C語言指針-從底層原理到花式技巧，用圖文和代碼幫你講解透徹

C語言與C++編程 今天

以下文章來源於IOT物聯網小鎮 ， 作者道哥

> **來自公眾號：IOT物聯網小鎮**

一、前言

如果問C語言中最重要、威力最大的概念是什麼，答案必將是指針！威力大，意味著使用方便、高效，同時也意味著語法複雜、容易出錯。指針用的好，可以極大的提高代碼執行效率、節約系統資源；如果用的不好，程序中將會充滿陷阱、漏洞。

這篇文章，我們就來聊聊指針。從最底層的內存存儲空間開始，一直到應用層的各種指針使用技巧，循序漸進、抽絲剝繭，以最直白的語言進行講解，讓你一次看過癮。

說明：為了方便講解和理解，文中配圖的內存空間的地址是隨便寫的，在實際計算機中是要遵循地址對齊方式的。

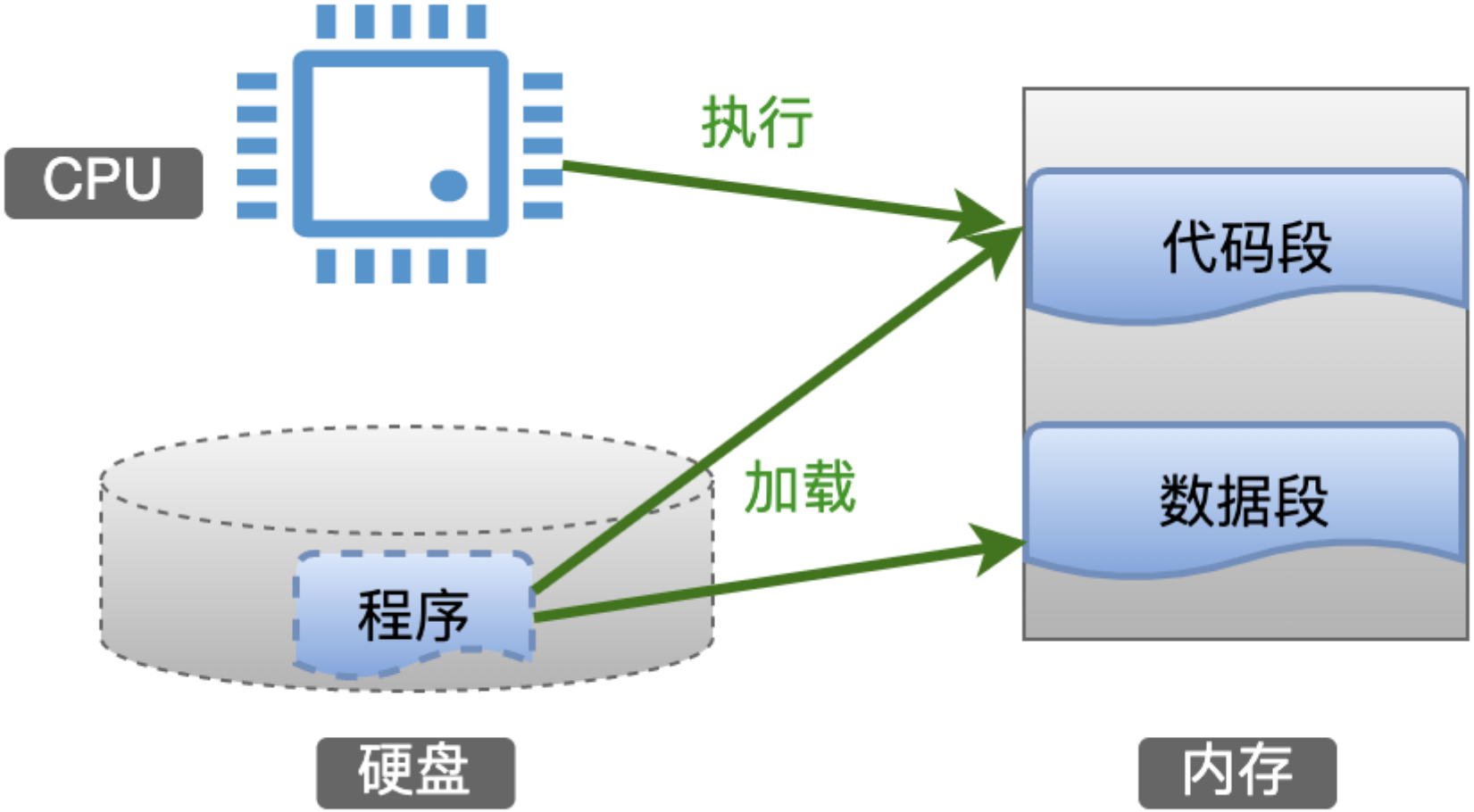
二、變量與指針的本質

1. 內存地址

我們編寫一個程序源文件之後，編譯得到的二進制可執行文件存放在電腦的硬盤上，此時它是一個靜態的文件，一般稱之為程序。

當這個程序被啟動的時候，操作系統將會做下面幾件事情：

1. 把程序的內容(代碼段、數據段)從硬盤複製到內存中；
2. 創建一個數據結構PCB(進程控制塊)，來描述這個程序的各種信息(例如：使用的資源，打開的文件描述符...);
3. 在代碼段中定位到入口函數的地址，讓CPU從這個地址開始執行。

