

# 第六讲

# 数组 (array)

文件	开始	插入	页面布局	公式	数据	审阅
I16						
	A	B	C	D	E	
1	姓名	学号	数学	物理		
2	张小三	20200001	85	88		
3	李小四	20200002	88	87		
4	王小五	20200003	96	92		
5	赵小六	20200004	91	90		
6						
7						
8						
9						
10						
11						
12						
13						
14						
15						

AC 编程

主页

小组

题目

赛事

E4-2020级程序设计基础训练第四次练习

比赛排名

更新中, 上次更新于 2020-11-03 20:14:20

简介

题目

排名

我的提交

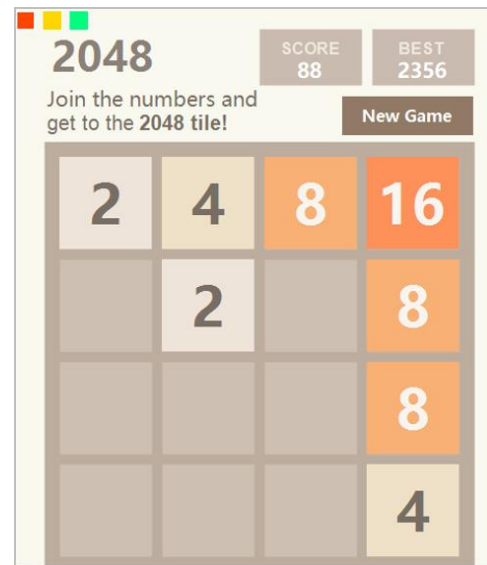
提问&&公告

服务器当前时间  
2020-11-03 20:14:46

比赛结束时间  
2020-11-07 12:00:00

比赛剩余时间  
87:45:13

排名	用户	得分	罚时	A 475/532	B 375/441	C 159/260
1		1000	32:00:22	0:35:06(+1)	0:58:16	4:12:59(+2)
2		1000	32:05:03	0:39:43	0:52:38(+1)	3:50:38(+2)
3		1000	32:17:25	0:22:45	0:46:42(+1)	2:22:27(+4)
4		1000	35:12:18	2:06:42	2:11:37	2:28:07(+1)
5		1000	45:05:47	0:55:31	1:28:28(+2)	3:48:57(+2)
6		1000	53:26:50	24:01:39(+1)	0:07:15	4:18:10(+4)
7		1000	59:59:14	4:22:50	4:28:43	4:38:29(+1)
8		1000	60:34:12	0:51:45	1:05:36	1:27:04(+3)
9		1000	62:18:17	0:06:01	2:42:38(+1)	3:31:40(+1)



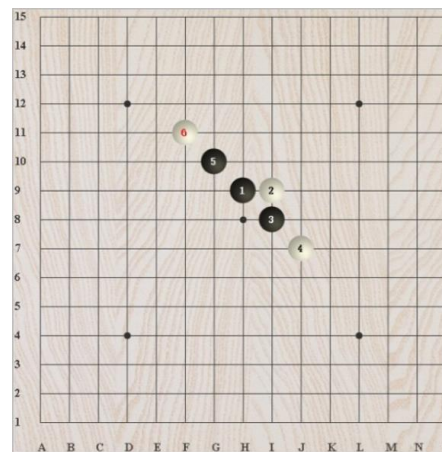
# 第六讲 数组

## 学习要点

1. 数组的结构、存储方式
2. 一维数组定义、初始化、访问
3. sizeof 的用法
4. 数组作为函数参数
5. 一些基本的算法设计：基本的查找和排序方法等
6. 字符串与字符数组的关系
7. 标准库字符串处理函数
8. 多维数组简介
9. 基于数组的简单数据结构（队、栈、散列表）\*

有哪些常见错误？

	6		4			9		
4		5			1			
	1			7				6
		4			8		3	
2				9				4
	7		6			2		
8				2			4	
			5			6		1
		6			7		8	



## 6.0 数组使用中的常见错误

# (1) 数组越界访问

数组越界访问，是比较隐蔽、很容易犯的一个错误。

```
int a[5] = {20, 21};  
int b[5] = {1, 2, 3, 4, 5, 6};
```

✗

```
for (int i = 0; i < 5; i++)  
    printf("%d, ", a[i]);
```

```
printf("\n");
```

```
for (int i = 0; i < 6; i++)  
    printf("%d, ", b[i]);
```

在定义数组a和b的同时进行初始化

- 定义数组a[5]，初始化前两个数组元素，其余元素就隐式初始化为0
- 定义数组b[5]，注意：初始化时元素个数多于数组长度（编译器忽略多余元素）

a[0]	a[1]	a[2]	a[3]	a[4]
20	21	0	0	0

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

输出结果：

20, 21, 0, 0, 0,  
1, 2, 3, 4, 5, 20,

?

# (1) 数组越界访问

数组是连续存储的一组同类型变量，这些变量统一以数组名+下标的形式访问。

```
int a[5] = {20, 21};  
int b[5] = {1, 2, 3, 4, 5, 6};
```

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

```
for (int i = 0; i < 5; i++)  
    printf("%d, ", a[i]);
```

```
printf("\n");
```

```
for (int i = 0; i < 6; i++)  
    printf("%d, ", b[i]);
```

数组越界访问可能导致严重问题

数组在内存中的可能存放方式示意（每个方格占4个字节）

b[0]	..	..	..	b[4]	a[0]	..	..	..	a[4]
1	2	3	4	5	20	21	0	0	0

输出结果：

20, 21, 0, 0, 0,  
1, 2, 3, 4, 5, 20,

b[5]

**b[5]是a[0]的地盘！**

**跑到别人家的地盘，  
偷了别人家的果实！**

**私闯民宅，错！**

# 数组越界访问：一个真实的例子

## G 多项式相加

时间限制：1000ms 内存限制：65536kb

通过率：118/402 (29.35%) 正确率：118/1356 (8.70%)

### 题目描述

一元多项式的定义如下：

- 设  $c_0, c_1, \dots, c_n$  都是数域  $F$  中的数， $n$  是非负整数，那么表达式

$$c_n \times x^n + c_{n-1} \times x^{n-1} + \dots + c_2 \times x^2 + c_1 \times x + c_0$$

就是数域  $F$  上关于变量  $x$  的多项式或一元多项式。

- 其中， $c_k \times x^k$  ( $1 \leq k \leq n$ ) 代表该一元多项式中的一个项，非零数  $c_k$  是该项的系数， $k$  是该项的指数。

现在给定两个整数数域上关于变量  $x$  的一元多项式  $f(x)$  和  $g(x)$ ，请你求出二者相加后产生的一元多项式  $f(x) + g(x)$ ，并要求不再输出系数为 0 的项。

### 输入格式

第一行两个整数  $N, M$  ( $1 \leq N, M \leq 100000$ )，分别代表  $f(x)$  和  $g(x)$  的项数。

第二行  $2 \times N$  个整数，第  $2 \times i - 1$  和  $2 \times i$  个整数分别代表  $f(x)$  中第  $i$  项的系数  $a_i$  和指数  $s_i$ ， $a_i$  和  $s_i$  在 `int` 范围内。

**多项式的项数  $n$  少 ( $\leq 100000$ )，  
多项式的指数  $s$  (次数) 大 (int范围内)**

```
int af[100020] = {};  
int ag[100020] = {};  
int main()  
{
```

af, 保存多项式 f 的系数：  
 $s$  次项的系数  $a$  保存到多项式的第  $s$  项  $af[s]$  中

```
    int n, m, i;  
    int s=0;  
    long long a=0;  
    scanf("%d %d", &n, &m);  
    for (i=0; i<n; i++)  
    {  
        scanf("%lld %d", &a, &s);  
        af[s] = a;  
    }
```

多项式的  $s$  次项 的系数存储到数组  $af$  的第  $s$  项中，混淆了多项式指数  $s$  和项数  $n$  的概念，把第  $i$  次项等同于了第  $i$  项。若多项式次数比较小，本代码正确。成于斯毁于斯。

