C 语言指针：从底层原理到花式技巧，用图文和代码帮你讲解透彻

程序员的那些事 1周前

以下文章来源于IOT物联网小镇 ， 作者道哥

IOT物联网小镇
深入的思考 + 直白的文字 + 实用的项目经验，这是我能为您提供的、最基本的知识服务...

一、前言

如果问C语言中最重要、威力最大的概念是什么，答案必将是指针！威力大，意味着使用方便、高效，同时也意味着语法复杂、容易出错。指针用的好，可以极大的提高代码执行效率、节约系统资源；如果用的不好，程序中将会充满陷阱、漏洞。

这篇文章，我们就来聊聊指针。从最底层的内存存储空间开始，一直到应用层的各种指针使用技巧，循序渐进、抽丝剥茧，以最直白的语言进行讲解，让你一次看过瘾。

说明：为了方便讲解和理解，文中配图的内存空间的地址是随便写的，在实际计算机中是要遵循地址对齐方式的。

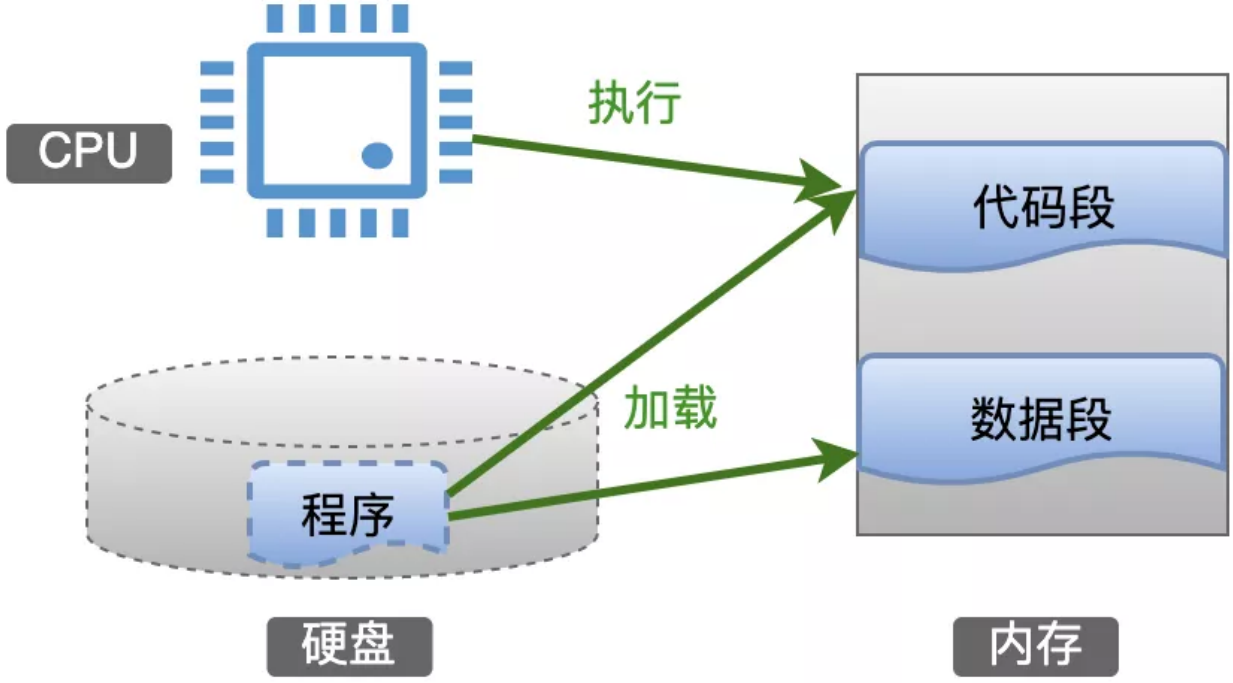
二、变量与指针的本质

1. 内存地址

我们编写一个程序源文件之后，编译得到的二进制可执行文件存放在电脑的硬盘上，此时它是一个静态的文件，一般称之为程序。

当这个程序被启动的时候，操作系统将会做下面几件事情：

> 1. 把程序的内容(代码段、数据段)从硬盘复制到内存中；
> 2. 创建一个数据结构PCB(进程控制块)，来描述这个程序的各种信息(例如：使用的资源，打开的文件描述符...)；
> 3. 在代码段中定位到入口函数的地址，让CPU从这个地址开始执行。