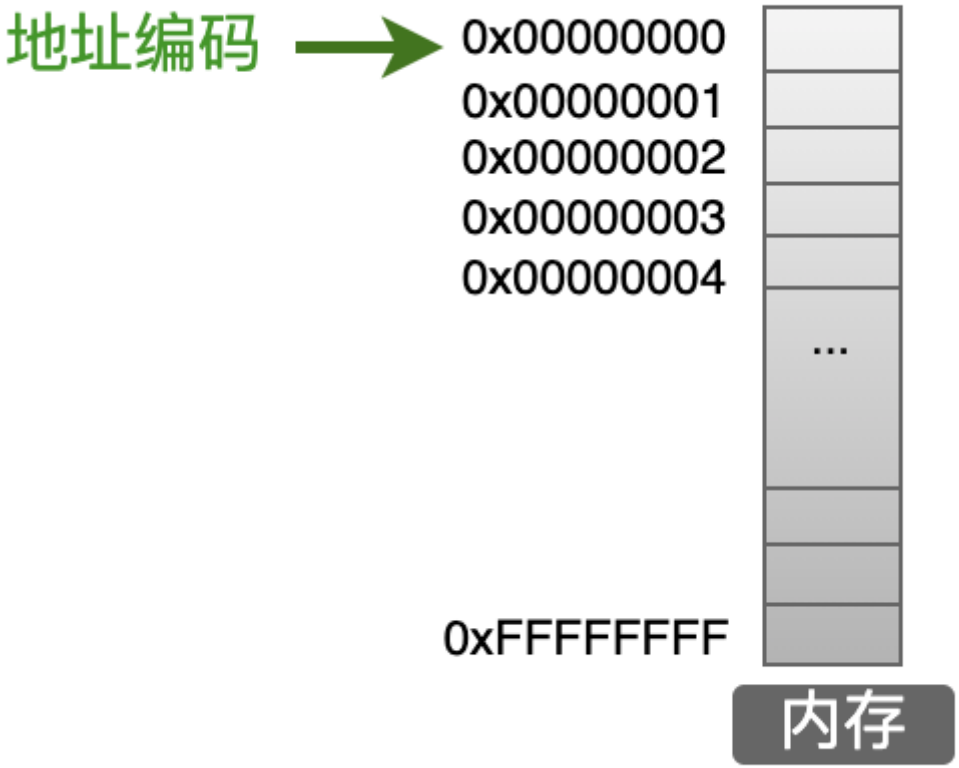


當程序開始被執行時，就變成一個動態的狀態，一般稱之為進程。

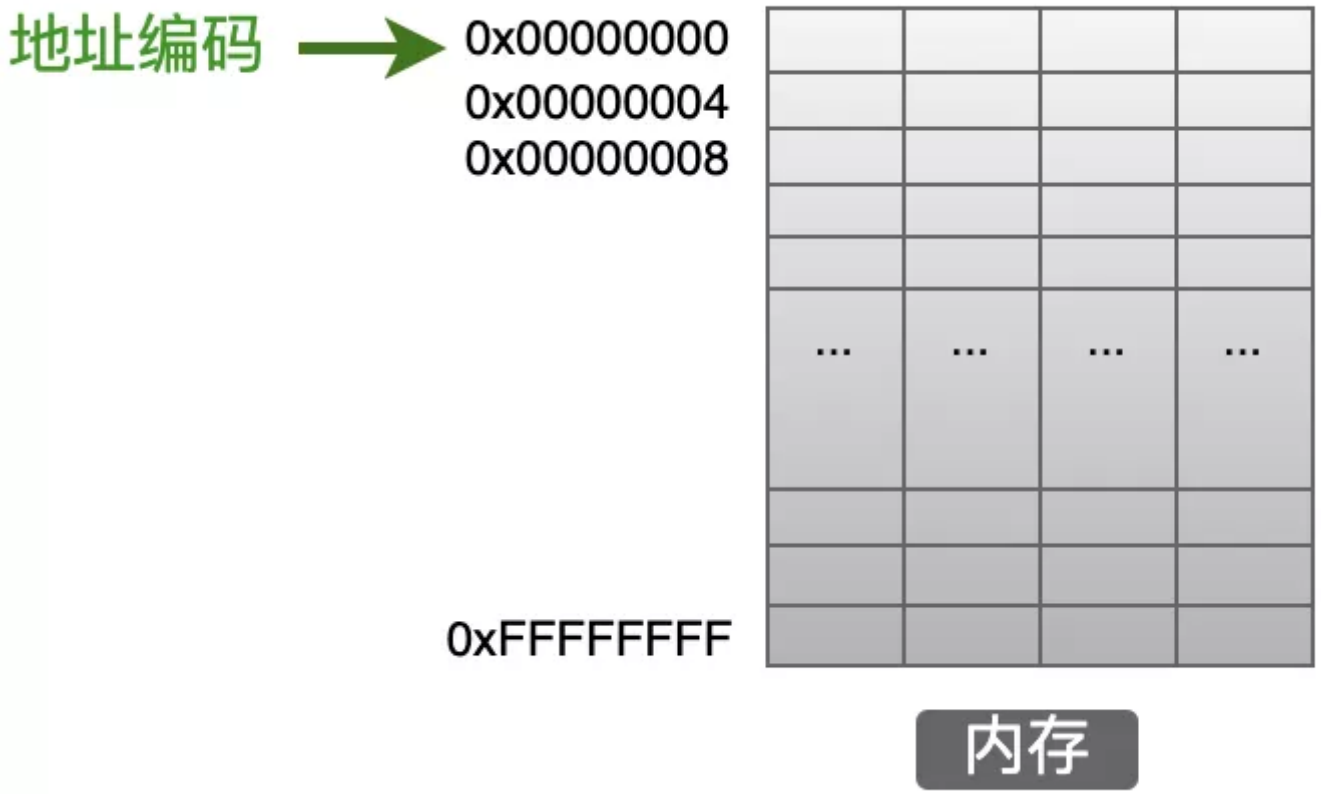
> 內存分為：物理內存和虛擬內存。操作系統對物理內存進行管理、包裝，我們開發者面對的是操作系統提供的虛擬內存。
> 這2個概念不妨礙文章的理解，因此就統一稱之為內存。

