

第11讲：深入理解指针(1)

目录：

1. 内存和地址
2. 指针变量和地址
3. 指针变量类型的意义
4. const修饰指针
5. 指针运算
6. 野指针
7. assert断言
8. 指针的使用和传址调用

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：

- ```
1 一楼：101, 102, 103...
2 二楼：201, 202, 203...
3 ...
```

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。



# 201

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

- 1 bit - 比特位
- 2 Byte - 字节
- 3 KB
- 4 MB
- 5 GB
- 6 TB
- 7 PB

- 1 1Byte = 8bit
- 2 1KB = 1024Byte
- 3 1MB = 1024KB
- 4 1GB = 1024MB
- 5 1TB = 1024GB
- 6 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

所以我们可以理解为：

钢琴、吉他上面没有写上“刹、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！

The diagram illustrates a computer system with a CPU (green bar on the left) and Memory (yellow bar on the right). Between them are three horizontal buses: the Address Bus (地址总线), Data Bus (数据总线), and Control Bus (控制总线). The Address Bus is shown with 15 lines, each labeled with a binary digit (1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0) from top to bottom. The Data Bus and Control Bus are shown as single lines.

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“刹、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。

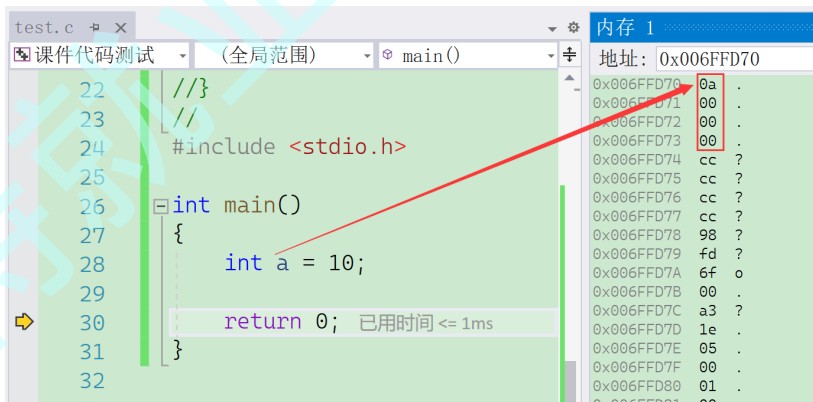
地址信息被下达给内存，在内存上，就可以找到该地址对应的数据，将数据在通过数据总线传入CPU内寄存器。

## 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>
2 int main()
3 {
4 int a = 10;
5 return 0;
6 }
```



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：

```
1 0x006FFD70
2 0x006FFD71
3 0x006FFD72
4 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

```
1 #include <stdio.h>
2 int main()
3 {
4 int a = 10;
5 &a; //取出a的地址
6 printf("%p\n", &a);
7 return 0;
8 }
```



## 2.2 指针变量和解引用操作符 (\*)

比特就业课主页：<https://m.cctalk.com/inst/s9yewhfr>

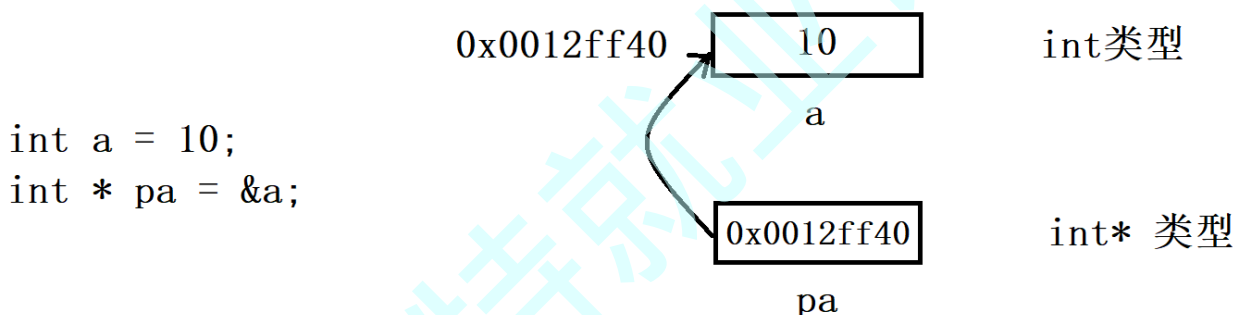
指针变量也是一种变量，这种变量就是用来存放地址的，存放在指针变量中的值都会理解为地址。

