第 5 章 指针

（视频讲解：5 小时）

前面我们掌握了整型，浮点型，字符型数据，看了这些数据的地址，这时候如果我们想把对应某个变量的地址存下来，这时 C 语言为我们提供了指针。掌握指针，对于以后学习操作系统，理解操作系统原理有非常大的帮助。通过本章，可以掌握：

- **指针的本质**
- **指针的使用场景---传递与偏移**
- **数组指针与二维数组**
- **二级指针的传递与偏移**
- **函数指针**

5.1 指针的本质

5.1.1 指针的定义

内存区的每一个字节有一个编号，这就是“地址” 。如果在程序中定义了一个变量，在对程序进行编译时，系统就会给这个变量分配内存单元。按变量地址存取变量值的方式称为“**直接访问**”方式，例如 printf(“%d”,i);scanf(“%d”,&i)；另一种存取变量值的方式称为“**间接访问**”的方式。即，将变量 i 的地址存放在另一个变量中。在 C 语言中，指针变量是一种特殊的变量，它是存放地址的。

定义一个指针变量

基类型 *指针变量名；例如 int *i_pointer;

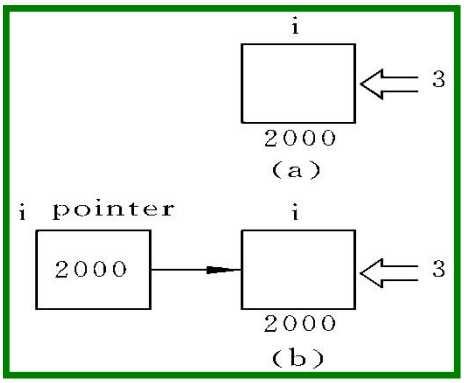

图 5.1.1-1

指针与指针变量是两个概念，一个变量的地址称为该变量的“指针”。例如，地址 2000 是变量 i 的指针。如果有一个变量专门用来存放另一变量的地址（即指针），则它称为“指针变量”。图 5.1.1-1 的 i_pointer 就是一个指针变量。

那么 i_pointer 本身占据多大的内存空间呢，因为我们后面写的都是 win32 控制台应用程序，寻址范围为 32 位，即 4 个字节，如果编写的程序是 64 位，那么就是 8 个字节。所以对于我们来说 sizeof(i_pointer)等于 4。

5.1.2 取地址与取值操作符

取地址操作符是&，也叫引用，通过该操作符我们可以获取一个变量的地址值；取值操作符为*，也叫解引用，通过该操作符我们可以拿到一个地址对应位置的数据。如图 5.1.2-1 所示，我们通过&i 获取整型变量 i 的地址值，然后对整型指针变量 p 进行初始化，p 中存储着整型变量 i 的地址值，所以通过第 12 行的*p，我们就可以拿到整型变量 i 的值，p 里边存储的是一个绝对地址值，为什么取值时，其会获取四个字节大小的空间呢，是因为 p 为整型变量指针，int 占用 4 个字节大小的空间，所以 p 在解引用时会访问四个字节大小的空间，同时以整型值对内存进行解析。

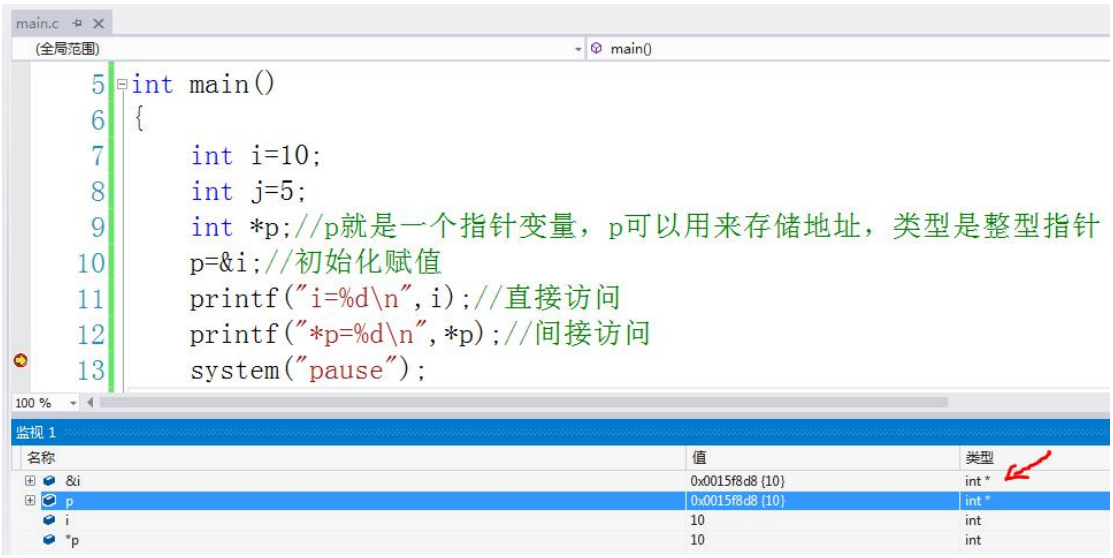

图 5.1.2-1

注意以下 4 点

1、指针变量前面的“*”，表示该变量的类型为指针型变量。

例: float *pointer_1；

指针变量名是 pointer_1 ，而不是* pointer_1 。

2、在定义指针变量时必须指定基类型。

需要特别注意的是，只有整型变量的地址才能放到指向整型变量的指针变量中。下面的赋值是错误的：

```
float  a;
int  * pointer_1;
pointer_1=&a;  //毫无意义，而且会造成出错，有兴趣的小伙伴可以自行尝试
```

3、如果已执行了语句 pointer_ 1 =& a ；

&* pointer_ 1 的含义是什么？

“&”和“*”两个运算符的优先级别相同，但按自右而左方向结合。因此，&* pointer_ 1 与& a 相同，即变量 a 的地址，也就是 pointer_1。

*& a 的含义是什么？

先进行& a 运算，得 a 的地址，再进行*运算。*& a 和*pointer_ 1 的作用是一样的，它们都等价于变量 a 。即*& a 与 a 等价。

4、C 在本质上是一种自由形式的语言，这很容易诱使你把星号写在靠近类型的一侧，如下所示： int *a 这个声明与前面一个声明具有相同的意思，而且看上去更为清晰， a 被

声明为类型为 int*的指针．但是，这并不是一个好支巧，原因如下：int *a,b,c,人们很自然地以为这条语句把所有三个变量声明为指向整型的指针，但事实上并非如此。我们 被它的形式愚弄了，星号实际上是表达式*a 的一部分，只对这个标识符有用，b 是一个指针，但其余两个变量只是普通的整型，要声明三个指针，正确的语句如下：

int *a,*b,*c;

5.2 指针的使用场景

很多同学很害怕指针，觉的很容易用错，其实是因为没有掌握指针的使用场景，指针的使用场景通过总结，只有两个，**传递与偏移**，在这两个场景下采用指针，这样就可以准确使用指针，你就会发觉指针其实很简单。

5.2.1 指针的传递

首先我们来看【例 5.2.1-1】，主函数中我们定义了整型变量 i，初始化值为 10，我们想通过子函数修改整型变量 i 的值，但是你会发现 after change 后，打印的值仍为 10，子函数 change 并没有改变变量 i 的值，为什么呢，我们通过执行来查看。