在我們的程序中，通過一個變量名來定義變量、使用變量。變量本身是一個確確實實存在的東西，變量名是一個抽象的概念，用來代表這個變量。就比如：我是一個實實在在的人，是客觀存在與這個地球上的，道哥是我給自己起的一個名字，這個名字是任意取得，只要自己覺得好听就行，如果我願意還可以起名叫：鳥哥、龍哥等等。

那麼，我們定義一個變量之後，這個變量放在哪裡呢？那就是內存的數據區。內存是一個很大的存儲區域，被操作系統劃分為一個一個的小空間，操作系統通過地址來管理內存。



內存中的最小存儲單位是字節(8個bit)，一個內存的完整空間就是由這一個一個的字節連續組成的。在上圖中，每一個小格子代表一個字節，但是好像大家在書籍中沒有這麼來畫內存模型的，更常見的是下面這樣的畫法：

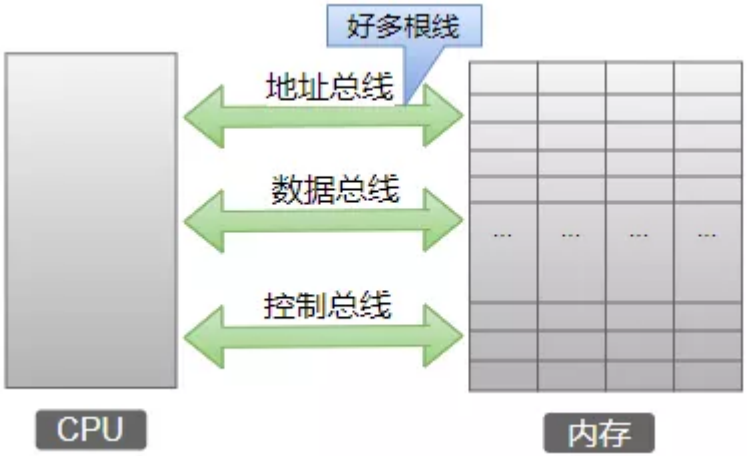


也就是把連續的4個字節的空間畫在一起，這樣就便於表述和理解，特別是深入到代碼對齊相關知識時更容易理解。(我認為根本原因應該是：大家都這麼畫，已經看順眼了~~)

2. 32位與64位系統

我們平時所說的計算機是32位、64位，指的是計算機的CPU中寄存器的最大存儲長度，如果寄存器中最大存儲32bit的數據，就稱之為32位系統。

在計算機中，數據一般都是在硬盤、內存和寄存器之間進行來回存取。CPU通過3種總線把各組成部分聯繫在一起：地址總線、數據總線和控制總線。地址總線的寬度決定了CPU的尋址能力，也就是CPU能達到的最大地址範圍。



剛才說了，內存是通過地址來管理的，那麼CPU想從內存中的某個地址空間上存取一個數據，那麼CPU就需要在地址總線上輸出這個存儲單元的地址。假如地址總線的寬度是8位，能表示的最大地址空間就是256個字節，能找到內存中最大的存儲單元是255這個格子(從0開始)。即使內存條的實際空間是2G字節，CPU也沒法使用後面的內存地址空間。如果地址總線的寬度是32位，那麼能表示的最大地址就是2的32次方，也就是4G字節的空間。

【注意】：這裡只是描述地址總線的概念，實際的計算機中地址計算方式要復雜的多，比如：虛擬內存中採用分段、分頁、偏移量來定位實際的物理內存，在分頁中還有大頁、小頁之分，感興趣的同學可以自己查一下相關資料。

3. 變量

我們在C程序中使用變量來“代表”一個數據，使用函數名來“代表”一個函數，變量名和函數名是程序員使用的助記符。變量和函數最終是要放到內存中才能被CPU使用的，而內存中所有的信息(代碼和數據)都是以二進制的形式來存儲的，計算機根據就不會從格式上來區分哪些是代碼、哪些是數據。CPU在訪問內存的時候需要的是地址，而不是變量名、函數名。

問題來了：在程序代碼中使用變量名來指代變量，而變量在內存中是根據地址來存放的，這二者之間如何映射(關聯)起來的？

答案是：編譯器！編譯器在編譯文本格式的C程序文件時，會根據目標運行平台(就是編譯出的二進製程序運行在哪裡？是x86平台的電腦？還是ARM平台的開發板？)來安排程序中的各種地址，例如：加載到內存中的地址、代碼段的入口地址等等，同時編譯器也會把程序中的所有變量名，轉成該變量在內存中的存儲地址。

變量有2個重要屬性：變量的類型和變量的值。

示例：代碼中定義了一個變量

```
int a = 20;
```

類型是int型，值是20。這個變量在內存中的存儲模型為：

我們在代碼中使用變量名a，在程序執行的時候就表示使用0x11223344地址所對應的那個存儲單元中的數據。因此，可以理解為變量名a就等價於這個地址0x11223344。換句話說，如果我們可以提前知道編譯器把變量a安排在地址0x11223344這個單元格中，我們就可以在程序中直接用這個地址值來操作這個變量。

在上圖中，變量a的值為20，在內存中佔據了4個格子的空間，也就是4個字節。為什麼是4個字節呢？在C標準中並沒有規定每種數據類型的變量一定要佔用幾個字節，這是與具體的機器、編譯器有關。

比如：32位的編譯器中：

> char: 1個字節；
> short int: 2個字節；
> int: 4個字節；
> long: 4個字節。

比如：64位的編譯器中：

> char: 1個字節；
> short int: 2個字節；
> int: 4個字節；
> long: 8個字節。

為了方便描述，下面都以32位為例，也就是int型變量在內存中佔據4個字節。

另外，0x11223344，0x11223345，0x11223346，0x11223347這連續的、從低地址到高地址的4個字節用來存儲變量a的數值20。在圖示中，使用十六進制來表示，十進制數值20轉成16進制就是：0x00000014，所以從開始地址依次存放0x00、0x00、0x00、0x14這4個字節(存儲順序涉及到大小端的問題，不影響文本理解)。