系数

指数

$s$  次项的系数  $a$  保存在数组  $af$  的第  $s$  项中

**指数  $s$  是 int 范围，输入的  $s$  可能让数组  $af$  越界了！  
如：  $f(x) = a_s x^s + a_0 = 6x^{300000} + 5$ ，只有两项，但有 30 万次。**

## “多项式相加”数组越界原因的进一步分析（课后阅读）

多项式通项： $c_k x^k + c_{k-1} x^{k-1} + \dots + c_1 x + c_0$ ，表示为系数形式  $(c_k, c_{k-1}, \dots, c_1, c_0)$

对多项式  $f(x)$ ，定义一个数组  $a[N]$  来表示其系数， $a[k]$  表示其  $k$  次项系数，则

$$f(x) : (a[k], a[k-1], \dots, a[1], a[0])$$

类似地， $g(x) : (b[k], b[k-1], \dots, b[1], b[0])$

因此， $f(x) + g(x) : (a[k] + b[k], a[k-1] + b[k-1], \dots, a[1] + b[1], a[0] + b[0])$

如果多项式的系数  $(c_k, c_{k-1}, \dots, c_1, c_0)$  都不为 0，多项式的次数就是多项式的项数；

如果最高次项的系数  $c_k$  不为 0，但有多项系数为 0，则多项式的项数少，次数高。

如： $f(x) = a_k x^k + a_0 = 6 x^{300000} + 5$ ，只有两项，但有 30 万次，表示系数的数组为  $(a[300000], a[300000-1], \dots, a[1], a[0]) = (6, 0, 0, \dots, 5)$ ，数组有 30 万项（但多项式只有两项），数组越界（因为定义的数组是 10 万项）！

把数组定义得再大些行不行？不可取，浪费存储空间！系数为 0 的次项也去比较，浪费时间！

# 数组越界带给我们的启示



德国Füssen小镇火车站



启示1：学编程，提醒我们要遵守规则！

没有检票口，马路上走过来就直接上火车！

德国人买不起检票闸机吗？

当然不是！不设检票口是为了进站高效！

但是，不购票上车被查到的后果很严重！（不良信用记录）

数组不进行越界检查，是因为C语言能力不行吗？不是！

C语言没有越界检查也是为了高效！

同样，数组越界，后果也很严重！

C语言允许做“任何事”，但你需要为自己的行为负责！

—— If you do not, bad things happen!



# 防止数组越界的有用一招

- 读取：对  $n$  个元素的数组，下标范围从  $0 \sim (n-1)$
- 长度：用循环进行数组遍历时，一般不用常数，而用带参数的宏，  
**#define ArrayNum(x) (sizeof(x)/sizeof(x[0]))**

```
#define ArrayNum(x) (sizeof(x)/sizeof(x[0]))
int main()
{
    float f[10];
    // for (int i = 0; i < 10; i++ )
    for (int i=0; i < ArrayNum(f); i++ )
    {
        f[i] = i*i;
        printf("%f\n", f[i]);
    }
    // 请观察如下输出，进一步理解sizeof
    printf("%d, %d\n", sizeof(i), sizeof(int));
    printf("%d, %d\n", sizeof(f), sizeof(f[0]));

    return 0;
}
```

一般不这样用

常用带参数的宏

sizeof(para): sizeof是一个运算符，计算参数para所占的字节数，参数可以是变量、数组、类型名称。



**注意：**数组的越界访问可能造成数据篡改或带来运行时错误。非常隐蔽，问题严重，一定避免！

## (2) 数组直接整体处理

### 数组复制

- 方法一：通过循环逐一复制数组中元素
- 方法二：通过内置函数memcpy( )实现整体复制

```
int a[5] = {1, 2, 3, 4};  
int b[5], i;  
b = a; // 错误  
b[5] = {1, 2, 3, 4}; // 错误  
for (i=0; i<5; i++)  
    b[i] = a[i]; // 正确  
memcpy(b, a, sizeof(a)); // 正确
```

```
char s[15] = "1234567890";  
memset(s, 'A', 6);  
printf("%s", s);
```

输出

AAAAAA7890


- `b = a`, 语法错误, 不能把数组整体赋值给另一个数组。
- `b[5] = {1, 2, 3, 4}`, 语法错误, 数组除定义时初始化外, 不能用 {数值列表} 进行整体赋值。
- 两个数组赋值需要通过循环逐一赋值数组元素。

`void *memcpy(void *dest, void *src, size_t count);`  
将src中的count个字节拷贝到dest, 内存拷贝, 效率高!

`void *memset(void *s, int ch, size_t n);`  
将s中当前位置后面的n个字节用 ch 替换并返回 s, 常用于清零等。

### (3) 用变量定义数组大小

```
int n;  
scanf("%d", &n);  
double s[n];  
double x[ ];
```



长度必须是常量或常量表达式，不能是变量。  
也不能定义长度为空的数组。

用变量定义数组长度，可能有时正确。不同的编译器由于版本不同，有很多扩展功能，可能造成跟C标准并不完全一致。

注意：C语言（C89标准）不支持动态数组，即数组的长度必须在编译时确定下来，而不是在运行中根据需要临时决定。但C语言提供了动态分配存储函数，利用它可实现动态申请空间[\*]

1) 在 ISO/IEC9899 标准的 6.7.5.2 Array declarators 中明确说明了数组的长度可以为变量的，称为变长数组（VLA，variable length array）。（注：这里的变长指的是数组的长度是在运行时才能决定，但一旦决定，在数组的生命周期内就不会再变。） 2) 在 GCC 标准规范的 6.19 Arrays of Variable Length 中指出，作为编译器扩展，GCC 在 C90 模式和 C++ 编译器下遵守 ISO C99 关于变长数组的规范。

\*\* C89是美国标准，之后被国际化组织认定为标准C90  
除了标准文档在印刷编排上的某些细节不同外，ISO C(C90) 和 ANSI C(C89) 在技术上完全一样

先定义常量，以常量作为数组长度，这种用法比较常见。

```
#include <stdio.h>  
#define LENGTH 10  
  
int main()  
{  
    double s[LENGTH];  
    .....
```

## (4) 数组定义时的大小问题

- 实际问题中的数据可能很大，如电商数据几亿用户M，几千万商品N，数组是否应定义为a[M][N]？
- 数组大小多大合适？取决于计算机的能力、算法设计、实际需要。
- 通常，**全局数组**可以比较大（比如几 MB），**局部数组**比较小（通常几十 KB）。
- 内存是宝贵的计算资源，应合理规划。

```
double globalArray[1 << 20];
int main()
{
    int localArray[1 << 10];
    ...
}
```

**\*\* 文库：c语言中的全局数组和局部数组：**今天在A一道题目的时候发现一个小问题，在main函数里面开一个 int[1000000] 的数组会提示stack overflow，但是将数组移到main函数外面，变为全局数组的时候则ok，就感到很迷惑，然后上网查了些资料，才得以理解。对于全局变量和局部变量，这两种变量存储的位置不一样。对于全局变量，是存储在内存中的静态区（static），而局部变量，则是存储在栈区（stack）。这里，顺便普及一下程序的内存分配知识：

C语言程序占用的内存分为几个部分：

1. 堆区（heap）：由程序员分配和释放，比如malloc函数
2. 栈区（stack）：由编译器自动分配和释放，一般用来存放局部变量、函数参数
3. 静态区（static）：用于存储全局变量和静态变量
4. 代码区：用来存放函数体的二进制代码

在C语言中，一个静态数组能开多大，决定于剩余内存的空间，在语法上没有规定。所以，能开多大的数组，就决定于它所在区的大小了。在WINDOWS下，栈区的大小为2M，也就是2\*1024\*1024=2097152字节，一个int占2个或4个字节，那么可想而知，在栈区中开一个int[1000000]的数组是肯定会overflow的。我尝试在栈区开一个2000 000/4=500 000的int数组，仍然显示overflow，说明栈区的可用空间还是相对小。所以在栈区（程序的局部变量），最好不要声明超过int[200000]的内存的变量。

