万字长文带你从内存看指针 C语言指针完全解析

> 链接：https://zhuanlan.zhihu.com/p/298363575

注：这篇文章好好看完一定会让你掌握好指针的本质

C语言最核心的知识就是指针，所以，这一篇的文章主题是「指针与内存模型」

说到指针，就不可能脱离开内存，学会指针的人分为两种，一种是不了解内存模型，另外一种则是了解。

不了解的对指针的理解就停留在"指针就是变量的地址"这句话，会比较害怕使用指针，特别是各种高级操作。

而了解内存模型的则可以把指针用得炉火纯青，各种 byte 随意操作，让人直呼 666。

想学好C语言，很关键就是搞懂内存、指针、还有各种编译链接，

一、内存本质

编程的本质其实就是操控数据，数据存放在内存中。

因此，如果能更好地理解内存的模型，以及 C 如何管理内存，就能对程序的工作原理洞若观火，从而使编程能力更上一层楼。

大家真的别认为这是空话，我大一整年都不敢用 C 写上千行的程序也很抗拒写 C。

因为一旦上千行，经常出现各种莫名其妙的内存错误，一不小心就发生了 coredump..... 而且还无从排查，分析不出原因。

相比之下，那时候最喜欢 Java，在 Java 里随便怎么写都不会发生类似的异常，顶多偶尔来个 **NullPointerException**，也是比较好排查的。

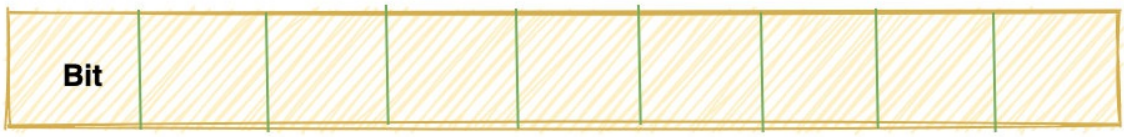
直到后来对内存和指针有了更加深刻的认识，才慢慢会用 C 写上千行的项目，也很少会再有内存问题了。（过于自信

「指针存储的是变量的内存地址」这句话应该任何讲 C 语言的书都会提到吧。

所以，要想彻底理解指针，首先要理解 C 语言中变量的存储本质，也就是内存。

1.1 内存编址

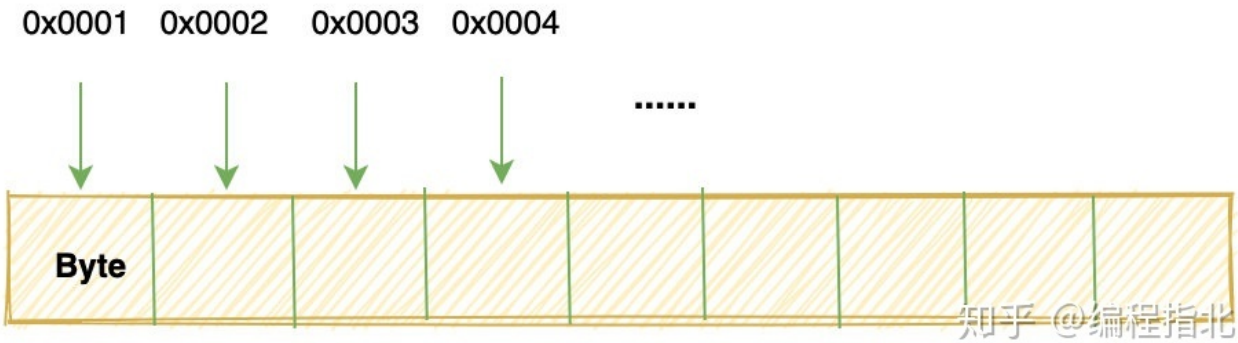
计算机的内存是一块用于存储数据的空间，由一系列连续的存储单元组成，就像下面这样，



每一个单元格都表示 1 个 Bit，一个 bit 在 EE 专业的同学看来就是高低电位，而在 CS 同学看来就是 0、1 两种状态。

由于 1 个 bit 只能表示两个状态，所以大佬们规定 8个 bit 为一组，命名为 byte。

并且将 byte 作为内存寻址的最小单元，也就是给每个 byte 一个编号，这个编号就叫内存的**地址**。



这就相当于，我们给小区里的每个单元、每个住户都分配一个门牌号： 301、302、403、404、501.....

在生活中，我们需要保证门牌号唯一，这样就能通过门牌号很精准的定位到一家人。

同样，在计算机中，我们也要保证给每一个 byte 的编号都是唯一的，这样才能够保证每个编号都能访问到唯一确定的 byte。

1.2 内存地址空间

上面我们说给内存中每个 byte 唯一的编号，那么这个编号的范围就决定了计算机可寻址内存的范围。

所有编号连起来就叫做内存的地址空间，这和大家平时常说的电脑是 32 位还是 64 位有关。

早期 Intel 8086、8088 的 CPU 就是只支持 16 位地址空间，**寄存器**和**地址总线**都是 16 位，这意味着最多对 2^16 = 64 Kb 的内存编号寻址。