根據這個圖示，如果在程序中想知道變量a存儲在內存中的什麼位置，可以使用取地址操作符&，如下：

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

這句話將會打印出：`&a = 0x11223344`。

考慮一下，在32位系統中：指針變量佔用幾個字節？

4. 指針變量

指針變量可以分2個層次來理解：

> 1. 指針變量首先是一個變量，所以它擁有變量的所有屬性：類型和值。它的類型就是指針，它的值是其他變量的地址。既然是一個變量，那麼在內存中就需要為這個變量分配一個存儲空間。在這個存儲空間中，存放著其他變量的地址。
> 2. 指針變量所指向的數據類型，這是在定義指針變量的時候就確定的。例如：int *p; 意味著指針指向的是一個int型的數據。

首先回答一下剛才那個問題，在32位系統中，一個指針變量在內存中佔據4個字節的空間。因為CPU對內存空間尋址時，使用的是32位地址空間(4個字節)，也就是用4個字節就能存儲一個內存單元的地址。而指針變量中的值存儲的就是地址，所以需要4個字節的空間來存儲一個指針變量的值。

示例：

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

在內存中的存儲模型如下：

對於指針變量pa來說，首先它是一個變量，因此在內存中需要有一個空間來存儲這個變量，這個空間的地址就是0x11223348；

其次，這個內存空間中存儲的內容是變量a的地址，而a的地址為0x11223344，所以指針變量pa的地址空間中，就存儲了0x11223344這個值。

這裡對兩個操作符&和*進行說明：

> &：取地址操作符，用來獲取一個變量的地址。上面代碼中&a就是用來獲取變量a在內存中的存儲地址，也就是0x11223344。
>
> *：這個操作符用在2個場景中：定義一個指針的時候，獲取一個指針所指向的變量值的時候。
>
> 1. int *pa;*這個語句中的表示定義的變量pa是一個指針，前面的int表示pa這個指針指向的是一個int類型的變量。不過此時我們沒有給pa進行賦值，也就是說此刻pa對應的存儲單元中的4個字節裡的值是沒有初始化的，可能是0x00000000，也可能是其他任意的數字，不確定；
> 2. printf語句中的*表示獲取pa指向的那個int類型變量的值，學名叫解引用，我們只要記住是獲取指向的變量的值就可以了。

5. 操作指針變量

對指針變量的操作包括3個方面：

1. 操作指針變量自身的值；
2. 獲取指針變量所指向的數據;
3. 以什麼樣數據類型來使用/解釋指針變量所指向的內容。

5.1 指針變量自身的值

`int a = 20;` 這個語句是定義變量a，在隨後的代碼中，只要寫下a就表示要操作變量a中存儲的值，操作有兩種：讀和寫。

`printf("a = %d \n", a);` 這個語句就是要讀取變量a中的值，當然是20；
`a = 100;` 這個語句就是要把一個數值100寫入到變量a中。

同樣的道理，`int *pa;` 語句是用來定義指針變量pa，在隨後的代碼中，只要寫下pa就表示要操作變量pa中的值：

`printf("pa = %d \n", pa);` 這個語句就是要讀取指針變量pa中的值，當然是0x11223344；
`pa = &a;` 這個語句就是要把新的值寫入到指針變量pa中。再次強調一下，指針變量中存儲的是地址，如果我們可以提前知道變量a的地址是0x11223344，那麼我們也可以這樣來賦值:pa = 0x11223344;

思考一下，如果執行這個語句`printf("&pa =0x%x \n", &pa);`，打印結果會是什麼？

上面已經說過，操作符&是用來取地址的，那麼&pa就表示獲取指針變量pa的地址，上面的內存模型中顯示指針變量pa是存儲在0x11223348這個地址中的，因此打印結果就是：`&pa = 0x11223348`。

5.2 獲取指針變量所指向的數據

指針變量所指向的數據類型是在定義的時候就明確的，也就是說指針pa指向的數據類型就是int型，因此在執行`printf("value = %d \n", *pa);`語句時，首先知道pa是一個指針，其中存儲了一個地址(0x11223344)，然後通過操作符*來獲取這個地址(0x11223344)對應的那個存儲空間中的值；又因為在定義pa時，已經指定了它指向的值是一個int型，所以我們就知道了地址0x11223344中存儲的就是一個int類型的數據。

5.3 以什麼樣的數據類型來使用/解釋指針變量所指向的內容

如下代碼：

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

根據以上的描述，我們知道printf的打印結果會是`value = 30000`，十進制的30000轉成十六進制是0x00007530，內存模型如下：

現在我們做這樣一個測試：

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

指針變量pc在定義的時候指明：它指向的數據類型是char型，pc變量中存儲的地址是0x11223344。當使用*pc獲取指向的數據時，將會按照char型格式來讀取0x11223344地址處的數據，因此將會打印 `value = 0` (在計算機中，ASCII碼是用等價的數字來存儲的)。

這個例子中說明了一個重要的概念：在內存中一切都是數字，如何來操作(解釋)一個內存地址中的數據，完全是由我們的代碼來告訴編譯器的。剛才這個例子中，雖然0x11223344這個地址開始的4個字節的空間中，存儲的是整型變量a的值，但是我們讓pc指針按照char型數據來使用/解釋這個地址處的內容，這是完全合法的。