【例 5.2.1-1】指针的传递使用场景

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

void change(int j)
{
    j=5;
}

int main()
{
    int i=10;
    printf("before change i=%d\n",i);
    change(i);
    printf("after change i=%d\n",i);
    system("pause");
    return 0;
}
```

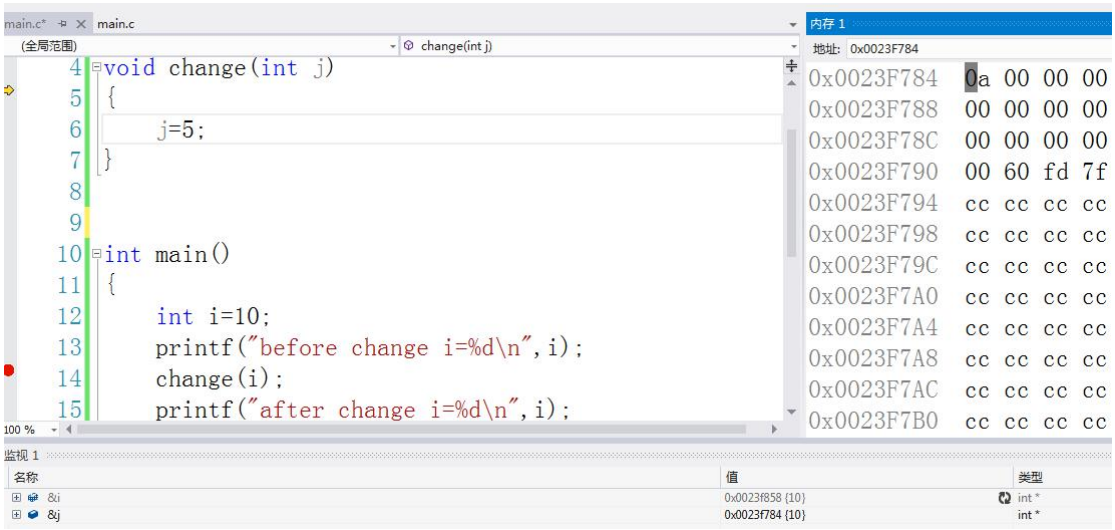

图 5.2.1-1

如图 5.2.1-1 所示，我们在第 14 行打断点，然后运行程序，在监视中输入&i，可以看到变量 i 的地址是 0x0023F858，然后我们按 F11 进入 change 函数，这时候变量 j 的值的确为 10，但是&j 的值为 0x0023F784，也就是 j 和 i 的地址并不相同，那么我们运行 j=5 后，实际 change 函数修改的是地址 0x0023F784 上的值，从 10 变为了 5，change 函数执行结束，变量 i 的值肯定不会发生改变，因为变量 i 的地址是 0x0023F858。

原理图如图 5.2.1-2，程序的执行过程其实就是内存的变化过程，我们需要关注栈空间的变化，当 main 函数开始执行时，系统会为 main 函数开辟函数栈空间，当程序走到 int i 时，main 函数的栈空间内就会为变量 i 分配 4 个字节大小的空间。当我们调用 change 函数时，系统会为 change 函数重新分配新的函数栈空间，并为形参变量 j 分配 4 个字节大小的空间，我们调用 change(i)时，实际是将 i 的值赋值给 j，我们把这种效果成为**值传递，C 语言的函数调用均为值传递**，因此当我们在 change 函数的函数栈空间内修改变量 j 的值后，change 函数执行结束，change 函数的栈空间就会释放，j 就灰飞烟灭了，i 的值不会改变。

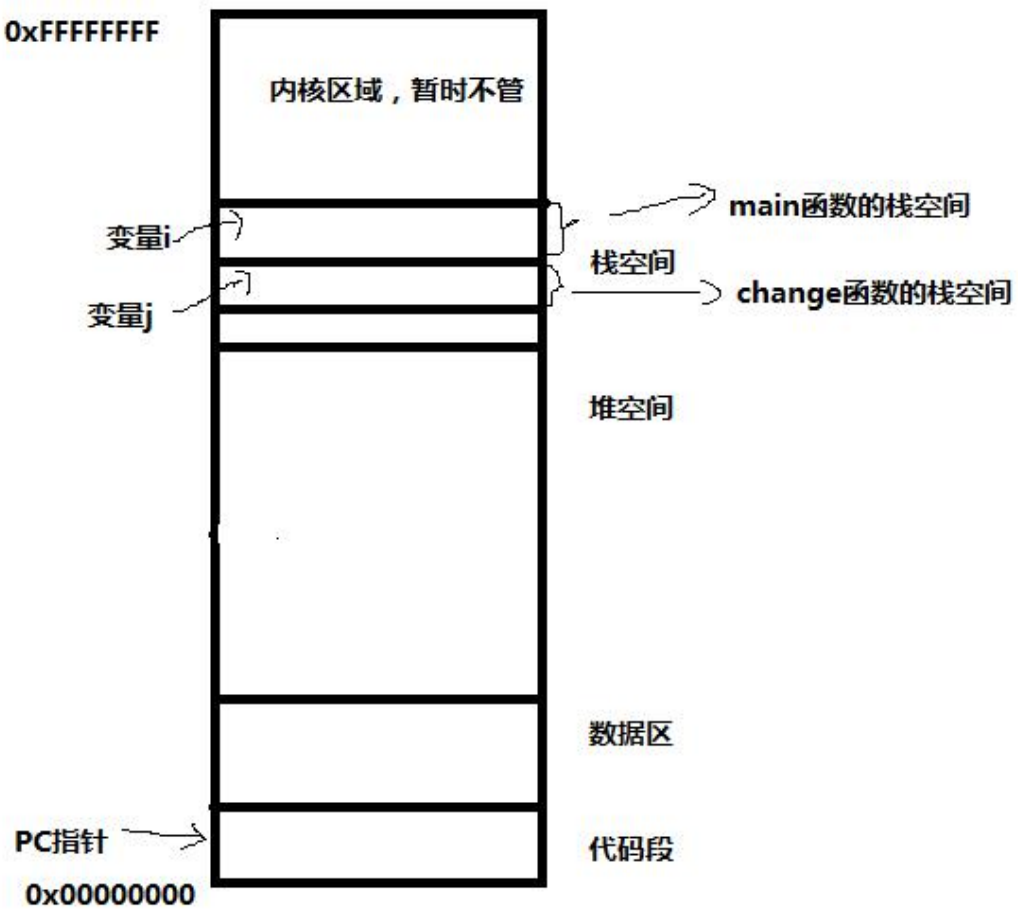

图 5.2.1-2

有些小伙伴已经着急了，难道我们就不能在子函数中修改 main 函数中某个变量的值了么，答案是可以的，我们将程序进行了简单的修改。

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

void change(int* j)
{
    *j=5;//间接访问拿到变量 i
}

//指针的传递
int main()
{
    int i=10;
    printf("before change i=%d\n",i);
    change(&i);//传递变量 i 的地址
    printf("after change i=%d\n",i);
    system("pause");
    return 0;
}
```

你会发现程序执行后，after change 打印的 i 的值为 5，难道 C 语言函数调用值传递的原理变了么？并没有！我们将变量 i 的地址传递给 change 函数，实际效果是 **j=&i**，依然是值传递，只是这时我们的 j 是一个指针变量，内部存储的是变量 i 的地址 0x0015ff0c，所以我们通过*j 就间接访问到变量 i 相同的区域，通过*j=5,就实现了对变量 i 的值的改变。在图 5.2.1-3 我们依然可以看到变量 j 自身的地址是 0x0015fe38，与变量 i 的地址依然是不相等的。