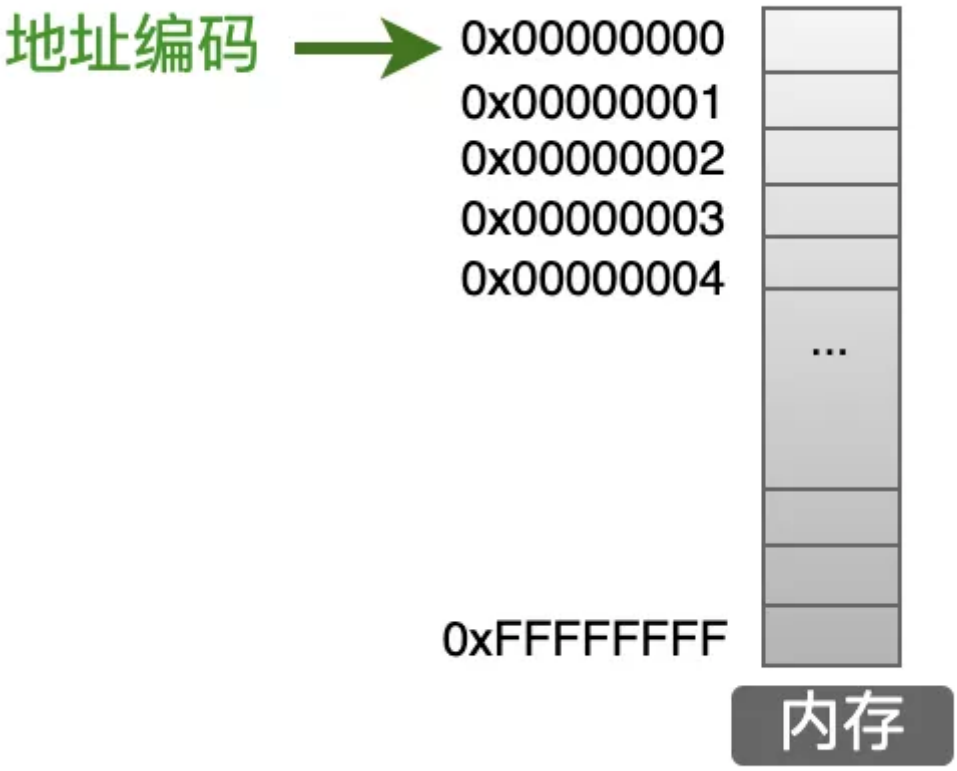


当程序开始被执行时，就变成一个动态的状态，一般称之为进程。

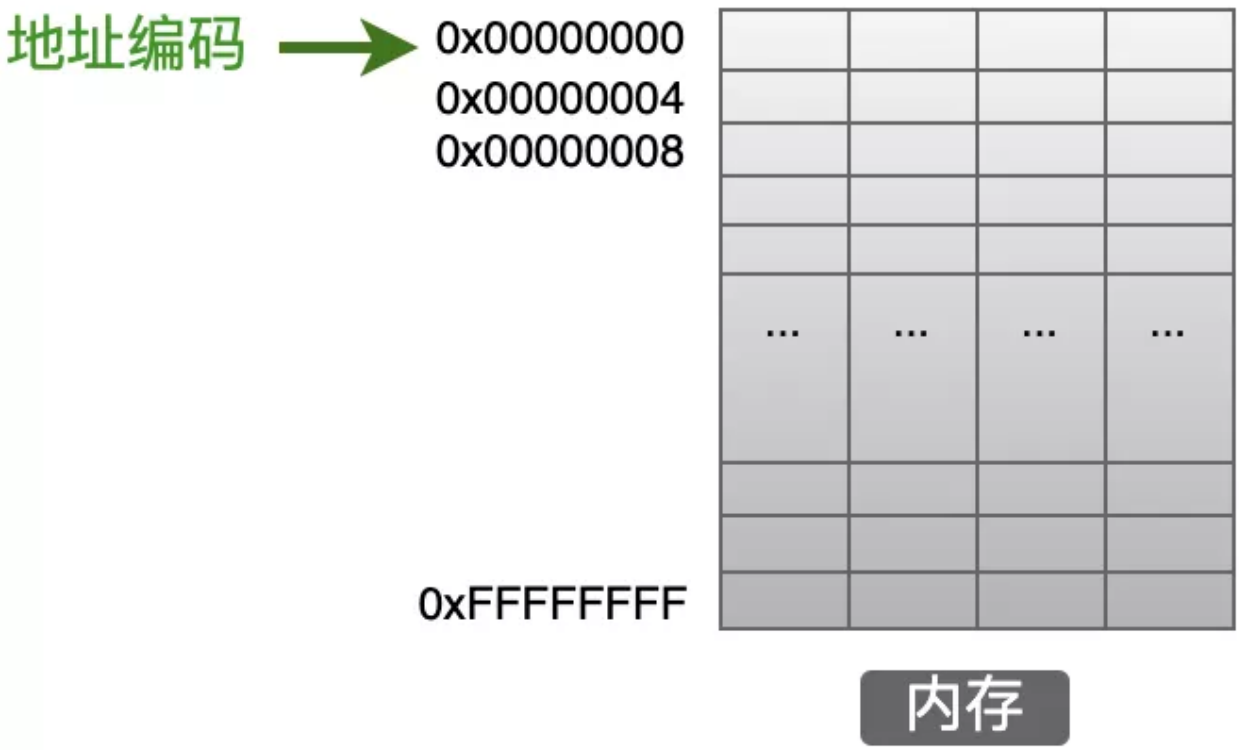
> 内存分为：物理内存和虚拟内存。操作系统对物理内存进行管理、包装，我们开发者面对的是操作系统提供的虚拟内存。
> 这2个概念不妨碍文章的理解，因此就统一称之为内存。

在我们的程序中，通过一个变量名来定义变量、使用变量。变量本身是一个确确实实存在的东西，变量名是一个抽象的概念，用来代表这个变量。就比如：我是一个实实在在的人，是客观存在与这个地球上的，道哥是我给自己起的一个名字，这个名字是任意取得，只要自己觉得好听就行，如果我愿意还可以起名叫：鸟哥、龙哥等等。

那么，我们定义一个变量之后，这个变量放在哪里呢？那就是内存的数据区。内存是一个很大的存储区域，被操作系统划分为一个一个的小空间，操作系统通过地址来管理内存。



内存中的最小存储单位是字节(8个bit)，一个内存的完整空间就是由这一个一个的字节连续组成的。在上图中，每一个小格子代表一个字节，但是好像大家在书籍中没有这么来画内存模型的，更常见的是下面这样的画法：

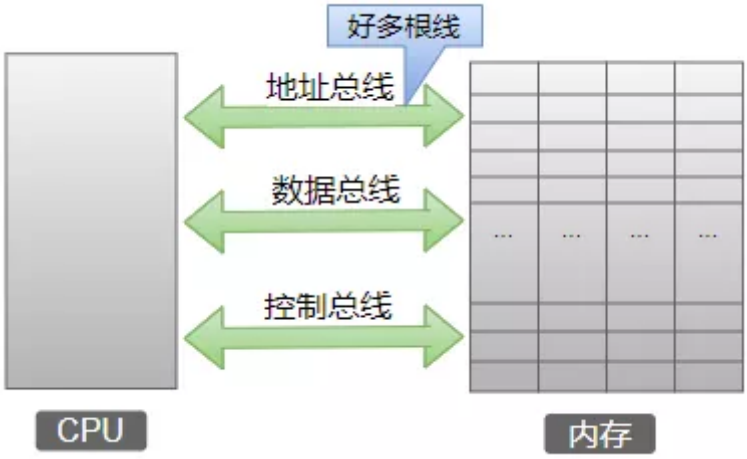


也就是把连续的4个字节的空间画在一起，这样就便于表述和理解，特别是深入到代码对齐相关知识时更容易理解。(我认为根本原因应该是：大家都这么画，已经看顺眼了~~)