以上內容，就是指針最根本的心法了。把這個心法整明白了，剩下的就是多見識、多練習的問題了。

三、指針的幾個相關概念

1. const屬性

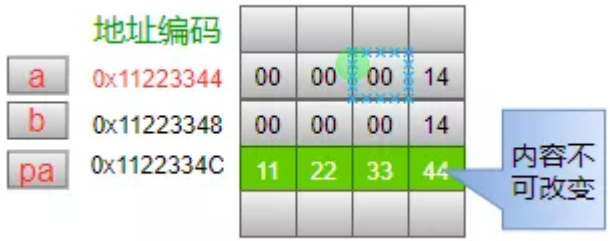
const標識符用來表示一個對象的不可變的性質，例如定義：

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

在後面的代碼中就不能改變變量b的值了，b中的值永遠是20。同樣的，如果用const來修飾一個指針變量：

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

內存模型如下：



這裡的const用來修飾指針變量p，根據const的性質可以得出結論：p在定義為變量a的地址之後，就固定了，不能再被改變了，也就是說指針變量pa中就只能存儲變量a的地址0x11223344。如果在後面的代碼中寫 `p = &b;` ，編譯時就會報錯，因為p是不可改變的，不能再被設置為變量b的地址。

但是，指針變量p所指向的那個變量a的值是可以改變的，即：`*p = 21;` 這個語句是合法的，因為指針p的值沒有改變(仍然是變量c的地址0x11223344)，改變的是變量c中存儲的值。

與下面的代碼區分一下：

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

這裡的const沒有放在p的旁邊，而是放在了類型int的旁邊，這就說明const符號不是用來修飾p的，而是用來修飾p所指向的那個變量的。所以，如果我們寫 `p = &b;` 把變量b的地址賦值給指針p，就是合法的，因為p的值可以被改變。

但是這個語句 `*p = 21` 就是非法了，因為定義語句中的const就限制了通過指針p獲取的數據，不能被改變，只能被用來讀取。這個性質常常被用在函數參數上，例如下面的代碼，用來計算一塊數據的CRC校驗，這個函數只需要讀取原始數據，不需要(也不可以)改變原始數據，因此就需要在形參指針上使用const修飾符：

```
short int getDataCRC(const char *pData, int len)
{
    short int crc = 0x0000;
```

```
    // 计算CRC
    return crc;
}
```

2. void型指針

關鍵字void並不是一個真正的數據類型，它體現的是一種抽象，指明不是任何一種類型，一般有2種使用場景：

> 1. 函數的返回值和形參;
> 2. 定義指針時不明確規定所指數據的類型，也就意味著可以指向任意類型。

指針變量也是一種變量，變量之間可以相互賦值，那麼指針變量之間也可以相互賦值，例如：

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

變量a賦值給變量b，指針p1賦值給指針p2，注意到它們的類型必須是相同的：a和b都是int型，p1和p2都是指向int型，所以可以相互賦值。那麼如果數據類型不同呢？必須進行強制類型轉換。例如：

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

內存模型如下：

p1指針指向的是int型數據，現在想把它的值(0x11223344)賦值給p2，但是由於在定義p2指針時規定它指向的數據類型是char型，因此需要把指針p1進行強制類型轉換，也就是把地址0x11223344處的數據按照char型數據來看待，然後才可以賦值給p2指針。

如果我們使用 `void *p2` 來定義p2指針，那麼在賦值時就不需要進行強制類型轉換了，例如：

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

指針p2是void*型，意味著可以把任意類型的指針賦值給p2，但是不能反過來操作，也就是不能把void*型指針直接賦值給其他確定類型的指針，而必須要強制轉換成被賦值指針所指向的數據類型，如下代碼，必須把p2指針強制轉換成int*型之後，再賦值給p3指針：

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

我們來看一個系統函數：

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

第一個參數類型是void*，這正體現了系統對內存操作的真正意義：它並不關心用戶傳來的指針具體指向什麼數據類型，只是把數據挨個存儲到這個地址對應的空間中。

第二個參數同樣如此，此外還添加了const修飾符，這樣就說明了memcpy函數只會從src指針處讀取數據，而不會修改數據。

3. 空指針和野指針

一個指針必須指向一個有意義的地址之後，才可以對指針進行操作。如果指針中存儲的地址值是一個隨機值，或者是一個已經失效的值，此時操作指針就非常危險了，一般把這樣的指針稱作野指針，C代碼中很多指針相關的bug就來源於此。

3.1 空指針：不指向任何東西的指針

在定義一個指針變量之後，如果沒有賦值，那麼這個指針變量中存儲的就是一個隨機值，有可能指向內存中的任何一個地址空間，此時萬萬不可以對這個指針進行寫操作，因為它有可能指向內存中的代碼段區域、也可能指向內存中操作系統所在的區域。

一般會將一個指針變量賦值為NULL來表示一個空指針，而C語言中，NULL實質是((void*)0) ， 在C++中，NULL實質是0。在標準庫頭文件stdlib.h中，有如下定義：

