第12讲：深入理解指针(2)

1. const修饰指针
2. 野指针
3. assert断言
4. 指针的使用和传址调用

正文开始

1. const 修饰指针

1.1 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。

```c
#include <stdio.h>
int main()
{
    int m = 0;
    m = 20;//m是可以修改的
    const int n = 0;
    n = 20;//n是不能被修改的
    return 0;
}
```

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。

```c
#include <stdio.h>
int main()
{
    const int n = 0;
```

```
    printf("n = %d\n", n);
    int* p = &n;
    *p = 20;
    printf("n = %d\n", n);
    return 0;
}
```

输出结果：

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？

1.2 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。

```
int * p;//没有const修饰?
int const * p;//const 放在*的左边做修饰
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：

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

//代码2 - 测试const放在*的左边情况
void test2()
```

```
{
    int n = 10;
    int m = 20;
    const int* p = &n;
    *p = 20;//ok?
    p = &m; //ok?
}

//代码3 - 测试const放在*的右边情况
void test3()
{
    int n = 10;
    int m = 20;
    int * const p = &n;
    *p = 20; //ok?
    p = &m;  //ok?
}

//代码4 - 测试*的左右两边都有const
void test4()
{
    int n = 10;
    int m = 20;
    int const * const p = &n;
    *p = 20; //ok?
    p = &m;  //ok?
}

int main()
{
    //测试无const修饰的情况
    test1();
    //测试const放在*的左边情况
    test2();
    //测试const放在*的右边情况
    test3();
    //测试*的左右两边都有const
    test4();
    return 0;
}
```

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。

2. 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）

2.1 野指针成因

1. 指针未初始化

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

2. 指针越界访问

```c
#include <stdio.h>
int main()
{
    int arr[10] = {0};
    int *p = &arr[0];
    int i = 0;
    for(i = 0; i <= 11; i++)
    {
        //当指针指向的范围超出数组arr的范围时，p就是野指针
        *(p++) = i;
    }
    return 0;
}
```

3. 指针指向的空间释放

```c
#include <stdio.h>

int* test()
{
```

```
    int n = 100;
    return &n;
}

int main()
{
    int*p = test();
    printf("%d\n", *p);
    return 0;
}
```

2.2 如何规避野指针

2.2.1 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。

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

初始化如下：

```
#include <stdio.h>

int main()
{
    int num = 10;
    int*p1 = &num;
    int*p2 = NULL;

    return 0;
}
```

2.2.2 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。

2.2.3 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。

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

2.2.4 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。

3. 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。

4. 指针的使用和传址调用

4.1 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：

```
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：

```
int my_strlen(const char * str)
{
    int count = 0;
    assert(str);
    while(*str)
    {
        count++;
        str++;
    }
    return count;
}

int main()
{
    int len = my_strlen("abcdef");
    printf("%d\n", len);
    return 0;
}
```

4.2 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：