这点内存空间显然不够用，后来，80286 在 8086 的基础上将**地址总线**和**地址寄存器**扩展到了20 位，也被叫做 A20 地址总线。

当时在写 mini os 的时候，还需要通过 BIOS 中断去启动 A20 地址总线的开关。

但是，现在的计算机一般都是 32 位起步了，32 位意味着可寻址的内存范围是 2^32 byte = 4GB。

所以，如果你的电脑是 32 位的，那么你装超过 4G 的内存条也是无法充分利用起来的。

好了，这就是内存和内存编址。

1.3 变量的本质

有了内存，接下来我们需要考虑，int、double 这些变量是如何存储在 0、1 单元格的。

在 C 语言中我们会这样定义变量：

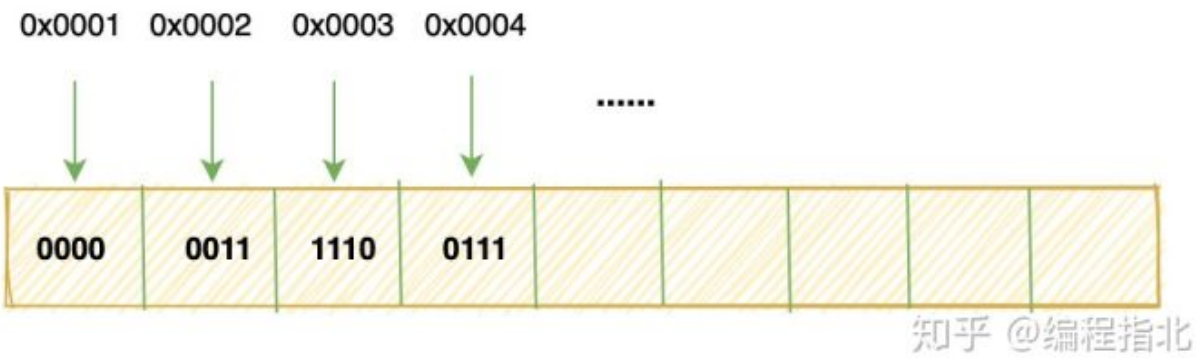
```
int a = 999;
char c = 'c';
```

当你写下一个变量定义的时候，实际上是向内存申请了一块空间来存放你的变量。

我们都知道 int 类型占 4 个字节，并且在计算机中数字都是用补码（不了解补码的记得去百度）表示的。

999 换算成补码就是：0000 0011 1110 0111

这里有 4 个byte，所以需要四个单元格来存储：



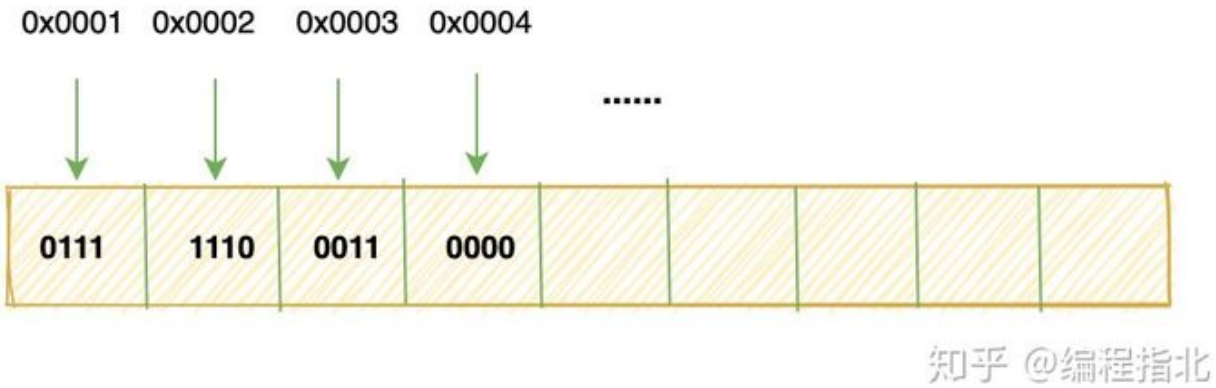
有没有注意到，我们把高位的字节放在了低地址的地方。

那能不能反过来呢？

当然，这就引出了**大端和小端。**

像上面这种将高位字节放在内存低地址的方式叫做**大端**

反之，将低位字节放在内存低地址的方式就叫做**小端**：



上面只说明了 int 型的变量如何存储在内存，而 float、char 等类型实际上也是一样的，都需要先转换为补码。

对于多字节的变量类型，还需要按照大端或者小端的格式，依次将字节写入到内存单元。

记住上面这两张图，这就是编程语言中所有变量的在内存中的样子，不管是 int、char、指针、数组、结构体、对象... 都是这样放在内存的。

觉得不错的话，可以关注下我公众号「编程指北」 ~

二、指针是什么东西?

2.1 变量放在哪?

上面我说，定义一个变量实际就是向计算机申请了一块内存来存放。

那如果我们要想知道变量到底放在哪了呢?

可以通过运算符&来取得变量实际的地址，这个值就是变量所占内存块的起始地址。

(PS: 实际上这个地址是虚拟地址，并不是真正物理内存上的地址

我们可以把这个地址打印出来:

```
printf("%x", &a);
```

大概会是像这样的一串数字:0x7ffcad3b8f3c

2.2 指针本质

上面说，我们可以通过&符号获取变量的内存地址，那获取之后如何来表示这是一个**地址**，而不是一个普通的值呢?

也就是在 C 语言中如何表示地址这个概念呢?

对，就是指针，你可以这样:

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

pa 中存储的就是变量 a 的地址，也叫做指向 a 的指针。

在这里我想谈几个看起来有点无聊的话题：

> 为什么我们需要指针？直接用变量名不行吗？

当然可以，但是变量名是有局限的。

> 变量名的本质是什么？

是变量地址的符号化，变量是为了让我们编程时更加方便，对人友好，可计算机可不认识什么变量 a，它只知道地址和指令。

所以当你去查看 C 语言编译后的汇编代码，就会发现变量名消失了，取而代之的是一串串抽象的地址。

你可以认为，编译器会自动维护一个映射，将我们程序中的变量名转换为变量所对应的地址，然后再对这个地址去进行读写。

也就是有这样一个映射表存在，将变量名自动转化为地址：

```
a  | 0x7ffcad3b8f3c
c  | 0x7ffcad3b8f2c
h  | 0x7ffcad3b8f4c
....
```

说的好！

可是我还是不知道指针存在的必要性，那么问题来了，看下面代码：

```
int func(...) {
  ...
};

int main() {
 int a;
 func(...);
};
```

假设我有一个需求：

> 要求在func 函数里要能够修改 main 函数里的变量 a，这下咋整，在 main 函数里可以直接通过变量名去读写 a 所在内存。 但是在 func 函数里是看不见a 的呀。

你说可以通过&取地址符号，将 a 的地址传递进去：

```
int func(int address) {
  ....
};

int main() {
 int a;
 func(&a);
};
```

这样在func 里就能获取到 a 的地址，进行读写了。

理论上这是完全没有问题的，但是问题在于:

编译器该如何区分一个 int 里你存的到底是 int 类型的值，还是另外一个变量的地址（即指针）。

这如果完全靠我们编程人员去人脑记忆了，会引入复杂性，并且无法通过编译器检测一些语法错误。

而通过int * 去定义一个指针变量，会非常明确：**这就是另外一个 int 型变量的地址。**

编译器也可以通过类型检查来排除一些编译错误。

这就是指针存在的必要性。

实际上任何语言都有这个需求，只不过很多语言为了安全性，给指针戴上了一层枷锁，将指针包装成了引用。

可能大家学习的时候都是自然而然的接受指针这个东西，但是还是希望这段啰嗦的解释对你有一定启发。

同时，在这里提点小问题:

既然指针的本质都是变量的内存首地址，即一个 int 类型的整数。

> 那为什么还要有各种类型呢？ 比如 int 指针，float 指针，这个类型影响了指针本身存储的信息吗？ 这个类型会在什么时候发挥作用？

2.3 解引用

上面的问题，就是为了引出指针解引用的。

pa中存储的是a变量的内存地址，那如何通过地址去获取a的值呢?

这个操作就叫做**解引用**，在 C 语言中通过运算符 *就可以拿到一个指针所指地址的内容了。

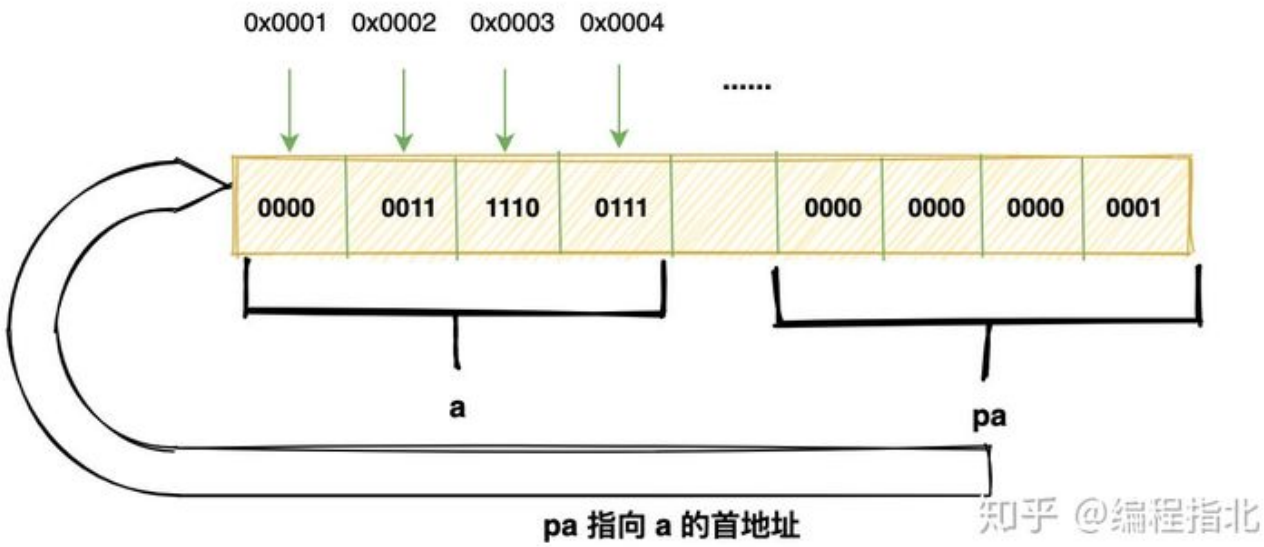
比如*pa就能获得a的值。

我们说指针存储的是变量内存的首地址，那编译器怎么知道该从首地址开始取多少个字节呢?

这就是指针类型发挥作用的时候，编译器会根据指针的所指元素的类型去判断应该取多少个字节。

如果是 int 型的指针，那么编译器就会产生提取四个字节的指令，char 则只提取一个字节，以此类推。

下面是指针内存示意图:



pa 指针首先是一个变量，它本身也占据一块内存，这块内存里存放的就是 a 变量的首地址。

当解引用的时候，就会从这个首地址连续划出 4 个 byte，然后按照 int 类型的编码方式解释。

2.4 活学活用

别看这个地方很简单，但却是深刻理解指针的关键。

举两个例子来详细说明：

比如：

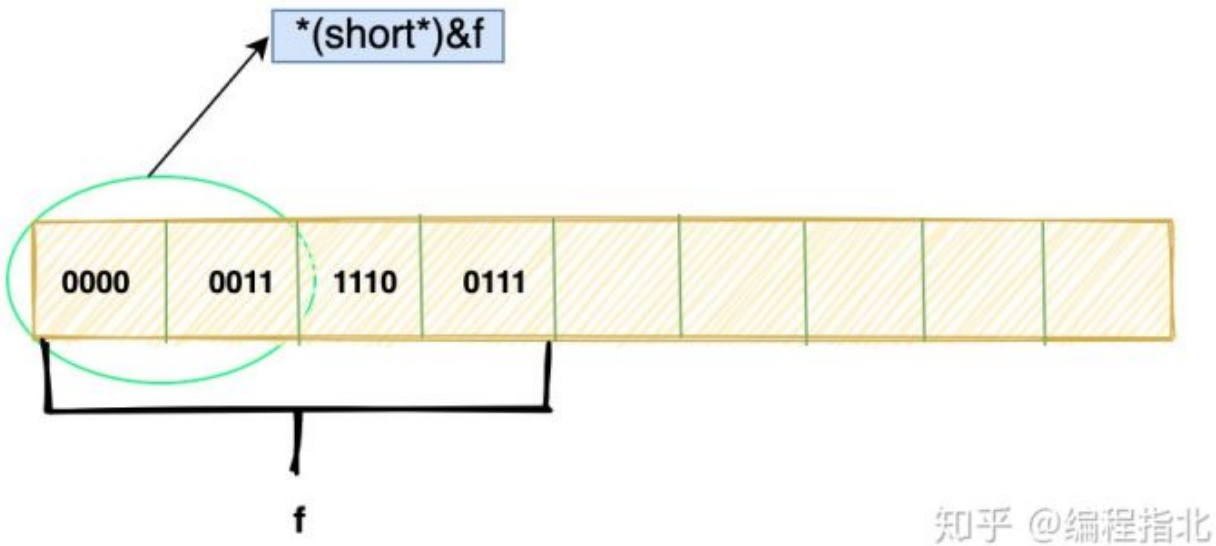
```
float f = 1.0;
short c = *(short*)&f;
```

