个数值有时候也是需要存储起来，方便后期再使用的，那我们把这样的地址值存放在哪里呢？答案是：**指针变量**中。

比如：

```
#include <stdio.h>
int main()
{
    int a = 10;
    int * pa = &a;//取出a的地址并存储到指针变量pa中

    return 0;
}
```

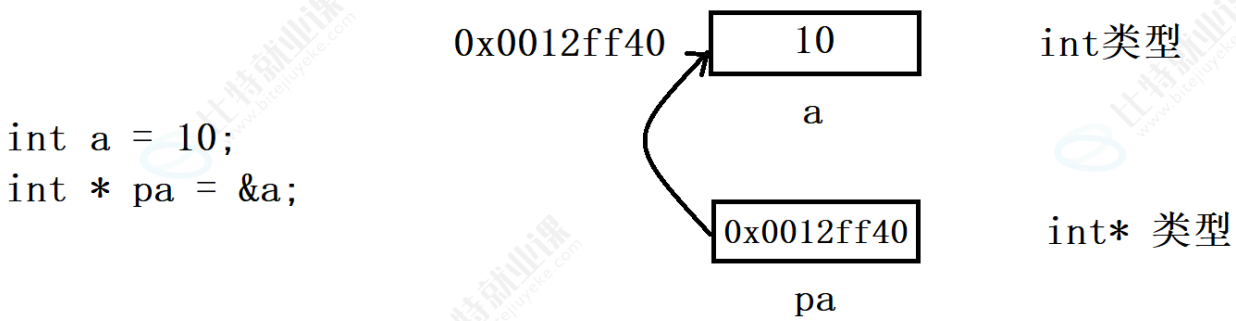
指针变量也是一种变量，这种变量就是用来存放地址的，存放在指针变量中的值都会理解为地址。

2.2.2 如何拆解指针类型

我们看到pa的类型是`int*`，我们该如何理解指针的类型呢？

```
int a = 10;
int * pa = &a;
```

这里pa左边写的是`int*`，`*`是在说明pa是指针变量，而前面的`int`是在说明pa指向的是整型(int)类型的对象。



那如果有一个char类型的变量ch，ch的地址，要放在什么类型的指针变量中呢？

```
char ch = 'w';
pc = &ch;//pc 的类型怎么写呢?
```

2.2.3 解引用操作符

我们将地址保存起来，未来是要使用的，那怎么使用呢?

在现实生活中，我们使用地址要找到一个房间，在房间里可以拿去或者存放物品。

C语言中其实也是一样的，我们只要拿到了地址（指针），就可以通过地址（指针）找到地址（指针）指向的对象，这里必须学习一个操作符叫解引用操作符(*)。

```
#include <stdio.h>

int main()
{
    int a = 100;
    int* pa = &a;
    *pa = 0;
    return 0;
}
```

上面代码中第7行就使用了解引用操作符，`*pa`的意思就是通过pa中存放的地址，找到指向的空间，*pa其实就是a变量了；所以*pa = 0，这个操作符是把a改成了0.

有同学肯定在想，这里如果目的就是把a改成0的话，写成`a = 0;`不就完了，为啥非要使用指针呢?

其实这里是把a的修改交给了pa来操作，这样对a的修改，就多了一种的途径，写代码就会更加灵活，后期慢慢就能理解了。

2.3 指针变量的大小

前面的内容我们了解到，32位机器假设有32根地址总线，每根地址线出来的电信号转换成数字信号后是1或者0，那我们把32根地址线产生的2进制序列当做一个地址，那么一个地址就是32个bit位，需要4个字节才能存储。

如果指针变量是用来存放地址的，那么指针变量的大小就得是4个字节的空间才可以。

同理64位机器，假设有64根地址线，一个地址就是64个二进制位组成的二进制序列，存储起来就需要8个字节的空间，指针变量的大小就是8个字节。

```
#include <stdio.h>
//指针变量的大小取决于地址的大小
//32位平台下地址是32个bit位（即4个字节）
```

```c
//64位平台下地址是64个bit位（即8个字节）

int main()
{
    printf("%zd\n", sizeof(char *));
    printf("%zd\n", sizeof(short *));
    printf("%zd\n", sizeof(int *));
    printf("%zd\n", sizeof(double *));
    return 0;
}
```

X86环境输出结果

X64环境输出结果

结论：

- 32位平台下地址是32个bit位，指针变量大小是4个字节
- 64位平台下地址是64个bit位，指针变量大小是8个字节
- 注意指针变量的大小和类型是无关的，只要指针类型的变量，在相同的平台下，大小都是相同的。