而在静态区（可以肯定比栈区大），用vs2010编译器试验，可以开2^32字节这么大的空间，所以开int[1000000]没有问题。

总而言之，当需要声明一个超过十万级的变量时，最好放在main函数外面，作为全局变量。否则，很有可能overflow。

字太小！  
课后读物！

## 6.1 数组作为函数参数

## 6.1 数组作为函数参数

### 【例6-1】计算两个n维向量的点积

$$a = (a_1, a_2, \dots, a_n)$$

$$b = (b_1, b_2, \dots, b_n)$$

$$d = \sum_{i=1}^n a_i b_i$$

```
#include <stdio.h>
#define LEN 5
double dot_vec(double [], double [], int);
//double dot_vec(double a[], double b[], int n);

int main()
{
    double a[LEN] = {1, 2, 3, 4, 5}, b[LEN];
    int i;
    for (i = 0; i < LEN; i++)
    {
        scanf("%lf", &b[i]);
    }
    printf("dot_vec: %f\n", dot_vec(a, b, LEN));
    return 0;
}
```

- 函数原型中的方括号，表示这个参数要求接受数组。
- 函数定义中，形参为数组不写长度值。
- 调用函数时，实参直接使用数组名（“不能”包括数组长度，如，不能写成 dot\_vec(a[5], b[5],...)。如果把参数写成a[5]这样，传递的就是数组元素（普通变量）。
- 数组传递时，数组名作为实参，仍然是值传递，它将数组的首地址传递给形参，即把 a 的值 (&a[0]) 传给 va，对 va 的访问，是从 a 的地址开始访问。

```
double dot_vec(double va[], double vb[], int n)
{
    double s=0; int i;
    for(i=0; i<n; i++)
        s += va[i]*vb[i]; // 第 i 项相乘并累加
    return s;
}
```

# 数组作为函数参数


## 数组作为函数参数：

- 要掌握其用法（调用与定义）
- 理解其含义（传递地址）


```
void f(int [], int);

int main()
{
    int a[10];
    ...
    f(a, ArrayNum(a));
    ...
}

void f(int b[], int size)
{
    ...
}
```



内存中的数组存放示意							
	.....	6422176 b		.....			
				6422232 a			



数组作为参数，通常也需要额外定义一个形参，来传递数组长度。

\*说明：把数组作为参数传递给函数，实际上只有数组的首地址作为指针传递给了函数中的形参。编译器无法获知数组长度，因此，需要额外定义形参来传递数组长度。数组作为函数参数，更多的原理将在下一讲（指针）进行介绍。

## 6.2 排序与查找



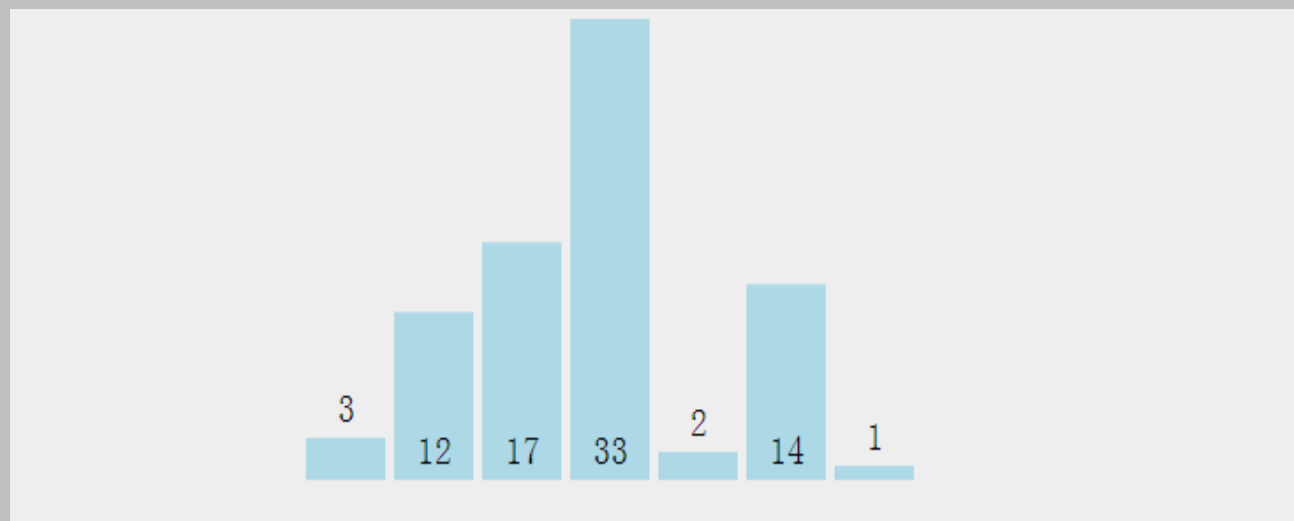
## 6.2 排序和查找

排序 (sort)

- 方法：冒泡排序、选择排序、插入排序、归并排序、快速排序、希尔排序、堆排序、...
- 排序是计算科学中一个非常重要的问题，是很多计算问题的基础，**必须熟练掌握！**

## 6.2.1 冒泡排序 (bubble sort, or sinking sort)

- 冒泡排序（升序）的算法思想：**在数组中多次操作，每一次都比较一对相邻元素**。如果某一对为升序（或数值相等），数值保持不变；如果某一对为降序，则将数值交换。
- 冒泡排序的特点：（密度）较小的数值快速从下往上“冒”，就像水中的气泡一样，而（密度）较大的值（如小固件）则往下沉。