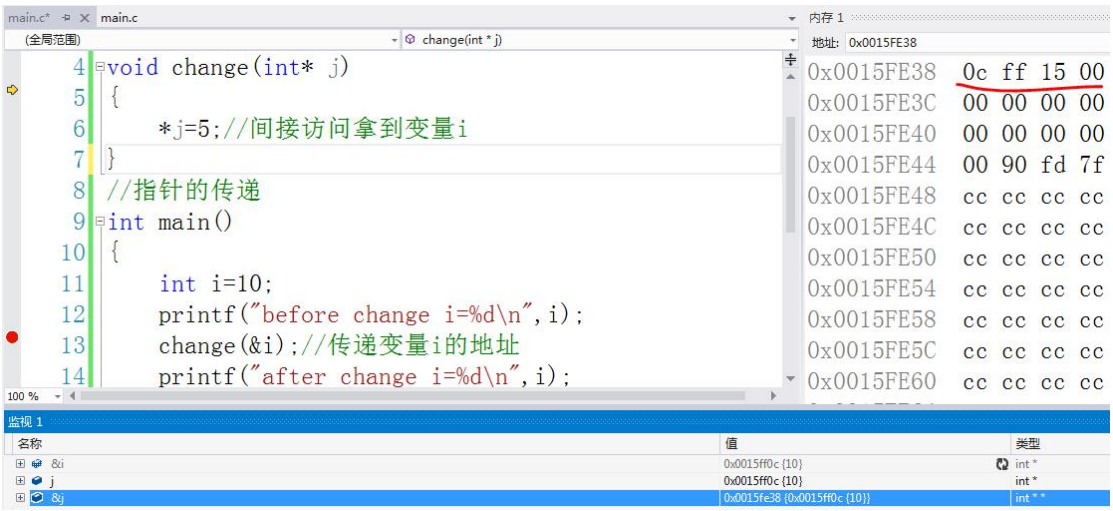

图 5.2.1-3

5.2.2 指针的偏移

前面有指针的传递，指针即地址，就像你找到了一栋楼，该栋楼叫 B 栋，那么往前就是 A 栋，往后就是 C 栋，所以指针的另一个场景就是对其进行加和减，地址进行乘除是没有意义的，就像你家的地址乘 5 是代表什么，没有意义。**工作中，我们把对指针的加减，称之为指针的偏移，加就是向后偏移，减就是向前偏移**。下面我们来看【例 5.2.2-1】

【例 5.2.2-1】指针的偏移使用场景

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

#define N 5
//指针的偏移
int main()
{
    int a[N]={1,2,3,4,5};
    int *p;
    int i;
    p=a;//保证等号两边的数值类型一致
    for(i=0;i<N;i++)//正序输出
    {
        printf("%3d",*(p+i));
    }
    printf("\n----------------\n");
    p=&a[4];//让 p 指向最后一个元素
```

```
    for(i=0;i<N;i++)//逆序输出
    {
        printf("%3d",*(p-i));
    }
    printf("\n");
    system("pause");
    return 0;
}
```

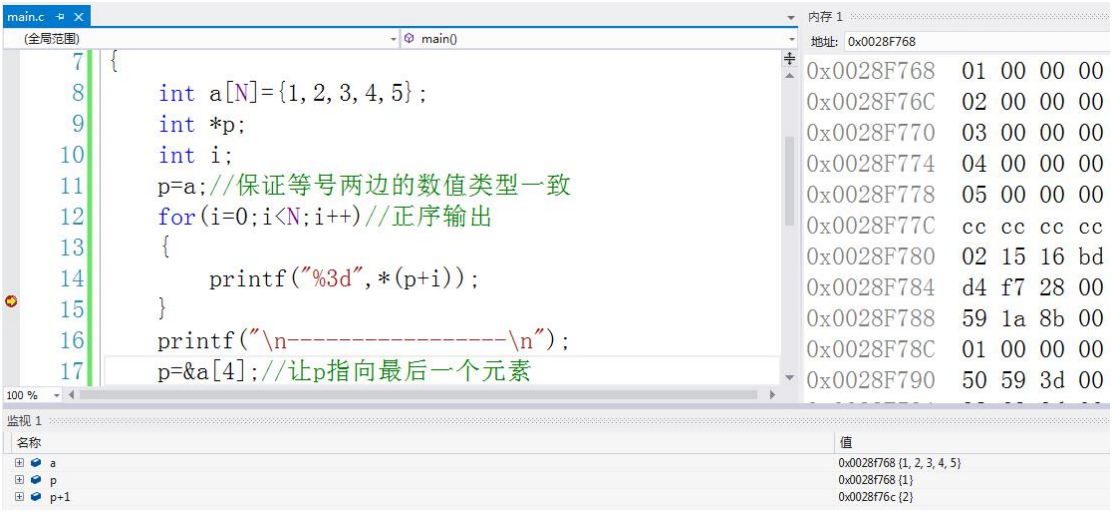

图 5.2.2-1

如图 5.2.2-1 所示，数组名中存着数组的起始地址 0x28F768，其类型为整型指针，所以我们可以将其赋值给整型指针变量 p，可以从监视中看到 p+1 的为 0x28F76C，为什么加 1 不是 69 呢，因为指针变量的加 1，所偏移的长度是其基类型的长度，也就是偏移 sizeof(int)，这样通过*(p+1)就可以拿到元素 a[1]。编译器在编译时，数组的取下标正是转换为指针偏移来完成的。

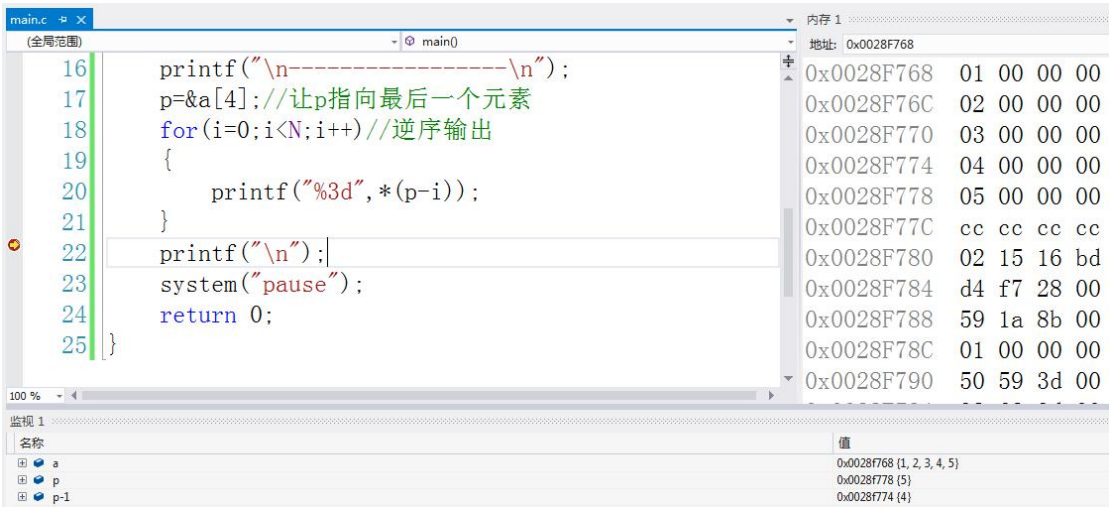

图 5.2.2-2

如图 5.2.2-2 所示，对数组最后一个元素 a[4]进行取地址操作后，得到地址 0x0028F778，通过减法运算，可以让数组元素逆序输出，在图 5.2.2-2 中可以看到 p-1 的值为 0x0028F774。

5.2.3 指针与自增自减运算符

既然已经掌握了指针的使用场景，为什么还要看指针与自增自减运算符的关系呢？其实这就像你掌握了乘法口诀，但是你依然要做各种乘法运算题一样，训练最常见的应用，从而避免在使用中发生错误，下面我们来看【例 5.2.3-1】

【例 5.2.3-1】指针与自增自减

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

//只有比后++，优先级高的操作符，才会作为一个整体，(),[]
int main()
{
    int a[3]={2,7,8};
    int *p;
    int j;
    p=a;
    j=*p++;//先把*p 的值赋给 j，然后再对 p 进行加 1
    printf("a[0]=%d,j=%d,*p=%d\n",a[0],j,*p);
    j=p[0]++;//p[0]赋值给 j，然后对 p[0]进行加 1
    printf("a[0]=%d,j=%d,*p=%d\n",a[0],j,*p);
    system("pause");
    return 0;
}
```