2. 32位与64位系统

我们平时所说的计算机是32位、64位，指的是计算机的CPU中寄存器的最大存储长度，如果寄存器中最大存储32bit的数据，就称之为32位系统。

在计算机中，数据一般都是在硬盘、内存和寄存器之间进行来回存取。CPU通过3种总线把各组成部分联系在一起：地址总线、数据总线和控制总线。地址总线的宽度决定了CPU的寻址能力，也就是CPU能达到的最大地址范围。



刚才说了，内存是通过地址来管理的，那么CPU想从内存中的某个地址空间上存取一个数据，那么CPU就需要在地址总线上输出这个存储单元的地址。假如地址总线的宽度是8位，能表示的最大地址空间就是256个字节，能找到内存中最大的存储单元是255这个格子(从0开始)。即使内存条的实际空间是2G字节，CPU也没法使用后面的内存地址空间。如果地址总线的宽度是32位，那么能表示的最大地址就是2的32次方，也就是4G字节的空间。

【注意】：这里只是描述地址总线的概念，实际的计算机中地址计算方式要复杂的多，比如：虚拟内存中采用分段、分页、偏移量来定位实际的物理内存，在分页中还有大页、小页之分，感兴趣的同学可以自己查一下相关资料。

3. 变量

我们在C程序中使用变量来“代表”一个数据，使用函数名来“代表”一个函数，变量名和函数名是程序员使用的助记符。变量和函数最终是要放到内存中才能被CPU使用的，而内存中所有的信息(代码和数据)都是以二进制的形式来存储的，计算机根据就不会从格式上来区分哪些是代码、哪些是数据。CPU在访问内存的时候需要的是地址，而不是变量名、函数名。

问题来了：在程序代码中使用变量名来指代变量，而变量在内存中是根据地址来存放的，这二者之间如何映射(关联)起来的？

答案是：编译器！编译器在编译文本格式的C程序文件时，会根据目标运行平台(就是编译出的二进制程序运行在哪里？是x86平台的电脑？还是ARM平台的开发板？)来安排程序中的各种地址，例如：加载到内存中的地址、代码段的入口地址等等，同时编译器也会把程序中的所有变量名，转成该变量在内存中的存储地址。

变量有2个重要属性：变量的类型和变量的值。

示例：代码中定义了一个变量

```
int a = 20;
```

类型是int型，值是20。这个变量在内存中的存储模型为：

我们在代码中使用变量名a，在程序执行的时候就表示使用0x11223344地址所对应的那个存储单元中的数据。因此，可以理解为变量名a就等价于这个地址0x11223344。换句话说，如果我们可以提前知道编译器把变量a安排在地址0x11223344这个单元格中，我们就可以在程序中直接用这个地址值来操作这个变量。

在上图中，变量a的值为20，在内存中占据了4个格子的空间，也就是4个字节。为什么是4个字节呢？在C标准中并没有规定每种数据类型的变量一定要占用几个字节，这是与具体的机器、编译器有关。

比如：32位的编译器中：

> char: 1个字节；
> short int: 2个字节；
> int: 4个字节；
> long: 4个字节。

比如：64位的编译器中：

> char: 1个字节；
> short int: 2个字节；
> int: 4个字节；
> long: 8个字节。

为了方便描述，下面都以32位为例，也就是int型变量在内存中占据4个字节。

另外，0x11223344，0x11223345，0x11223346，0x11223347这连续的、从低地址到高地址的4个字节用来存储变量a的数值20。在图示中，使用十六进制来表示，十进制数值20转成16进制就是：0x00000014，所以从开始地址依次存放0x00、0x00、0x00、0x14这4个字节(存储顺序涉及到大小端的问题，不影响文本理解)。

根据这个图示，如果在程序中想知道变量a存储在内存中的什么位置，可以使用取地址操作符&，如下：

```
printf("&a = 0x%x \n", &a);
```

这句话将会打印出： `&a = 0x11223344` 。

考虑一下，在32位系统中：指针变量占用几个字节？

4. 指针变量

指针变量可以分2个层次来理解：

> 1. 指针变量首先是一个变量，所以它拥有变量的所有属性：类型和值。它的类型就是指针，它的值是其他变量的地址。 既然是一个变量，那么在内存中就需要为这个变量分配一个存储空间。在这个存储空间中，存放着其他变量的地址。
> 2. 指针变量所指向的数据类型，这是在定义指针变量的时候就确定的。例如：int *p; 意味着指针指向的是一个int型的数据。

首先回答一下刚才那个问题，在32位系统中，一个指针变量在内存中占据4个字节的空间。因为CPU对内存空间寻址时，使用的是32位地址空间(4个字节)，也就是用4个字节就能存储一个内存单元的地址。而指针变量中的值存储的就是地址，所以需要4个字节的空间来存储一个指针变量的值。

示例：

```
int a = 20;
int *pa;
pa = &a;
printf("value = %d \n", *pa);
```

在内存中的存储模型如下：

对于指针变量pa来说，首先它是一个变量，因此在内存中需要有一个空间来存储这个变量，这个空间的地址就是0x11223348；

其次，这个内存空间中存储的内容是变量a的地址，而a的地址为0x11223344，所以指针变量pa的地址空间中，就存储了0x11223344这个值。

这里对两个操作符&和*进行说明：

> &：取地址操作符，用来获取一个变量的地址。上面代码中&a就是用来获取变量a在内存中的存储地址，也就是0x11223344。
>
> *：这个操作符用在2个场景中：定义一个指针的时候，获取一个指针所指向的变量值的时候。
>
> > 1. int *pa;* 这个语句中的表示定义的变量pa是一个指针，前面的int表示pa这个指针指向的是一个int类型的变量。不过此时我们没有给pa进行赋值，也就是说此刻pa对应的存储单元中的4个字节里的值是没有初始化的，可能是0x00000000，也可能是其他任意的数字，不确定；
> > 2. printf语句中的*表示获取pa指向的那个int类型变量的值，学名叫解引用，我们只要记住是获取指向的变量的值就可以了。