### 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？

```
1 int a = 10;
2 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？

```
1 char ch = 'w';
2 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(`*`)。

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

```
2
3 int main()
4 {
5 int a = 100;
6 int* pa = &a;
7 *pa = 0;
8 return 0;
9 }
```

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。

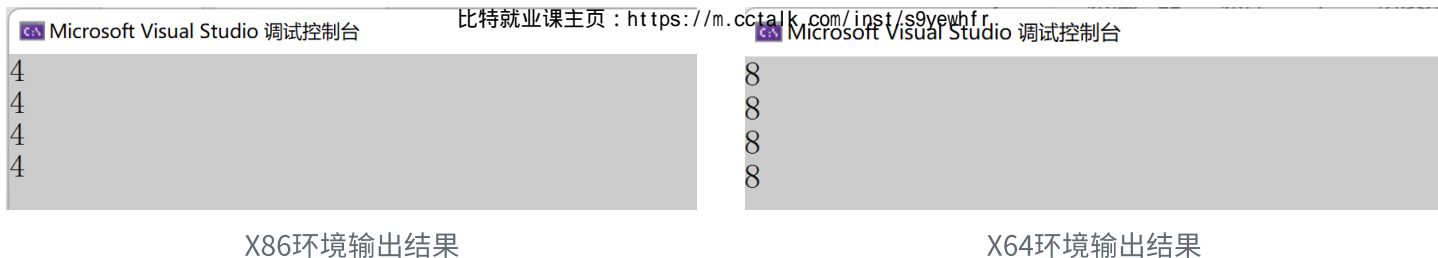
## 2.3 指针变量的大小

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。

```
1 #include <stdio.h>
2 //指针变量的大小取决于地址的大小
3 //32位平台下地址是32个bit位（即4个字节）
4 //64位平台下地址是64个bit位（即8个字节）
5
6 int main()
7 {
8 printf("%zd\n", sizeof(char *));
9 printf("%zd\n", sizeof(short *));
10 printf("%zd\n", sizeof(int *));
11 printf("%zd\n", sizeof(double *));
12 return 0;
13 }
```



- 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。

```
1 //代码1
2 #include <stdio.h>
3
4 int main()
5 {
6 int n = 0x11223344;
7 int *pi = &n;
8 *pi = 0;
9 return 0;
10 }
```

```
1 //代码2
2 #include <stdio.h>
3
4 int main()
5 {
6 int n = 0x11223344;
7 char *pc = (char *)&n;
8 *pc = 0;
9 return 0;
10 }
```

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）。

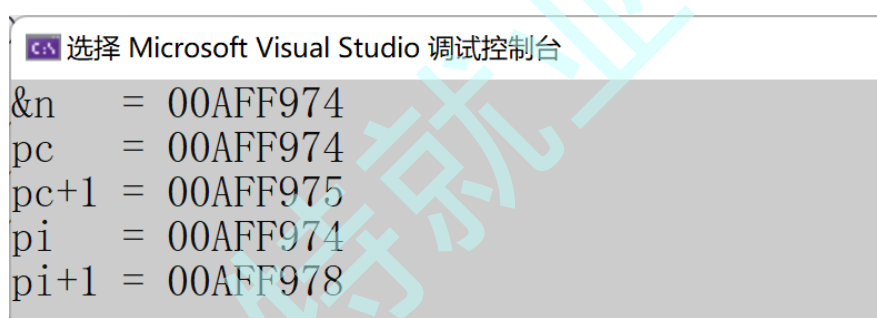
比如：`char*` 的指针解引用就只能访问一个字节，而 `int*` 的指针的解引用就能访问四个字节。

#### 3.2 指针+-整数



```
1 #include <stdio.h>
2 int main()
3 {
4 int n = 10;
5 char *pc = (char*)&n;
6 int *pi = &n;
7
8 printf("%p\n", &n);
9 printf("%p\n", pc);
10 printf("%p\n", pc+1);
11 printf("%p\n", pi);
12 printf("%p\n", pi+1);
13 return 0;
14 }
```

代码运行的结果如下：



选择 Microsoft Visual Studio 调试控制台

```
&n = 00AFF974
pc = 00AFF974
pc+1 = 00AFF975
pi = 00AFF974
pi+1 = 00AFF978
```

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：

```
1 #include <stdio.h>
2
```

```

3 int main()
4 {
5 int a = 10;
6 int* pa = &a;
7 char* pc = &a;
8 return 0;
9 }

```

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題。

输出

显示输出来源(S): 生成

已启动生成...

