

# 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。



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中，又是怎么样呢？

我们知道计算上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**，一个人字节空间里面能放**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。

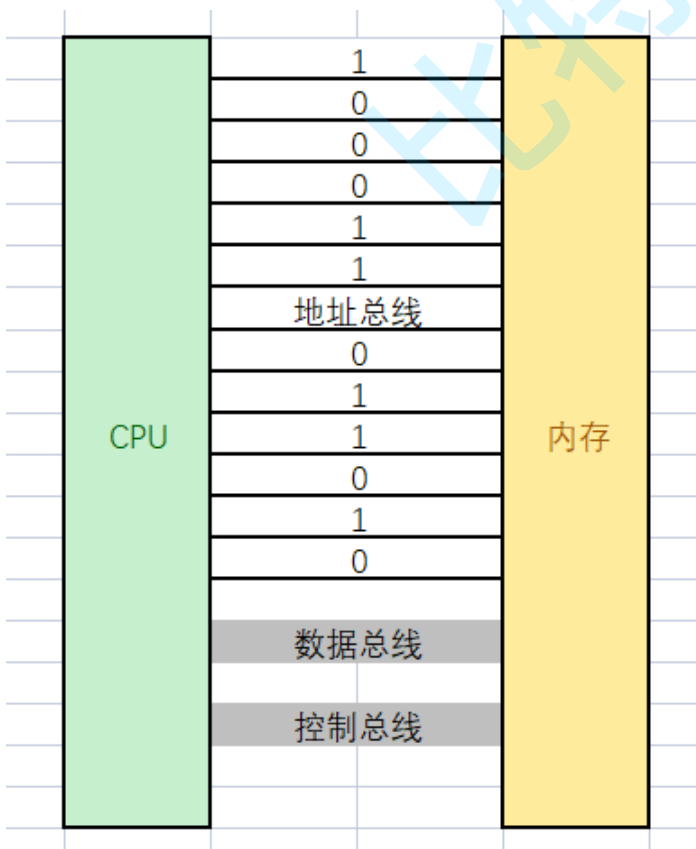
[illegible]

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：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“都瑞咪发嗦啦”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！

硬件编址也是如此

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**。

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。

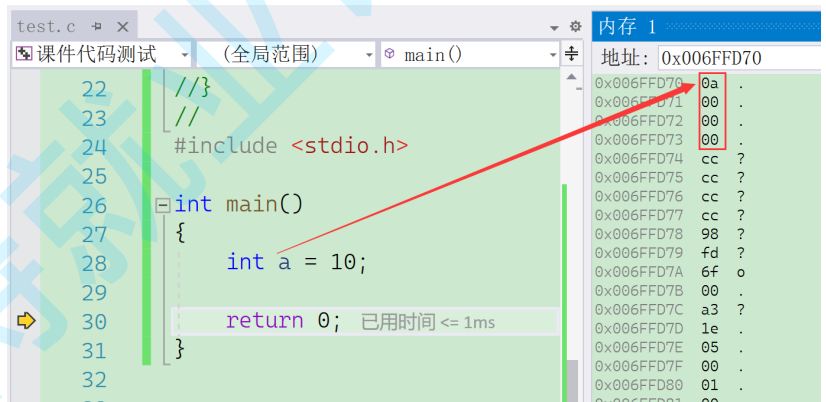
地址信息被下达给内存，在内存上，就可以找到该地址对应的数据，将数据在通过数据总线传入CPU内寄存器。

