

第 4 章 数组

（视频讲解：1.5 小时）

假如你的衣柜最下面一排你准备全部放鞋子，放在一堆肯定不合适，所以你把衣柜最下面一排切分为了连续的 10 个格子，你叫你的保姆帮你拿鞋子时，可以说最下面一排，第三个格子，她就知道了。而当我们有 10 个整数在内存中存着，为了方便存取，C 为我们提供了数组，这样我们通过一个符号就可以访问多个元素了。学习本章，你将掌握：

- 一维数组的内存原理及使用
- 二维数组的原理及使用
- 字符数组使用方法
- `str` 等字符串系列函数的使用

4.1 一维数组的定义和引用

4.1.1 数组定义

一个班学生的学习成绩，一行文字，一个矩阵

这些数据的特点是：

1. 具有相同的数据类型
2. 使用过程中需要保留原始数据

C 语言为这些数据，提供了一种构造数据类型：数组。所谓数组就是一组具有相同数据类型的数据的有序集合。

一维数组的定义格式为：

类型说明符 数组名 [常量表达式]；

例如：`int a[10];`

它表示定义了一个整形数组，数组名为 `a`，此数组有 10 个元素。

说明：

1. 数组名定名规则和变量名相同，遵循标识符定名规则。
2. 在定义数组时，需要指定数组中元素的个数，方括弧中的常量表达式用来表示元素的个数，即数组长度。
3. 常量表达式中可以包括常量和符号常量，但不能包含变量。也就是说，C 语言不允许对数组的大小作动态定义，即数组的大小不依赖于程序运行过程中变量的值。

以下是错误示例：

```
int n;
scanf( "%d" , &n); /*在程序中临时输入数组的大小 */
int a[n];
```

数组声明中其他常见的错误：

- ① `float a[0]; /* 数组大小为 0 没有意义 */`
- ② `int b(2)(3); /* 不能使用圆括号 */`
- ③ `int k=3, a[k]; /* 不能用变量说明数组大小 */`

4.1.2 一维数组在内存中的存放

一维数组 `int mark[100]` 在内存中的存放情况如图 4.1.2-1，每个元素都是整型元素，占用 4 个字节，数组元素的引用方式是 **数组名 [下标]**，所以我们访问数组 `mark` 的 100 元素的方式是 `mark[0]` 一直到 `mark[99]`，注意没有 `mark[100]` 元素，数组元素从 0 编号开始的。

下面来看一下一维数组的初始化方法

1. 在定义数组时对数组元素赋以初值。

例如: `int a [10] = {0,1,2,3,4,5,6,7,8,9};` 不能写成 `int a[10];a[10]={0,1,2,3,4,5,6,7,8,9}`

2. 可以只给一部分元素赋值。

例如: `int a [10] = {0, 1, 2, 3, 4};`

定义 `a` 数组有 10 个元素，但花括弧内只提供 5 个初值，这表示只给前面 5 个元素赋初值，后 5 个元素值为 0

3. 如果想使一个数组中全部元素值为 0，可以写成:

`int a [10] = {0,0,0,0,0,0,0,0,0,0};`

或 `int a [10] = {0};`

4. 在对全部数组元素赋初值时，由于数据的个数已经确定，因此可以不指定数组长度。

例如 `int a [] = {1, 2, 3, 4, 5};`

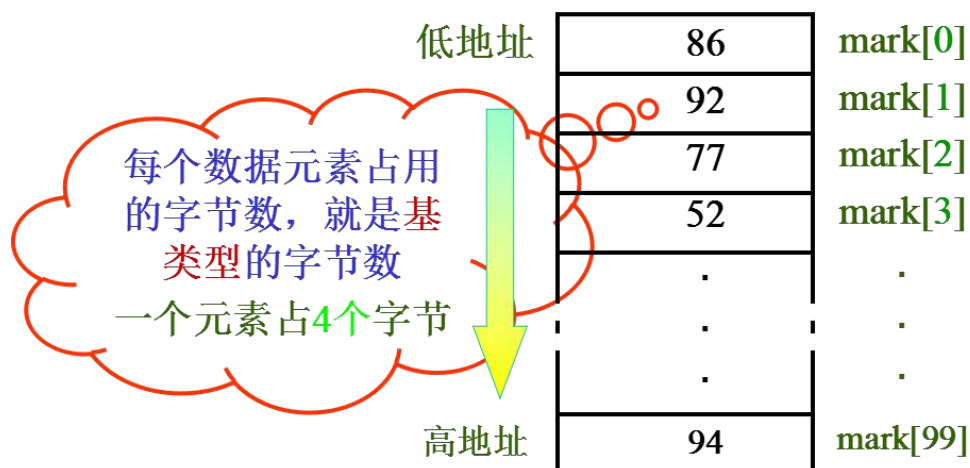


图 4.1.2-1

下面来看一个数组的实例，掌握数组元素的赋值，访问，数组传递，【例 4.1.2-1】是全部代码，由于截图无法全部显示，首先展示全部代码，然后每一部分进行单独解析。

【例 4.1.2-1】一维数组的存储及函数传递

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

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

```
//一维数组的传递，数组长度无法传递给子函数
```

```
//C 语言的函数调用是值传递
```

```
void print(int b[], int len)
```

```
{
```

```
    int i;
```

```
    for(i=0;i<len;i++)
```

```

    {
        printf("%3d", b[i]);
    }
    b[4]=20; //在子函数中修改数组元素
    printf("\n");
}
//数组越界
//一维数组的传递
#define N 5
int main()
{
    int j=10;
    int a[5]={1, 2, 3, 4, 5}; //定义数组时候, 必须长度固定
    int i=3;
    a[5]=20; //越界访问
    a[6]=21;
    a[7]=22; //越界访问会造成数据异常
    print(a, 5);
    printf("a[4]=%d\n", a[4]); //a[4]发生改变
    system("pause");
}