1>----- 已启动生成: 项目: test, 配置: Debug Win32 -----

1>test.c

1>D:\code\test\test\test.c(11,11): warning C4133: “初始化”: 从“int\*”到“char\*”的类型不兼容

1>已完成生成项目“test.vcxproj”的操作。

===== 生成: 1 成功, 0 失败, 0 最新, 0 已跳过 =====

===== 生成 开始于 10:16 AM, 并花费了 01.795 秒 =====

|

VS2022编译的结果

使用void\*类型的指针接收地址:

```

1 #include <stdio.h>
2
3 int main()
4 {
5 int a = 10;
6 void* pa = &a;
7 void* pc = &a;
8
9 *pa = 10;
10 *pc = 0;
11 return 0;
12 }

```

VS编译代码的结果:

显示输出来源(S): 生成

已启动生成...

1&gt;----- 已启动生成: 项目: test, 配置: Debug Win32 -----

1&gt;test.c

1&gt;D:\code\test\test\test\test.c(14,5): error C2100: 非法的间接寻址 1

1&gt;D:\code\test\test\test\test.c(14,13): warning C4047: "=": "void \*" 与 "int" 的间接级别不同

1&gt;D:\code\test\test\test\test.c(15,5): error C2100: 非法的间接寻址 2

1&gt;已完成生成项目 "test.vcxproj" 的操作 - 失败。

===== 生成: 0 成功, 1 失败, 0 最新, 0 已跳过 =====

===== 生成 开始于 10:28 AM, 并花费了 00.243 秒 =====

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(4)》中我们会讲解。

## 4. const 修饰指针

### 4.1 const修饰变量

变量是可以修改的, 如果把变量的地址交给一个指针变量, 通过指针变量的也可以修改这个变量。

但是如果我们希望一个变量加上一些限制, 不能被修改, 怎么做呢? 这就是const的作用。

```
1 #include <stdio.h>
2 int main()
3 {
4 int m = 0;
5 m = 20; //m是可以修改的
6 const int n = 0;
7 n = 20; //n是不能被修改的
8 return 0;
9 }
```

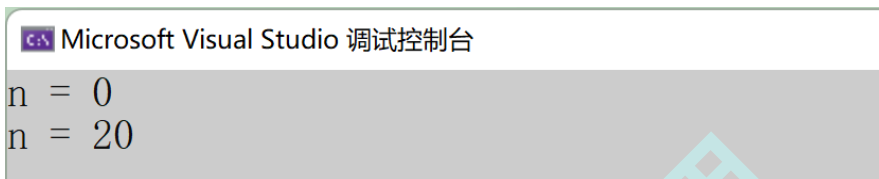
上述代码中n是不能被修改的, 其实n本质是变量, 只不过被const修饰后, 在语法上加了限制, 只要我们在代码中对n就行修改, 就不符合语法规则, 就报错, 致使没法直接修改n。

但是如果我们绕过n, 使用n的地址, 去修改n就能做到了, 虽然这样做是在打破语法规则。

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

```
2 int main()
3 {
4 const int n = 0;
5 printf("n = %d\n", n);
6 int*p = &n;
7 *p = 20;
8 printf("n = %d\n", n);
9 return 0;
10 }
```

输出结果:



```
Microsoft Visual Studio 调试控制台
n = 0
n = 20
```

程序运行结果

我们可以看到这里一个确实修改了,但是我们还是要思考一下,为什么n要被const修饰呢?就是为了不能被修改,如果p拿到n的地址就能修改n,这样就打破了const的限制,这是不合理的,所以应该让p拿到n的地址也不能修改n,那接下来怎么做呢?