3. 指针变量类型的意义

指针变量的大小和类型无关，只要是指针变量，在同一个平台下，大小都是一样的，为什么还要有各种各样的指针类型呢？

其实指针类型是有特殊意义的，我们接下来继续学习。

3.1 指针的解引用

对比，下面2段代码，主要在调试时观察内存的变化。

```c
//代码1
#include <stdio.h>

int main()
{
    int n = 0x11223344;
    int *pi = &n;
    *pi = 0;
```

```c
//代码2
#include <stdio.h>

int main()
{
    int n = 0x11223344;
    char *pc = (char *)&n;
    *pc = 0;
```

```
    return 0;
}
```

```
    return 0;
}
```

调试我们可以看到，代码1会将n的4个字节全部改为0，但是代码2只是将n的第一个字节改为0。

结论：指针的类型决定了，对指针解引用的时候有多大的权限（一次能操作几个字节）。

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。

3.2 指针+-整数

先看一段代码，调试观察地址的变化。

```c
#include <stdio.h>
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;
}
```

代码运行的结果如下：

我们可以看出，`char*` 类型的指针变量+1跳过1个字节，`int*` 类型的指针变量+1跳过了4个字节。这就是指针变量的类型差异带来的变化。指针+1，其实跳过1个指针指向的元素。指针可以+1，那也可以-1。

结论：指针的类型决定了指针向前或者向后走一步有多大（距离）。

3.3 void* 指针

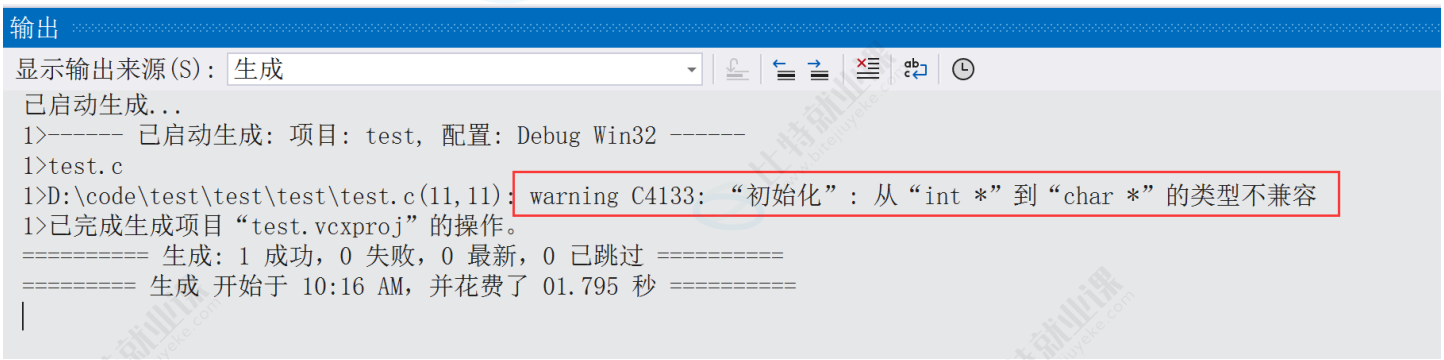
在指针类型中有一种特殊的类型是 `void *` 类型的，可以理解为无具体类型的指针（或者叫泛型指针），这种类型的指针可以用来接受任意类型地址。但是也有局限性， `void*` 类型的指针不能直接进行指针的+-整数和解引用的运算。

举例：

```c
#include <stdio.h>

int main()
{
    int a = 10;
    int* pa = &a;
    char* pc = &a;
    return 0;
}
```

在上面的代码中，将一个int类型的变量的地址赋值给一个char*类型的指针变量。编译器给出了一个警告（如下图），是因为类型不兼容。而使用void*类型就不会有这样的问题。