5. 操作指针变量

对指针变量的操作包括3个方面：

> 1. 操作指针变量自身的值；
> 2. 获取指针变量所指向的数据;
> 3. 以什么样数据类型来使用/解释指针变量所指向的内容。

5.1 指针变量自身的值

`int a = 20;` 这个语句是定义变量a，在随后的代码中，只要写下a就表示要操作变量a中存储的值，操作有两种：读和写。

> `printf("a = %d \n", a);` 这个语句就是要读取变量a中的值，当然是20；
> `a = 100;` 这个语句就是要把一个数值100写入到变量a中。

同样的道理， `int *pa;` 语句是用来定义指针变量pa，在随后的代码中，只要写下pa就表示要操作变量pa中的值：

> `printf("pa = %d \n", pa);` 这个语句就是要读取指针变量pa中的值，当然是0x11223344；
> `pa = &a;` 这个语句就是要把新的值写入到指针变量pa中。再次强调一下，指针变量中存储的是地址，如果我们可以提前知道变量a的地址是 0x11223344，那么我们也可以这样来赋值:pa = 0x11223344;

思考一下，如果执行这个语句 *`printf("&pa =0x%x \n", &pa);`* ，打印结果会是什么？

上面已经说过，操作符&是用来取地址的，那么&pa就表示获取指针变量pa的地址，上面的内存模型中显示指针变量pa是存储在0x11223348这个地址中的，因此打印结果就是： `&pa = 0x11223348` 。

5.2 获取指针变量所指向的数据

指针变量所指向的数据类型是在定义的时候就明确的，也就是说指针pa指向的数据类型就是int型，因此在执行 `printf("value = %d \n", *pa);` 语句时，首先知道pa是一个指针，其中存储了一个地址(0x11223344)，然后通过操作符*来获取这个地址(0x11223344)对应的那个

存储空间中的值；又因为在定义pa时，已经指定了它指向的值是一个int型，所以我们就知道了地址0x11223344中存储的就是一个int类型的数据。

5.3 以什么样的数据类型来使用/解释指针变量所指向的内容

如下代码：

```
int a = 30000;
int *pa = &a;
printf("value = %d \n", *pa);
```

根据以上的描述，我们知道printf的打印结果会是 `value = 30000` ，十进制的30000转成十六进制是0x00007530，内存模型如下：

现在我们做这样一个测试：

```
char *pc = 0x11223344;
printf("value = %d \n", *pc);
```

指针变量pc在定义的时候指明：它指向的数据类型是char型，pc变量中存储的地址是0x11223344。当使用*pc获取指向的数据时，将会按照char型格式来读取0x11223344地址处的数据，因此将会打印 `value = 0` (在计算机中，ASCII码是用等价的数字来存储的)。

这个例子中说明了一个重要的概念：在内存中一切都是数字，如何来操作(解释)一个内存地址中的数据，完全是由我们的代码来告诉编译器的。刚才这个例子中，虽然0x11223344这个地址开始的4个字节的空间中，存储的是整型变量a的值，但是我们让pc指针按照char型数据来使用/解释这个地址处的内容，这是完全合法的。

以上内容，就是指针最根本的心法了。把这个心法整明白了，剩下的就是多见识、多练习的问题了。

三、指针的几个相关概念

1. const属性

const标识符用来表示一个对象的不可变的性质，例如定义：

```
const int b = 20;
```

在后面的代码中就不能改变变量b的值了，b中的值永远是20。同样的，如果用const来修饰一个指针变量：

```
int a = 20;
int b = 20;
int * const p = &a;
```

内存模型如下：

这里的const用来修饰指针变量p，根据const的性质可以得出结论：p在定义为变量a的地址之后，就固定了，不能再被改变了，也就是说指针变量pa中就只能存储变量a的地址0x11223344。如果在后面的代码中写 `p = &b;` ，编译时就会报错，因为p是不可改变的，不能再被设置为变量b的地址。

但是，指针变量p所指向的那个变量a的值是可以改变的，即： `*p = 21;` 这个语句是合法的，因为指针p的值没有改变(仍然是变量c的地址0x11223344)，改变的是变量c中存储的值。

与下面的代码区分一下：

```
int a = 20;
int b = 20;
const int *p = &a;
p = &b;
```

这里的const没有放在p的旁边，而是放在了类型int的旁边，这就说明const符号不是用来修饰p的，而是用来修饰p所指向的那个变量的。所以，如果我们写 `p = &b;` 把变量b的地址赋值给指针p，就是合法的，因为p的值可以被改变。

但是这个语句 `*p = 21` 就是非法了，因为定义语句中的const就限制了通过指针p获取的数据，不能被改变，只能被用来读取。这个性质常常被用在函数参数上，例如下面的代码，用来计算一块数据的CRC校验，这个函数只需要读取原始数据，不需要(也不可以)改变原始数据，因此就需要在形参指针上使用const修饰符：

```
short int getDataCRC(const char *pData, int len)
```

```
{
    short int crc = 0x0000;
    // 计算CRC
    return crc;
}
```

2. void型指针

关键字void并不是一个真正的数据类型，它体现的是一种抽象，指明不是任何一种类型，一般有2种使用场景：

> 1. 函数的返回值和形参;
> 2. 定义指针时不明确规定所指数据的类型，也就意味着可以指向任意类型。

指针变量也是一种变量，变量之间可以相互赋值，那么指针变量之间也可以相互赋值，例如：

```
int a = 20;
int b = a;
int *p1 = &a;
int *p2 = p1;
```

变量a赋值给变量b，指针p1赋值给指针p2，注意到它们的类型必须是相同的：a和b都是int型，p1和p2都是指向int型，所以可以相互赋值。那么如果数据类型不同呢？必须进行强制类型转换。例如：

```
int a = 20;
int *p1 = &a;
char *p2 = (char *)p1;
```