输出结果如图 5.2.3-1 所示，第一次输出为什么是 j=2，*p=7 呢（输出结果如图 5.2.3-1），首先我们前面讲过当我们遇到后++时，就是成两步来看，第一步去掉后++，即 j=*p，因为 p 指向数组的第一个元素，所以 j=2，那第二步是对 p++，还是对*p 整体，也就是数组第一个元素的值加 1 呢，实际是对 p++，因为*操作符和++操作符是相同优先级，只有比++优先级高的操作符（参照附录 2），才会作为一个整体，目前我们使用过的，比++优先级高的就只有(),[]两个操作符。

思考题：如果想让 a[0]为 3，*p 也为 3，怎么修改表达式？

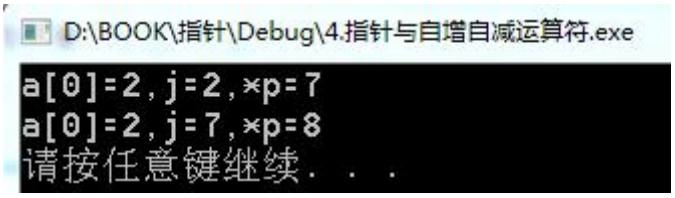

图 5.2.3-1

第二次输出为什么是 j=7，*p=8 呢，首先 p 指向的是元素 a[1]，对于 j=p[0]++,首先去掉++，也就是 j=p[0]，所以 j=7，接着是对 p++，还是对 p[0]++呢，当然是对 p[0]++,所以*p 等于 8。后减减操作符与加加类似，因为编写习惯问题，前++和前--我们使用的很少。

5.2.4 指针与一维数组

为什么一维数组在函数调用进行传递时，它的长度子函数无法知道呢，其实**一维数组名中存储的是数组的首地址**，如图 5.2.4-1，数组名 c 中存的地址为 0x0015faf4，所以在子函数 change 中我们接受一个地址，可以定义一个指针变量，指针变量的基类型要和数组的数据

类型保持一致，通过取值操作，就可以将 h 改为 H，称为**指针法**，获取数组元素，也可以用取下标获取数组元素进行修改，我们称之为**下标法**。

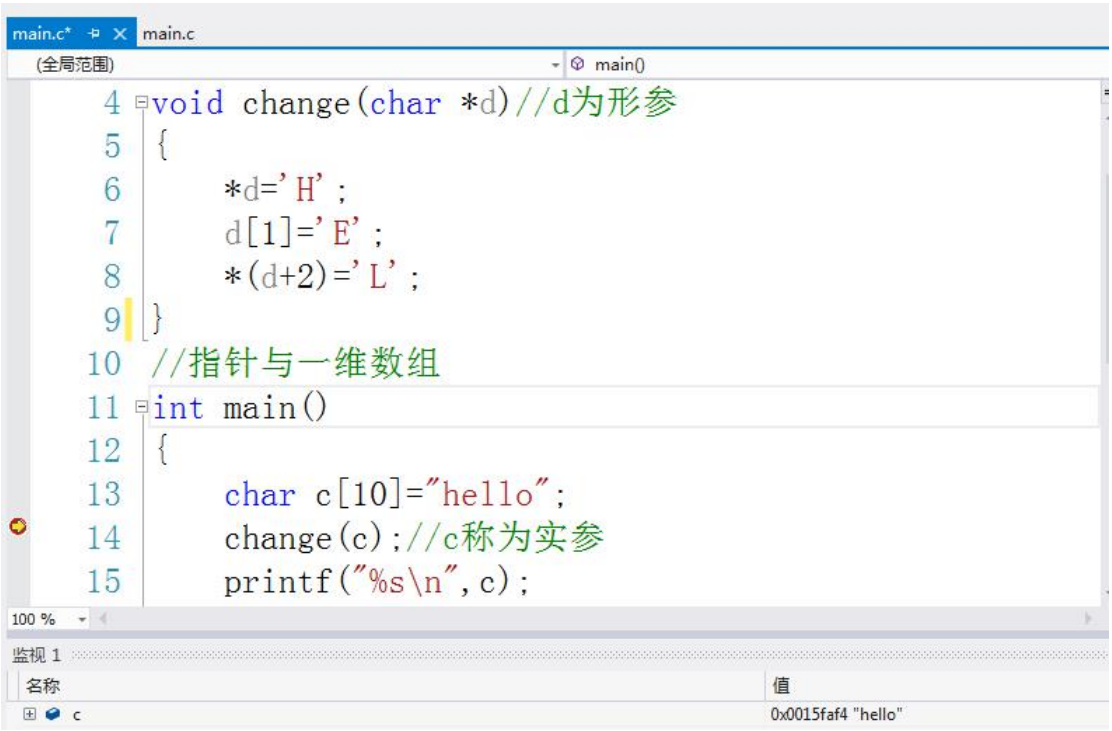

图 5.2.4-1

思考题：假如在子函数中将 hello 中小写的 o 变为大写的 O 如何操作？如果把 hello 翻转怎么操作？就是将 hello 变为 olleh

5.2.5 指针与动态内存申请

很多同学一直都非常想使用动态数组，觉的数组长度固定很不爽，其实是因为我们定义的整型，浮点型，字符型变量，数组变量，都在栈空间，栈空间的使用是在编译时确定大小，如果使用的空间大小不确定，那么我们就需要使用**堆空间**。下面我们来看【例 5.2.5-1】

