数组

一位数组

数组就是把相同数据类型的变量组合在一起而产生的数据集合。我们知道，每个变量在内存中都有对应的存放地址，而数组就是从某个地址开始连续若个位置形成的元素集合。

一维数组的定义格式如下：

```
数据类型 数组名[数组大小];
```

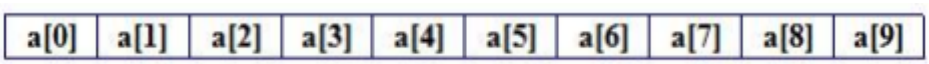
注意数组大小必须是正整数，不可以是变量。几种常见数据类型的一维数组定义举例：

```
int a[10];
double db[100];
char str[1000];
bool HashTable[10000];
```

这样，我们就可以把 int a[10]理解为定义了 10 个 int 型数据，且以下面的格式访问：

```
数组名称[下标];
```

还需要知道，在定义了长度为 `size` 的一维数组后，只能访问下标为 `0` 到 `size - 1` 之间的元素。例如定义 `int a[10]` 之后，我们允许正常访问的元素是 `a[0]`、`a[1]`、...、`a[9]`，而不允许访问 `a[10]`，在初学时要特别注意这点。



下面谈谈一维数组的初始化。一维数组的初始化，需要给出用逗号隔开的从第一个元素开始的若干个元素的初值，并用大括号括住。后面未被赋初值的元素将会由不同编译器内部实现的不同而被赋以不同的初值（可能是很大的随机数），而一般情况默认初值为 `0`。

举例如下：

```
#include <stdio.h>
int main() {
    int a[10] = {5, 3, 2, 6, 8, 4};
    for(int i = 0; i < 10; i++) {
        printf("a[%d] = %d  ", i, a[i]);
    }
    return 0;
}
```

输出结果：

```
a[0] = 5  a[1] = 3  a[2] = 2  a[3] = 6  a[4] = 8  a[5] = 4  a[6] = 0  a[7] = 0  a[8] = 0
 a[9] = 0
```

上面的程序对数组 `a` 的前六个元素进行了赋初值，而后面没有赋值的部分默认赋为 `0`。但是如果数组一开始没有赋初值的话，数组中的每个元素都可能会是一个随机数，并不一定默认为 `0`。因此，如果我们想要给整个数组都赋初值 `0` 的话，只需要把第一个元素赋为 `0`，或者只用一个大括号来表示

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

数组中每个元素都可以被赋值、被运算，可以被当成普通变量进行相同的操作。下面的程序实现了输入 `a[0]`，并将数组中后续元素都赋值为其前一个元素的两倍的功能。

```
#include <stdio.h>
int main() {
    int a[10];
    scanf("%d", &a[0]);
    for(int i = 1; i < 10; i++) {
        a[i] = a[i - 1] * 2;
    }
    for(int i = 0; i < 10; i++) {
        printf("a[%d] = %d  ", i, a[i]);
    }
    return 0;
}
```

当我们输入 `1` 时，输出结果：

```
a[0] = 1  a[1] = 2  a[2] = 4  a[3] = 8  a[4] = 16  a[5] = 32  a[6] = 64  a[7] = 128  a[8] =
256  a[9] = 512
```

冒泡排序

排序是指将一个无序序列按某个规则进行有序排列，而冒泡排序是排序算法中最基础的一种。在这里，我们给出一个序列 `a`，其中元素的个数为 `n`，要求将它们按从小到大的顺序排序。

冒泡排序的本质在于交换，即每次通过交换的方式把当前剩余元素的最大值移动到一端；而当剩余元素减少为 `0` 时，排序结束。为了使排序过程更加清楚，我们举一个例子。

现在有一个数组 `a`，其中有 `5` 个元素，分别为 `a[0] = 3`，`a[1] = 4`，`a[2] = 1`，`a[3] = 5`，`a[4] = 2`，要求把它们按从小到大的顺序排列。下面的过程中，每趟将最大数交换到最右边：