内存模型如下：

p1指针指向的是int型数据，现在想把它的值(0x11223344)赋值给p2，但是由于在定义p2指针时规定它指向的数据类型是char型，因此需要把指针p1进行强制类型转换，也就是把地址0x11223344处的数据按照char型数据来看待，然后才可以赋值给p2指针。

如果我们使用 `void *p2` 来定义p2指针，那么在赋值时就不需要进行强制类型转换了，例如：

```
int a = 20;
```

```
int *p1 = &a;
void *p2 = p1;
```

指针p2是void*型，意味着可以把任意类型的指针赋值给p2，但是不能反过来操作，也就是不能把void*型指针直接赋值给其他确定类型的指针，而必须要强制转换成被赋值指针所指向的数据类型，如下代码，必须把p2指针强制转换成int*型之后，再赋值给p3指针：

```
int a = 20;
int *p1 = &a;
void *p2 = p1;
int *p3 = (int *)p2;
```

我们来看一个系统函数：

```
void* memcpy(void* dest, const void* src, size_t len);
```

第一个参数类型是void*，这正体现了系统对内存操作的真正意义：它并不关心用户传来的指针具体指向什么数据类型，只是把数据挨个存储到这个地址对应的空间中。

第二个参数同样如此，此外还添加了const修饰符，这样就说明了memcpy函数只会从src指针处读取数据，而不会修改数据。

3. 空指针和野指针

一个指针必须指向一个有意义的地址之后，才可以对指针进行操作。如果指针中存储的地址值是一个随机值，或者是一个已经失效的值，此时操作指针就非常危险了，一般把这样的指针称作野指针，C代码中很多指针相关的bug就来源于此。

3.1 空指针：不指向任何东西的指针

在定义一个指针变量之后，如果没有赋值，那么这个指针变量中存储的就是一个随机值，有可能指向内存中的任何一个地址空间，此时万万不可以对这个指针进行写操作，因为它有可能指向内存中的代码段区域、也可能指向内存中操作系统所在的区域。

一般会将一个指针变量赋值为NULL来表示一个空指针，而C语言中，NULL实质是 ((void*)0) ，在C++中，NULL实质是0。在标准库头文件stdlib.h中，有如下定义：

```
#ifdef __cplusplus
    #define NULL    0
#else
    #define NULL    ((void *)0)
#endif
```

3.2 野指针：地址已经失效的指针

我们都知道，函数中的局部变量存储在栈区，通过malloc申请的内存空间位于堆区，如下代码：

```
int *p = (int *)malloc(4);
*p = 20;
```

内存模型为：

在堆区申请了4个字节的空间，然后强制类型转换为int*型之后，赋值给指针变量p，然后通过*p设置这个地址中的值为14，这是合法的。如果在释放了p指针指向的空间之后，再使用*p来操作这段地址，那就是非常危险了，因为这个地址空间可能已经被操作系统分配给其他代码使用，如果对这个地址里的数据强行操作，程序立刻崩溃的话，将会是我们最大的幸运！

```
int *p = (int *)malloc(4);
*p = 20;
free(p);
// 在free之后就不可以再操作p指针中的数据了。
p = NULL;  // 最好加上这一句。
```

四、指向不同数据类型的指针

1. 数值型指针

通过上面的介绍，指向数值型变量的指针已经很明白了，需要注意的就是指针所指向的数据类型。

2. 字符串指针

字符串在内存中的表示有2种：

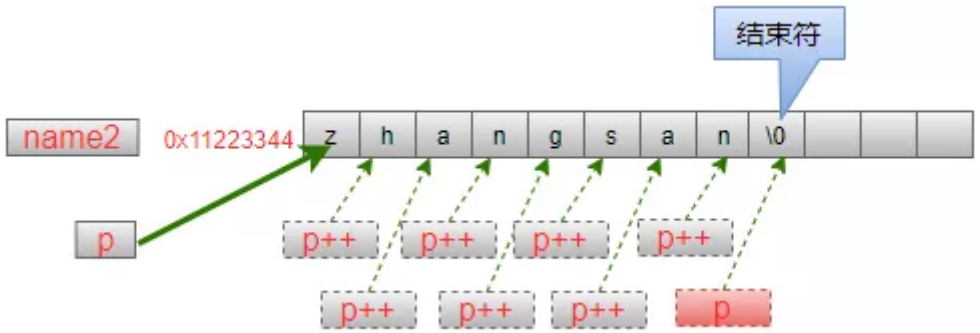
1. 用一个数组来表示，例如：char name1[8] = "zhangsan";
2. 用一个char *指针来表示，例如：char *name2 = "zhangsan";

name1在内存中占据8个字节，其中存储了8个字符的ASCII码值；name2在内存中占据9个字节，因为除了存储8个字符的ASCII码值，在最后一个字符'n'的后面还额外存储了一个'\0'，用来标识字符串结束。

对于字符串来说，使用指针来操作是非常方便的，例如：变量字符串name2:

```
char *name2 = "zhangsan";
char *p = name2;
while (*p != '\0')
{
    printf("%c ", *p);
    p = p + 1;
}
```

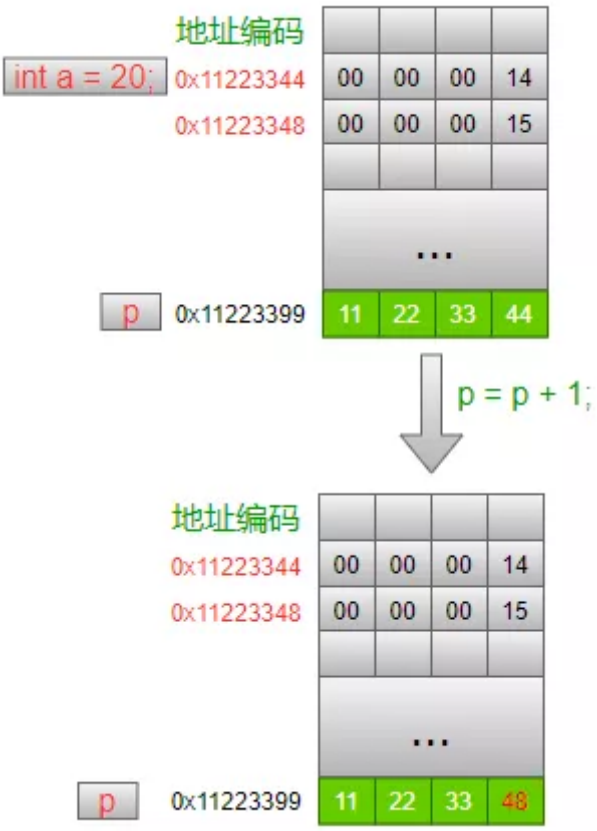
在while的判断条件中，检查p指针指向的字符是否为结束符'\0'。在循环体重，打印出当前指向的字符之后，对指针比那里进行自增操作，因为指针p所指向的数据类型是char，每个char在内存中占据一个字节，因此指针p在自增1之后，就指向下一个存储空间。