```

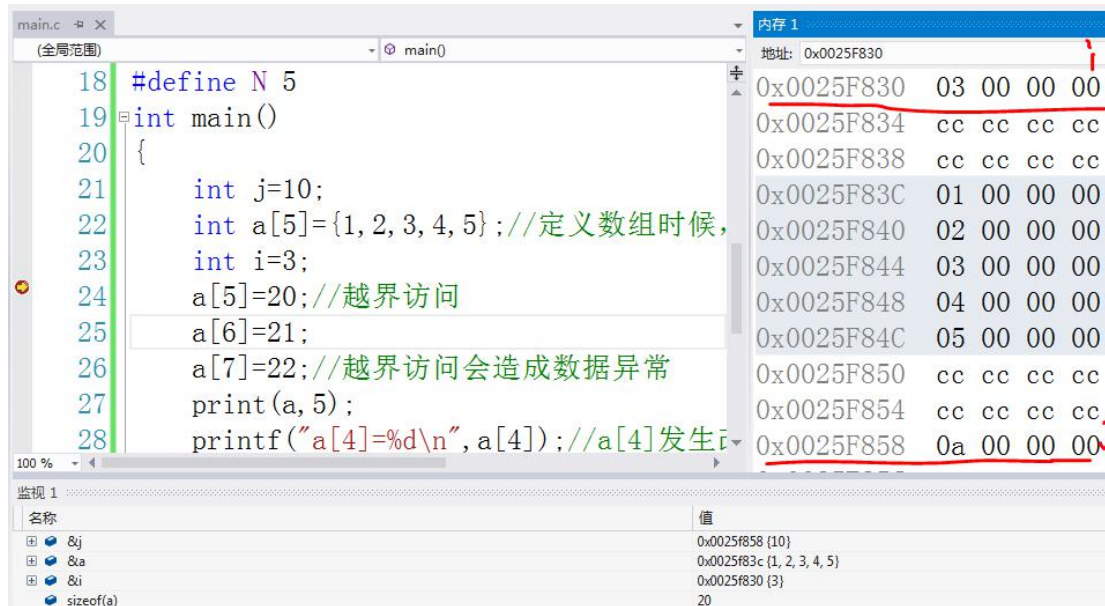


图 4.1.2-2

如图 4.1.2-2, 我们在 24 行左键打断点, 然后点击运行, 在监视窗口我们一次输入 &j, &a, &i 来查看整型变量 j, 整型数组 a, 整型变量 i 的地址, 左键拖动对应地址到内存窗口即可看到三个变量的地址, 就像我们给衣柜的每一个格子的编号, 第一格, 第二格依次到柜子的最后一格, 操作系统给内存的每一个位置也给了一个编号, 对于 win32 控制台应用程序来说, 这个编号的范围是从 0x00 00 00 00 到 0xFF FF FF FF, 总计为 2 的 32 次方, 大小为 4G, 我们把这些编号称为地址。

我们可以看到先定义的变量 j 的地址大于后定义的变量 i 的地址, 所以先定义的变量放

在高地址，后定义的变量在低地址，变量 j，变量 a，变量 i 都是在 main 函数中，其实每个函数开始执行时，系统会为其分配对应的函数栈空间，而变量 j，变量 a，变量 i 都是在 main 函数的栈空间中的，由于后定义的变量在上面，我们把这种效果成为栈先上增长。

我们在监视中输入 sizeof(a)，可以看到数组 a 的大小为 20 个字节，计算方法其实就是 sizeof(int)*5，数组中有 5 个整型元素，每个大小为 4 个字节，所以总计 20 个字节，访问元素依次是从 a[0]到 a[4]，a[5]=20,a[6]=21 均为访问越界，（如图 4.1.2-3）当执行到第 26 行的时候，我们可以看到数组 a 与变量 j 中间的 8 个字节的保护空间已经被赋值（微软的编译器在不同的变量间设置了保护空间），当我们执行到 27 行时，我们发现变量 j 的值被修改了，这就是访问越界的危险性，没有对变量 j 进行赋值，其值却发生了改变！

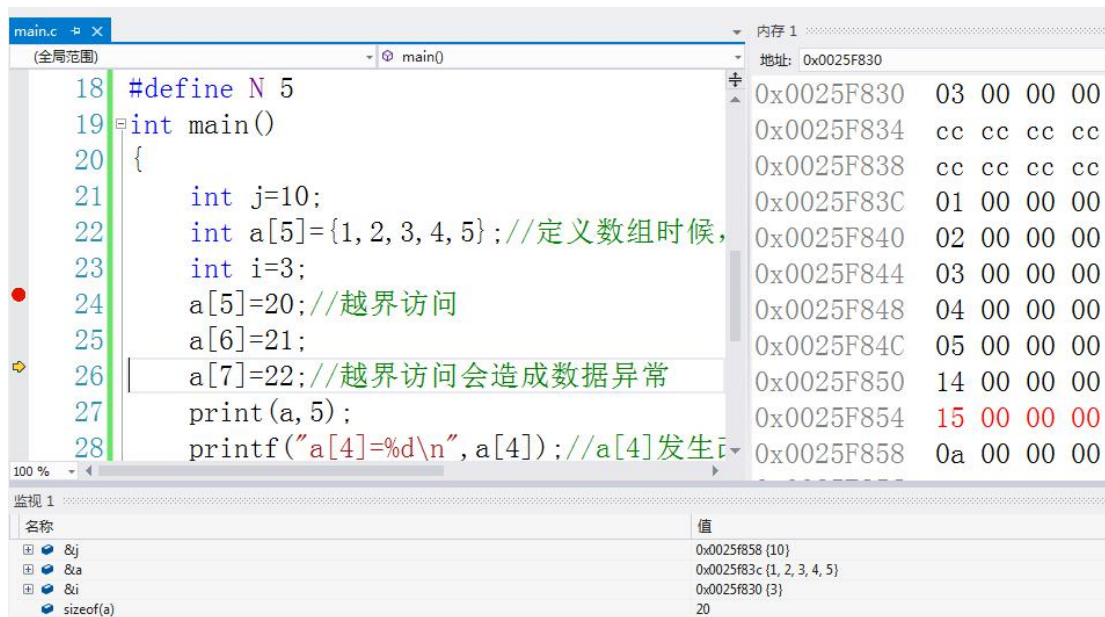


图 4.1.2-3

数组另一个值得关注的地方是，编译器并不检查程序对数组下标的引用是否在数组的合法范。这种不加检查的行为有好处也有坏处。好处是不需要浪费时间对有些已知是正确的数组下标进行检查。坏处是这样做将使无效的下标引用无法被检测出来。一个良好的经验法则则是：如果下标值是从那些已知是正确的值计算得来，那么就无需检查，从用户输入的数据产生而来的，那么在使用它之前必须进行检测，确保它们位于有效范围内（对于后端开发工程师，如果这个下标是从前端接收过来的，一定要判断）