【例 5.2.5-1】动态内存申请

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
    int i;
    char *p;
    scanf("%d",&i);//输入要申请的空间大小
    p=(char*)malloc(i);//使用 malloc 动态申请堆空间
    strcpy(p,"malloc success");
    puts(p);
    free(p);//free 时必须使用 malloc 申请时返回的指针值，不能进行任何偏移
    printf("free success\n");
    system("pause");
```

}

首先我们来看下 malloc 函数，**#include <stdlib.h>　void *malloc(size_t size);**我们需要给 malloc 传递的参数是一个整型，因为 size_t 类型即为 int，返回值为 void*类型的指针，void*类型指针只能用来存储一个地址，不能进行偏移，原因是 malloc 并不知道我们申请的空间用来存放什么类型的数据，所以当我们要用来存储什么类型数据时，我们会将 void*强转为对应类型，在【例 5.2.5-1】中，我们用来存储字符，所以将其强转为 char*类型。

如图 5.2.5-1 所示，我们定义的整型变量 i，指针变量 p 均在 main 函数的栈空间，通过 malloc 申请的空间会返回一个堆空间的首地址，我们把首地址存入变量 p 中，知道了首地址，我们就可以通过 strcpy 往对应的空间里放字符数据。

既然都是内存空间，为什么还要分栈和堆呢？栈是机器系统提供的数据结构，计算机会在底层对栈提供支持：分配专门的寄存器存放栈的地址，压栈出栈都有专门的指令执行，这就决定了栈的效率比较高。堆则是 C/C++函数库提供的，它的机制是很复杂的，例如为了分配一块内存，库函数会按照一定的算法（具体的算法可以参考数据结构/操作系统）在堆内存中搜索可用的足够大小的空间，如果没有足够大小的空间（可能是由于内存碎片太多），就有可能调用系统功能去增加程序数据段的内存空间，这样就有机会分到足够大小的内存，然后进行返回。显然，堆的效率比栈要低得多。

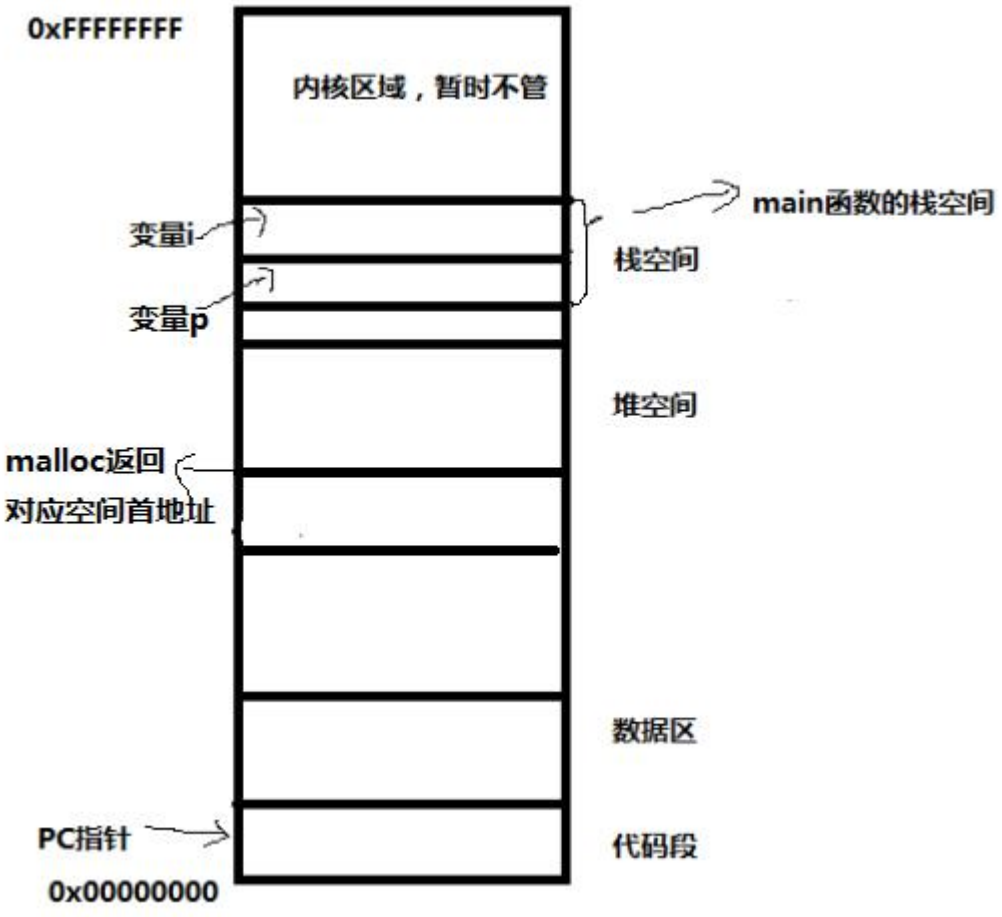

图 5.2.5-1

栈空间由系统自动管理，而堆空间申请和释放需要我们自行管理，所以在例子中我们需要通过 free 进行堆空间的释放，free 函数头文件及格式为 **#include <stdlib.h>　void free(void *ptr);**其入参为 void 类型指针，任何指针均可自动转为 void*类型指针，所以我们把 p 传递给 free 函数时，不用强制类型转换，关键是 p 的地址值必须是 malloc 当时返回给我们的地

址值，不能进行偏移，也就是在 malloc 和 free 之间不能进行 p++等改变变量 p 的操作，原因是申请一段堆内存空间，内核帮我们记录的是起始地址和大小，所以释放时，内核拿对应的首地址进行匹配，匹配不上，进程就会崩溃！可能你会说我要偏移放数据怎么办，可以定义两个指针变量。

思考题：有兴趣的小伙伴可以偏移一下 p，free 看看崩溃的效果哦~

在 free(p)之后，我们需要把 p 置为 NULL，也就是在 free(p)之后加上 p=NULL，如果不置为 NULL，我们这时的 p 称为**野指针**，接下来我们来看【例 5.2.5-2】，free(p)之后没有把 p 置为 NULL 的一个错误场景。

【例 5.2.5-2】野指针的危险

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

int main()
{
    int *p1,*p2,*p3;//定义三个整型指针
    p1=(int*)malloc(4);//申请4个字节空间，存放整型数1
    *p1=1;
    p2=(int*)malloc(4);//申请4个字节空间，存放整型数2
    *p2=2;
    free(p1);//释放原有为p1申请的空间，但是我们没有对p1赋值为NULL
    p3=(int*)malloc(4);
    *p3=3;
    printf("*p3=%d\n",*p3);//这次打印值为3
    *p1=100;
    printf("*p3=%d\n",*p3);//我们并没有对*p3进行修改，但是这里打印值为100
    system("pause");
}
```

代码执行结果如下：

为什么我们没有修改*p3 的值，但是第二次打印却是 100，原因在于 malloc 内部的内存管理算法，把我们刚刚 free(p1)的空间，分配给了 p3，因此 p1 这时候和 p3 指向同一块内存空间，如果我们没对 p1 赋值为 NULL，那么修改代码的人，如果未看到 p1 已经 free，不小心使用 p1，将会导致不可预知的错误，因为我们要把 free 后的指针，置为 NULL。

接下来我们看一个栈空间和堆空间在时间的有效性的差异，下面的例子【例 5.2.5-2】，执行打印结果如图 5.2.5-2，为什么第二次打印会有异常，而且每次执行打印效果都不一样，原因是 print_stack 函数中的字符串存放在栈空间，当函数执行结束后，栈空间会被释放，字符数组 c 原有空间已经被分配给其他函数使用，因此在调用 print_stack()之后，puts(p)会出现打印乱码。而 print_malloc 函数中字符串存放在堆空间，**堆只有我们 free 才会释放，否则在进程执行过程中一直有效**。

【例 5.2.5-2】栈空间与堆空间的差异

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

```
#include <stdlib.h>
#include <string.h>

//函数栈空间释放后，函数内的所有局部变量消失
char* print_stack()
{
    char c[]="I am print_stack";
    puts(c);
    return c;
}
//堆空间不会因函数执行结束而释放
char* print_malloc()
{
    char *p;
    p=(char*)malloc(20);
    strcpy(p,"I am print_malloc");
    puts(p);
    return p;
}
int main()
{
    char *p;
    p=print_stack();//数据放在栈空间
    puts(p);//打印有异常，出现乱码
    p=print_malloc();//数据放在堆空间
    puts(p);
    system("pause");
}
```