```
#include <stdio.h>

void Swap1(int x, int y)
{
    int tmp = x;
    x = y;
    y = tmp;
```

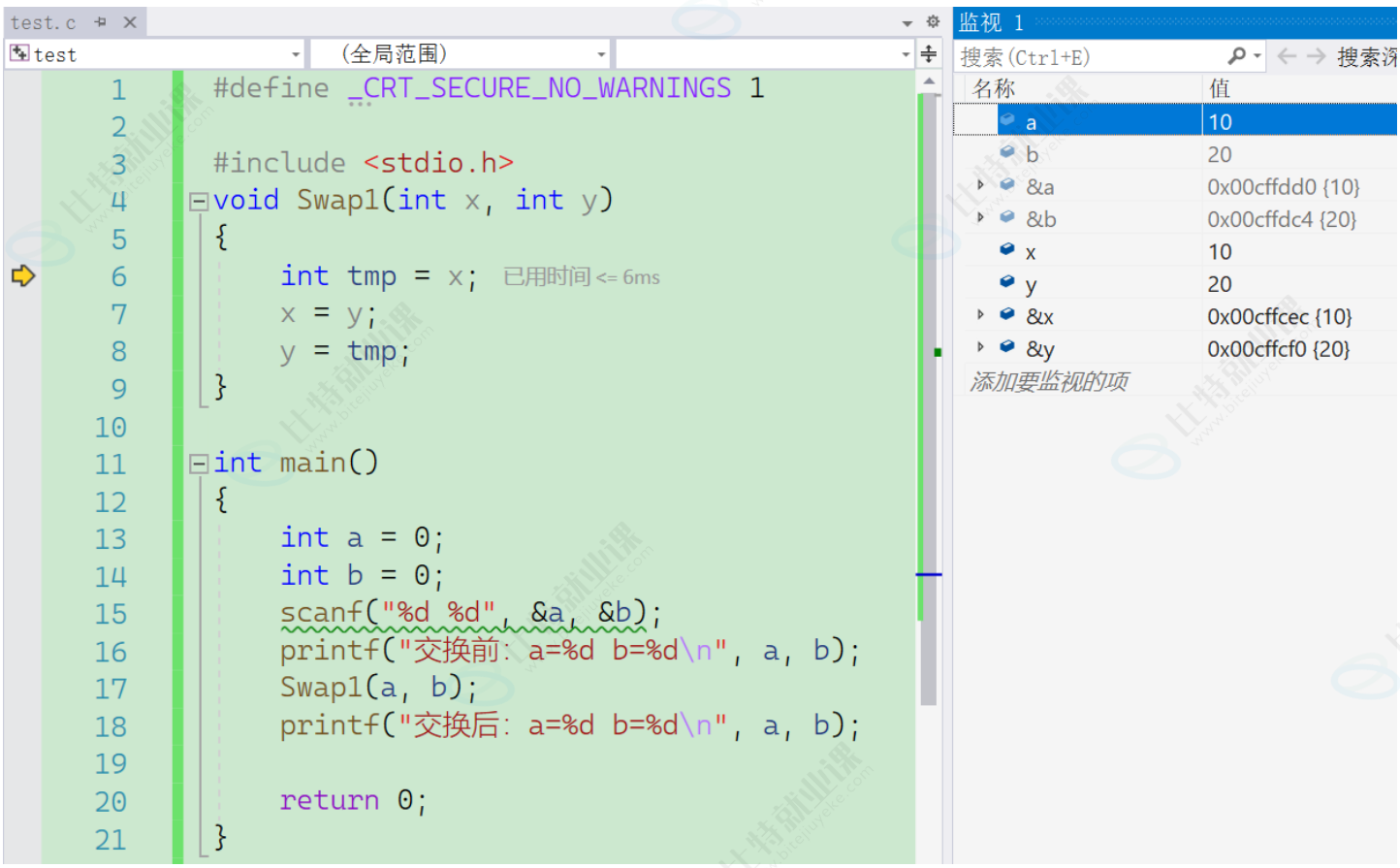
```c
}

int main()
{
    int a = 0;
    int b = 0;
    scanf("%d %d", &a, &b);
    printf("交换前: a=%d b=%d\n", a, b);
    Swap1(a, b);
    printf("交换后: a=%d b=%d\n", a, b);

    return 0;
}
```

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？



我们发现在main函数内部，创建了a和b，a的地址是0x00cffdd0，b的地址是0x00cffdc4，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0x00cffcec，y的地址是0x00cffcf0，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。

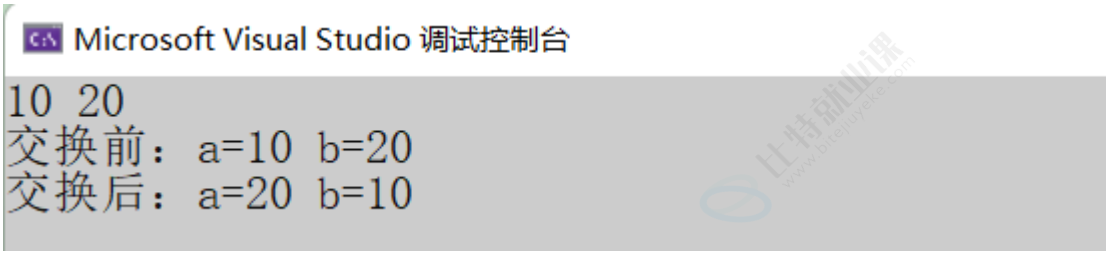
所以Swap1是失败的了。

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。

```c
#include <stdio.h>

void Swap2(int*px, int*py)
{
    int tmp = 0;
    tmp = *px;
    *px = *py;
    *py = tmp;
}

int main()
{
    int a = 0;
    int b = 0;
    scanf("%d %d", &a, &b);
    printf("交换前: a=%d b=%d\n", a, b);
    Swap2(&a, &b);
    printf("交换后: a=%d b=%d\n", a, b);

    return 0;
}
```

首先看输出结果：



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