## 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()
```

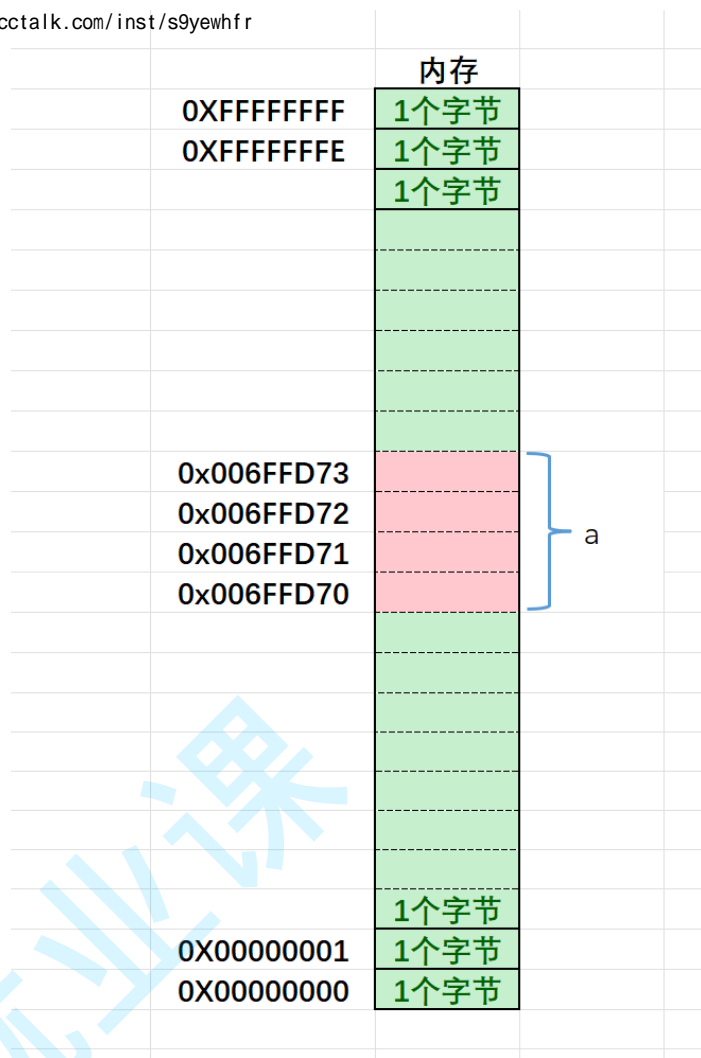
```

3 {
4     int a = 10;
5     &a; //取出a的地址
6     printf("%p\n", &a);
7     return 0;
8 }

```

按照我画图的样子，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。



变量在内存中的存储

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：

```

1 #include <stdio.h>
2 int main()
3 {
4     int a = 10;
5     int* pa = &a; //取出a的地址并存储到指针变量pa中
6
7     return 0;

```

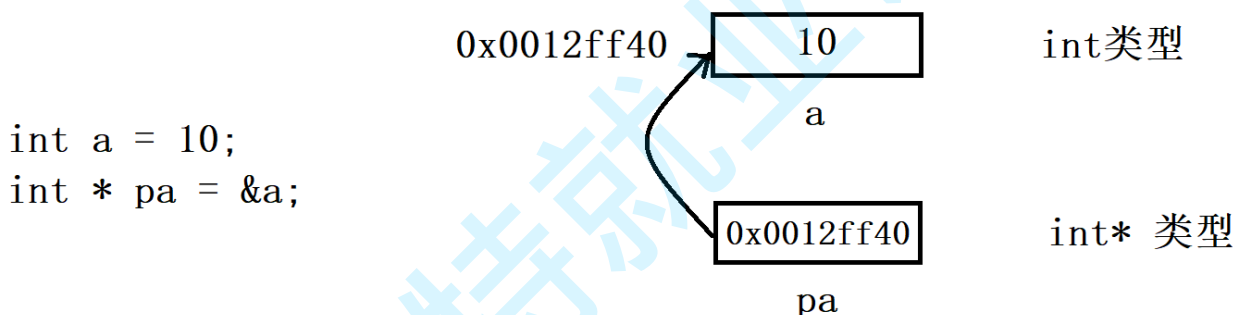
指针变量也是一种变量，这种变量就是用来存放地址的，存放在指针变量中的值都会理解为地址。

### 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？