你能解释清楚上面过程，对于 f 变量，在内存层面发生了什么变化吗？

或者 c 的值是多少？1？

实际上，从内存层面来说，f 什么都没变。

如图：



假设这是f 在内存中的位模式，这个过程实际上就是把 f 的前两个 byte 取出来然后按照 short 的方式解释，然后赋值给 c。

详细过程如下：

1. &f取得f 的首地址
2. (short*)&f

上面第二步什么都没做，这个表达式只是说：

“噢，我认为f这个地址放的是一个 short 类型的变量”

最后当去解引用的时候*(short)*&f时，编译器会取出前面两个字节，并且按照 short 的编码方式去解释，并将解释出的值赋给 c 变量。

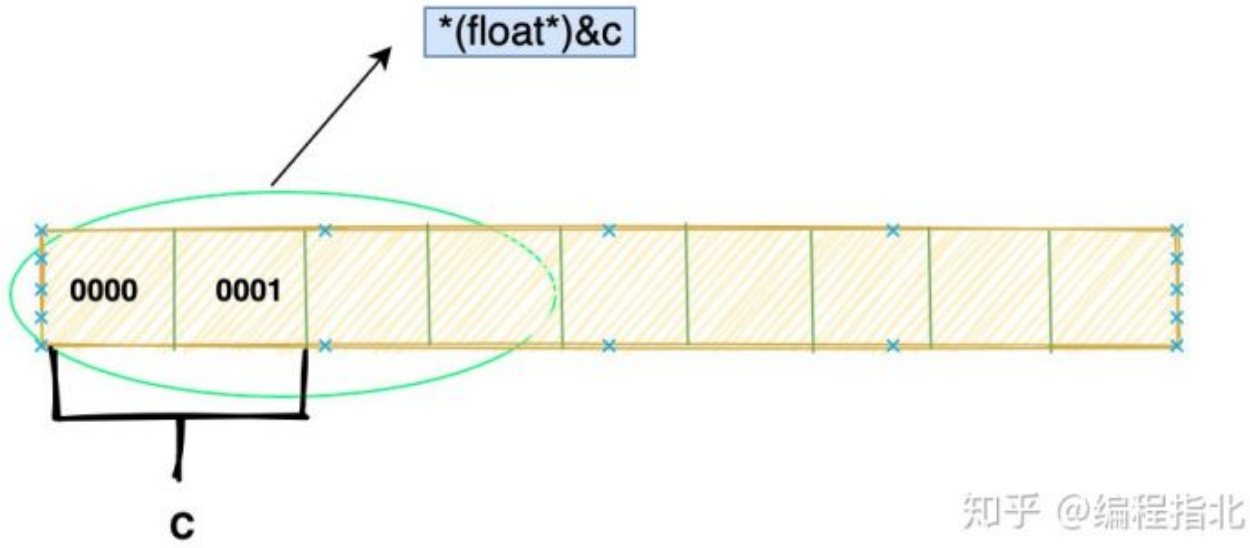
这个过程 f的位模式没有发生任何改变，变的只是解释这些位的方式。

当然，这里最后的值肯定不是 1，至于是什么，大家可以去真正算一下。

那反过来，这样呢？

```
short c = 1;
float f = *(float*)&c;
```

如图：



具体过程和上述一样，但上面肯定不会报错，这里却不一定。

为什么？

(float*)&c会让我们从c 的首地址开始取四个字节，然后按照 float 的编码方式去解释。

但是c是 short 类型只占两个字节，那肯定会访问到相邻后面两个字节，这时候就发生了内存访问越界。

当然，如果只是读，大概率是没问题的。

但是，有时候需要向这个区域写入新的值，比如：

```
*(float*)&c = 1.0;
```

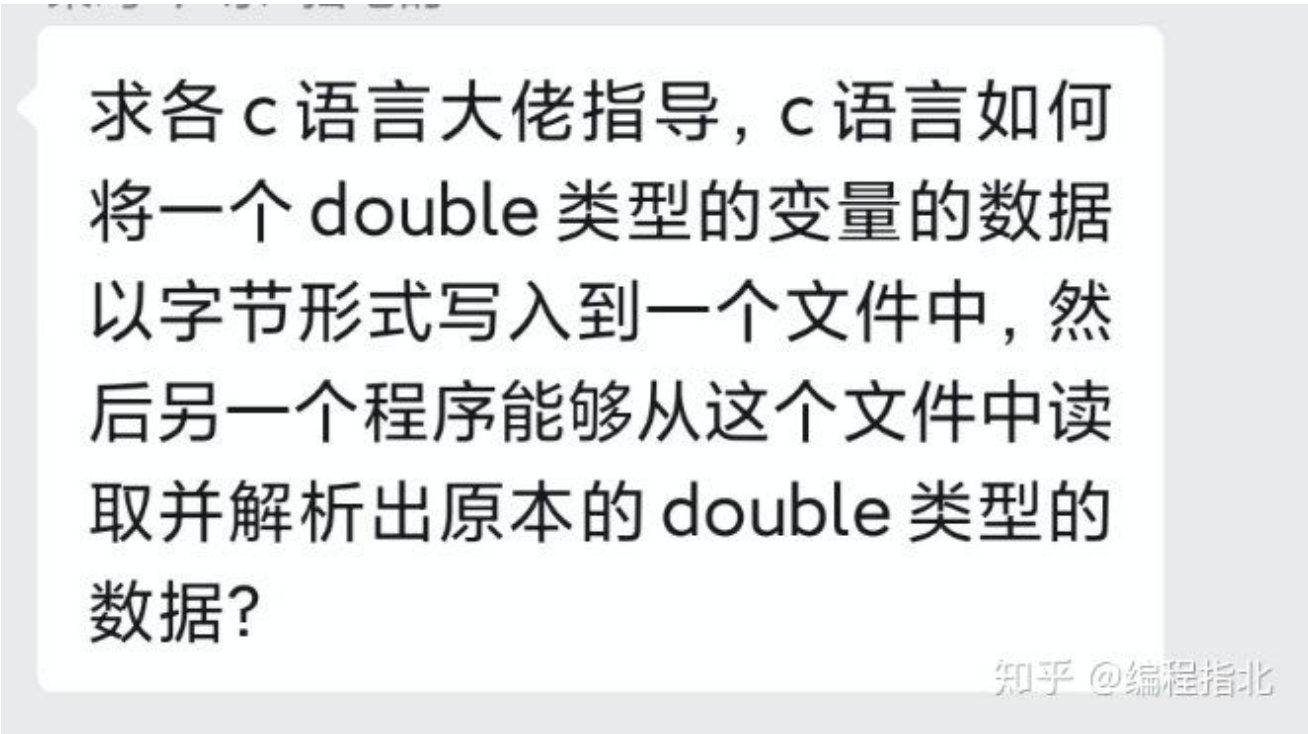
那么就可能发生 coredump，也就是访存失败。

另外，就算是不会 coredump，这种也会破坏这块内存原有的值，因为很可能这是是其它变量的内存空间，而我们去覆盖了人家的内容，肯定会导致隐藏的 bug。

如果你理解了上面这些内容，那么使用指针一定会更加的自如。

2.6 看个小问题

讲到这里，我们来看一个问题，这是一位群友问的，这是他的需求：



这是他写的代码：

```
FILE *fp;
float d1 = 0.1568;
printf( _Format: "%f %x\n", d1, d1);
char buffer[4];

fp = fopen( _Filename: "file.txt", _Mode: "w+");