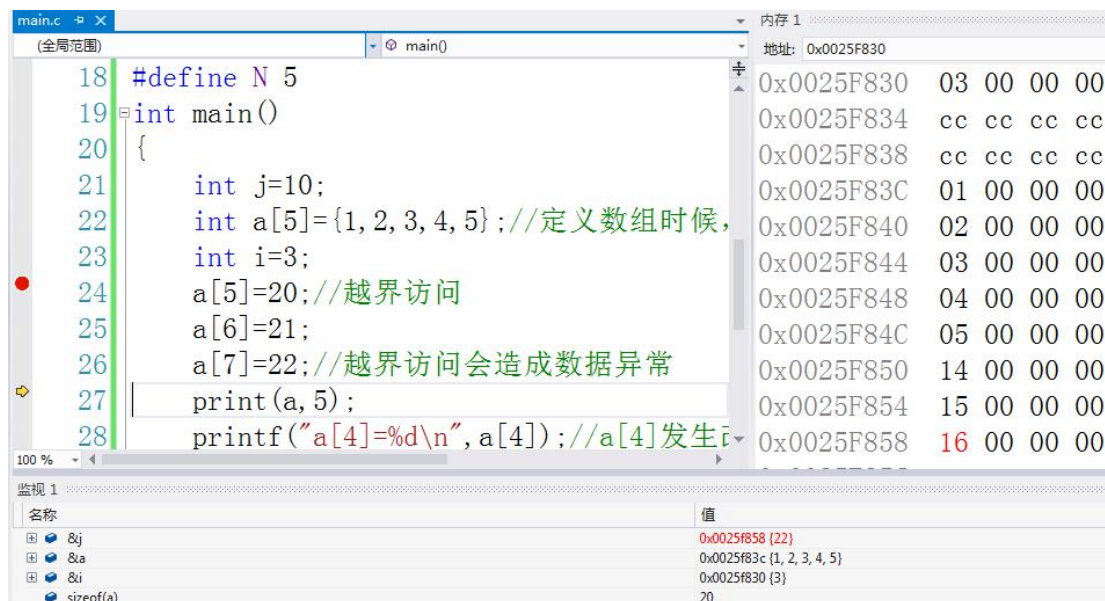


图 4.1.2-4

如图 4.1.2-4，在 27 行，我们按 F11，进入 print 函数，这个时候，我们发现数组 b 的大小变为 4 个字节（如图 4.1.2-5），其实是因为一维数组在传递的时候，它的长度是传递不过去的，所以我们通过 len 来传递数组中的元素个数。实际数组名里存的是数组的首地址，在调用函数传递时，是将数组的首地址给了变量 b（其实变量 b 是指针类型，具体原理我们到下一章会进行讲解），b[] 的方括号中填写任何数字都是没有意义的。这时我们在 print 函数内修改元素 b[4]=20，因为我们可以看到数组 b 的起始地址和 main 函数的数组 a 是同一个，在内存中就是同一个位置，当函数执行结束时，数组 a 中的元素 a[4] 就得到了修改。

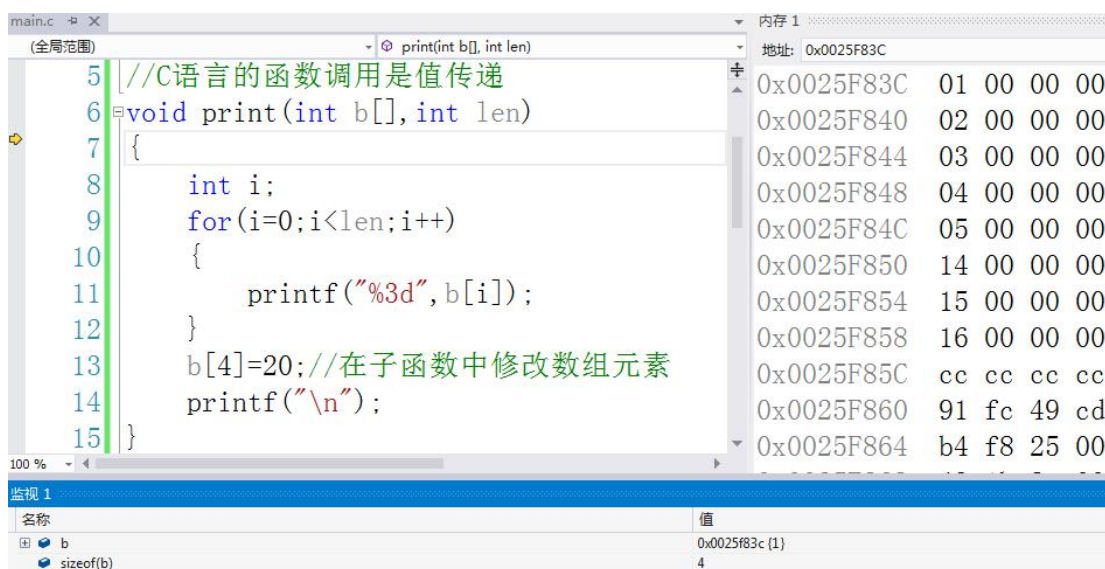


图 4.1.2-5

4.1.3 栈空间不能定义过大数组

Windows 的单个函数栈空间大小是 1M，Linux 的单个函数默认栈空间大小默认是 10M（Linux 下可以修改），超出 1M 后会出现什么情况呢，请【例 4.1.3-1】代码，假如我们的 N 设置的为 25 万，这时运行没有任何异常，但是当我们把 N 的值改为 26 万时，执行就会

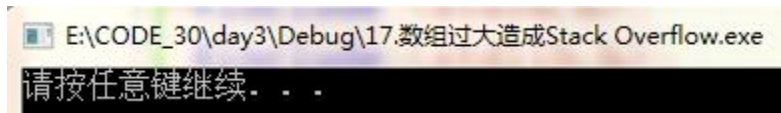
出现 Stack Overflow（指栈空间溢出），因为我们定义的数组是在栈空间上，当数组的大小为 26 万时，超出了单个函数的栈空间，因此就会发生栈空间溢出，所以大家在使用栈空间时，**不要使用过大数组**，如果需要使用大数组，在下一章指针章节讲会讲解，使用堆空间即可。

【例 4.1.3-1】单个函数栈空间上限

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

#define N 250000
int main()
{
    int arr[N]={0};
    system("pause");
}
```

运行效果：



【例 4.1.3-2】单个函数栈空间访问越界

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

#define N 260000//超出了单个函数的栈空间限制
int main()
{
    int arr[N]={0};
    system("pause");
}
```