输入

3	12	17	33	2	14	1
---	----	----	----	---	----	---



1	2	3	12	14	17	33
---	---	---	----	----	----	----

输出

# 冒泡排序算法与执行过程

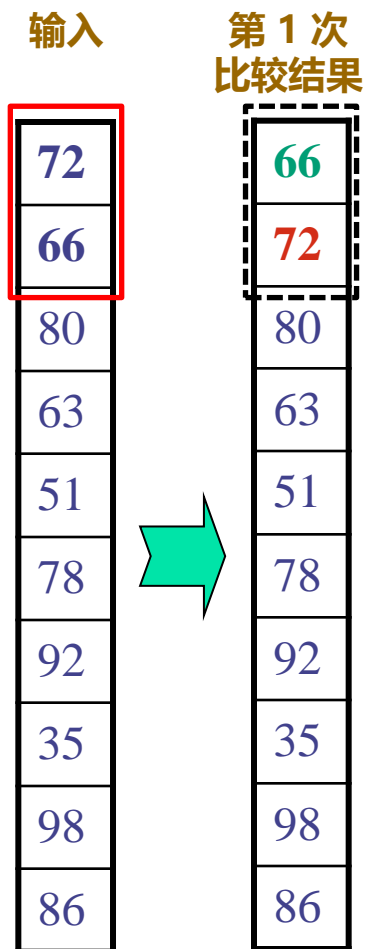
- 升序冒泡排序过程：输入

输入

72
66
80
63
51
78
92
35
98
86

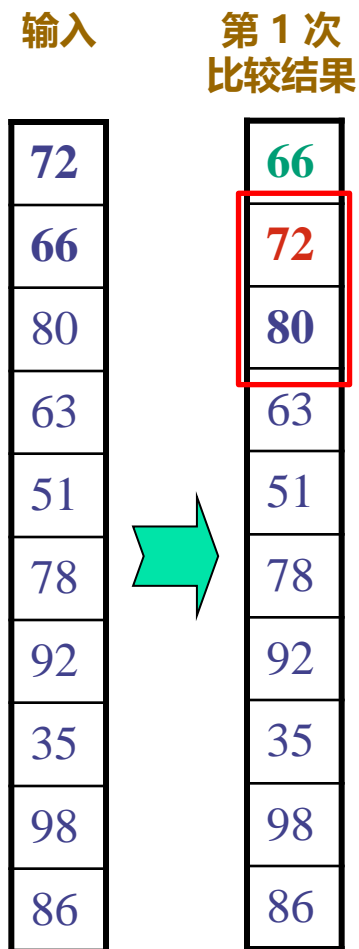
# 冒泡排序算法与执行过程

- 升序冒泡排序过程：第 1 次比较及其结果



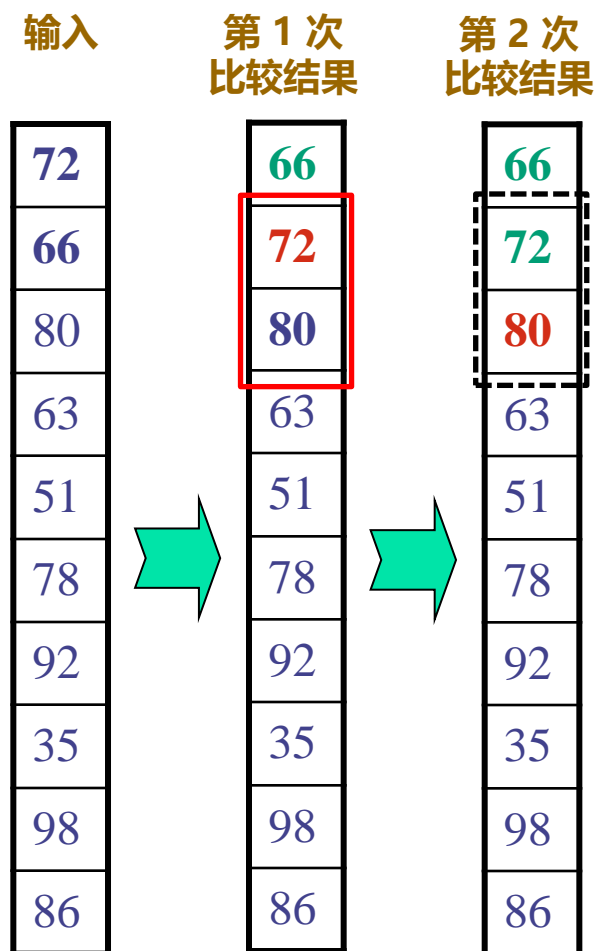
# 冒泡排序算法与执行过程

- 升序冒泡排序过程：第 2 次比较



# 冒泡排序算法与执行过程

- 升序冒泡排序过程：第 2 次比较结果



# 冒泡排序算法与执行过程

- 升序冒泡排序过程：第 3 次比较

输入	第 1 次 比较结果	第 2 次 比较结果
72	66	66
66	72	72
80	80	80
63	63	63
51	51	51
78	78	78
92	92	92
35	35	35
98	98	98
86	86	86

# 冒泡排序算法与执行过程

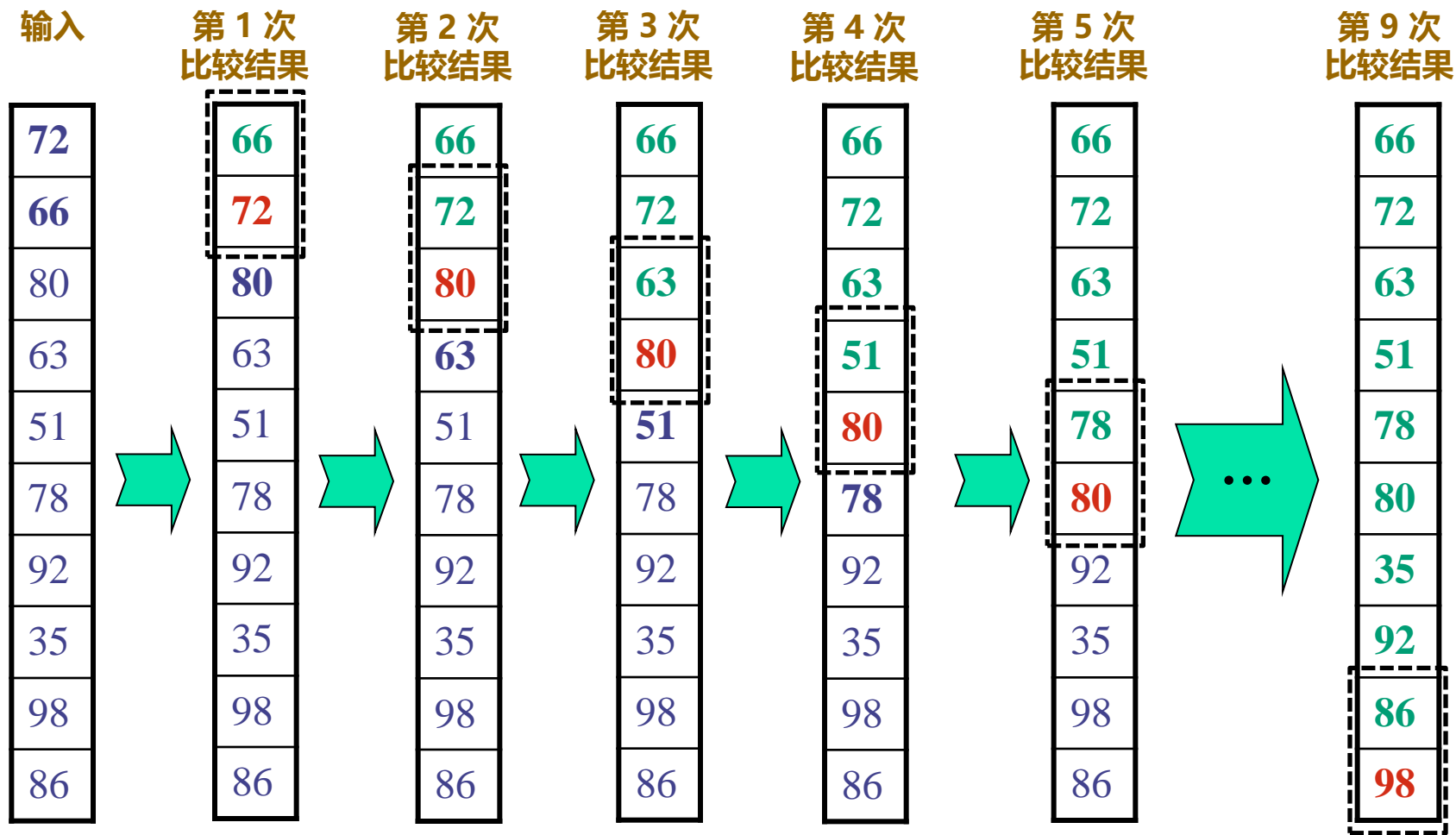
- 升序冒泡排序过程：第 3 次比较结果

输入	第 1 次 比较结果	第 2 次 比较结果	第 3 次 比较结果
72	66	66	66
66	72	72	72
80	80	80	63
63	63	63	80
51	51	51	51
78	78	78	78
92	92	92	92
35	35	35	35
98	98	98	98
86	86	86	86



# 冒泡排序算法与执行过程

- 升序冒泡排序过程：第 9 次比较结果（第一遍扫描结束）



# 冒泡排序代码实现

升序冒泡排序过程：遍历第1遍，9次比较结果(1.9)；第2遍8次(2.8)；.....