fwrite(&d1, _Size: 4, _Count: 1, fp);

fseek(fp, _Offset: 0, SEEK_SET);

fread(buffer, _ElementSize: 4, _Count: 1, fp);
printf( _Format: "%f %x\n", *buffer, *buffer);
fclose(fp);
E:\c_project\test>gcc test.c&a.exe
0.156800 c0000000
0.000000 2e
```


他把 double 写进文件再读出来，然后发现打印的值对不上。

而关键的地方就在于这里:

```
char buffer[4];
...
printf("%f %x\n", *buffer, *buffer);
```

他可能认为 buffer 是一个指针（准确说是数组），对指针解引用就该拿到里面的值，而里面的值他认为是从文件读出来的 4 个byte，也就是之前的 float 变量。

注意，这一切都是他认为的，实际上编译器会认为：

“哦，buffer 是 char类型的指针，那我取第一个字节出来就好了”。

然后把第一个字节的值传递给了 printf 函数，printf 函数会发现，%f 要求接收的是一个 float 浮点数，那就会自动把第一个字节的值转换为一个浮点数打印出来。

这就是整个过程。

错误关键就是，这个同学误认为，任何指针解引用都是拿到里面“我们认为的那个值”，实际上编译器并不知道，编译器只会傻傻的按照指针的类型去解释。

所以这里改成：

```
printf("%f %x\n", *(float*)buffer, *(float*)buffer);
```

相当于明确的告诉编译器：

“buffer指向的这个地方，我放的是一个 float，你给我按照 float 去解释”

三、 结构体和指针

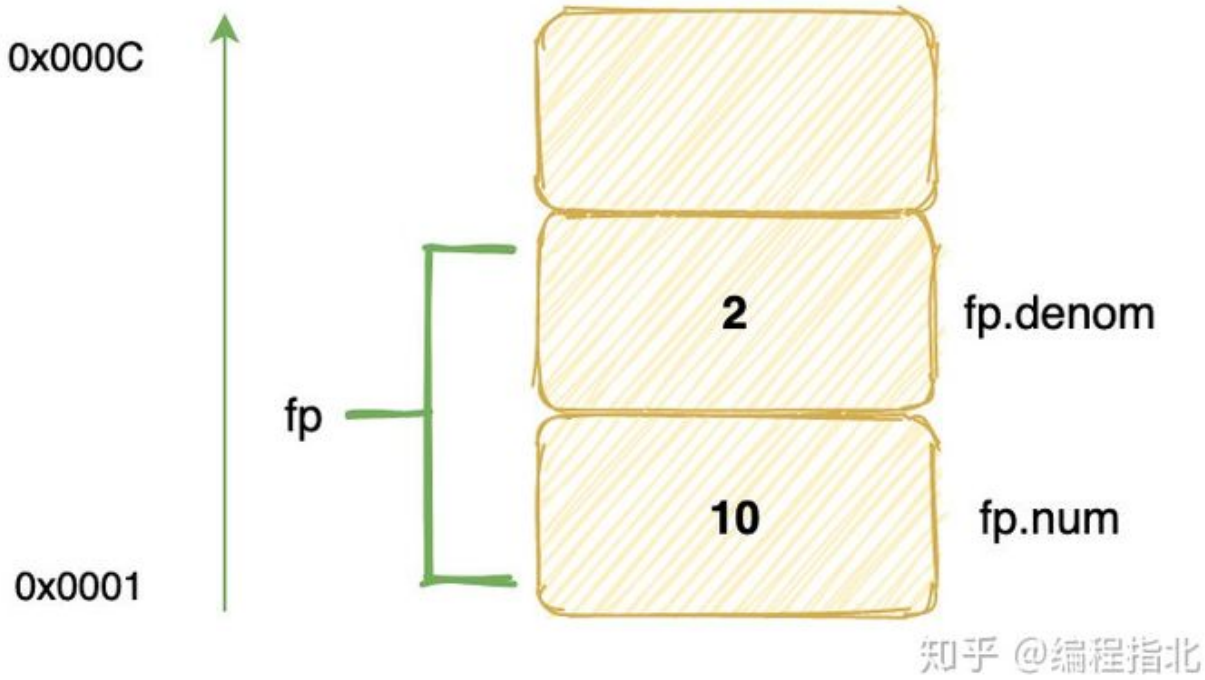
结构体内包含多个成员，这些成员之间在内存中是如何存放的呢？

比如：

```
struct fraction {
 int num; // 整数部分
 int denom; // 小数部分
};

struct fraction fp;
fp.num = 10;
fp.denom = 2;
```