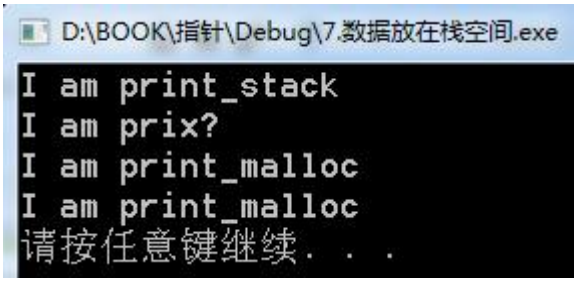

图 5.2.5-2

5.2.6 字符指针与字符数组的初始化

字符指针可以初始化赋值一个字符串，字符数组初始化也可以赋值一个字符串，如下面例子【例 5.2.6-1】所示，char *p="hello"和 char c[10]="hello"有什么区别呢，如图 5.2.6-1 所示，编译器在时，对于字符串常量是存储在数据区中的常量区的，好处是相同的字符串，比如 hello 只会存储一遍，常量区的含义就是字符串本身是不可修改的，所以我们称之为字符串常量，hello 存在字符串常量区，占用 6 个字节，有自己的首地址，char *p="hello"是**将字符串常量"hello"的首地址赋值给 p**，对于 char c[10]="hello"，字符数组 c 在栈空间有

10 个字节大小的空间，这个初始化是将字符串 hello 通过 strcpy 给字符数组 c，因此我们可以将 c[0]修改为 H，而 p[0]拿到是常量区的空间，所以不可以修改。

p 是一个指针变量，因此我们可以将字符串 world 的首地址重新赋值给 p，而数组名 c 本身存储的就是数组的首地址，是确定的，不可修改的，c 等价于符号常量，因此如果 c=”world”打开就会造成编译不通。

【例 5.2.6-1】字符指针与字符数组初始化

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
    char *p="hello";//是把字符串常量"hello"的首地址赋值给 p
    char c[10]="hello";//等价于 strcpy(c,"hello");
    c[0]='H';
    printf("c[0]=%c\n",c[0]);
    printf("p[0]=%c\n",p[0]);
    //p[0]='H';//不可以对常量区数据进行修改
    p="world";//将字符串 world 的地址赋值给 p
    //c="world";//非法
    system("pause");
    return 0;
}
```

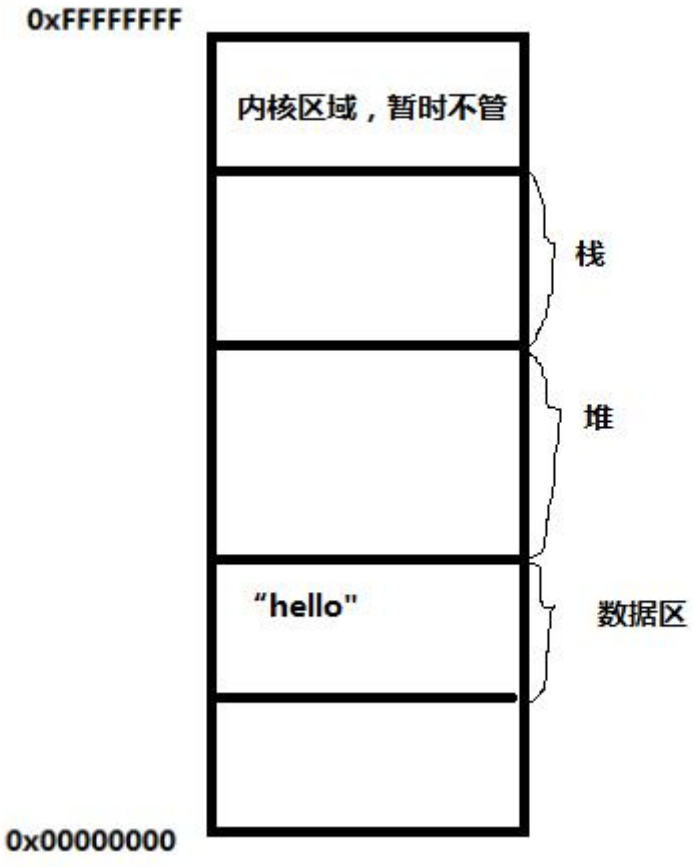

图 5.2.6-1

思考题：有兴趣的小伙伴可以打开 p[0]='H'，运行查看报错提示，是否理解对应的报错提示呢？可以把自己的理解发送到 QQ 群内，与各位读者进行交流。

5.2.7 const 的详细讲解

下面我们来看一下 const int i 的意义：

const int 类型一旦定义以后就不能修改，int 类型是随时可以修改的。

因为 const int 是用来保存一些全局常量的，这些常量在编译期可以改，在运行期不能改。

听起来这像宏，其实这确实就是用来取代宏的：#define PI 3　和 const int Pi = 3; 如果你的代码里用到了 100 次 PI（宏），你的代码中会保存 100 个 3 这个常数。

鉴于使用常数进行运算的机器代码很多时候会比使用变量来的长，如果你换用 100 次 Pi（const int)，程序编译后的机器码里就不需要出现 100 次常量 3，只要在需要的时候引用存有 3 的常量就行了。

const 定义常量从汇编的角度来看，只是给出了对应的内存地址，而不是像#define 一样给出的是立即数，所以，const 定义的常量在程序运行过程中只有一份拷贝，而#define 定义的常量在内存中有若干份拷贝。编译器通常不为普通 const 常量分配存储空间，而是将它们保存在符号表中，这使得它成为一个编译期间的常量，没有了存储与读内存的操作，使得它的效率也很高。

针对 const 修饰指针，存在两种情况

1、const char *ptr;

定义一个指向字符常量的指针，这里，ptr 是一个指向 char* 类型的常量，所以不能用 ptr 来修改所指向的内容，换句话说，*ptr 的值为 const，不能修改。但是 ptr 的声明并不意味着它指向的值实际上就是一个常量，而只是意味着对 ptr 而言，这个值是常量。实验【例 5.2.7-1】如下：ptr 指向 str，而 str 不是 const，可以直接通过 str 变量来修改 str[0]的值，但是确不能通过 ptr 指针来修改。（char const *ptr 与 const char *ptr 等价，通常大家使用 const char *ptr）

【例 5.2.7-1】const 修饰指针

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    char str[]="hello world";
    const char *ptr=str;
    str[0]='H';//操作合法
    puts(ptr);
    ptr[0]='n';//操作非法,编译错误，提示 error C2166: 左值指定 const 对象
    puts(ptr);
    system("pause");
}
```

2、char * const ptr;