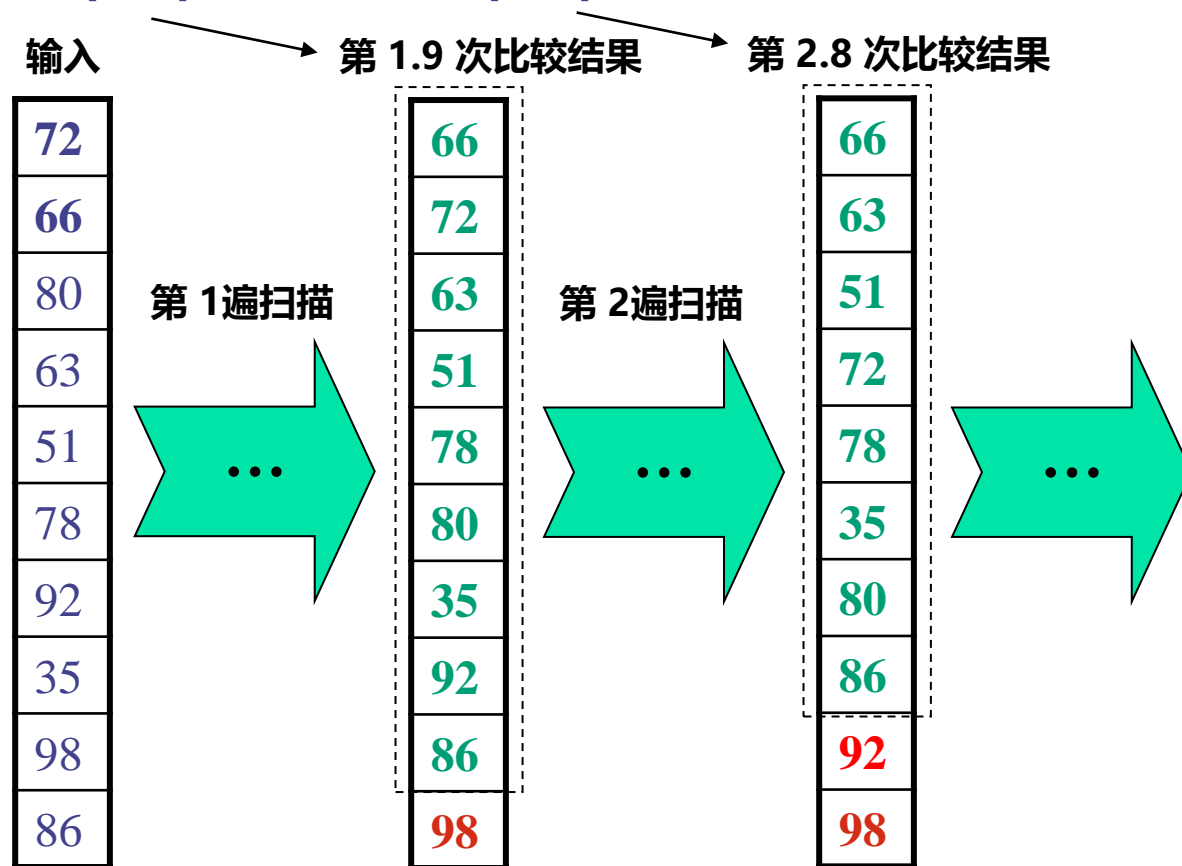
```
void bubbleSort(int a[], int n)
{
    int i, j, hold;
    for(i = 0; i < n - 1; i++)
        for(j = 0; j < n-1-i; j++)
            if(a[j] > a[j + 1])
            {
                hold = a[j];
                a[j] = a[j + 1];
                a[j + 1] = hold;
            }
}
```

算法：

- 两两比较相邻数据
- 反序则交换
- 直到全部遍历结束

算法过程：

- 对当前还未排好序的前缀子数组，自上而下对相邻的两个数依次进行比较
- 让较大的数往下沉，较小的往上冒



# “完整”的冒泡排序实现

```
#include <stdio.h>
#define N 10
void bubble(int [], int);
int main()
{
    int i, a[N];
    // 输入数据到数组 a, 代码段略

    bubble(a, N);

    for(i=0; i<N; i++) //输出排序后的数组a
        printf("%d, ", a[i]);

    return 0;
}
```

```
void bubble(int b[], int n)
{
    int i, j, hold;
    for(i=0; i<n-1; i++)
        for(j=0; j<n-1-i; j++)
            if(b[j]>b[j+1])
            {
                hold = b[j];
                b[j] = b[j+1];
                b[j+1] = hold;
            }
}
```

## 【例6-2】

请同学们补充完成该代码，然后运行，观察结果。补充工作：

1. 根据需要处理的数组元素个数修改 define 中 N 的值；
2. 输入数据到数组 a，（若 N 比较大，则应把 a 定义为全局数组）。

# 冒泡排序能否再优化？ 给我们什么启示？

## 经典的冒泡算法

```
// 经典的冒泡算法
void bubbleSort(int a[], int n)
{
    int i, j, hold;
    for(i = 0; i < n - 1; i++)
        for(j = 0; j < n-1-i; j++)
            if(a[j] > a[j + 1])
            {
                hold = a[j];
                a[j] = a[j + 1];
                a[j + 1] = hold;
            }
}
```

输入	第一遍	第二遍
1	1	1
5	3	3
3	5	5
8	7	7
7	8	8

还需要继续扫描吗？

## 优化后的冒泡算法

```
// 优化的冒泡算法
void bubbleSort(int a[], int n)
{
    int i, j, hold, flag;
    for (i = 0; i < n-1; i++)
    {
        flag = 0;
        for (j = 0; j < n-1-i; j++)
        {
            if (a[j] > a[j + 1])
            {
                hold = a[j];
                a[j] = a[j + 1];
                a[j + 1] = hold;
                flag = 1;
            }
        }
        if (0 == flag)
            break;
    }
}
```



启示2：一个程序写完后，通常还可以再优化，优化后的方法可以有效地提高效率。  
**学编程，培养我们精益求精的做事态度。**

# 冒泡排序性能分析简述

- 比较次数

$$(n-1) + (n-2) + \dots + 2 + 1 = \frac{n(n-1)}{2} \iff O(n^2)$$

- 交换次数

- ◆ 最好情况（输入是正序）：0次

- ◆ 最坏情况（输入是逆序）： $(n-1) + (n-2) + \dots + 2 + 1 = \frac{n(n-1)}{2}$ ， $O(n^2)$

- 时间复杂度（比较+交换次数）

$$O(n^2)$$

易知，当  $n$  比较小的时候，冒泡排序很有效；当  $n$  比较大时，冒泡排序就很慢。如  $n$  为  $10^5$ ，则冒泡排序需要做  $10^{10}$ （100亿）次比较（设计算机每秒比较10亿次，即 $10^9$ 次），则需要10秒才能完成冒泡排序。OJ上会 TLE。

// 经典的冒泡算法

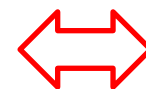
```
void bubbleSort(int a[], int n)
{
    int i, j, hold;
    for(i=0; i < n-1; i++)
        for(j=0; j<n-1-i; j++)
            if(a[j]>a[j+1])
            {
                Swap(a[j], a[j+1])
            }
}
```

比较

交换

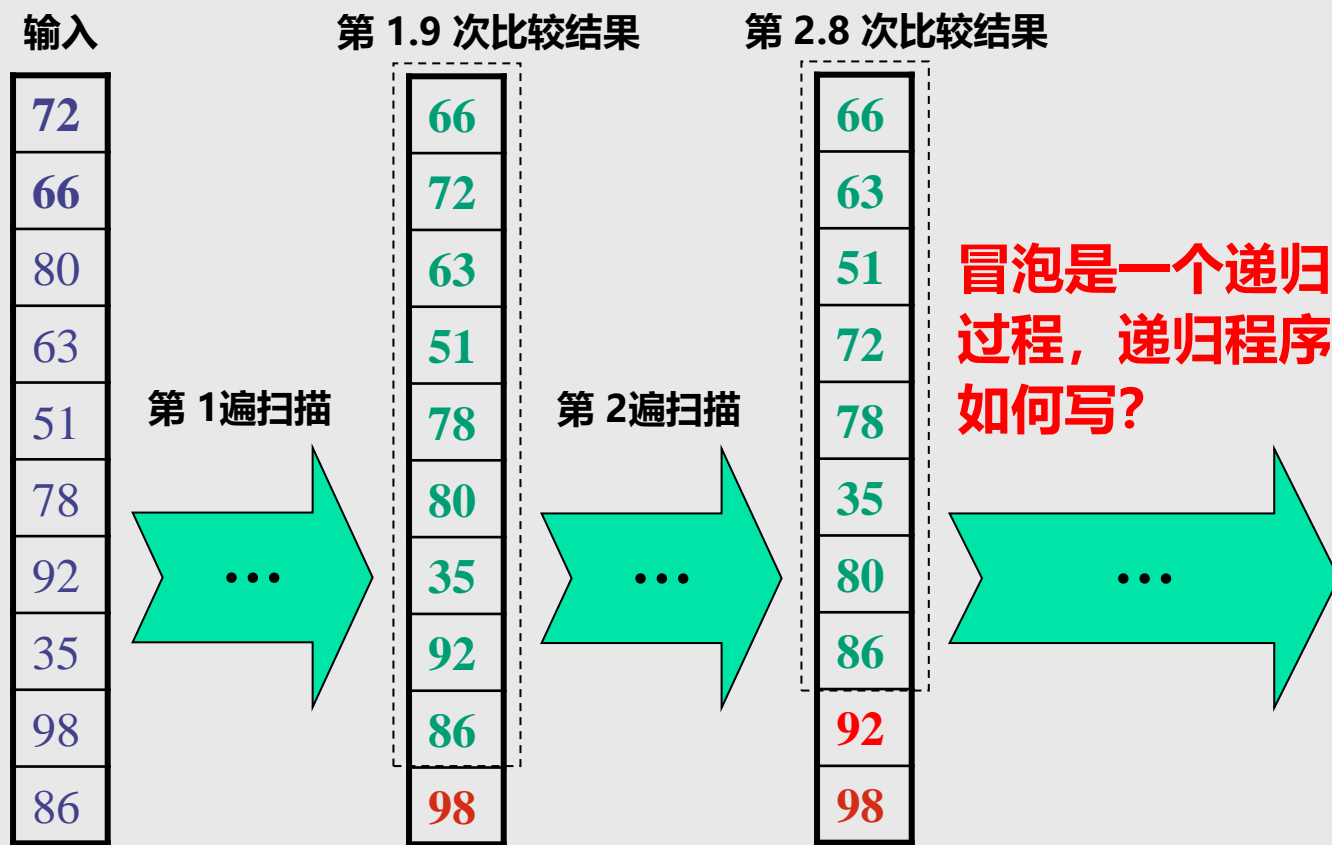
	输入		输出
最好情况示例	1		1
	3	→	3
	5		5
	7		7
	8		8