这是一个定点小数结构体，它在内存占 8 个字节（这里不考虑内存对齐），两个成员域是这样存储的：



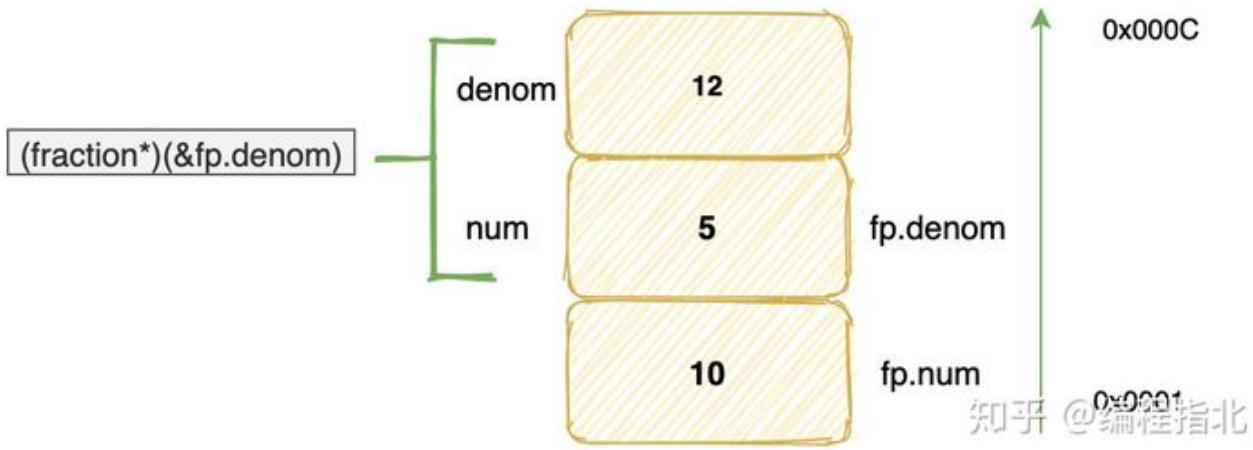
我们把 10 放在了结构体中基地址偏移为 0 的域，2 放在了偏移为 4 的域。

接下来我们做一个正常人永远不会做的操作：

```
((fraction*)(&fp.denom))->num = 5;
((fraction*)(&fp.denom))->denom = 12;
printf("%d\n", fp.denom); // 输出多少?
```

上面这个究竟会输出多少呢？自己先思考下噢~

接下来我分析下这个过程发生了什么：



首先，&fp.denom表示取结构体 fp 中 denom 域的首地址，然后以这个地址为起始地址取 8 个字节，并且将它们看做一个 fraction 结构体。

在这个新结构体中，最上面四个字节变成了 denom 域，而 fp 的 denom 域相当于新结构体的 num 域。

因此：

((fraction*)(&fp.denom))->num = 5

实际上改变的是 fp.denom，而

((fraction*)(&fp.denom))->denom = 12

则是将最上面四个字节赋值为 12。

当然，往那四字节内存写入值，结果是无法预测的，可能会造成程序崩溃，因为也许那里恰好存储着函数调用栈帧的关键信息，也可能那里没有写入权限。

大家初学 C 语言的很多 coredump 错误都是类似原因造成的。

所以最后输出的是 5。

为什么要讲这种看起来莫名其妙的代码?

就是为了说明结构体的本质其实就是一堆的变量打包放在一起，而访问结构体中的域，就是通过结构体的起始地址，也叫基地址，然后加上域的偏移。

其实，C++、Java 中的对象也是这样存储的，无非是他们为了实现某些面向对象的特性，会在数据成员以外，添加一些 Head 信息，比如C++ 的虚函数表。

实际上，我们是完全可以用 C 语言去模仿的。

这就是为什么一直说 C 语言是基础，你真正懂了 C 指针和内存，对于其它语言你也会很快的理解其对象模型以及内存布局。

四、多级指针

说起多级指针这个东西，我以前大一，最多理解到 2 级，再多真的会把我绕晕，经常也会写错代码。

你要是给我写个这个：int **p 能把我搞崩溃，我估计很多同学现在就是这种情况

其实，多级指针也没那么复杂，就是指针的指针的指针的指针……非常简单。

今天就带大家认识一下多级指针的本质。

首先，我要说一句话，没有多级指针这种东西，指针就是指针，多级指针只是为了我们方便表达而取的逻辑概念。

首先看下生活中的快递柜：

这种大家都用过吧，丰巢或者超市储物柜都是这样，每个格子都有一个编号，我们只需要拿到编号，然后就能找到对应的格子，取出里面的东西。

这里的格子就是内存单元，编号就是地址，格子里放的东西就对应存储在内存中的内容。

假设我把一本书，放在了 03 号格子，然后把 03 这个编号告诉你，你就可以根据 03 去取到里面的书。

那如果我把书放在 05 号格子，然后在 03 号格子只放一个小纸条，上面写着：「书放在 05 号」。

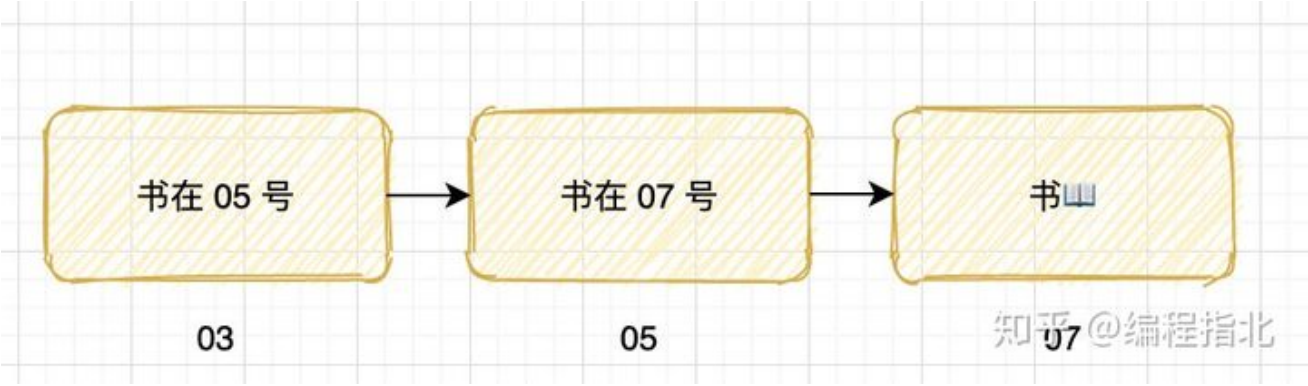
你会怎么做?