运行效果如图 4.1.3-1：

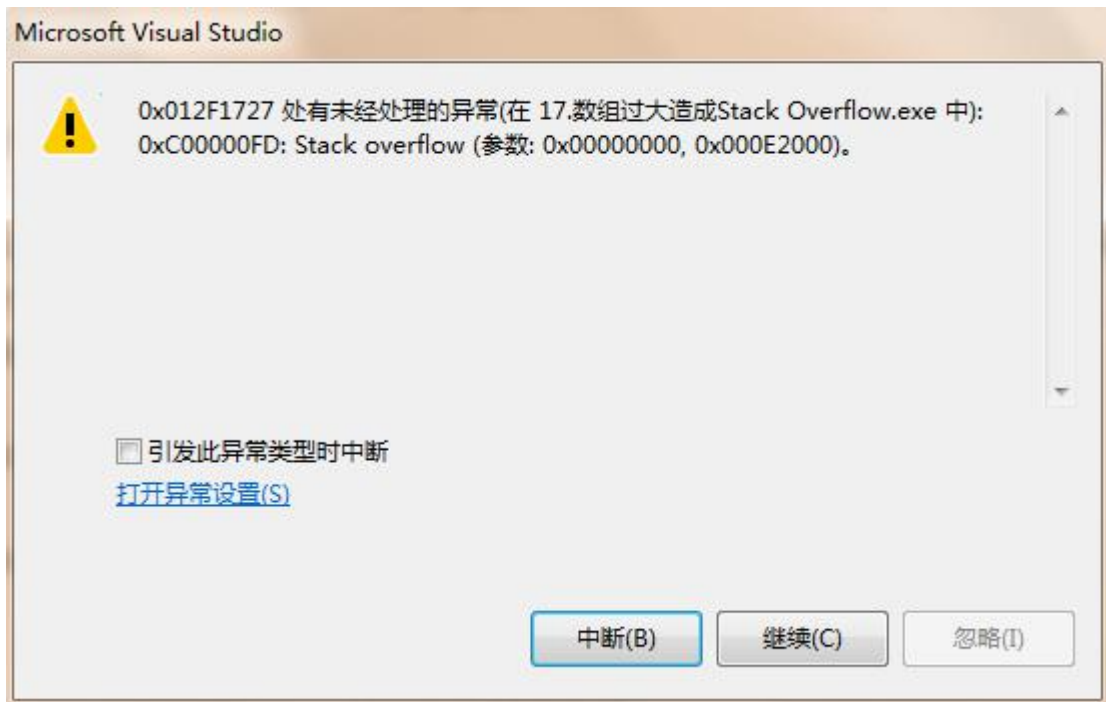


图 4.1.3-1

4.2 二维数组

4.2.1 二维数组的定义与引用

二维数组定义的一般形式为

类型说明符 数组名 [常量表达式] [常量表达式] ;

例如：定义 a 为 3×4 (3 行 4 列) 的数组，b 为 5×10 (5 行 10 列) 的数组。如下：

float a [3] [4] , b [5] [10] ;

注意：我们可以把二维数组看作是一种特殊的一维数组：它的元素又是一个一维数组。

例如：可以把二维数组 a[3][4] 看作是一个一维数组，它有 3 个元素：a [0]、a [1]、a [2]，每个元素又是一个包含 4 个元素的一维数组。如图 4.2.1-1

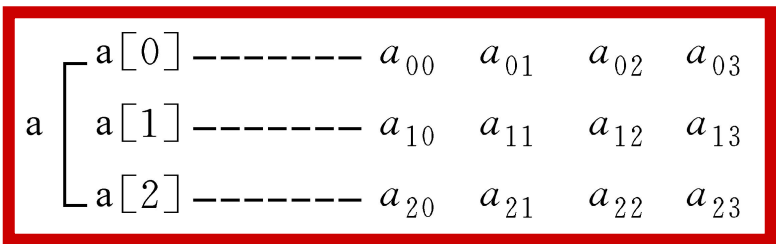


图 4.2.1-1

二维数组中的元素在内存中的排列顺序是：按行存放，即先顺序存放第一行的元素，再存放第二行的元素，数组元素获取依次是从 a[0][0]，a[0][1] 到最后一个元素 a[2][3]。

图 4.2.1-2 表示对 a [3] [4] 数组中每个元素存放的顺序

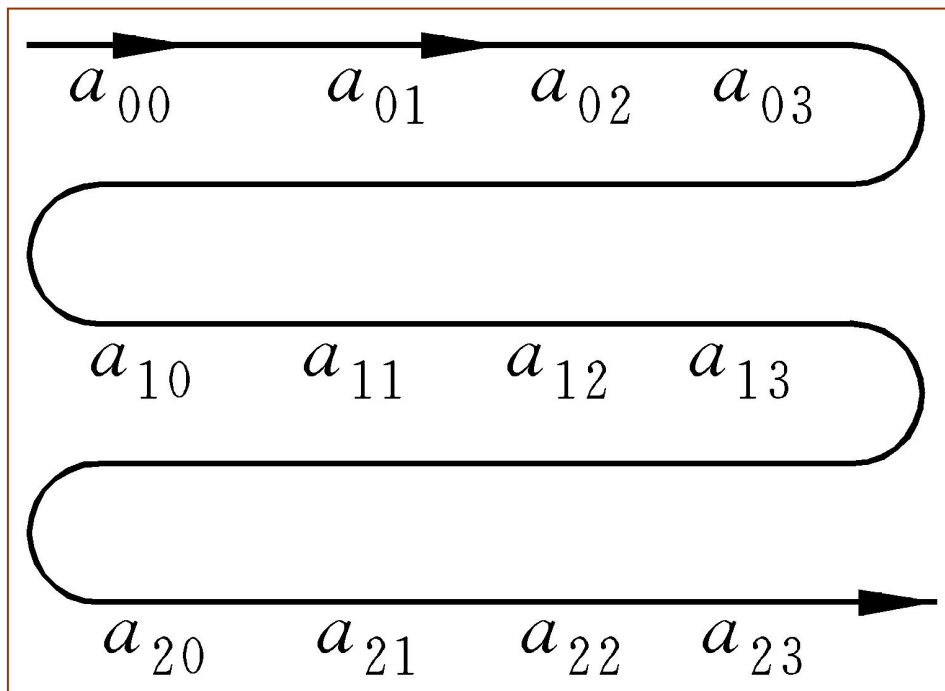


图 4.2.1-2

4.2.2 二维数组初始化及传递

可以用下面 4 种方法对二维数组初始化:

1. 分行给二维数组赋初值。

例如: `int a [3] [4] = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}};`

2. 可以将所有数据写在一个花括号内, 按数组排列的顺序对各元素赋初值。

例如: `int a [3] [4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};`

3. 可以对部分元素赋初值。

例如: `int a [3] [4] = {{1}, {5}, {9}};` 效果如图 4.2.2-1

1	0	0	0
5	0	0	0
9	0	0	0

图 4.2.2-1

4. 如果对全部元素都赋初值, 则定义数组时对第一维的长度可以不指定, 但第二维的长度不能省。

例如: `int a [3] [4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};` 它等价于: `int a [] [4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};`

接下来我们通过一个实例【例 4.2.2-1】来查看二维数组的初始化, 函数间传递, 内存存储情况。

【例 4.2.2-1】二维数组的存储与传递

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



```

#include <stdlib.h>

//二维数组的首地址，赋值给 b
//行不能传递过去，列一定要写
void print(int b[][4], int row)
{
    int i, j;
    for(i=0; i<row; i++) //外层循环为行
    {
        for(j=0; j<sizeof(b[0])/sizeof(int); j++) //内层循环为列
        {
            printf("%3d", b[i][j]);
        }
        printf("\n");
    }
}

//二维数组的存储结构
//二维数组的传递
int main()
{
    int a[3][4]={1, 3, 5, 7, 2, 4, 6, 8, 9, 11, 13, 15};
    float b[4]={1, 2, 3, 4};
    int c[3][4]={ {1}, {5, 9} }; //可以只对部分元素进行初始化
    print(a, 3);
    printf("a[2][3]=%d\n", a[2][3]); //打印最后面一个元素
    system("pause");
    return 0;
}

```

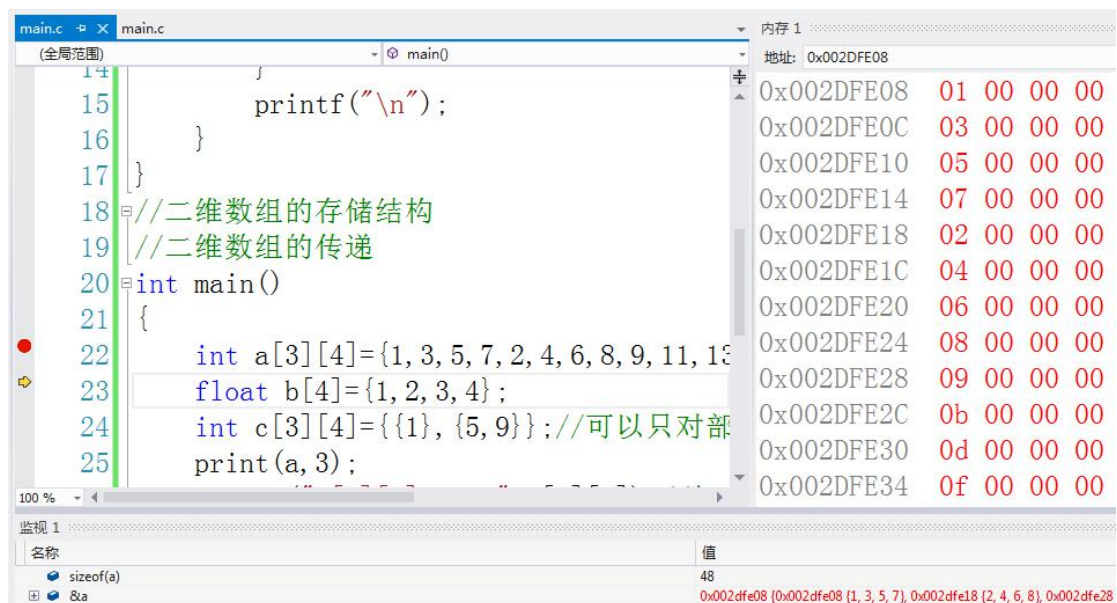


图 4.2.2-2

通过图 4.2.2-2 可以发现二维数组中每一个元素按顺序存储，从低地址到高地址，大小为 `sizeof(int)*元素个数`，元素个数为行乘以列，总计 12 个元素，所以大小为 48 个字节。当执行到 `print` 函数时，按 F11 即可进入 `print` 函数，二维数组在传递时，列数一定要写，因为二维数组传递时也是以指针变量（数组指针，下章讲解）形式传递的，当然列数要与主函数中的二维数组 `a` 的列数相同，我们在图 4.2.2-3 中可以看到 `b` 的地址值与 `a` 相等，同时 `sizeof(b[0])` 为 16，其实 `b[0]` 代表第一行，是一个一维数组，除以 `sizeof(int)` 就可以得到一行有几个元素，二维数组在打印时，外层为行，内层为列，这样可以以矩阵效果打印二维数组每一个元素。

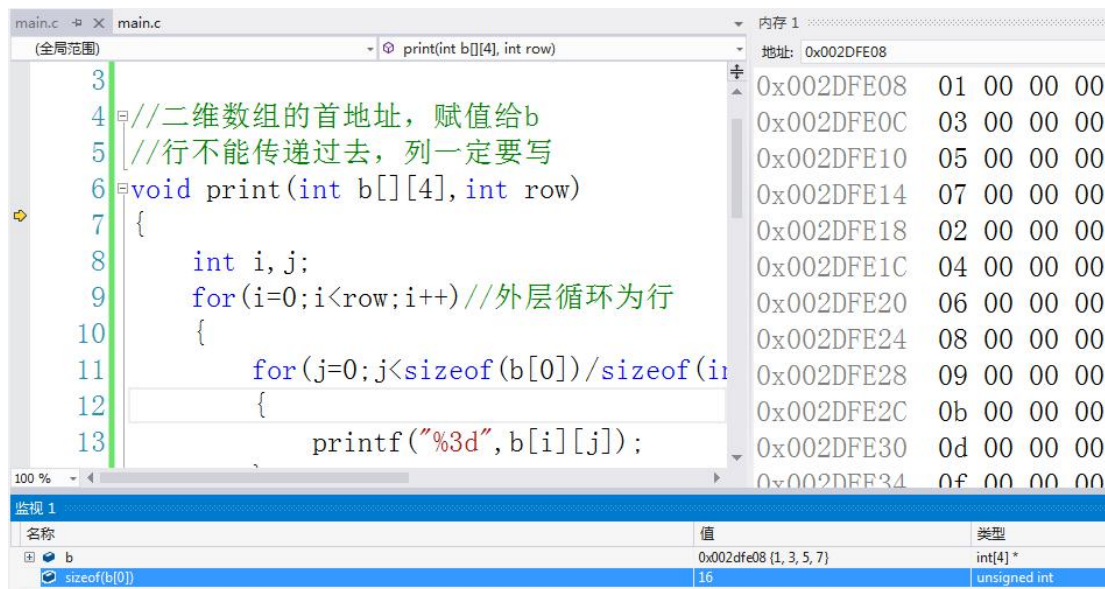


图 4.2.2-3

4.3 字符数组

4.3.1 字符数组的定义及初始化

定义方法与前面介绍的类似。例如：

```
char c [10] ;
```

1. 可以对每个字符单独赋值进行初始化

```
c[0]=' l' ;c[1]=' ' ;c[2]=' a' ;c[3]=' m' ;c[4]=' ' ;c[5]=' h' ;c[6]=' a' ;
c [7]=' p' ;c [8]=' p' ;c [9]=' y' ;
```