(1)第一趟

`a[0]` 与 `a[1]` 比较(3 与 4 比较)，`a[1]` 大，因此不动，此时序列为 `{3, 4, 1, 5, 2}`；

`a[1]` 与 `a[2]` 比较(4 与 1 比较)，`a[1]` 大，因此把 `a[1]` 和 `a[2]` 交换，此时序列为 `{3, 1, 4, 5, 2}`；

`a[2]` 与 `a[3]` 比较(4 与 5 比较)，`a[3]` 大，因此不动，此时序列为 `{3, 1, 4, 5, 2}`；

`a[3]` 与 `a[4]` 比较(5 与 2 比较)，`a[4]` 大，因此把 `a[3]` 和 `a[4]` 交换，此时序列为 `{3, 1, 4, 2, 5}`；

由此，第一趟结束，共进行了 4 次比较。

(2)第二趟

`a[0]` 与 `a[1]` 比较(3 与 1 比较)，`a[0]` 大，因此把 `a[0]` 和 `a[1]` 交换，此时序列为 `{1, 3, 4, 2, 5}`;

`a[1]` 与 `a[2]` 比较(3 与 4 比较)，`a[2]` 大，因此不动，此时序列为 `{1, 3, 4, 2, 5}`;

`a[2]` 与 `a[3]` 比较(4 与 2 比较)，`a[2]` 大，因此把 `a[2]` 和 `a[3]` 交换，此时序列为 `{1, 3, 2, 4, 5}`;

由此，第二趟结束，共进行了 3 次比较。

(3)第三趟

`a[0]` 与 `a[1]` 比较(1 与 3 比较)，`a[1]` 大，因此不动，此时序列为 `{1, 3, 2, 4, 5}`;

`a[1]` 与 `a[2]` 比较(3 与 2 比较)，`a[1]` 大，因此把 `a[1]` 和 `a[2]` 交换，此时序列为 `{1, 2, 3, 4, 5}`;

由此，第三趟结束，共进行了 2 次比较。

(4)第四趟

`a[0]` 与 `a[1]` 比较(1 与 2 比较)，`a[2]` 大，因此不动，此时序列为 `{1, 2, 3, 4, 5}`;

由此，第四趟结束，共进行了 1 次比较。

到此为止，已经无法再继续比较，序列已经有序，冒泡排序结束。

下面来看看如何实现冒泡排序。首先学习如何交换两个数。一般来说，交换两个数需要借助中间变量，即先定义中间变量 `temp` 存放其中一个数 `a`，然后再把另一个数 `b` 赋值给已被转移数据的 `a`，最后再把存有 `a` 的中间变量 `temp` 赋值给 `b`。这样 `a` 和 `b` 就完成了交换。下面这段代码就实现了这个功能：

```
#include <stdio.h>
int main() {
    int a = 1, b = 2;
    int temp = a;
    a = b;
    b = temp;
    printf("a = %d, b = %d\n", a, b);
    return 0;
}
```

输出结果：

```
a=2, b=1
```

然后来实现冒泡排序。从上面的例子中我们可以发现，整个过程执行 `n - 1` 趟，每一趟从左到右依次比较相邻的两个数，如果大的数在左边，则交换这两个数，当该趟结束时，该趟最大数被移动到当前剩余数的最右边。具体实现如下：

```
#include <stdio.h>
int main() {
    int a[10] = {3, 1, 4, 5, 2};
    for(int i = 1; i <= 4; i++) {   //进行 n - 1 趟
    //第 i 趟时从 a[0]到 a[n - i]都与它们下一个数比较
```

```
        for(int j = 0; j < 5 - i; j++) {
            if(a[j] > a[j + 1]) {   //如果左边的数更大，则交换 a[j]和 a[j + 1]
                int temp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = temp;
            }
        }
    }
    for(int i = 0; i < 5; i++) {
        printf("%d ", a[i]);
    }
    return 0;
}
```