Swap(a[j], a[j+1])



```
hold = a[j];
a[j] = a[j+1];
a[j+1] = hold;
```

# 另一种思考：冒泡排序的递归实现



冒泡排序的算法：

- 对当前还未排好序的前缀子数组，自上而下对相邻的两个数依次进行比较
- 让较大的数往下沉，较小的往上冒

```
void bubble(int n)
{
    int j, hold, flag=0;
    if(n<=1) // 只剩一个数，已排序，直接返回
        return;
    for(j=0; j<n-1; j++) // 一遍扫描，找最大值
        if(a[j]>a[j+1]) // a 是全局数组
        {
            hold = a[j];
            a[j] = a[j+1];
            a[j+1] = hold;
            flag = 1;
        }
    if(0 == flag)
        return; // 无交换，则已排好序，返回

    bubble(n-1); // 递归，对剩下n-1个数继续冒泡
}
```



启示3：一个问题，多种实现方式。  
兵无常势，水无常形。  
程序设计，犹如用兵。

## 扩展知识2：两个关键字索引的排序

OJ排序依据：

**得分是第一关键字**，分越高排名越靠前。

**罚时是第二关键字**，得分相同的情况下，  
罚时越少排名越靠前。

用冒泡排序，核心算法如何写？

得分数组为 `int a[N]`;

罚时数组为 `int t[N]`; // 值已换算为秒数

AC 编程

主页

小组

题目

赛事

比赛排名

OJ 排序如何实现的？

«

‹

1

2

3

4

5

6

7

...

12

›

»

排名	用户	得分	罚时	A	B	C	D
	<div></div>			1182/1195	1153/1187	1102/1141	713/1
1	昊	1000	6:26:53	0:00:42	0:02:57	0:05:53	0:15:56
2	华	900	4:26:37	0:00:15	0:01:40	0:05:09	0:14:22
3	浜	900	6:08:20	0:00:37	0:00:27	0:04:47(+1)	0:58:32
4	杰	871.4286	9:09:32	0:00:08	0:04:04	0:09:30	0:40:56
5	杰	842.8571	8:26:25	0:02:04(+1)	0:09:07	0:14:05	0:32:57
6	渝	800	4:03:47	0:03:33	0:06:18	0:09:32	1:30:47
7	森	800	4:25:11	0:03:56	0:07:20	0:13:52	0:33:56
8	升	800	5:01:49	0:03:02	0:05:23	0:10:45(+1)	1:05:56
9	泽	800	5:07:19	0:04:15	0:06:52	0:10:50	0:22:56
10	霞	800	5:25:34	0:04:28	0:09:27	0:12:51	0:57:45

## 扩展知识2：两个关键字索引的排序解析

### 请填空

```
for(i = 0; i < N-1; i++)  
    for(j = 0; j < N-1-i; j++)  
        if(_____)  
        {  
            _____  
        }  
}
```

将来学习完二维数组，或结构体数组，把每一行作为一个整体，会更方便。

得分数组为 int a[N]; //值已换算为秒数  
//罚时数组为 int t[N];

比赛排名			
« < 1 2 3 4 5			
排名	用户	得分	罚时
	<input type="text"/>		
1	昊	1000	6:26:53
2	华	900	4:26:37
3	浜	900	6:08:20
4	杰	871.4286	9:09:32
5	杰	842.8571	8:26:25
6	渝	800	4:03:47



# 6.2.2 查找(finding, searching)

- 功能：寻找数组中是否存在一个元素等于给定关键字(key value)的过程
- 应用：根据学号、姓名找学生， ...

a[0]	a[1]	a[2]	...
------	------	------	-----

key == a[i]

### 查找所需四要素

- ✓ 被查找的数据集合：数组（或其他）
- ✓ 需要查找的关键字：键值
- ✓ 查找方法：线性、折半
- ✓ 查找结果：返回找到（位置） or 未找到

比赛排名

« < 1 2 3 4 5 6 7 ... 12 > »

排名	用户	得分	罚时	A	B	C	D
	张三			1182/1195	1153/1187	1102/1141	713/1
1	昊	1000	6:26:53	0:00:42	0:02:57	0:05:53	0:15:56
2	华	900	4:26:37	0:00:15	0:01:40	0:05:09	0:14:22
3	浜	900	6:08:20	0:00:37	0:00:27	0:04:47(+1)	0:58:32
4	杰	871.4286	9:09:32	0:00:08	0:04:04	0:09:30	0:40:56
5	杰	842.8571	8:26:25	0:02:04(+1)	0:09:07	0:14:05	0:32:57
6	渝	800	4:03:47	0:03:33	0:06:18	0:09:32	1:30:47
7	永	800	4:25:11	0:03:56	0:07:20	0:13:52	0:33:33
8	升	800	5:01:49	0:03:02	0:05:23	0:10:45(+1)	1:05:33
9	泽	800	5:07:19	0:04:15	0:06:52	0:10:50	0:22:33
10	霞	800	5:25:34	0:04:28	0:09:27	0:12:51	0:57:45

# 线性（顺序）查找

- 算法：从第一个元素开始扫描数组，依次将数组中每个元素与关键字相比较，若相等，查找成功，返回索引（位置）；否则，到最后都没有找到关键字，查找失败，返回失败标记

- 前提条件：无，可以是无序或有序数组

a[0]	a[1]	a[2]	.....
------	------	------	-------

key == a[i]

$$\frac{\sum_1^n k}{n} = \frac{n(n+1)}{2n} = \frac{n+1}{2}$$

- 时间复杂度： $O(n)$

- 适用：小型数据集查找，对大数组，线性查找的效率不高（尤其对需要频繁查找的情况）
- 缺点： $n$  很大时，若执行多次查找，效率低
- 优点：算法简单，对查找对象没有要求

key: 3

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



# 线性（顺序）查找

```
int find(int x[], int key, int SIZE)
{
    int j;
    for ( j = 0; j < SIZE; j++ )
        if ( x[j] == key )
            return j;
    return -1;
}
```

key 是要查找的值,  
SIZE 是数组大小

a[0]	a[1]	a[2]	.....
------	------	------	-------

key == a[i]