2. 也可以 `char c[10]={ 'l' , ' a' , ' m' , ' h' , ' a' , ' p' , ' p' , ' y' }`

工作中我们不用以上两种初始化方式，原因是字符数组我们用来存取字符串，通常所采用的初始化方式是 `char c[10]="hello"`，为什么采用这个初始化方式呢，因为 C 语言规定字符串结束标志为 `'\0'`，而系统对字符串常量自动加一个 `'\0'`，为了保证处理方法一致，我们会人为的在字符数组添加 `'\0'`，所以字符数组存储的字符串长度必须比字符数组小一个字节，比如，例如 `char c[10]` 最长存储 9 个字符，剩余 1 个字符用来存储 `'\0'`，下面我们通过实例【例 4.3.1-1】来查看

【例 4.3.1-1】字符数组初始化及传递

```
#include <stdio.h>
#include <stdlib.h>
void print(char c[])
```

```

{
    int i=0;
    while(c[i])
    {
        printf("%c",c[i]);
        i++;
    }
    printf("\n");
}

//字符数组存储字符串，必须存储结束符，'\0'
//scanf 读取字符串使用%s
int main()
{
    char c[5]={'h','e','l','l','o'};
    char d[5]="how";
    printf("%s----%s\n",c,d);//会发现打很多烫
    scanf("%s%s",c,d);
    printf("%s----%s\n",c,d);
    print(c);
    system("pause");
    return 0;
}

```

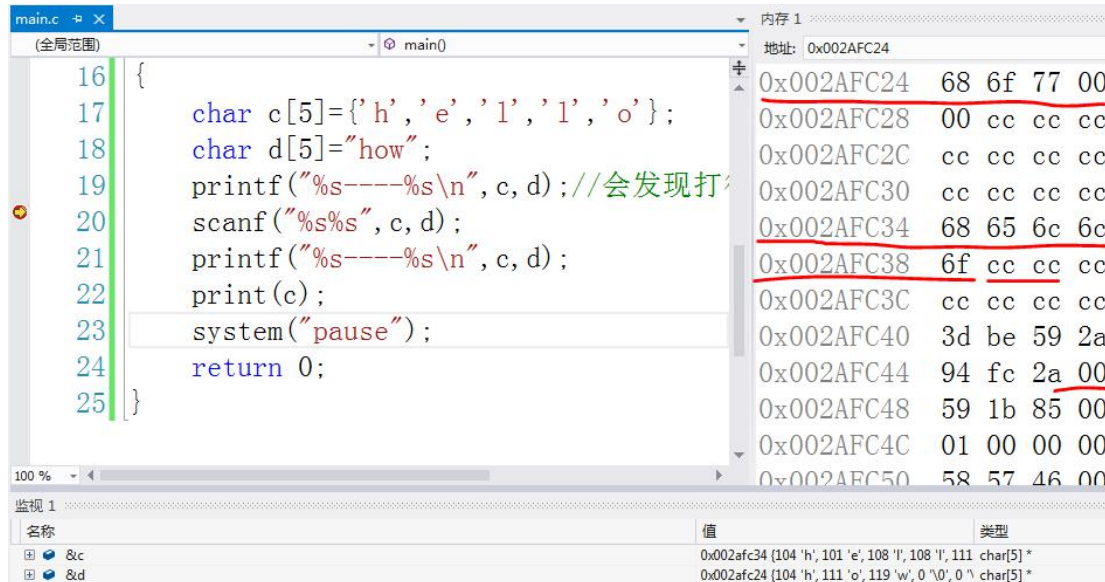


图 4.3.1-1



图 4.3.1-2

执行结果如图 4.3.1-2，为什么我们赋值 hello，打印出现很多烫，因为 printf 通过 %s 打印字符串时，原理是依次输出每一个字符，当读到结束符 '\0' 时，结束打印，scanf 通过 %s

读取字符串，我们对 c 和 d 分别输入 are you，中间加一个空格，scanf 使用 %s 读取字符串时，会忽略空格和回车，我们通过 print 函数模拟实现 printf 的 %s 效果，如图 4.3.1-3 所示，当 c[i] 为 '\0' 时，其值也是 0，循环结束，当然也可以写为 c[i]!='\0'。