## 4.2 const修饰指针变量

一般来讲const修饰指针变量,可以放在\*的左边,也可以放在\*的右边,意义是不一样的。

```
1 int * p; //没有const修饰?
2 int const * p; //const 放在*的左边做修饰
3 int * const p; //const 放在*的右边做修饰
```

我们看下面代码,来分析具体分析一下:

```
1 #include <stdio.h>
2 //代码1 - 测试无const修饰的情况
3 void test1()
4 {
5 int n = 10;
6 int m = 20;
7 int *p = &n;
8 *p = 20; //ok?
9 p = &m; //ok?
10 }
```

```
11
12 //代码2 - 测试const放在*的左边情况
13 void test2()
14 {
15 int n = 10;
16 int m = 20;
17 const int* p = &n;
18 *p = 20; //ok?
19 p = &m; //ok?
20 }
21
22 //代码3 - 测试const放在*的右边情况
23 void test3()
24 {
25 int n = 10;
26 int m = 20;
27 int * const p = &n;
28 *p = 20; //ok?
29 p = &m; //ok?
30 }
31
32 //代码4 - 测试*的左右两边都有const
33 void test4()
34 {
35 int n = 10;
36 int m = 20;
37 int const * const p = &n;
38 *p = 20; //ok?
39 p = &m; //ok?
40 }
41
42 int main()
43 {
44 //测试无const修饰的情况
45 test1();
46 //测试const放在*的左边情况
47 test2();
48 //测试const放在*的右边情况
49 test3();
50 //测试*的左右两边都有const
51 test4();
52 return 0;
53 }
```

结论：const修饰指针变量的时候

- const如果放在\*的左边,修饰的是指针指向的内容,保证指针指向的内容不能通过指针来改变。但是指针变量本身的内容可变。
- const如果放在\*的右边,修饰的是指针变量本身,保证了指针变量的内容不能修改,但是指针指向的内容,可以通过指针改变。

## 5. 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

### 5.1 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 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 下标 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9  |

数组元素和下标

```
1 #include <stdio.h>
2 //指针+- 整数
3 int main()
4 {
5 int arr[10] = {1,2,3,4,5,6,7,8,9,10};
6 int *p = &arr[0];
7 int i = 0;
8 int sz = sizeof(arr)/sizeof(arr[0]);
9 for(i=0; i<sz; i++)
10 {
11 printf("%d ", *(p+i)); //p+i 这里就是指针+整数
12 }
13 return 0;
```

## 5.2 指针-指针

```
1 //指针-指针
2 #include <stdio.h>
3 int my_strlen(char *s)
4 {
5 char *p = s;
6 while(*p != '\0')
7 p++;
8 return p-s;
9 }
10
11 int main()
12 {
13 printf("%d\n", my_strlen("abc"));
14 return 0;
15 }
```

## 5.3 指针的关系运算

```
1 //指针的关系运算
2 #include <stdio.h>
3
4 int main()
5 {
6 int arr[10] = {1,2,3,4,5,6,7,8,9,10};
7 int *p = &arr[0];
8 int sz = sizeof(arr)/sizeof(arr[0]);
9 while(p<arr+sz) //指针的大小比较
10 {
11 printf("%d ", *p);
12 p++;
13 }
14 return 0;
15 }
```

## 6. 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)

## 6.1 野指针成因

### 1. 指针未初始化

```
1 #include <stdio.h>
2 int main()
3 {
4 int *p; //局部变量指针未初始化, 默认为随机值
5 *p = 20;
6 return 0;
7 }
```

### 2. 指针越界访问

```
1 #include <stdio.h>
2 int main()
3 {
4 int arr[10] = {0};
5 int *p = &arr[0];
6 int i = 0;
7 for(i=0; i<=11; i++)
8 {
9 //当指针指向的范围超出数组arr的范围时, p就是野指针
10 *(p++) = i;
11 }
12 return 0;
13 }
```

### 3. 指针指向的空间释放

```
1 #include <stdio.h>
2
3 int* test()
4 {
5 int n = 100;
6 return &n;
7 }
8
9 int main()
10 {
11 int*p = test();
```



```
12 printf("%d\n", *p);
13 return 0;
14 }
```

## 6.2 如何规避野指针

### 6.2.1 指针初始化

如果明确知道指针指向哪里就直接赋值地址, 如果不知道指针应该指向哪里, 可以给指针赋值NULL.