定义一个指向字符的指针常数，即 const 指针，实验得知，不能修改 ptr 指针，但是可以修改该指针指向的内容。实验【例 5.2.7-2】如下：

【例 5.2.7-2】const 修饰变量

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    char str[]="hello world";
    char str1[]="how do you do";
    char * const ptr=str;
    str[0]='H';
    puts(ptr);
    ptr[0]='n';//合法
    puts(ptr);
    ptr=str1;//非法，编译错误，error C2166: 左值指定 const 对象
    system("pause");
}
```

通过【例 5.2.7-1】可以看到，const 直接修饰指针时，指针 ptr 指向的内容可以修改，但是指针 ptr 在第一次初始化后，后面不能够再对 ptr 进行赋值，否则会出现编译报错，当然这种场景使用的并不多。

5.2.8 memcpy 与 memmove 的差异

有同学非常疑惑，为什么有了 memcpy 后，还给一个 memmove 的接口呢？其实 memmove 针对源内存和目的内存发生重叠时，依然可以使用，但是这时 memcpy 就不能够使用了，这时有同学又有疑问，既然 memmove 比 memcpy 强大，为啥不去掉 memcpy 呢，原因是 memmove 内部增加了判断，如果内存没有重叠，用 memmove，效率就会低于 memcpy，接下来我们自己来实现以下 memmove，代码见【例 5.2.8-1】，相信各位小伙伴就会一目了然。

【例 5.2.8-1】memmove 的实现

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void mymemmove(void* to,void* from,size_t count)
{
    char* pTo,*pFrom;
    if(to>from)
    {
        pTo=(char*)to+count-1;//注意这里是容易出错的地方
        pFrom=(char*)from+count-1;
        while(count>0)
        {
            *pTo=*pFrom;
            pFrom--;
            pTo--;
            count--;
        }
    }else{
        pTo=(char*)to;
        pFrom=(char*)from;
        while(count>0)
        {
            *pTo=*pFrom;
            pFrom++;
            pTo++;
            count--;
        }
    }
}
#define N 5
int main()
{
    int a[N]={1,2,3,4,5};
```

```
    int b[N]={1,2,3,4,5};
    int i;
    memmove(b+2,b+1,8);
    for(i=0;i<N;i++)
    {
        printf("%3d",b[i]);
    }
    printf("\n");
    mymemmove(a+2,a+1,8);
    for(i=0;i<N;i++)
    {
        printf("%3d",a[i]);
    }
    printf("\n");
    system("pause");
}
```

memmove 之所以能够实现重叠时也能够复制正确，是因为我们需要通过源地址和目的地址的大小进行内存复制，当目的地址的值大于源地址时，我们从后往前复制，代码中备注地方是容易出错的地方，注意减一，最后一个复制的字节是加 count 减 1 的位置，这是某个大公司的面试题，考察的就是对指针偏移的理解。

5.3 数组指针与二维数组

5.3.1 数组指针应用

很多同学在学习 C 语言时，都认为二维数组和二级指针是一回事，二维数组的实现是用二级指针偏移实现的，这是错误的！二维数组通过两次偏移获取到数组中的某一个元素，所使用的指针是数组指针，**数组指针是一级指针**，下面我们来看【例 5.3.1-1】实例

【例 5.3.1-1】二维数组的传递场景

```
#include <stdio.h>
#include <stdlib.h>
//列数必须写
void print(int p[][4],int row)
{
    int i,j;
    for(i=0;i<row;i++)
    {
        for(j=0;j<sizeof(*p)/sizeof(int);j++)
        {
            printf("%3d",p[i][j]);
        }
        printf("\n");
    }
}
//数组指针用于二维数组的传递和偏移
```

```
int main()
{
    int a[3][4]={1,3,5,7,9,11,13,15,17,19,21,23};
    int b[4]={1,2,3,4};
    int i=10;
    int (*p)[4];//定义一个数组指针
    p=a;
    print(a,3);
    system("pause");
    return 0;
}
```

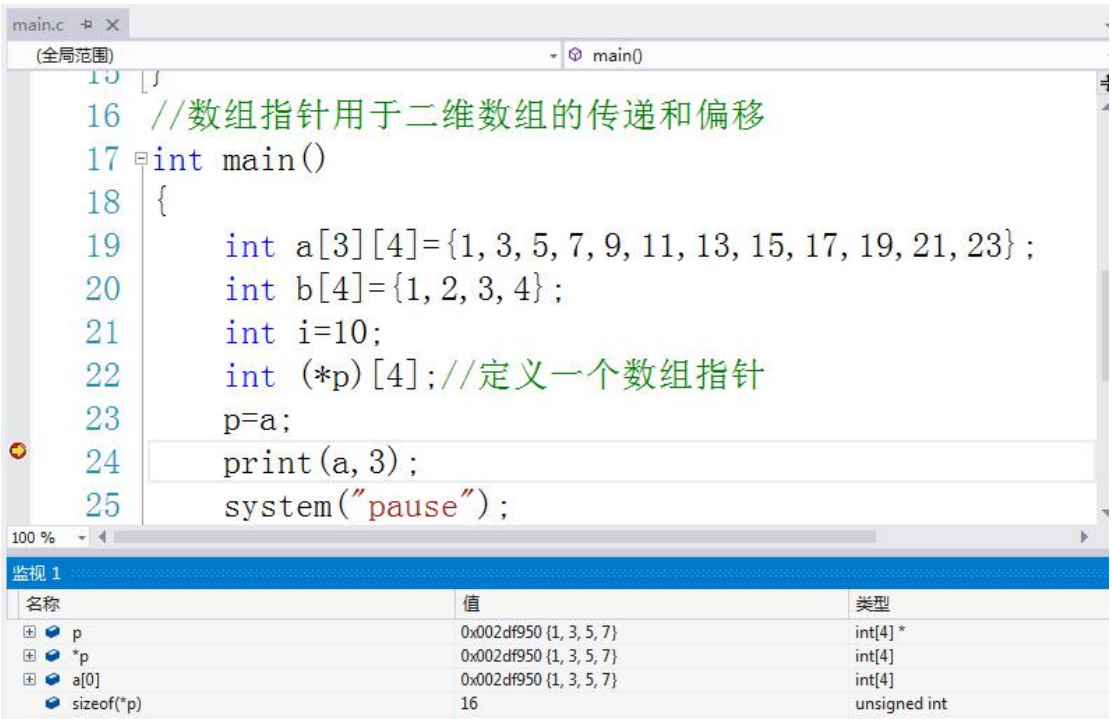

图 5.3.1-1

如图 5.3.1-1 所示，p 是一个数组指针，其指向一个大小为 4 个整型元素的数组，所以 *p 代表一个长度为 4 的整型数组，通过 sizeof(*p)可以看到其大小为 16 个字节，前面我们讲过指针的偏移，p+1 偏移的长度为其基类型的大小，因为 p 指向一个大小为 4 个整型元素的数组，所以 p+1 偏移 16 个字节，因为二维数组名 a 中存储的地址类型为数组指针，所以我们将 a 赋值给 p 不会有编译警告。

通过 print 函数将二维数组打印成矩阵形式，当然我们可以把形参中的 int p[][4]改为 int (*p)[4]，是等价的，二维数组的行数依然无法传递过去的，所以我们通过整型变量 row 传递行数，对于打印位置的 p[i][j],我们可以写成*(*(p+i)+j),当然对于使用二维数组，我们使用 p[i][j]较多，首先 p+i 偏移到对应的行，然后*(p+i)就是拿到对应行，等价于一维数组，而一维数组的数组名存储的就是一级指针，所以*(p+i)+j 就偏移到对应的元素，然后再解引用就拿到对应的元素值。

5.3.2 二维数组的偏移计算

定义 int a[3][4]={1,3,5,7,9,11,13,15,17,19,21,23};假设 a 的地址,0x2000，那么表

5.3.2-1 展示了各种写法偏移后对应的地址值，如果定义数组指针变量 int (*p)[4],将 p=a，那么下面第一列表达形式中，把 a 换出 p，一切成立。

表 示 形 式	含义	地址值
a	二维数组名，指向一维数组 a[0]，即 0 行首地址	0x2000
a[0]， *(a+0)， *a	0 行 0 列元素地址	0x2000
a+1，&a [1]	1 行首地址	0x2010
a [1] ,*(a+1)	1 行 0 列元素 a[1][0] 的地址	0x2010
a[1]+2, *(a+1)+2, &a[1][2]	1 行 2 列元素 a[1][2] 的地址	0x2018
*(a[1]+2), *(*(a+1)+2), a[1][2]	1 行 2 列元素 a [1] [2] 的值	元素值为 13