也可以把循环体中的2条语句写成1条语句：

```
printf("%c ", *p++);
```

假如一个指针指向的数据类型为int型，那么执行 `p = p + 1;` 之后，指针p中存储的地址值将会增加4，因为一个int型数据在内存中占据4个字节的空间，如下所示：



思考一个问题：void*型指针能够递增吗？如下测试代码：

```
int a[3] = {1, 2, 3};
void *p = a;
printf("1: p = 0x%x \n", p);
p = p + 1;
printf("2: p = 0x%x \n", p);
```

打印结果如下：

```
1: p = 0x733748c0
2: p = 0x733748c1
```

说明void*型指针在自增时，是按照一个字节的跨度来计算的。

3. 指针数组与数组指针

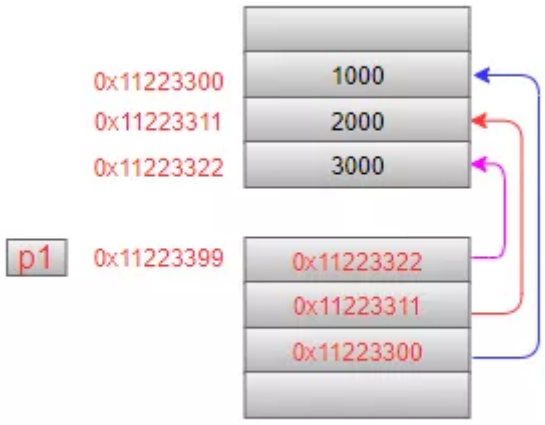
这2个说法经常会混淆，至少我是如此，先看下这2条语句：

```
int *p1[3];   // 指针数组
int (*p2)[3]; // 数组指针
```

3.1 指针数组

第1条语句中：中括号[]的优先级高，因此与p1先结合，表示一个数组，这个数组中有3个元素，这3个元素都是指针，它们指向的是int型数据。可以这样来理解：如果有这个定义 `char p`

[3]，很容易理解这是一个有3个char型元素的数组，那么把char换成int*，意味着数组里的元素类型是int*型(指向int型数据的指针)。内存模型如下(注意：三个指针指向的地址并不一定是连续的)：

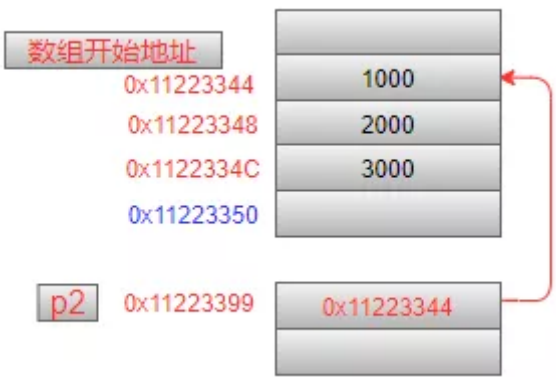


如果向指针数组中的元素赋值，需要逐个把变量的地址赋值给指针元素：

```
int a = 1, b = 2, c = 3;
char *p1[3];
p1[0] = &a;
p1[1] = &b;
p1[2] = &c;
```

3.2 数组指针

第2条语句中：小括号让p2与*结合，表示p2是一个指针，这个指针指向了一个数组，数组中有3个元素，每一个元素的类型是int型。可以这样来理解：如果有这个定义 `int p[3]` ，很容易理解这是一个有3个char型元素的数组，那么把数组名p换成是*p2，也就是p2是一个指针，指向了这个数组。内存模型如下(注意：指针指向的地址是一个数组，其中的3个元素是连续放在内存中的)：



在前面我们说到取地址操作符&，用来获得一个变量的地址。凡事都有特殊情况，对于获取地址来说，下面几种情况不需要使用&操作符：

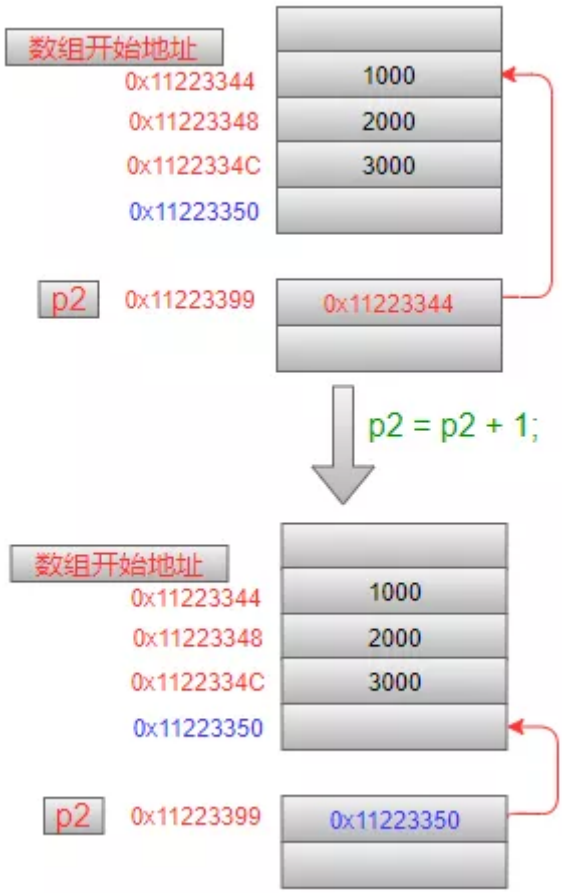
1. 字符串字面量作为右值时，就代表这个字符串在内存中的首地址；
2. 数组名就代表这个数组的地址，也等于这个数组的第一个元素的地址；
3. 函数名就代表这个函数的地址。