输出结果：

```
1 2 3 4 5
```

第二个 `for` 循环的理解：从前面的例子可以看出，第一趟从 `a[0]` 到 `a[3]` 都需要与下一个数比较，第二趟从 `a[0]` 到 `a[2]` 都需要与下一个数比较，第三趟从 `a[0]` 到 `a[1]` 都需要与下一个数比较，第四趟只有 `a[0]` 需要与下一个数比较。因此很容易找到规律，即当第 `i` 趟时，从 `a[0]` 到 `a[n - i]` 都需要与下一个数比较。

二维数组

二维数组其实就是一维数组的扩展，定义格式如下：

```
数据类型 数组名[第一维大小][第二维大小];
```

下面是常见的数据类型的二维数组定义：

```
int a[5][6];
double db[10][10];
char [256][256];
bool vie[1000][1000];
```

二维数组中元素的访问和一维数组类似，只需要给出第一维和第二维的下标：

```
数组名[下标1][下标2];
```

需要注意的是，对定义为 `int a[size1][size2]` 的二维数组，其第一维的下标取值只能是 `0 ~ size1 - 1`，其第二维的下标取值只能是 `0 ~ size2 - 1`。

怎么理解二维数组？其实我们可以把二维数组当成是一维数组的每一个元素都是一个一维数组，例如定义为 `int a[5][6]` 的二维数组就可以看成是 5 个长度为 6 的一维数组。

a[0]	a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]	a[0][5]
a[1]	a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]	a[1][5]
a[2]	a[2][0]	a[2][1]	a[2][2]	a[2][3]	a[2][4]	a[2][5]
a[3]	a[3][0]	a[3][1]	a[3][2]	a[3][3]	a[3][4]	a[3][5]
a[4]	a[4][0]	a[4][1]	a[4][2]	a[4][3]	a[4][4]	a[4][5]

和一维数组一样，二维数组也可以再定义的时候进行初始化。二维数组在初始化的时候，需要按第一维的顺序依次用大括号给出第二维初始化情况，然后将它们用逗号分隔，并用大括号全部括住，而在这些被赋初值的元素之外的部分将被默认赋值为 `0`。举一个例子会比较容易理解：

```
#include <stdio.h>
int main() {
    int a[5][6] = {{3, 1, 2}, {8, 4}, {}, {1, 2, 3, 4, 5}};
    for(int i = 0; i < 5; i++) {
        for(int j = 0; j < 6; j++) {
            printf("%d", a[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

输出结果：

```
3 1 2 0 0 0
8 4 0 0 0 0
0 0 0 0 0 0
1 2 3 4 5 0
0 0 0 0 0 0
```

可以看到，第 1、2、4 行都都赋予了初值，第 3 行使用大括号 `{}` 跳过了（并且如果不写大括号是无法通过编译的）。剩下的部分均被默认赋为 0。

接下来，我们实现一个程序，用以实现将两个二维数组对应位置的元素相加，并存放到另一个二维数组中：

```
#include <stdio.h>
int main() {
    int a[3][3], b[3][3];
    for(int i = 0; i < 3; i++){
        for(int j = 0; j < 3; j++){
            scanf("%d", &a[i][j]);  //输入二维数组 a 的元素
        }
    }
    for(int i = 0; i < 3; i++) {
        for(int j = 0; j < 3; j++){
            scanf("%d", &b[i][j]);  //输入二维数组 b 的元素
        }
    }
    int c[3][3];
```

```
    for(int i = 0; i < 3; i++) {
        for(int j = 0; j < 3; j++){
            c[i][j] = a[i][j] + b[i][j];    //对应位置元素相加并放到二维数组 c 中
        }
    }
    for(int i = 0; i < 3; i++) {
        for(int j = 0; j < 3; j++){
            printf("%d ", c[i][j]); //输出二维数组 c 的元素
        }
        printf("\n");
    }
    return 0;
}
```

输入如下两个 3*3 的矩阵：