表 5.3.2-1

5.4 二级指针

一级指针的使用场景是传递与偏移，服务的对象是整型变量，浮点型变量，字符型变量等，那么二级指针既然是指针，其作用也是传递与偏移，服务对象更加简单，只服务于一级指针的传递与偏移！

5.4.1 二级指针的传递

请看实例【例 5.4.1-1】，整型指针 pi，指向整型变量 i，整型指针 pj 指向整型变量 j，通过子函数 change，我们想改变指针变量 pi 的值，让其指向 j，我们知道 C 语言的函数调用是值传递，因此要想在 change 中改变变量 pi 的值，那么必须把 pi 的地址传递给 change，如图 5.4.1-1 所示，pi 是一级指针，&pi 的类型即为 2 级指针，左键拖至内存区域可以看到，0x0031F800 就是指针变量 pi 本身的地址，对应存储的地址是红色标注位置 1 整型变量 i 的地址值（因为小端所以低位在前），将其传入函数 change，change 函数形参 p 必须定义为二级指针，然后在 change 函数内，对 p 进行解引用，就可以拿到 pi，进而对其存储的地址值进行改变。

对于二级指针的传递使用场景，把握两点，**第一：二级指针变量定义是在形参，第二：在调用函数中往往不定义二级指针，如果定义，初始化注意是一级指针的取地址。**

【例 5.4.1-1】二级指针的传递场景

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

void change(int **p,int* pj)
{
```

```
    int i=5;
    *p=pj;
}
//要想在子函数改变一个变量的值，必须把该变量的地址传进去
//要想在子函数改变一个指针变量的值，必须把该指针变量的地址传进去
int main()
{
    int i=10;
    int j=5;
    int *pi;
    int *pj;
    pi=&i;
    pj=&j;
    printf("i=%d,*pi=%d,*pj=%d\n",i,*pi,*pj);//等于 10
    change(&pi,pj);
    printf("after change i=%d,*pi=%d,*pj=%d\n",i,*pi,*pj);
    system("pause");
    return 0;
}
```

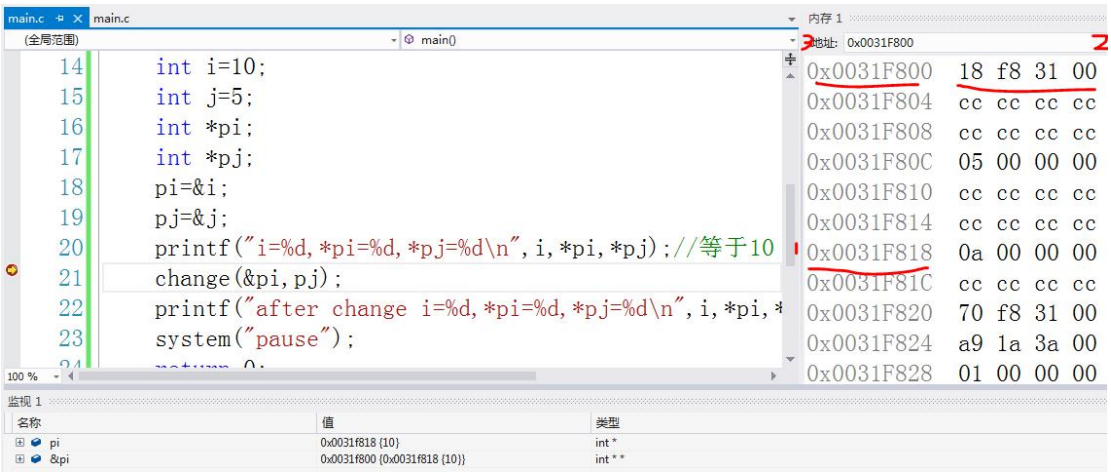

图 5.4.1-1

思考题：猜一下 after change 后，i，*pi，*pj 打印出的值分别是多少？

5.4.2 二级指针的偏移

一级指针的偏移服务于数组，整型一级指针服务于整型数组，所以二级指针的偏移也服务于数组，**服务对象为指针数组**，实际在淘宝购物过程中，大家搜索的商品信息存在内存中，如果以某个查询条件搜索商品，淘宝需要把商品按你的要求进行排序，比如价格从低到高，这时交换内存中商品的信息会极大的降低效率，因为不同用户会有不同的查询需求，每件商品本身的信息存储又较大，假如我们让每个指针指向商品信息，排序比较时，我们比较实际的商品信息，但交换的是指针，这样交换成本将会极大降低，这种思想称为**索引式排序**，在【例 5.4.2-1】中把字符串看成商品信息即可，将指针数组 p 赋值给二级指针 p2，目的是为了演示二级指针的偏移，p2+1 偏移的就是一个指针变量空间的大小，即 sizeof(char*)，对于 win32 控制台应用程序，就是偏移 4 个字节。

【例 5.4.2-1】二级指针的偏移场景

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void print(char *p[])//这里可以写成 char **p
{
    int i;
    for(i=0;i<5;i++)
    {
        puts(p[i]);
    }
}
//二级指针的偏移，服务的是指针数组
int main()
{
    char *p[5];//定义了一个指针数组
    char b[5][10]={"lele","lili","lilei","hanmeimei","zhousi"};
    int i,j,tmp;
    char *t;
    char **p2;//定义一个二级指针
    int a[5]={3,7,9,2,4};
    for(i=0;i<5;i++)//让指针数组中的每一个指针都指向一个字符串
    {
        p[i]=b[i];
    }
    p2=p;
    for(i=4;i>0;i--)//冒泡法排序
    {
        for(j=0;j<i;j++)
        {
            if(strcmp(p2[j],p2[j+1])==1)//判断 p2[j]是否大于 p2[j+1]
            {
                t=p2[j];
                p2[j]=p2[j+1];
                p2[j+1]=t;
            }
        }
    }
    print(p2);//先打印排序结果
    puts("--------------------------");
    for(i=0;i<5;i++)//再看数据存储本身有没有变
    {
        puts(b[i]);
```

```
    }
    system("pause");
    return 0;
}
```

5.5 函数指针

由于本书给大家指导的路线是学完以后，学习 Linux 系统编程和 C++，走 C++后台开发工程师路线，因此函数指针并不重要，因为 C++的重载使用起来更加方便，如果走**嵌入式路线**，那么学完函数指针，在 Linux 系统编程阶段深入研究一下回调函数，对于嵌入式，函数指针要熟练掌握。下面我们来看【例 5.5-1】实例，定义函数指针 p，将函数 b 赋值给 p，为什么可以赋值呢，其实是因为**函数名本身存储的即为函数入口地址**，将 p 传递给函数 a，相当于把一个行为传递给函数 a，之前我们传递给函数的都是数据，通过函数指针可以将行为传递给一个函数，这样我们调用函数 a 可以就可以执行函数 b 的行为，当然也可以实现执行其他函数的行为。

【例 5.5-1】函数指针的使用

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

void b()
{
    printf("I am func b\n");
}
void a(void (*p)())
{
    p();
}
//定义函数指针，初始化只能赋函数名
int main()
{
    void (*p)();//定义了一个函数指针变量
    p=b;//函数指针的返回值及入参要与函数保持一致
    a(p);
    system("pause");
    return 0;
}
```