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

3.2 野指針：地址已經失效的指針

我們都知道，函數中的局部變量存儲在棧區，通過malloc申請的內存空間位於堆區，如下代碼：

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

內存模型為：

在堆區申請了4個字節的空間，然後強制類型轉換為int*型之後，賦值給指針變量p，然後通過*p設置這個地址中的值為14，這是合法的。如果在釋放了p指針指向的空間之後，再使用*p來操作這段地址，那就是非常危險了，因為這個地址空間可能已經被操作系統分配給其他代碼使用，如果對這個地址裡的數據強行操作，程序立刻崩潰的話，將會是我們最大的幸運！

```
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、指向不同數據類型的指針

1. 數值型指針

通過上面的介紹，指向數值型變量的指針已經很明白了，需要注意的就是指針所指向的數據類型。

2. 字符串指針

字符串在內存中的表示有2種：

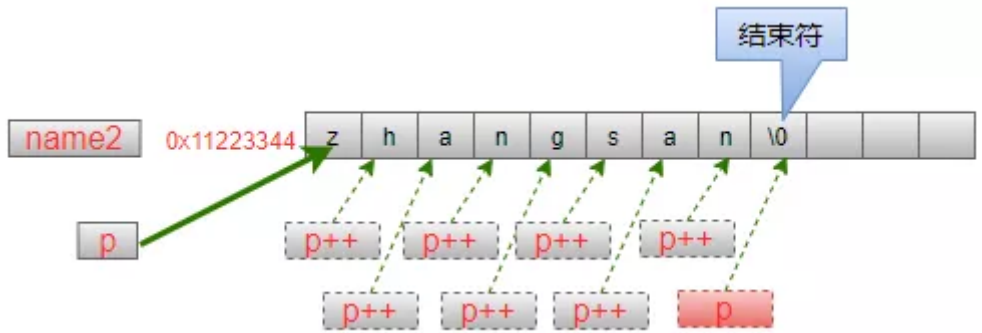
> 1. 用一個數組來表示，例如：char name1[8] = "zhangsan";
> 2. 用一個char *指針來表示，例如：char *name2 = "zhangsan";

name1在內存中佔據8個字節，其中存儲了8個字符的ASCII碼值；name2在內存中佔據9個字節，因為除了存儲8個字符的ASCII碼值，在最後一個字符'n'的後面還額外存儲了一個'\0'，用來標識字符串結束。

對於字符串來說，使用指針來操作是非常方便的，例如：變量字符串name2:

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

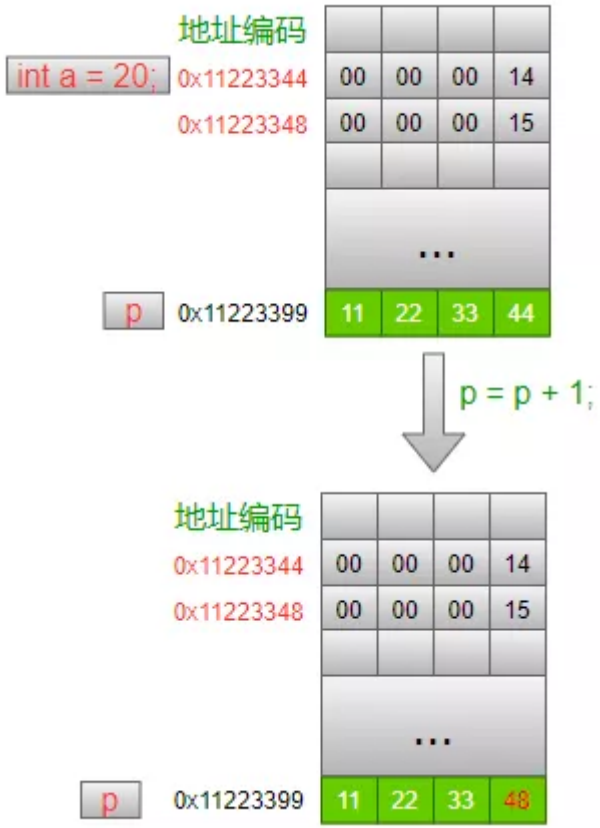
在while的判斷條件中，檢查p指針指向的字符是否為結束符'\0'。在循環體重，打印出當前指向的字符之後，對指針比那裡進行自增操作，因為指針p所指向的數據類型是char，每個char在內存中佔據一個字節，因此指針p在自增1之後，就指向下一個存儲空間。



也可以把循環體中的2條語句寫成1條語句：

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

假如一個指針指向的數據類型為int型，那麼執行 `p = p + 1;` 之後，指針p中存儲的地址值將會增加4，因為一個int型數據在內存中佔據4個字節的空間，如下所示：



思考一個問題：void*型指針能夠遞增嗎？如下測試代碼：

```
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);
```

打印結果如下：

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

說明void*型指針在自增時，是按照一個字節的跨度來計算的。

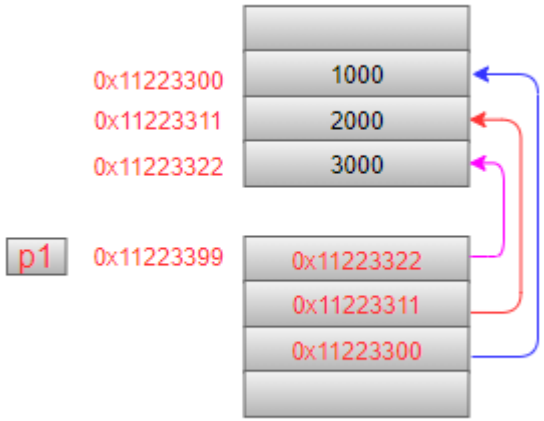
3. 指針數組與數組指針

這2個說法經常會混淆，至少我是如此，先看下這2條語句：

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

3.1 指針數組

第1條語句中：中括號[]的優先級高，因此與p1先結合，表示一個數組，這個數組中有3個元素，這3個元素都是指針，它們指向的是int型數據。可以這樣來理解：如果有這個定義 `char p[3]` ，很容易理解這是一個有3個char型元素的數組，那麼把char換成int*，意味著數組裡的元素類型是int*型(指向int型數據的指針)。內存模型如下(注意：三個指針指向的地址並不一定是連續的)：



如果向指針數組中的元素賦值，需要逐個把變量的地址賦值給指針元素：

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

3.2 數組指針

第2條語句中：小括號讓p2與*結合，表示p2是一個指針，這個指針指向了一個數組，數組中有3個元素，每一個元素的類型是int型。可以這樣來理解：如果有這個定義 `int p[3]` ，很容易理解這是一個有3個char型元素的數組，那麼把數組名p換成是*p2，也就是p2是一個指針，指向了這個數組。內存模型如下(注意：指針指向的地址是一個數組，其中的3個元素是連續放在內存中的)：

在前面我們說到取地址操作符&，用來獲得一個變量的地址。凡事都有特殊情況，對於獲取地址來說，下面幾種情況不需要使用&操作符：

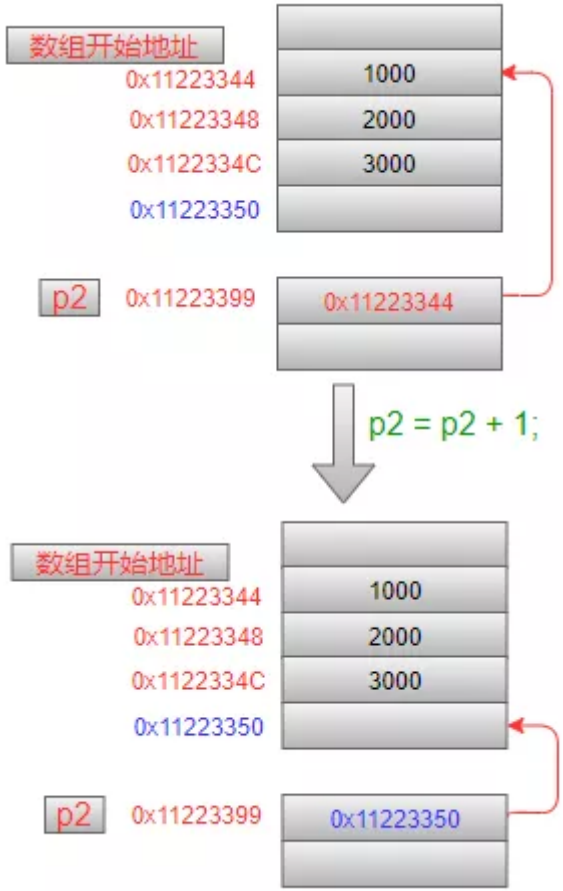
> 1. 字符串字面量作為右值時，就代表這個字符串在內存中的首地址；
> 2. 數組名就代表這個數組的地址，也等於這個數組的第一個元素的地址;
> 3. 函數名就代表這個函數的地址。

因此，對於一下代碼，三個printf語句的打印結果是相同的：