```
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。

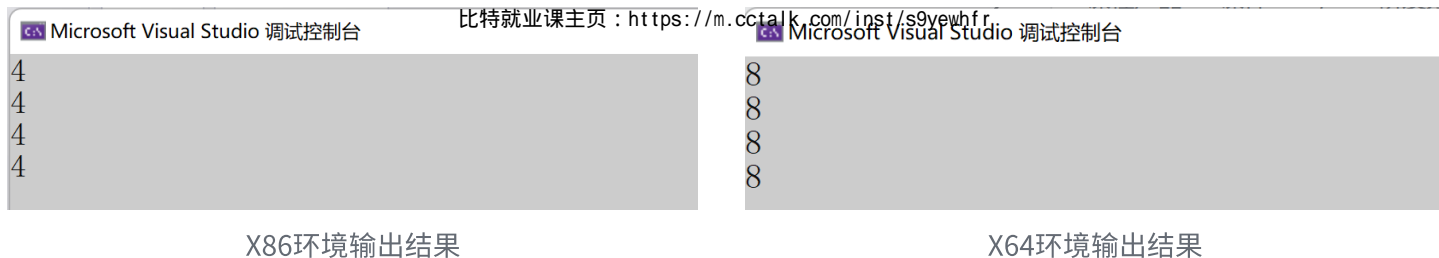
## 2.3 指针变量的大小

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

先看一段代码，调试观察地址的变化。



```
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。

**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)。

## 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;
```

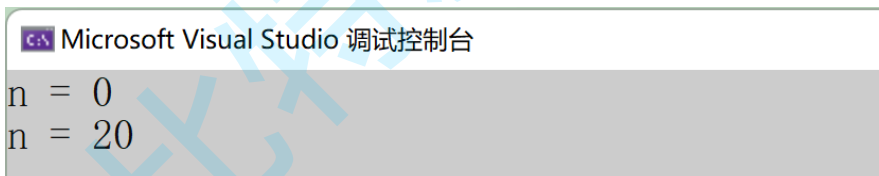
```
7     n = 20; //n是不能被修改的
8     return 0;
9 }
```

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

我们看下面代码，来分析

```
1 #include <stdio.h>
2 //代码1
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 void test2()
12 {
13     //代码2
14     int n = 10;
15     int m = 20;
16     const int* p = &n;
17     *p = 20; //ok?
18     p = &m; //ok?
19 }
20 void test3()
21 {
22     int n = 10;
23     int m = 20;
24     int *const p = &n;
25     *p = 20; //ok?
26     p = &m; //ok?
27 }
28
29 void test4()
30 {
31     int n = 10;
32     int m = 20;
33     int const * const p = &n;
34     *p = 20; //ok?
35     p = &m; //ok?
36 }
37
38 int main()
39 {
40     //测试无const修饰的情况
41     test1();
42     //测试const放在*的左边情况
43     test2();
44     //测试const放在*的右边情况
45     test3();
46     //测试*的左右两边都有const
47     test4();
48     return 0;
49 }
```

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;
14 }
```

## 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 i = 0;
9     int sz = sizeof(arr)/sizeof(arr[0]);
10    while(p<arr+sz) //指针的大小比较
11    {
12        printf("%d ", *p);
13        p++;
14    }
15    return 0;
16 }
```