VS2022编译的结果

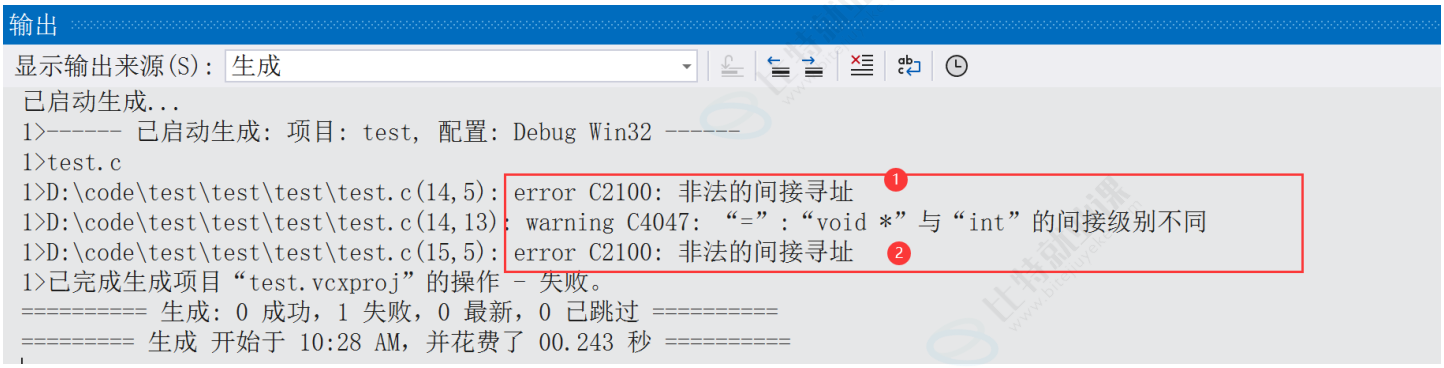
使用void*类型的指针接收地址：

```c
#include <stdio.h>

int main()
{
    int a = 10;
    void* pa = &a;
    void* pc = &a;

    *pa = 10;
    *pc = 0;
    return 0;
}
```

VS编译代码的结果：



VS2022编译的结果

这里我们可以看到，`void*`类型的指针可以接收不同类型的地址，但是无法直接进行指针运算。

那么`void*`类型的指针到底有什么用呢？

一般`void*`类型的指针是使用在**函数参数的部分**，用来接收不同类型数据的地址，这样的设计可以实现泛型编程的效果。使得一个函数来处理多种类型的数据，在《深入理解指针(5)》中我们会讲解。

4. 指针运算

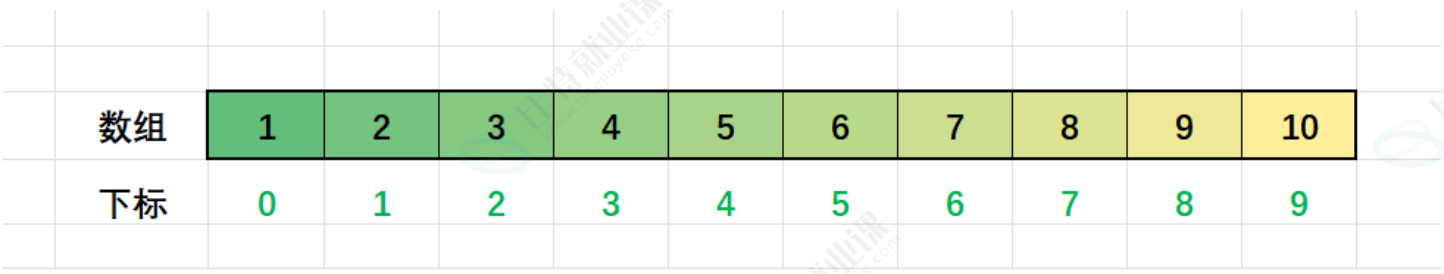
指针的基本运算有三种，分别是：

- 指针+- 整数
- 指针-指针
- 指针的关系运算

4.1 指针+- 整数

因为数组在内存中是连续存放的，只要知道第一个元素的地址，顺藤摸瓜就能找到后面的所有元素。

```
1   int arr[10] = {1,2,3,4,5,6,7,8,9,10};
```



数组元素和下标

```c
#include <stdio.h>
//指针+- 整数
int main()
{
    int arr[10] = {1,2,3,4,5,6,7,8,9,10};
    int *p = &arr[0];
    int i = 0;
    int sz = sizeof(arr)/sizeof(arr[0]);
    for(i = 0; i < sz; i++)
    {
        printf("%d ", *(p + i));//p+i 这里就是指针+整数
    }
    return 0;
}
```

4.2 指针 - 指针

```c
//指针-指针
#include <stdio.h>
int my_strlen(char *s)
{
   char *p = s;
   while(*p != '\0' )
          p++;
   return p-s;
}

int main()
{
    printf("%d\n", my_strlen("abc"));
    return 0;
}
```

4.3 指针的关系运算

```c
//指针的关系运算
#include <stdio.h>

int main()
{
    int arr[10] = {1,2,3,4,5,6,7,8,9,10};
    int *p = &arr[0];
    int sz = sizeof(arr)/sizeof(arr[0]);
```

```
    while(p < arr + sz) //指针的大小比较
    {
        printf("%d ", *p);
        p++;
    }
    return 0;
}
```

完