```
1 2 3

4 5 6

7 8 9

1 4 7

2 5 8

3 6 9
```

输出结果：

```
2 6 10
6 10 14
10 14 18
```

特别提醒：如果数组大小较大（大概10^6级别）的话，需要把它定义在主函数外面，否则会使程序异常退出，原因暂时不必掌握。例如下面的代码就把10^6大小的数组定义在了主函数外面。

```
#include <stdio.h>
int a[1000000];
    int main(){
        for(int i = 0; i < 1000000; i++) {
            a[i] = i;
        }
    return 0;
}
```

最后再提一下多维数组，即维度高于二维的数组。多维数组跟二维数组类似，只是把维度增加了若干维，使用方法和二维数组基本无二。例如下面这段代码就定义了一个三维数组并在输入的基础上每个元素都增加了 `1`：

```
#include <stdio.h>
int main(){
```

```
    int a[3][3][3];
    for(int i = 0; i < 3; i++) {
        for(int j = 0; j < 3; j++){
            for(int k = 0; k < 3; k++){
                scanf("%d", &a[i][j][k]);   //输入三维数组 a 的元素
                a[i][j][k]++;   //自增
            }
        }
    }
    for(int i = 0; i < 3; i++){
        for(int j = 0; j < 3; j++){
            for(int k = 0; k < 3; k++){
                printf("%d\n", a[i][j][k]); //输出三维数组 a 的元素
            }
        }
    }
    return 0;
}
```

memset----对数组中每一个元素赋相同的值

如果需要对数组中每一个元素赋以相同的值，例如对数组初始化为 `0`、或是其他的一些数，就可能要使用相关的函数。`memset` 函数的格式：

```
memset(数组名，值，sizeof(数组名));
```

不过也要记住，使用`memset` 需要在程序开头添加 `string.h` 头文件，且只建议初学者使用 `memset` 赋 `0` 或 `-1`。这是因为 `memset` 使用的是按字节赋值，`0` 的二进制补码为全 `0`、`-1` 的二进制补码为全 `1`，不容易弄错。如果要对数组赋其他的数字（例如 1），那么请使用 `fill` 函数（但 `memset` 的执行速度快）。实例如下：

```
#include <stdio.h>
#include <string.h>
int main(){
    int a[5] = {1, 2, 3, 4, 5};
    //赋初值 0
    memset(a, 0, sizeof(a));
    for(int i = 0; i < 5; i++){
        printf("%d ", a[i]);
    }
    printf("\n");
    //赋初值-1
    memset(a, -1, sizeof(a));
    for(int i = 0; i < 5; i++){
        printf("%d ", a[i]);
    }
    printf("\n");
    return 0;
}
```

输出结果：

```
0 0 0 0 -1 -1 -1 -1 -1
```

学弟妹不妨把 `memset` 里面的 `-1` 改为 `1`，看看结果会有什么不同。另外，对二维数组或多维数组的赋值方法也是一样的（仍然只需要写数组名），不需要改变任何东西。

字符数组

字符数组的初始化

和普通数组一样，字符数组也可以初始化，方法相同，下面给一个简单的例子说明：

```
#include <stdio.h>
int main(){
    char str[15] = {'G', 'o', 'o', 'd', ' ', 's', 't', 'o', 'r', 'y', '!'};
    for(int i = 0; i < 11; i++){
        printf("%c", str[i]);
    }
    return 0;
}
```

输出结果：

```
Good Story!
```

除此之外，字符数组也可以通过直接复制字符串来初始化（仅限于初始化，程序其他位置不允许这样直接赋值整个字符串），这样比普通数组更为方便，如下所示。

```
#include <stdio.h>
int main(){
    char str[15] = "Good Story!";
    for(int i = 0; i < 11; i++) {
        printf("%c", str[i]);
    }
    return 0;
}
```

输出结果：