因此，对于一下代码，三个printf语句的打印结果是相同的：

```
int a[3] = {1, 2, 3};
int (*p2)[3] = a;
printf("0x%x \n", a);
printf("0x%x \n", &a);
printf("0x%x \n", p2);
```

思考一下，如果对这里的p2指针执行 `p2 = p2 + 1;` 操作，p2中的值将会增加多少？

答案是12个字节。因为p2指向的是一个数组，这个数组中包含3个元素，每个元素占据4个字节，那么这个数组在内存中一共占据12个字节，因此p2在加1之后，就跳过12个字节。



4. 二维数组和指针

一维数组在内存中是连续分布的多个内存单元组成的，而二维数组在内存中也是连续分布的多个内存单元组成的，从内存角度来看，一维数组和二维数组没有本质差别。

和一维数组类似，二维数组的数组名表示二维数组的第一维数组中首元素的首地址，用代码来说明：

```
int a[3][3] = {{1,2,3}, {4,5,6}, {7,8,9}}; // 二维数组
int (*p0)[3] = NULL;   // p0是一个指针，指向一个数组
int (*p1)[3] = NULL;   // p1是一个指针，指向一个数组
int (*p2)[3] = NULL;   // p2是一个指针，指向一个数组
p0 = a[0];
p1 = a[1];
p2 = a[2];
printf("0: %d %d %d \n", *(*p0 + 0), *(*p0 + 1), *(*p0 + 2));
printf("1: %d %d %d \n", *(*p1 + 0), *(*p1 + 1), *(*p1 + 2));
printf("2: %d %d %d \n", *(*p2 + 0), *(*p2 + 1), *(*p2 + 2));
```

打印结果是：

```
0: 1 2 3
1: 4 5 6
2: 7 8 9
```

我们拿第一个printf语句来分析：p0是一个指针，指向一个数组，数组中包含3个元素，每个元素在内存中占据4个字节。现在我们想获取这个数组中的数据，如果直接对p0执行加1操作，那么p0将会跨过12个字节(就等于p1中的值了)，因此需要使用解引用操作符*，把p0转为指向int型的指针，然后再执行加1操作，就可以得到数组中的int型数据了。

5. 结构体指针

C语言中的基本数据类型是预定义的，结构体是用户定义的，在指针的使用上可以进行类比，唯一有区别的就是在结构体指针中，需要使用 -> 箭头操作符来获取结构体中的成员变量，例如：

```
typedef struct
{
    int age;
    char name[8];
} Student;

Student s;
s.age = 20;
strcpy(s.name, "lisi");
Student *p = &s;
printf("age = %d, name = %s \n", p->age, p->name);
```

看起来似乎没有什么技术含量，如果是结构体数组呢？例如：

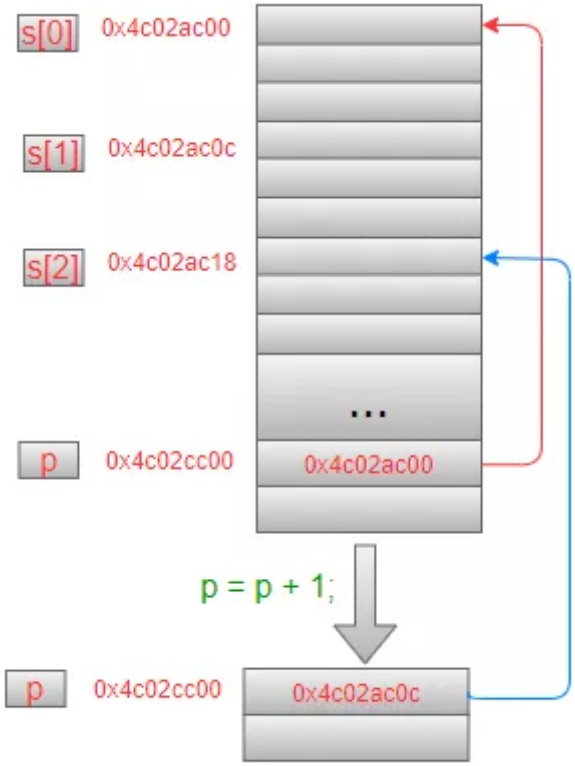
```
Student s[3];
Student *p = &s;
printf("size of Student = %d \n", sizeof(Student));
printf("1: 0x%x, 0x%x \n", s, p);
```

```
p++;
printf("2: 0x%x \n", p);
```

打印结果是：

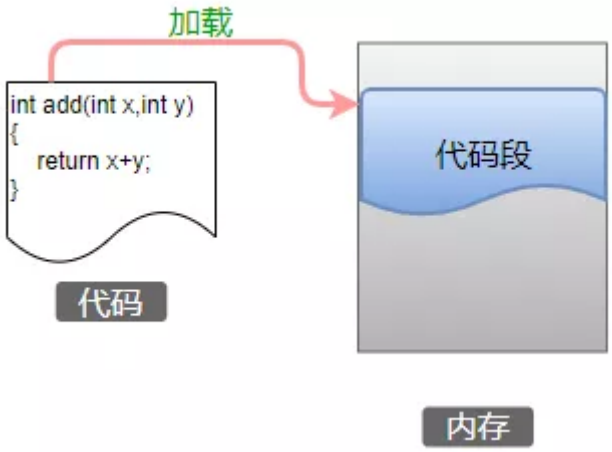
```
size of Student = 12
1: 0x4c02ac00, 0x4c02ac00
2: 0x4c02ac0c
```