当然是打开 03 号格子，然后取出了纸条，根据上面内容去打开 05 号格子得到书。

这里的 03 号格子就叫指针，因为它里面放的是指向其它格子的小纸条（地址）而不是具体的书。

明白了吗?

那我如果把书放在 07 号格子，然后在 05 号格子 放一个纸条：「书放在 07号」，同时在03号格子放一个纸条「书放在 05号」



这里的 03 号格子就叫二级指针，05 号格子就叫指针，而 07 号就是我们平常用的变量。

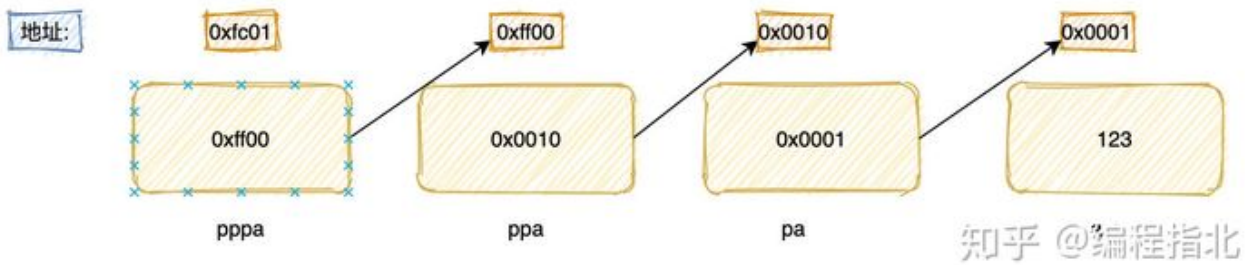
依次，可类推出 N 级指针。

所以你明白了吗？同样的一块内存，如果存放的是别的变量的地址，那么就叫指针，存放的是实际内容，就叫变量。

```
int a;
int *pa = &a;
int **ppa = &pa;
int ***pppa = &ppa;
```

上面这段代码，pa就叫一级指针，也就是平时常说的指针，ppa 就是二级指针。

内存示意图如下:



不管几级指针有两个最核心的东西:

- 指针本身也是一个变量，需要内存去存储，指针也有自己的地址
- 指针内存存储的是它所指向变量的地址

这就是我为什么多级指针是逻辑上的概念，实际上一块内存要么放实际内容，要么放其它变量地址，就这么简单。

怎么去解读int **a这种表达呢?

int * *a 可以把它分为两部分看，即int* 和 *a，后面 *a 中的*表示 a 是一个指针变量，前面的 int* 表示指针变量a

只能存放 int* 型变量的地址。

对于二级指针甚至多级指针，我们都可以把它拆成两部分。

首先不管是多少级的指针变量，它首先是一个指针变量，指针变量就是一个，*其余的*表示的是这个指针变量只能存放什么类型变量的地址。

比如inta表示指针变量 a 只能存放int*** 型变量的地址。

觉得不错的话，关注公众号「编程指北」，获取更多精彩文章噢~

五、指针与数组

5.1 一维数组

数组是 C 自带的基本数据结构，彻底理解数组及其用法是开发高效应用程序的基础。

数组和指针表示法紧密关联，在合适的上下文中可以互换。

如下：

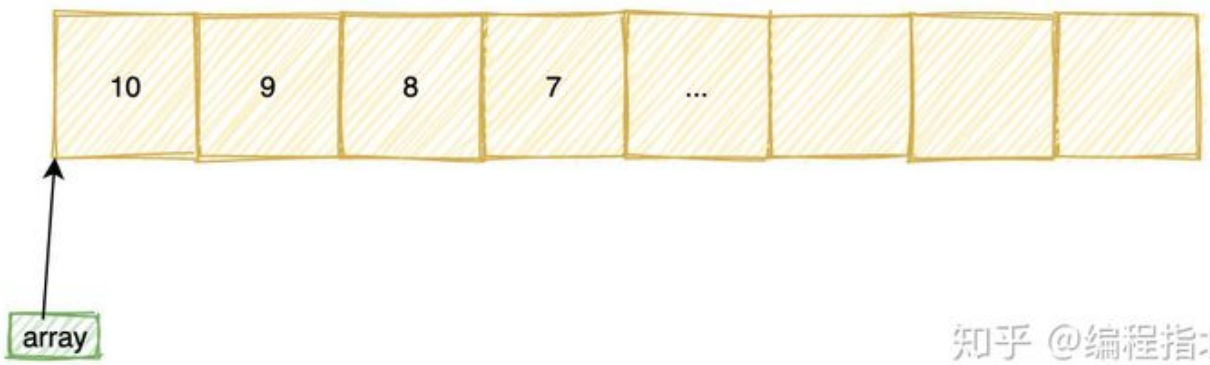
```
int array[10] = {10, 9, 8, 7};
printf("%d\n", *array);  //  输出 10
printf("%d\n", array[0]);  // 输出 10

printf("%d\n", array[1]);  // 输出 9
printf("%d\n", *(array+1)); // 输出 9

int *pa = array;
printf("%d\n", *pa);  //  输出 10
printf("%d\n", pa[0]);  // 输出 10

printf("%d\n", pa[1]);  // 输出 9
printf("%d\n", *(pa+1)); // 输出 9
```

在内存中，数组是一块连续的内存空间：



第 0 个元素的地址称为数组的首地址，数组名实际就是指向数组首地址，当我们通过array[1]或者*(array + 1) 去访问数组元素的时候。

实际上可以看做 address[offset]，address 为起始地址，offset 为偏移量，但是注意这里的偏移量offset 不是直接和 address相加，而是要乘以数组类型所占字节数，也就是： address + sizeof(int) * offset。

学过汇编的同学，一定对这种方式不陌生，这是汇编中寻址方式的一种：基址变址寻址。

看完上面的代码，很多同学可能会认为指针和数组完全一致，可以互换，这是完全错误的。

尽管数组名字有时候可以当做指针来用，但数组的名字不是指针。

最典型的地方就是在 sizeof:

```
printf("%u", sizeof(array));
printf("%u", sizeof(pa));
```

第一个将会输出 40，因为 array包含有 10 个int类型的元素，而第二个在 32 位机器上将会输出 4，也就是指针的长度。

为什么会这样呢?