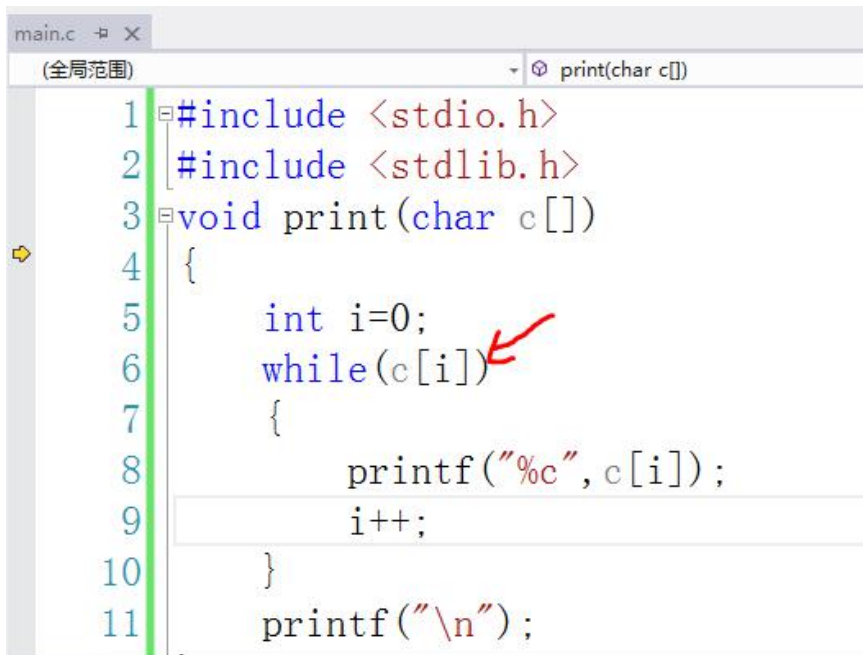


图 4.3.1-3

4.3.2 gets 与 puts

gets 函数类似 scanf 函数，用于读取标准输入，前面我们已经掌握 scanf 函数在读取字符串时遇到空格，就认为读取结束，所以当输入的字符串存在空格时，我们使用 gets 函数进行读取，char *gets(char *str); gets() 函数从 STDIN(标准输入)读取字符并把它们加载到 str(字符串)里,直到遇到新行(\n)或到达 EOF. 新行字符翻译为一个 null 中断符。如图 4.3.2-1，在第 8 行打断点，执行后，我们输入 how are you，总计 11 个字符，可以看到 gets 会读取空格，同时可以看到我们并未给数组进行初始化赋值，但是最后有 '\0'，因为 gets 遇到 \n 后，不会存储 \n，而是翻译为 '\0' 空字符。

int puts(char *str); 函数 puts() 把 str(字符串)写到 STDOUT(标准输出)上. puts() 成功时返回非负值，失败时返回 EOF. 如图 4.3.2-1 所示，puts 会将数组 c 中存储的 hao are you 字符串打印到屏幕上，同时会打印换行，相对于 printf 函数，puts 只能用于输出字符串，同时会多打一个换行符。

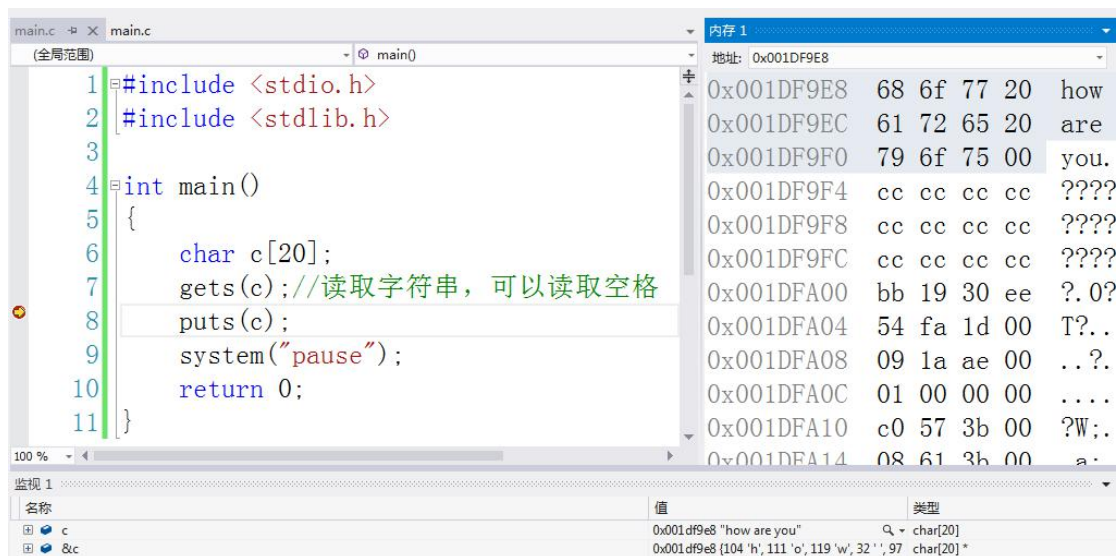


图 4.3.2-1

4.3.3 str 系列字符串操作函数

str 系列字符串操作函数主要包括 strlen, strcpy, strcmp, strcat 等函数（所有函数可以加入前言的 QQ 群，群内有 C / C++ 函数大全），strlen 用于统计字符串长度，strcpy 用于将某个字符串复制到字符数组，strcmp 用于比较两个字符串大小，strcat 用于将两个字符串连接到一起。

```
#include <string.h>
size_t strlen( char *str );
char *strcpy( char *to, const char *from );
int strcmp( const char *str1, const char *str2 );
char *strcat( char *str1, const char *str2 );
```

针对传参类型为 char*，直接放入字符数组的数组名即可。

接下来我们通过一个实例【例 4.3.3-1】来学习 str 系列函数，掌握每一个函数的内部实现。

【例 4.3.3-1】str 系列接口使用

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

int mystrlen(char c[])
{
    int i=0;
    while(c[i++]);
    return i-1;
}
//strlen 统计字符串长度
int main()
{
    int len;//用于存储字符串长度
```



```

char c[20];
char d[100]="world";
while(gets(c)!=NULL)
{
    puts(c);
    len=strlen(c);
    printf("len=%d\n", len);
    len=mystrlen(c);
    printf("mystrlen len=%d\n", len);
    strcat(c, d);
    strcpy(d, c); //c 中的字符串复制给 d
    puts(d);
    printf("c?d %d\n", strcmp(c, d));
    puts(c);
}
system("pause");
return 0;
}

```

通过循环 gets 读取字符串是为了方便大家不断的输入不同的字符串查看程序执行效果，以及当大家修改了程序中某一部分时，可以多种输入测试。如果要结束循环，输入 ctrl+z 即可结束循环。图 4.3.3-1 为我们输入 hello 的执行结果，strlen 计算字符串长度为 5，我们自己写的函数就是 strlen 的计算原理，通过判断结束符确定字符串长度。



图 4.3.3-1

strcpy 是将字符串中的字符，一个字符一个字符的赋值给目标字符数组，例子中我们将 c 复制给 d，就是将 c 中的每一个字符依次赋值给 d，也会将结束符赋值给 d。**注意目标数组一定要大于字符串大小，也就是 sizeof(d)>strlen(c)，否则会造成访问越界。**

strcmp 比较两个字符串的大小，由于字符数组 c 中的字符串与 d 相等，所以返回 0，如果 c 中的字符串大于 d，那么返回值为 1，如果 c 中的字符串小于 d，那么返回值为 -1，如何比较两个字符串的大小呢，其实是从头开始比较，比较相同位置字符的 ASCII 码值，如果发现不相等就会直接返回，否则会接着往后比较，例如 strcmp("hello", "how"), 返回值就是 -1，也就是 **hello 小于 how**，因为第一个字符 h 相等，就接着往后比，e 的 ASCII 码小于 o，然后就返回 -1。

strcat 是将一个字符串接到另外一个字符串的末尾，例子中字符数组 c 中为 hello，我们

将 d 中的 world 拼接，最终结果为 helloworld，注意目标数组必须大于拼接后的字符串大小，也就是 `sizeof(c)>strlen("helloworld")`。

思考题：假如我们 COPY 的不是字符串，而是把一个整型数组中的内容复制到另外一个整型数组中，是否可以使用 `strcpy`，如果不行，那应该怎么办？

4.3.4 strn 系列字符串操作函数

`strn` 系列函数包括 `strncpy`，`strncmp`，`strncat`，当我们只想复制一部分源字符串的字符时，我们需要用 `strncpy`，当只需要比较两个字符串的一部分是否相等时，我们使用 `strncmp`，当需要将一个字符串的部分字符拼接到另一个字符串时，我们需要使用 `strncat`，下面是这三个函数的用法：