在执行 p++ 操作后，p需要跨过的空间是一个结构体变量在内存中占据的大小(12个字节)，所以此时p就指向了数组中第2个元素的首地址，内存模型如下：



6. 函数指针

每一个函数在经过编译之后，都变成一个包含多条指令的集合，在程序被加载到内存之后，这个指令集合被放在代码区，我们在程序中使用函数名就代表了这个指令集合的开始地址。



函数指针，本质上仍然是一个指针，只不过这个指针变量中存储的是一个函数的地址。函数最重要特性是什么？可以被调用！因此，当定义了一个函数指针并把一个函数地址赋值给这个指针时，就可以通过这个函数指针来调用函数。

如下示例代码：

```
int add(int x,int y)
{
    return x+y;
}

int main()
{
    int a = 1, b = 2;
    int (*p)(int, int);
    p = add;
    printf("%d + %d = %d\n", a, b, p(a, b));
}
```

前文已经说过，函数的名字就代表函数的地址，所以函数名add就代表了这个加法函数在内存中的地址。`int (*p)(int, int);`这条语句就是用来定义一个函数指针，它指向一个函数，这个函数必须符合下面这2点(学名叫：函数签名)：

> 1. 有2个int型的参数;
> 2. 有一个int型的返回值。

代码中的add函数正好满足这个要求，因此，可以把add赋值给函数指针p，此时p就指向了内存中这个函数存储的地址，后面就可以用函数指针p来调用这个函数了。

在示例代码中，函数指针p是直接定义的，那如果想定义2个函数指针，难道需要像下面这样定义吗？

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

```
int (*p2)(int, int);
```

这里的参数比较简单，如果函数很复杂，这样的定义方式岂不是要烦死？可以用typedef关键字来定义一个函数指针类型：

```
typedef int (*pFunc)(int, int);
```

然后用这样的方式 `pFunc p1, p2;` 来定义多个函数指针就方便多了。注意：只能把与函数指针类型具有相同签名的函数赋值给p1和p2，也就是参数的个数、类型要相同，返回值也要相同。

注意：这里有几个小细节稍微了解一下：

> 1. 在赋值函数指针时，使用p = &a;也是可以的；
> 2. 使用函数指针调用时，使用(*p)(a, b);也是可以的。

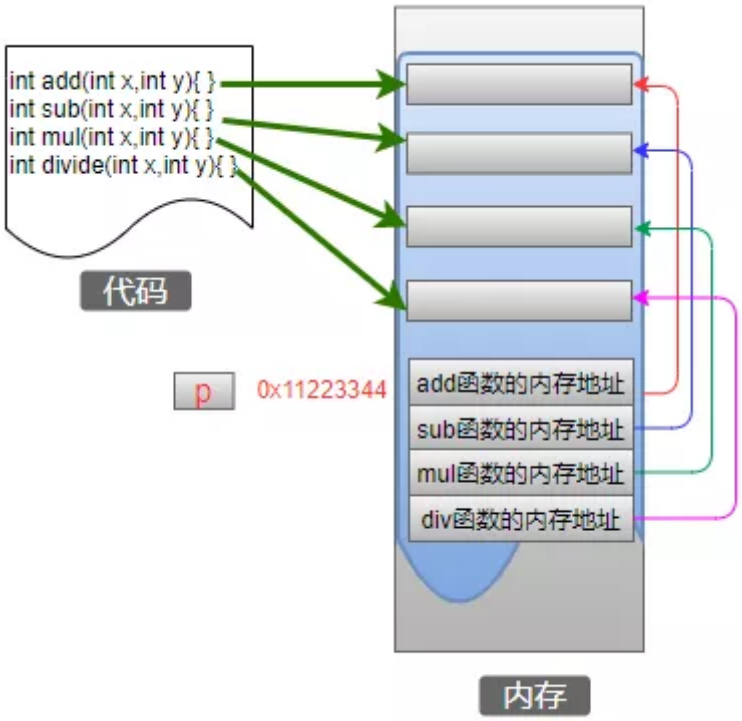
这里没有什么特殊的原理需要讲解，最终都是编译器帮我们处理了这里的细节，直接记住即可。

函数指针整明白之后，再和数组结合在一起：函数指针数组。示例代码如下：

```
int add(int a, int b) { return a + b; }
int sub(int a, int b) { return a - b; }
int mul(int a, int b) { return a * b; }
int divide(int a, int b) { return a / b; }

int main()
{
    int a = 4, b = 2;
    int (*p[4])(int, int);
    p[0] = add;
    p[1] = sub;
    p[2] = mul;
    p[3] = divide;
    printf("%d + %d = %d \n", a, b, p[0](a, b));
    printf("%d - %d = %d \n", a, b, p[1](a, b));
    printf("%d * %d = %d \n", a, b, p[2](a, b));
    printf("%d / %d = %d \n", a, b, p[3](a, b));
}
```

这条语句不太好理解: `int (*p[4])(int, int);` ，先分析中间部分，标识符p与中括号[]结合(优先级高)，所以p是一个数组，数组中有4个元素；然后剩下的内容表示一个函数指针，那么就说明数组中的元素类型是函数指针，也就是其他函数的地址，内存模型如下：



如果还是难以理解，那就回到指针的本质概念上：指针就是一个地址！这个地址中存储的内容是什么根本不重要，重要的是你告诉计算机这个内容是什么。如果你告诉它：这个地址里存放的内容是一个函数，那么计算机就去调用这个函数。那么你是如何告诉计算机的呢，就是在定义指针变量的时候，仅此而已！

五、总结

我已经把自己知道的所有指针相关的概念、语法、使用场景都作了讲解，就像一个小酒馆的掌柜，把自己的美酒佳肴都呈现给你，但愿你已经酒足饭饱！

如果以上的内容太多，一时无法消化，那么下面的这两句话就作为饭后甜点为您奉上，在以后的编程中，如果遇到指针相关的困惑，就想一想这两句话，也许能让你茅塞顿开。

> 1. 指针就是地址，地址就是指针。
> 2. 指针就是指向内存中的一块空间，至于如何来解释/操作这块空间，由这个指针的类型来决定。

另外还有一点嘱咐，那就是学习任何一门编程语言，一定要弄清楚内存模型，内存模型，内存模型！

- EOF -

推荐阅读 — 点击标题可跳转