站在编译器的角度讲，变量名、数组名都是一种符号，它们都是有类型的，它们最终都要和数据绑定起来。

变量名用来指代一份数据，数组名用来指代一组数据（数据集合），它们都是有类型的，以便推断出所指代的数据的长度。

对，数组也有类型，我们可以将 int、float、char 等理解为基本类型，将数组理解为由基本类型派生得到的稍微复杂一些的类型，

数组的类型由元素的类型和数组的长度共同构成。而 sizeof 就是根据变量的类型来计算长度的，并且计算的过程是在编译期，而不会在程序运行时。

编译器在编译过程中会创建一张专门的表格用来保存变量名及其对应的数据类型、地址、作用域等信息。

sizeof 是一个操作符，不是函数，使用 sizeof 时可以从这张表格中查询到符号的长度。

所以，这里对数组名使用sizeof可以查询到数组实际的长度。

pa 仅仅是一个指向 int 类型的指针，编译器根本不知道它指向的是一个整数，还是一堆整数。

虽然在这里它指向的是一个数组，但数组也只是一块连续的内存，没有开始和结束标志，也没有额外的信息来记录数组到底多长。

所以对 pa 使用 sizeof 只能求得的是指针变量本身的长度。

也就是说，编译器并没有把 pa 和数组关联起来，pa 仅仅是一个指针变量，不管它指向哪里，sizeof求得的永远是它本身所占用的字节数。

5.2 二维数组

大家不要认为二维数组在内存中就是按行、列这样二维存储的，实际上，不管二维、三维数组... 都是编译器的语法糖。

存储上和一维数组没有本质区别，举个例子:

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

或许你以为在内存中 array 数组会像一个二维矩阵:

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

可实际上它是这样的:

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

和一维数组没有什么区别，都是一维线性排列。

当我们像 array[1][1]这样去访问的时候，编译器会怎么去计算我们真正所访问元素的地址呢?

为了更加通用化，假设数组定义是这样的:

int array[n][m]

访问: array[a][b]

那么被访问元素地址的计算方式就是: array + (m * a + b)

这个就是二维数组在内存中的本质，其实和一维数组是一样的，只是语法糖包装成一个二维的样子。

六、神奇的 void 指针

想必大家一定看到过 void 的这些用法：

```
void func();
int func1(void);
```

在这些情况下，void 表达的意思就是没有返回值或者参数为空。

但是对于 void 型指针却表示通用指针，可以用来存放任何数据类型的引用。

下面的例子就 是一个 void 指针：

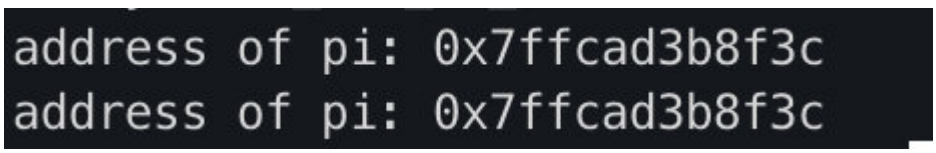
```
void *ptr;
```

void 指针最大的用处就是在 C 语言中实现泛型编程，因为任何指针都可以被赋给 void 指针，void 指针也可以被转换回原来的指针类型， 并且这个过程指针实际所指向的地址并不会发生变化。

比如:

```
int num;
int *pi = &num;
printf("address of pi: %p\n", pi);
void* pv = pi;
pi = (int*) pv;
printf("address of pi: %p\n", pi);
```

这两次输出的值都会是一样:



平常可能很少会这样去转换，但是当你用 C 写大型软件或者写一些通用库的时候，一定离不开 void 指针，这是 C 泛型的基石，比如 std 库里的 sort 函数申明是这样的:

```
void qsort(void *base,int nelem,int width,int (*fcmp)(const void *,const void *));
```

所有关于具体元素类型的地方全部用 void 代替。

void 还可以用来实现 C 语言中的多态，这是一个挺好玩的东西。

不过也有需要注意的:

- 不能对 void 指针解引用

比如:

```
int num;
void *pv = (void*)&num;
*pv = 4; // 错误
```

为什么?

因为解引用的本质就是编译器根据指针所指的类型，然后从指针所指向的内存连续取 N 个字节，然后将这 N 个字节按照指针的类型去解释。

比如 int *型指针，那么这里 N 就是 4，然后按照 int 的编码方式去解释数字。

但是 void，编译器是不知道它到底指向的是 int、double、或者是一个结构体，所以编译器没法对 void 型指针解引用。

七、花式秀技

很多同学认为 C 就只能面向过程编程，实际上利用指针，我们一样可以在 C 中模拟出对象、继承、多态等东西。

也可以利用 void 指针实现泛型编程，也就是 Java、C++ 中的模板。

大家如果对 C 实现面向对象、模板、继承这些感兴趣的话，可以积极一点，点赞，留言~ 呼声高的话，我就再写一篇。

实际上也是很有趣的东西，当你知道了如何用 C 去实现这些东西，那你对 C++ 中的对象、Java 中的对象也会理解得更加透彻。

比如为啥有 this 指针，或者 Python 中的 self 究竟是个啥?

关于指针想写的内容还有很多，这其实也只算是开了个头，限于篇幅，以后有机会补齐以下内容:

- 二维数组和二维指针
- 数组指针和指针数组
- 指针运算
- 函数指针
- 动态内存分配: malloc 和 free
- 堆、栈
- 函数参数传递方式
- 内存泄露
- 数组退化成指针
- const 修饰指针
- ...

C++必读书籍推荐| 附下载方式