```
Good Story!
```

字符数组的输入输出

字符数组就是 `char` 数组，当维度是一维时可以当作“字符串”。当维度是二维时可以当成是字符串数组，即若干字符串。字符数组的输入除了使用 `scanf` 外，还可以用 `getchar` 或者 `gets`；其输出除了使用 `printf` 外，还可以用 `putchar` 或者 `puts`。下面对上面的这几种方式分别介绍：

(1) `scanf` 输入，`printf` 输出

`scanf` 对字符类型有`%c` 和`%s` 两种格式（`printf` 同理，下同），其中`%c` 用来输入单个字符，`%s` 用来输入一个字符串并存在字符数组里。`%c` 格式能够识别空格跟换行并将其输入，而`%s` 通过空格或换行来识别一个字符串的结束。下面是一个实例：

```
#include <stdio.h>
int main(){
    char str[10]; scanf("%s", str);
    printf("%s", str);
    return 0;
}
```

输入下面这个字符串：

```
TAT TAT TAT
```

输出结果：

```
TAT
```

可以看见，`%s` 识别空格作为字符串的结尾，因此三个 `TAT`的后两个不会被读入。另外， `scanf` 在使用`%s`的时候，后面对应数组名前面是不需要加&取地址运算符的。

(2) `getchar` 输入，`putchar` 输出

`getchar` 和 `putchar` 分别用来输入和输出单个字符，这点在之前已经说过，这里简单举一个二维字符数组的例子。

```
#include <stdio.h>
int main(){
    char str[5][5];
    for(int i = 0; i <3; i++) {
        for(int j = 0; j < 3; j++) {
            str[i][j] = getchar();
        }
        getchar();  //这句是为了把输入中每行的换行吸收掉
    }
    for(int i = 0; i < 3; i++) {
        for(int j = 0; j < 3; j++) {
            putchar(str[i][j]);
        }
        putchar('\n');
    }
    return 0;
}
```

输入下面的字符矩阵：

```
^_^ _^_ ^_^
```

输出的结果和输入相同。

(3) `gets` 输入，`puts` 输出

`gets` 用来输入一行字符串，并将其存放于一维数组（或二维数组的一维）中；`puts` 用来输出一行字符串，即将一维数组（或二维数组的一维）在屏幕上输出，并紧跟一个换行。下面给出实例：

```
#include <stdio.h>
int main(){
    char str1[20];
    char str2[5][10];
    gets(str1);
    for(int i = 0; i < 3; i++){
        gets(str2[i]);
    }
    puts(str1);
    for(int i = 0;i < 3; i++) {
        puts(str2[i]);
    }
    return 0;
}
```

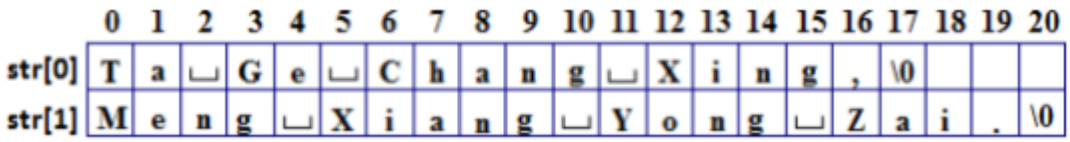
输入下面四个字符串：