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

#### 6.2.4 避免返回局部变量的地址

如果造成野指针的第3个例子。

## 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时程序的效率。

## 8. 指针的使用和传址调用

### 8.1 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);
16     Swap1(a, b);
17     printf("交换后: a=%d b=%d\n", a, b);
18
19     return 0;
20 }
```

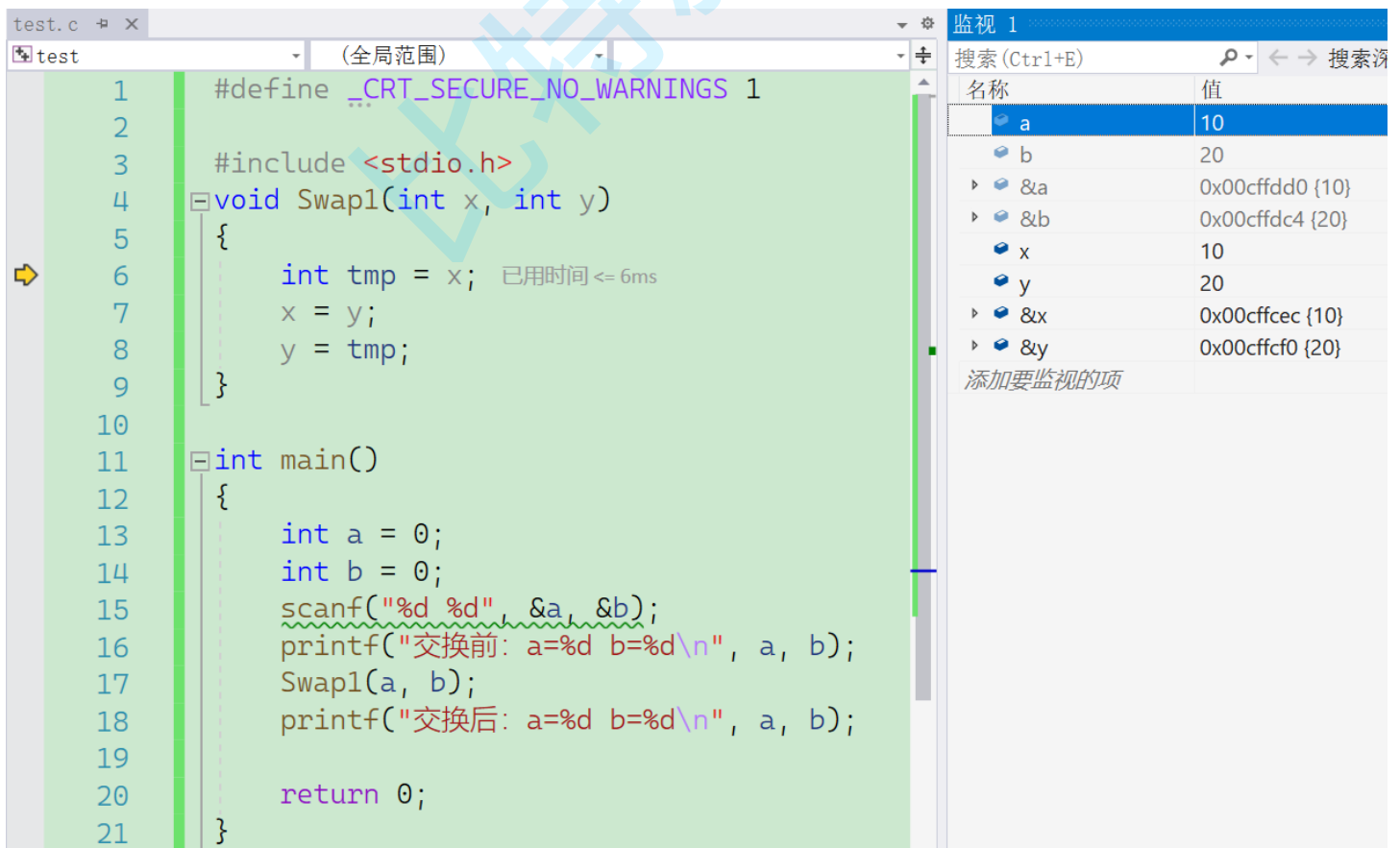
当我们运行代码，结果如下：

 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？



test.c 1 X

test (全局范围)

```
1 #define _CRT_SECURE_NO_WARNINGS 1
2
3 #include <stdio.h>
4 void Swap1(int x, int y)
5 {
6     int tmp = x; 已用时间 <= 6ms
7     x = y;
8     y = 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     Swap1(a, b);
18     printf("交换后: a=%d b=%d\n", a, b);
19
20     return 0;
21 }
```

监视 1

搜索 (Ctrl+E) 搜索

| 名称 | 值                |
|----|------------------|
| a  | 10               |
| b  | 20               |
| &a | 0x00cfffdd0 {10} |
| &b | 0x00cfffdc4 {20} |
| x  | 10               |
| y  | 20               |
| &x | 0x00cfffcec {10} |
| &y | 0x00cfffcf0 {20} |

添加要监视的项

我们发现在main函数内部，创建了a和b，a的地址是0x00cfffdd0，b的地址是0x00cfffdc4，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0x00cfffcec，y的地址是0x00cfffcf0，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Swap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     Swap1(&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**。

## 8.2 strlen的模拟实现

```
1 //计数器方式
2 int my_strlen(const char * str)
3 {
4     int count = 0;
5     assert(str);
6     while(*str)
7     {
8         count++;
9         str++;
10    }
11    return count;
12 }
13
14 int main()
15 {
16     int len = my_strlen("abcdef");
17     printf("%d\n", len);
18     return 0;
19 }
```

完