## 线性查找

- 被查找的数据范围: int x[]
- 需要查找的关键字: int key
- 查找方法: 线性, 即遍历整个数组
- 查找结果:
  - ◆ 找到, 返回元素下标
  - ◆ 未找到, 返回-1 (也可以返回其他标记, 根据实际需要来决定)

# 折半查找（二分查找）



**猜数字游戏（我的心里想了一个 1~1024 的整数，猜猜是多少）。**  
**我只回答你：猜小了，或猜大了，或猜中。多少次内肯定能猜中？**

- 折半查找算法：

将关键字  $key$  和查找表  $a$  中间位置的数  $a[mid]$  相比较，如果相等，查找成功；  
如果  $key < a[mid]$ ，则在查找表的前半个子表  $a[low .. mid-1]$  中继续折半查找；  
否则，在后半个子表  $a[mid+1 .. high]$  中继续折半查找。  
不断重复上述过程，直到查找成功或失败。

- 前提：数组有序

- 时间复杂度： $O(\log_2 n)$

- 优点：查找效率高

折半查找在每次比较之后排除所查找数组的一半元素

$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	$a[6]$	$a[7]$	$a[8]$	$a[9]$	$a[10]$	$a[11]$	$a[12]$	$a[13]$
1	3	4	7	9	10	13	34	35	37	46	49	50	53

# 折半查找的效率

- 最糟糕的情况下，查找  $2^{10} = 1024$  个元素的数组只要进行 11 次比较（通常简略说成10次）

1趟: ( $1024=2^{10}$ ): a[0] a[1] a[2] a[3] ..... a[510] a[511] a[512] ..... a[1023]  
2趟: ( $512=2^{10-1}$ ): a[0] a[1] a[2] a[3] ... a[255] ... a[510] a[512] ..... a[1023]  
3趟: ( $256=2^{10-2}$ ): ...  
.....  
10趟: ( $2=2^{10-9}$ ): ...  
11趟: ( $1=2^{10-10}$ ): ...

- 每查找一次，排除所查找数组的一半元素，即除以2
- 查找10亿（约  $2^{30}$ ）个元素的数组：线性查找，10亿次比较 vs 折半查找，30次比较
- 折半查找需要排序数组，而排序数组的成本比线性查找要高（快速排序的效率  $n\log_2 n$ ），如果需要在数组中多次查找，则预先排序数组是值得的（磨刀不误砍柴工）（厚积薄发，一劳永逸）
- 查找次数:  $\log_2 n$        $n / \log_2 n \rightarrow \infty$  ( when  $n \rightarrow \infty$  )

# 折半查找的实现

```
// binary find, recursive version
int rec_bin_find(int b[], int key, int low, int high)
{
    int mid;

    if( low > high )
        return -1;

    mid = (low + high)/2;
    if( key == b[mid] )
        return mid;
    else if( key < b[mid] )
        return rec_bin_find(b, key, low, mid-1);
    else
        return rec_bin_find(b, key, mid+1, high);
}
```

```
// non-recursive version
int bin_find(int b[], int key, int low, int high)
{
    int mid;
    while( low <= high )
    {
        mid = (low + high)/2;
        if( key == b[mid] )
            return mid;
        else if (key < b[mid])
            high = mid-1;
        else
            low = mid+1;
    }
    return -1;
}
```

- 被查找的数据范围: int b[]
- 需要查找的关键字: int key
- 查找方法: 折半, 即每次只查找当前范围的一半
- 查找结果: 找到, 返回元素下标; 未找到, 返回-1



# “完整”的折半查找程序

```
#include <stdio.h>
#define LEN 1000
int bin_find( int[], int, int, int);
int rec_bin_find( int [], int, int, int);
int count_find = 0;

int main()
{
    int a[LEN], key, result, i;

    // ... 输入数组a, 输入查找关键字key

    result = rec_bin_find(a, key, 0, LEN-1);
    // result = bin_find(a, key, 0, LEN-1);

    if(result != -1)
        printf("Found at [%d]", result);
    else
        printf("Key not found");
    printf("\nfind times: %d", count_find);
    return 0;
}
```

```
// binary find, recursive version
int rec_bin_find(int b[], int key, int low, int high)
{
    int mid;

    if( low > high )
        return -1;

    mid = (low + high)/2;
    if( key == b[mid] )
        return mid;
    else if( key < b[mid] )
        return rec_bin_find(b, key, low, mid-1);
    else
        return rec_bin_find(b, key, mid+1, high);
}
```

```
// non-recursive version
int bin_find(int b[], int key,
int low, int high)
{
    int mid;
    while( low <= high )
    {
        mid = (low + high)/2;
        if( key == b[mid] )
            return mid;
        else if (key < b[mid])
            high = mid-1;
        else
            low = mid+1;
    }
    return -1;
}
```

## 【例6-3】

请同学们补充 main 函数，然后运行，观察结果。补充工作：

1. 根据需要处理的数组元素个数修改 define 中 LEN 的值；
2. 输入数据到数组 a ，输入 key （若 N 比较大，则应把 a 定义为全局数组）。

## 扩展知识3：折半查找的其他应用（不仅仅限于数组）

### 【例】单调函数的方程求解

$f(x) = 2\sin(x) + \sin(2x) + \sin(3x) + (x-1)^2 - 20 = 0$ ，  
对方程求解（解析解不易求得）。

什么是  
解析解？

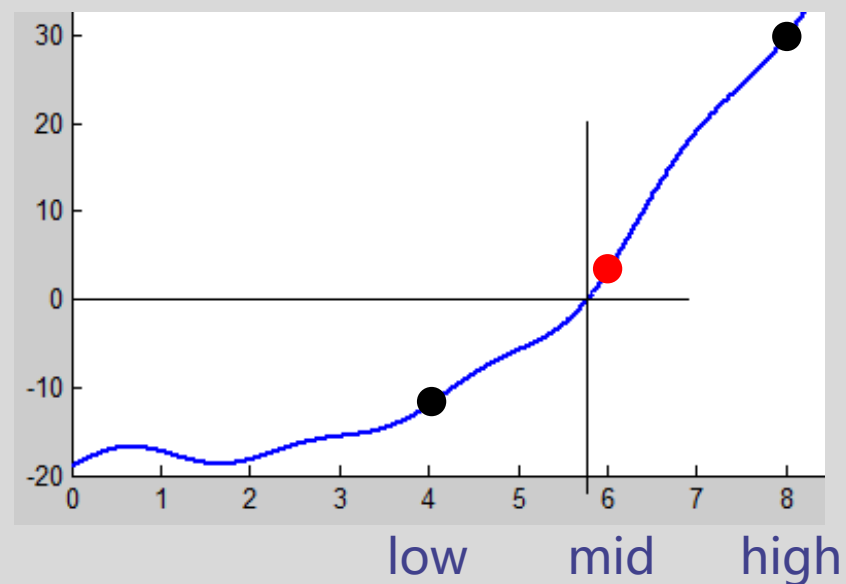
如，一元二次方程求根  
 $ax^2 + bx + c = 0$

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

分析可知：f(x) 连续，且在  $x \in [4, 8]$  之间单调递增（主要由 $(x-1)^2$ 决定），且  $f(4) < 0$ ， $f(8) > 0$ ，那么在4和8之间一定会有方程的根。

用折半查找法进行方程**数值求解**，其算法思想：

- ① 找到适当的初始搜索区间，如  $[low \ high] \leftarrow [4 \ 8]$ ；
- ② 计算中点  $mid = (high + low) / 2$ ，  
若  $high - low < eps$ ，mid 为近似解（eps是预设精度）；  
若  $f(mid) > 0$ ，把高点重置为mid，即  $high \leftarrow mid$ ；  
若  $f(mid) < 0$ ，把低点重置为mid，即  $low \leftarrow mid$ ；
- ③ 重复第 ② 步。





## 扩展知识3：折半查找解方程的递归实现（非数组版）

```
#include <stdio.h>
#include <math.h>
#define eps 1e-6
double f(double);
double solve_f(double, double);

int main()
{
    double x, low=4, high=8;

    x = solve_f(low, high);

    printf("%f\n", x);

    return 0;
}
```