NULL 是C语言中定义的一个标识符常量, 值是0, 0也是地址, 这个地址是无法使用的, 读写该地址会报错。

```
1 #ifdef __cplusplus
2 #define NULL 0
3 #else
4 #define NULL ((void *)0)
5 #endif
```

初始化如下:

```
1 #include <stdio.h>
2
3 int main()
4 {
5 int num = 10;
6 int*p1 = #
7 int*p2 = NULL;
8
9 return 0;
10 }
```

### 6.2.2 小心指针越界

一个程序向内存申请了哪些空间, 通过指针也就只能访问哪些空间, 不能超出范围访问, 超出了就是越界访问。

### 6.2.3 指针变量不再使用时, 及时置NULL, 指针使用之前检查有效性

当指针变量指向一块区域的时候，我们可以通过指针访问该区域，后期不再使用这个指针访问空间的时候，我们可以把该指针置为NULL。因为约定俗成的一个规则就是：只要是NULL指针就不去访问，同时使用指针之前可以判断指针是否为NULL。

我们可以把野指针想象成野狗，野狗放任不管是非常危险的，所以我们可以找一棵树把野狗拴起来，就相对安全了，给指针变量及时赋值为NULL，其实就类似把野狗栓起来，就是把野指针暂时管理起来。

不过野狗即使拴起来我们也要绕着走，不能去挑逗野狗，有点危险；对于指针也是，在使用之前，我们也要判断是否为NULL，看看是不是被拴起来起来的野狗，如果是不能直接使用，如果不是我们再去使用。

```
1 int main()
2 {
3 int arr[10] = {1,2,3,4,5,6,7,8,9,10};
4 int *p = &arr[0];
5 int i = 0;
6 for(i=0; i<10; i++)
7 {
8 *(p++) = i;
9 }
10 //此时p已经越界了，可以把p置为NULL
11 p = NULL;
12 //下次使用的时候，判断p不为NULL的时候再使用
13 //...
14 p = &arr[0]; //重新让p获得地址
15 if(p != NULL) //判断
16 {
17 //...
18 }
19 return 0;
20 }
```

#### 6.2.4 避免返回局部变量的地址

如造成野指针的第3个例子，不要返回局部变量的地址。

## 7. assert 断言

`assert.h` 头文件定义了宏 `assert()`，用于在运行时确保程序符合指定条件，如果不符合，就报错终止运行。这个宏常常被称为“断言”。

```
1 assert(p != NULL);
```

上面代码在程序运行到这一行语句时，验证变量 `p` 是否等于 `NULL`。如果确实不等于 `NULL`，程序继续运行，否则就会终止运行，并且给出报错信息提示。

`assert()` 宏接受一个表达式作为参数。如果该表达式为真（返回值非零），`assert()` 不会产生任何作用，程序继续运行。如果该表达式为假（返回值为零），`assert()` 就会报错，在标准错误流 `stderr` 中写入一条错误信息，显示没有通过的表达式，以及包含这个表达式的文件名和行号。

`assert()` 的使用对程序员是非常友好的，使用 `assert()` 有几个好处：它不仅能自动标识文件和出问题的行号，还有一种无需更改代码就能开启或关闭 `assert()` 的机制。如果已经确认程序没有问题，不需要再做断言，就在 `#include <assert.h>` 语句的前面，定义一个宏 `NDEBUG`。

```
1 #define NDEBUG
2 #include <assert.h>
```

然后，重新编译程序，编译器就会禁用文件中所有的 `assert()` 语句。如果程序又出现问题，可以移除这条 `#define NDEBUG` 指令（或者把它注释掉），再次编译，这样就重新启用了 `assert()` 语句。

`assert()` 的缺点是，因为引入了额外的检查，增加了程序的运行时间。

一般我们可以在 `Debug` 中使用，在 `Release` 版本中选择禁用 `assert` 就行，在 `VS` 这样的集成开发环境中，在 `Release` 版本中，直接就是优化掉了。这样在 `debug` 版本写有利于程序员排查问题，在 `Release` 版本不影响用户使用程序效率。

## 8. 指针的使用和传址调用

### 8.1 strlen的模拟实现

库函数 `strlen` 的功能是求字符串长度，统计的是字符串中 `\0` 之前的字符的个数。

函数原型如下：

```
1 size_t strlen (const char * str);
```

参数 `str` 接收一个字符串的起始地址，然后开始统计字符串中 `\0` 之前的字符个数，最终返回长度。

如果要模拟实现只要从起始地址开始向后逐个字符的遍历，只要不是 `\0` 字符，计数器就+1，这样直到 `\0` 就停止。

参考代码如下：

```
1 int my_strlen(const char * str)
2 {
3 int count = 0;
4 assert(str);
5 while(*str)
6 {
7 count++;
8 str++;
9 }
10 return count;
11 }
12
13 int main()
14 {
15 int len = my_strlen("abcdef");
16 printf("%d\n", len);
17 return 0;
18 }
```