```
Hello World!
QAQ
T_T
Helo!
```

这段代码中，我们通过 `gets(str1)` 将第一个字符串存入字符数组 `str1` 中，然后通过 `for` 循环将三个字符串分别存于 `str2[0]`、`str2[1]`、`str2[2]`。之后使用 `puts` 来将这些字符串原样输出。

字符数据的存放方式

由于字符数组是由若干个 `char` 类型的元素组成，因此字符数组的每一位都是一个 `char` 字符。除此之外，在一维字符数组（或是二维字符数组的第二维）的末尾都有一个空字符 `\0`，以表示存放的字符串的结尾。空字符 `\0` 在使用 `gets` 或 `scanf` 输入字符串的时候会自动添加在输入的字符串后面，并占用一个字符位，而 `puts` 与 `printf` 的输出就是识别 `\0` 作为字符串的结尾。我们以二维字符数组存储两个字符串 `"Ta Ge Chang Xing,"`、`"Meng Xiang Yong Zai."` 来说明。



特别提醒 1：结束符 `\0` 的 `ASCII` 码为 0，占用一个字符位，因此开字符数组的时候千万要记得字符数组的长度一定要比实际存储字符串的长度至少多 1。

特别提醒 2：如果不是使用 `scanf` 函数的 `%s` 格式或 `gets` 函数输入字符串（例如使用 `getchar`），请一定要在输入的每个字符串后加入 `'\0'`，不然 `printf` 和 `puts` 输出字符串会无法识别字符串末尾而输出一大堆乱码，例如下面这个程序：

```c
#include <stdio.h>
int main(){
    char str[15];
    for(int i = 0; i < 3; i++) {
        str[i] = getchar();
    }
    puts(str);
    return 0;
}
```

输入字符串：

```
T^T
```

输出结果会是类似下面这样的（T^T 后面全是乱码）：

```
T^T 腫?w?@
```

string.h 头文件

`string.h` 头文件中包含了许多用于字符数组的函数，使用这些函数需要在程序开头添加 `string.h` 头文件。

strlen()

`strlen` 函数可以得到字符数组中字符的个数，格式如下：

```
strlen(字符数组);
```

实例如下：

```c
#include <stdio.h>
#include <string.h>
int main(){
    char str[10];
    gets(str);
    int len = strlen(str);
    printf("%d\n", len);
    return 0;
}
```

输入字符串：

```
member
```

输出结果：

```
6
```

strcmp()

`strcmp` 函数返回两个字符串大小的比较结果，比较原则是按字典序，格式如下：

```
strcmp(字符数组1, 字符数组2);
```

所谓字典序就是字符串在字典中的顺序，因此如果有两个字符数组`str1`和`str2`，且满足`str1[0...k-1] == str2[0...k-1]`、`str1[k] < str2[k]`，那么我们就说`str1`的字典序小于`str2`。例如`"a"`的字典序小于`"b"`、`"aaaa"`的字典序小于`"aab"`。`strcmp`的不同返回结果如下。

如果`字符数组 1 < 字符数组 2`，则返回一个负整数（不同编译器处理不同，不一定是-1）；如果`字符数组 1 == 字符数组 2`，则返回 0；如果`字符数组 1 > 字符数组 2`，则返回一个正整数（不同编译器处理不同，不一定是+1）。实例如下：

```
#include <stdio.h>
#include <string.h>
int main(){
    char str1[10], str2[10];
    gets(str1);
    gets(str2);
    int cmp = strcmp(str1, str2);
    if(cmp < 0)
        printf("str1 < str2\n");
    else if(cmp > 0)
        printf("str1 > str2\n");
    else
        printf("str1 == str2\n");
    return 0;
}
```

输入字符串：

```
Alice
Bob
```

输出结果：

```
str1 > str2
```

strcpy()

`strcpy` 函数可以把一个字符串复制给另一个字符串，格式如下：

```
strcpy(字符数组 1, 字符数组 2);
```

注意是把字符数组 2 复制给字符数组 1，这里的“复制”包括了结束符`\0`。实例如下：

```c
#include <stdio.h>
#include <string.h>
int main(){
    char str1[15], str2[15];
    gets(str1);
    gets(str2);
    strcpy(str1, str2);
    puts(str1);
    return 0;
}
```

输入字符串：