```
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，如果對這裡的p2指針執行 `p2 = p2 + 1;` 操作，p2中的值將會增加多少？

答案是12個字節。因為p2指向的是一個數組，這個數組中包含3個元素，每個元素佔據4個字節，那麼這個數組在內存中一共佔據12個字節，因此p2在加1之後，就跳過12個字節。



4. 二維數組和指針

一維數組在內存中是連續分佈的多個內存單元組成的，而二維數組在內存中也是連續分佈的多個內存單元組成的，從內存角度來看，一維數組和二維數組沒有本質差別。

和一維數組類似，二維數組的數組名表示二維數組的第一維數組中首元素的首地址，用代碼來說明：

```
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));
```

打印結果是：

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

我們拿第一個printf語句來分析：p0是一個指針，指向一個數組，數組中包含3個元素，每個元素在內存中佔據4個字節。現在我們想獲取這個數組中的數據，如果直接對p0執行加1操作，那麼p0將會跨過12個字節(就等於p1中的值了)，因此需要使用解引用操作符*，把p0轉為指向int型的指針，然後再執行加1操作，就可以得到數組中的int型數據了。

5. 結構體指針

C語言中的基本數據類型是預定義的，結構體是用戶定義的，在指針的使用上可以進行類比，唯一有區別的就是在結構體指針中，需要使用`->`箭頭操作符來獲取結構體中的成員變量，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);
```

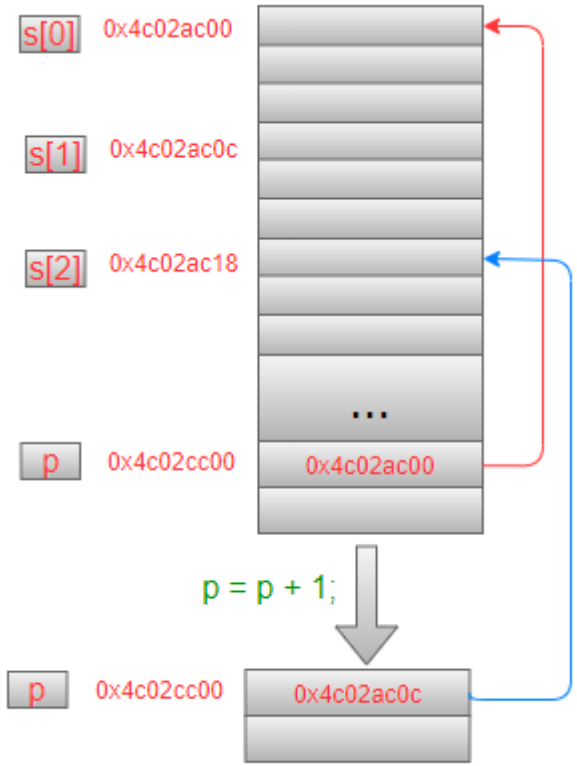
看起來似乎沒有什麼技術含量，如果是結構體數組呢？例如：

```
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);
```

打印結果是：

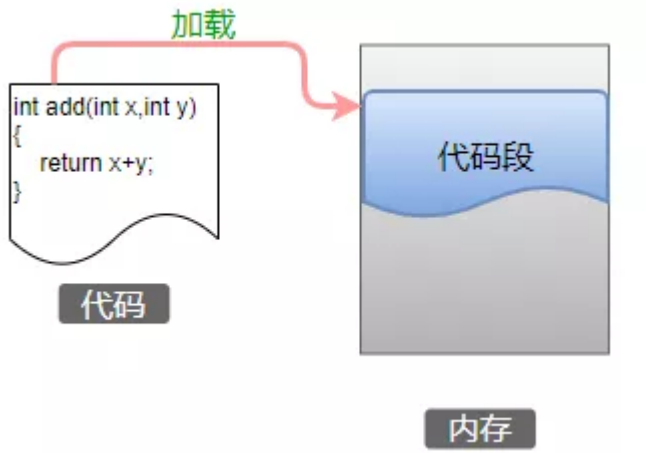
```
size of Student = 12
1: 0x4c02ac00, 0x4c02ac00
2: 0x4c02ac0c
```

在執行 `p++` 操作後，p需要跨過的空間是一個結構體變量在內存中佔據的大小(12個字節)，所以此時p就指向了數組中第2個元素的首地址，內存模型如下：



6. 函數指針

每一個函數在經過編譯之後，都變成一個包含多條指令的集合，在程序被加載到內存之後，這個指令集合被放在代碼區，我們在程序中使用函數名就代表了這個指令集合的開始地址。



函數指針，本質上仍然是一個指針，只不過這個指針變量中存儲的是一個函數的地址。函數最重要特性是什麼？可以被調用！因此，當定義了一個函數指針並把一個函數地址賦值給這個指針時，就可以通過這個函數指針來調用函數。

如下示例代碼：