```
char *strncpy( char *to, const char *from, size_t count );
```

功能：将字符串 *from* 中至多 *count* 个字符复制到字符串 *to* 中。如果字符串 *from* 的长度小于 *count*，其余部分用 `'\0'` 填补。返回处理完成的字符串。

```
int strncmp( const char *str1, const char *str2, size_t count );
```

功能：比较字符串 *str1* 和 *str2* 中至多 *count* 个字符。返回值如表 4.3.4-1：

返回值	解释
less than 0	str1 is less than str2
equal to 0	str1 is equal to str2
greater than 0	str1 is greater than str2

表 4.3.4-1

如果参数中任一字符串长度小于 *count*，那么当比较到第一个空值结束符时，就结束处理。

```
char *strncat( char *str1, const char *str2, size_t count );
```

功能：将字符串 *from* 中至多 *count* 个字符连接到字符串 *to* 中，追加空值结束符。返回处理完成的字符串。

4.3.5 mem 系列操作函数

虽然放到字符数组这一节，但是我们的 `mem` 系列函数是任何数组都可以进行操作的，无论是字符数组，还是整型数组，浮点型数组，或者后面我们讲的结构体数组。当需要对一个数组进行全部置位零时，我们需要使用到 `memset`，接口如下：

```
void *memset( void *buffer, int ch, size_t count );
```

功能：函数拷贝 *ch* 到 *buffer* 从头开始的 *count* 个字符里，并返回 *buffer* 指针。`memset()` 可以应用在将一段内存初始化为某个值。例如：

```
memset( the_array, '\0', sizeof(the_array) );
```

这是将一个数组的所有元素设置成零的很便捷的方法。注意 `memset` 是将每一个字节都设置为 *ch*。

当我们需要把一个整型数组，或者浮点型数组的数据，或者某一部分元素的数据搬到另外一个数组时，这时我们不能使用 `strcpy`，需要使用 `memcpy`，接口如下：

```
void *memcpy( void *to, const void *from, size_t count );
```

功能：函数从 *from* 中复制 *count* 个字符到 *to* 中，并返回 *to* 指针。如果 *to* 和 *from*

重叠，则函数行为不确定。请看实例【例 4.3.5-1】

【例 4.3.5-1】memcpy 的使用

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

//strcpy不能用于整型数组, 浮点型数组
int main()
{
    int a[5]={1,2,3,4,5};
    int b[5];
    int i;
    memcpy(b, a, sizeof(a));
    for(i=0;i<5;i++)
    {
        printf("%3d",b[i]);
    }
    printf("\n");
    system("pause");
}
```

【例 4.3.5-1】运行结果可以得到 b 数组中内容打印与 a 一致，但是如果改为 strcpy 打印内容不对。

思考题：为什么上面 a 数组内容复制到 b 数组中，用 strcpy 不可以，最终打印结果又是多少呢？

当我们复制的内容发生重叠时，我们需要使用 memmove，不能使用 memcpy，接口如下：

void *memmove(void *to, const void *from, size_t count);

功能：与 memcpy 相同，不同的是当 to 和 from 重叠，函数正常仍能工作。

int memcmp(const void *buffer1, const void *buffer2, size_t count);

功能：函数比较 buffer1 和 buffer2 的前 count 个字符。返回值如表 4.3.5-1:

Value	解释
less than 0	buffer1 is less than buffer2
equal to 0	buffer1 is equal to buffer2
greater than 0	buffer1 is greater than buffer2

表 4.3.5-1

memcmp 可以比较任何类型数组，当然主要是比较两个数组是否相同，其内部原理实际是一个字节一个字节比较大小的，因此一般用于比较相等。count 是用来控制比较的字节数目。

4.3.6 sscanf 与 sprintf 使用

前面讲过 scanf 读取标准输入时，我们往控制台黑窗口输入的是字符串，然后 scanf 会将字符串中的整型数，浮点数等转换为对应的整型存入整型变量，浮点型存储到浮点型变量

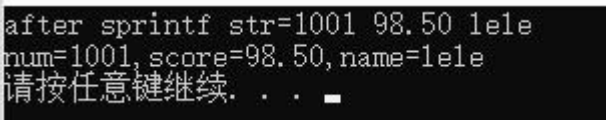
中，例如字符串“1001 98.50 lele”，如果我们想把 1001 存入整型变量 num，98.5 存入浮点型变量 score，lele 存储字符数组 name 中，我们就需要使用 sscanf，当我们需要将不同类型的变量拼接为一个字符串时，就需要使用 sprintf 实现，具体见【例 4.3.6-1】。

【例 4.3.6-1】sscanf 与 sprintf 的使用

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

int main(int argc, char *argv[])
{
    int num=1001;
    float score=98.5;
    char name[20]="lele";
    char str[1000];
    sprintf(str, "%d %5.2f %s", num, score, name); //将不同类型数据变为一个字符串
    printf("after sprintf str=%s\n", str);
    num=0, score=0;
    memset(name, 0, sizeof(name)); //设置 num, score, name 均为零
    sscanf(str, "%d%f%s", &num, &score, name); //将字符串格式化为不同的数据类型并放入对应变量的中
    printf("num=%d, score=%5.2f, name=%s\n", num, score, name);
    num=atoi("123"); //字符串转整型
    score=atof("92.4"); //字符串转浮点数
    system("pause");
}
```

【例 4.3.6-1】代码执行结果输出为：



sscanf 是帮忙处理字符串的利器，如果在考研机试，或者校招 OJ 过程中，可以通过 sscanf 进行字符串处理，同时例子中给出了两个接口，如果把字符串转为整型数，我们用 atoi 即可，如果转为浮点数，用 atof 即可。下面给出了 sscanf 与 sprintf 的参数形式，可以看出与 scanf 与 printf 的差别在多了 buffer。

```
#include <stdio.h>
int sscanf( const char *buffer, const char *format, ... );
int sprintf( char *buffer, const char *format, ... );
```