```
Rage Your Dream
You will success
```

输出结果：

```
You will success
```

strcat()

`strcat()` 可以把一个字符串接到另一个字符串后面，格式如下：

```
strcat(字符数组 1, 字符数组 2);
```

注意是把字符数组 2 接到字符数组 1 后面，实例如下：

```c
#include <stdio.h>
#include <string.h>
int main(){
    char str1[15], str2[15];
    gets(str1);
    gets(str2);
    strcat(str1, str2);
    puts(str1);
    return 0;
}
```

输入字符串：

```
Alice
Bob
```

输出结果：

```
AliceBob
```

sscanf 与 sprintf

`sscanf` 与 `sprintf` 是处理字符串问题的利器，我们很有必要学会它们（`sscanf` 可以理解为 `string + scanf`，`sprintf` 可以理解为 `string + printf`，均在 `stdio.h` 头文件下）。先来回顾一下 `scanf` 与 `printf`，如果想要从屏幕输入 `int` 型变量 `n`、将 `int` 型变量 `n` 输出到屏幕的话，写法是下面这样的：

```
scanf("%d", &n);
printf("%d", n);
```

事实上，上面的写法其实可以表示成下面的样子，其中 `screen` 表示屏幕：

```
scanf(screen, "%d", &n);
printf(screen, "%d", n);
```

可以发现，`scanf` 的输入其实是把 `screen` 的内容以 `"%d"` 的格式传输到 `n` 中（即从左至右），而 `printf` 的输出则是把 `n` 以 `"%d"` 的格式传输到 `screen` 上（即从右至左）。

我们要介绍的 `sscanf` 与 `sprintf` 与上面的格式是相同的，只不过把 `screen` 换成了字符数组（假设我们定义了一个 `char` 数组 `str[100]`），如下所示：

```
sscanf(str, "%d", &n);
sprintf(str, "%d", n);
```

上面 `sscanf` 写法的作用是把字符数组 `str` 中的内容以 `"%d"` 的格式写到 `n` 中（还是从左至右），实例如下：

```
#include <stdio.h>
int main() {
    int n;
    char str[100] = "123";
    sscanf(str, "%d", &n);
    printf("%d\n", n);
    return 0;
}
```

输出结果：

```
123
```

而 `sprintf` 写法的作用是把 `n` 以 `"%d"` 的格式写到 `str` 字符数组中（还是从右至左），实例如下：

```
#include <stdio.h>
int main() {
    int n = 233;
    char str[100];
    sprintf(str, "%d", n);
    printf("%s\n", str);
    return 0;
}
```

输出结果：

```
233
```

上面只是一些简单的应用，事实上我们可以像使用 `scanf` 与 `printf` 那样进行复杂的格式输入和输出。例如下面的代码使用 `sscanf` 将字符数组 `str` 中的内容按 `"%d:%lf,%s"` 的格式写到 `int` 型变量 `n`、`double` 型变量 `db`、`char` 型数组 `str2` 中：

```
#include <stdio.h>
int main() {
    int n;
    double db;
    char str[100] = "2048:3.14,hello", str2[100];
    sscanf(str, "%d:%lf,%s", &n, &db, str2);
    printf("n = %d, db = %.2f, str2 = %s\n", n, db, str2);
    return 0;
}
```

输出结果：

```
n = 2048, db = 3.14, str2 = hello
```

类似地，下面的代码使用 `sprintf` 将 `int` 型变量 `n`、`double` 型变量 `db`、`char` 型数组 `str2` 按 `"%d:%.2f,%s"` 的格式写到字符数组 `str` 中：

```
#include <stdio.h>
int main() {
    int n = 12;
    double db = 3.1415;
    char str[100], str2[100] = "good";
    sprintf(str, "%d:%.2f,%s", n, db, str2);
    printf("str = %s\n", str);
    return 0;
}
```

输出结果：

```
str=12:3.14,good
```

最后指出，`sscanf` 还支持正则表达式，如果配合正则表达式来进行字符串的处理的话，很多字符串的题目都将迎刃而解。不过正则表达式不是本书想要讨论的内容，因此不作深入探讨，有兴趣的学弟妹可以自己去了解。