```
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));
}
```

前文已經說過，函數的名字就代表函數的地址，所以函數名add就代表了這個加法函數在內存中的地址。`int (*p)(int, int);`這條語句就是用來定義一個函數指針，它指向一個函數，這個函數必須符合下面這2點(學名叫：函數簽名)：

> 1. 有2個int型的參數;
> 2. 有一個int型的返回值。

代碼中的add函數正好滿足這個要求，因此，可以把add賦值給函數指針p，此時p就指向了內存中這個函數存儲的地址，後面就可以用函數指針p來調用這個函數了。

在示例代碼中，函數指針p是直接定義的，那如果想定義2個函數指針，難道需要像下面這樣定義嗎？

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

這裡的參數比較簡單，如果函數很複雜，這樣的定義方式豈不是要煩死？可以用typedef關鍵字來定義一個函數指針類型：

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

然後用這樣的方式 `pFunc p1, p2;` 來定義多個函數指針就方便多了。注意：只能把與函數指針類型具有相同簽名的函數賦值給p1和p2，也就是參數的個數、類型要相同，返回值也要相同。

注意：這裡有幾個小細節稍微了解一下：

> 1. 在賦值函數指針時，使用p = &a;也是可以的；
> 2. 使用函數指針調用時，使用(*p)(a, b);也是可以的。

這裡沒有什麼特殊的原理需要講解，最終都是編譯器幫我們處理了這裡的細節，直接記住即可。

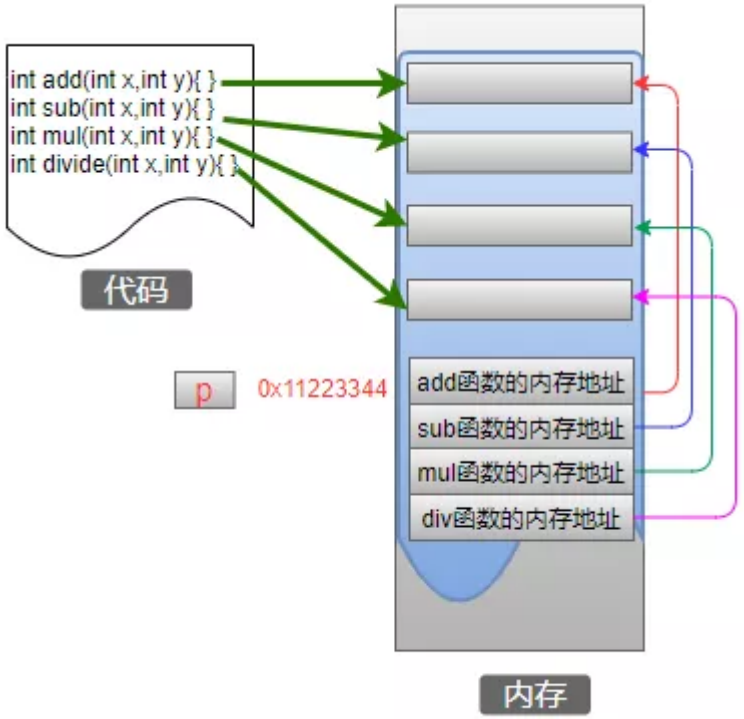
函數指針整明白之後，再和數組結合在一起：函數指針數組。示例代碼如下：

```
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));
}
```

這條語句不太好理解: `int (*p[4])(int, int);` ，先分析中間部分，標識符p與中括號[]結合(優先級高)，所以p是一個數組，數組中有4個元素；然後剩下的內容表示一個函數指針，那麼就說明數組中的元素類型是函數指針，也就是其他函數的地址，內存模型如下：



如果還是難以理解，那就回到指針的本質概念上：指針就是一個地址！這個地址中存儲的內容是什麼根本不重要，重要的是你告訴計算機這個內容是什麼。如果你告訴它：這個地址裡存放的內容是一個函數，那麼計算機就去調用這個函數。那麼你是如何告訴計算機的呢，就是在定義指針變量的時候，僅此而已！

五、總結

我已經把自己知道的所有指針相關的概念、語法、使用場景都作了講解，就像一個小酒館的掌櫃，把自己的美酒佳餚都呈現給你，但願你已經酒足飯飽！

如果以上的內容太多，一時無法消化，那麼下面的這兩句話就作為飯後甜點為您奉上，在以後的編程中，如果遇到指針相關的困惑，就想一想這兩句話，也許能讓你茅塞頓開。

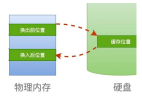
1. 指針就是地址，地址就是指針。
2. 指針就是指向內存中的一塊空間，至於如何來解釋/操作這塊空間，由這個指針的類型來決定。

另外還有一點囑咐，那就是學習任何一門編程語言，一定要弄清楚內存模型，內存模型，內存模型！

--- EOF ---

Linux系統是如何用虛擬內存來欺騙應用程序的?

IOT物聯網小鎮



右值引用的意義!

程序喵大人