## 8.2 传值调用和传址调用

学习指针的目的是使用指针解决问题，那什么问题，非指针不可呢？

**例如：**写一个函数，交换两个整型变量的值

一番思考后，我们可能写出这样的代码：

```
1 #include <stdio.h>
2
3 void Swap1(int x, int y)
4 {
5 int tmp = x;
6 x = y;
7 y = tmp;
8 }
9
10 int main()
11 {
12 int a = 0;
13 int b = 0;
14 scanf("%d %d", &a, &b);
15 printf("交换前: a=%d b=%d\n", a, b);
```

比特就业课主页: <https://m.cctalk.com/inst/s9yewhfr>

```
16 Swap1(a, b);
17 printf("交换后: a=%d b=%d\n", a, b);
18
19 return 0;
20 }
```

当我们运行代码，结果如下：

选择 Microsoft Visual Studio 调试控制台

```
10 20
交换前: a=10 b=20
交换后: a=10 b=20
```

我们发现其实没产生交换的效果，这是为什么呢？

调试一下，试试呢？

The screenshot shows the Visual Studio IDE with a C program named `test.c`. The code defines a `Swap1` function and calls it in the `main` function. The debug console on the right shows the state of variables `a`, `b`, `x`, and `y`.

| 名称                  | 值               |
|---------------------|-----------------|
| <code>a</code>      | 10              |
| <code>b</code>      | 20              |
| <code>&amp;a</code> | 0x00cffd0 {10}  |
| <code>&amp;b</code> | 0x00cfdc4 {20}  |
| <code>x</code>      | 10              |
| <code>y</code>      | 20              |
| <code>&amp;x</code> | 0x00cffeec {10} |
| <code>&amp;y</code> | 0x00cfff0 {20}  |

我们发现在main函数内部，创建了a和b，a的地址是0x00cffd0，b的地址是0x00cfdc4，在调用Swap1函数时，将a和b传递给了Swap1函数，在Swap1函数内部创建了形参x和y接收a和b的值，但是x的地址是0x00cffeec，y的地址是0x00cfff0，x和y确实接收到了a和b的值，不过x的地址和a的地址不一样，y的地址和b的地址不一样，相当于x和y是独立的空间，那么在Swap1函数内部交换x和y的值，自然不会影响a和b，当Swap1函数调用结束后回到main函数，a和b的没法交换。Swap1函数在使用

的时候，是把变量本身直接传递给了函数，这种调用函数的方式我们之前在函数的时候就知道了，这种叫**传值调用**。

**结论：实参传递给形参的时候，形参会单独创建一份临时空间来接收实参，对形参的修改不影响实参。**

所以Swap1是失败的了。

那怎么办呢？

我们现在要解决的就是当调用Swap函数的时候，Swap函数内部操作的就是main函数中的a和b，直接将a和b的值交换了。那么就可以使用指针了，在main函数中将a和b的地址传递给Swap函数，Swap函数里边通过地址间接的操作main函数中的a和b，并达到交换的效果就好了。

```
1 #include <stdio.h>
2
3 void Swap2(int*px, int*py)
4 {
5 int tmp = 0;
6 tmp = *px;
7 *px = *py;
8 *py = tmp;
9 }
10
11 int main()
12 {
13 int a = 0;
14 int b = 0;
15 scanf("%d %d", &a, &b);
16 printf("交换前: a=%d b=%d\n", a, b);
17 Swap2(&a, &b);
18 printf("交换后: a=%d b=%d\n", a, b);
19
20 return 0;
21 }
```

首先看输出结果：

```
10 20
交换前： a=10 b=20
交换后： a=20 b=10
```

我们可以看到实现成Swap2的方式，顺利完成了任务，这里调用Swap2函数的时候是将变量的地址传递给了函数，这种函数调用方式叫：**传址调用**。

传址调用，可以让函数和主调函数之间建立真正的联系，在函数内部可以修改主调函数中的变量；所以未来函数中只是需要主调函数中的变量值来实现计算，就可以采用传值调用。如果函数内部要修改主调函数中的变量的值，就需要传址调用。

完