一个很通用的函数

```
double solve_f(double low, double high)
{
    double mid = (high+low)/2;
    if(high-low < eps)
        return mid;
    else if( f(mid)>0 )
        return solve_f(low, mid);
    else
        return solve_f(mid, high);
}

double f(double x)
{
    return 2*sin(x)+sin(2*x)+sin(3*x)+(x-1)*(x-1)-20;
}
```

折半查找  
解方程

前半  
中查找

待求解的函数

折半查找功能强大，务必掌握，灵活运用！

## 扩展知识3：折半查找解方程的两种实现

### 循环实现

```
double solve_f(double low, double high)
{
    double mid = (high+low)/2;
    while(high-low > eps)
    {
        mid = (high+low)/2;
        f(mid)>0 ? (high = mid) : (low = mid);
    }
    return (high+low)/2;
}
```

### 递归实现

```
double solve_f(double low, double high)
{
    double mid = (high+low)/2;
    if(high-low < eps)
        return mid;
    else if( f(mid)>0 )
        return solve_f(low, mid);
    else
        return solve_f(mid, high);
}
```

像数学一样优雅，像诗歌一样优美

## 6.3 字符串和字符数组

## 6.3 字符串和字符数组

- 字符串：由 '\0' 结尾的有限长序列，例如，"Hello, world", "A", "123456", char s[] = "123" 等
  - ◆ **字符串结束标志符：** '\0'，表示空字符 (null character)，是由编译器自动添加到字符串结尾处

H	e	l	l	o	,	w	o	r	l	d	\0
---	---	---	---	---	---	---	---	---	---	---	----

- 字符串到底是什么？先看它可以是什么

- ◆ **1：字符数组，** char s1[64];

**特点：**借助数组预先分配的若干连续字符空间，存储字符串（即：字符串是由多个连续字符组成的），所能存储的字符的个数是有限的

**2：字符串常量，**如"Hello, world"

**3：字符型指针，**如char \*s2; (后面介绍)

"A"      'A'

A \0      VS      A

字符串和字符区别：

- 字符串：双引号表示且必须以 '\0' 结尾
- 字符：单引号表示

说明：C语言没有专用的字符串数据类型，都是通过字符数组或字符指针实现

# 字符串和字符数组

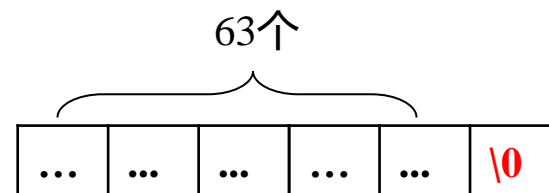
## 定义和初始化字符数组

(1) 显式定义字符串长度, `char s1[64];`

(2) 隐式定义字符串长度, `char a[] = "Hello,world";`

数组长度由编译器根据字符串中实际字符个数确定, 共11个字符, 结尾隐藏'\0',

实际长度12, 等价于`char a[] = { 'H', 'e', 'l', 'l', 'o', ',', 'w', 'o', 'r', 'l', 'd', '\0' };`



a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]
H	e	l	l	o	,	w	o	r	l	d	\0

### 字符数组的长度问题:

- 显示定义中, 字符串最多只能存储“长度-1”个实际字符
- 隐式定义中, 字符串长度= “” 中的字符个数+1
- 用字符数组定义字符串时, 需提供足够空间存储字符串中的实际字符和空字符
- 已知字符串, 建议用隐式方法定义
- 数组一旦定义, 空间已确定, 只能修改存储内容, 不能修改存储大小
- 长度错误为逻辑错误, 编译器不报错

### 字符数组的赋值问题:

- 只有在初始化时, 可直接用字符串整体赋值
- 其他情况, 都必须通过循环逐个元素赋值

```
char s1[3] = "first"; //逻辑错误, 编译器不报错
printf("%s", s1); // 数组越界, 结果未知
s1[] = "second"; // 整体赋值, 语法错误
char s2[10];
s2[10]= "first"; // 把地址赋值给单个“变量”, 逻辑错误
```

# 字符数组定义与初始化

		全局字符数组	局部字符数组
“只定义” 时		默认每个元素赋值为 '\0'	默认每个元素赋值为随机值
“定义+初始化” 时	恰够存	用指定字符串赋值，结尾自动添加'\0'	
	足够存	剩余部分全部用'\0'填充	
	不够存	截断，按实际长度存储，结尾不存'\0'	

```
char G1[10];
char G2[10] = "Hi";
char G3[10] = {'H', 'i'};
char G4[10] = {'H', 'i', '\0'};
char G5[] = "Hi";
char G6[] = {'H', 'i'};
int main()
{
    char La[10];
    char Lb[10] = "Hi";
    char Lc[10] = {'H', 'i'};
    char Ld[10] = {'H', 'i', '\0'};
    char Le[] = "Hi";
    char Lf[] = {'H', 'i'};
    ...
}
```


计算字符串长度：

- strlen(G)：返回字符串实际长度，不计算'\0'
- sizeof(G)：返回字符数组所占字节数

# 字符数组访问

字符数组元素的访问跟其他类型数组元素的访问一样，此时每个数组元素就相当于一个字符变量。

```
char s[ ] = "first";  
// char s[ ] = {'f', 'i', 'r', 's', 't', '\0'};  
for(i = 0; s[i] != '\0'; i++)  
    printf("%c", s[i]);
```



字符串定义的三大条件：

- 序列
- char类型
- '\0' 结尾

判断字符串结束的通常用法

# 字符串和字符数组的关系

- 字符串与字符数组：字符串是内容定义，字符数组是类型定义。
- 字符数组可用于定义字符串，但未必所有字符数组都是字符串，只有以 '\0' 结尾的字符数组才“被认为”是字符串，否则只能称之为字符序列。
- 数组可以是int, float, char等很多类型，字符串可以看成是一个char类型的数组。

字符数组：  
char s1[ ] = {"first"}; // 用串常量"first"初始化字符数组 s1  
char s2[ ] = {'f', 'i', 'r', 's', 't', '\0'};  
char s3[ ] = {'f', 'i', 'r', 's', 't'};

s1, s2, s3都是字符数组  
s1和s2可作为字符串处理  
s3不是字符串

	字符串	字符数组
常量或变量	常量或变量	变量
定义方式	常量字符串、0结尾的字符数组、char*指针	char s[] = "first"; /* 用字符串常量对字符数组初始化, "first" 是常量, 但s[]是变量。 printf("%s", s); ≡ printf("%s", "first"); */
读写方式	常量字符不可更改	当无const限定时, 可读, 可写
结束标志	'\0'	无要求



# 字符数组示例

## 【例6-5】一行字符串倒置

```
#include <stdio.h>
#include <string.h>
void str_rev(char []);

int main()
{
    char a[100];
    int i, hi=0, low=0;
    gets(a);

    puts(a);
    str_rev(a);
    puts(a);
    return 0;
}
```

gets: 读入一行字符串（回车结束，支持空格读入）

scanf: 空格不能读入

→ 如果写成 `scanf("%s", a);` 结果怎么样?

puts vs printf?

```
void str_rev(char s[])
{
    int hi = 0, low = 0;
    char temp;
    while (s[hi] != '\0')
        hi++;
    for (hi--; hi > low; low++, hi--)
    {
        temp = s[low];
        s[low] = s[hi];
        s[hi] = temp;
    }
}
```

## 6.4 常用的标准字符串函数

`#include <string.h>`

- puts, fputs
- gets, fgets
- scanf, sscanf
- printf, sprintf
- strcpy, strncpy
- strcmp, strncmp
- strlen
- strchr, strrchr
- strstr

- ✓ 内容较多，难以记忆。
- ✓ 尽量理解，熟悉名称。
- ✓ 学会自查，灵活运用。
- ✓ 初学时，在字符串处理中一定会犯很多错误！
- ✓ 要习惯，关键是要在错误中成长！

# 字符串输入输出函数: gets, puts

- 行(hang)输入函数 `char * gets(char s[ ]);`
  - ◆ 从标准输入读取完整的一行（遇到换行符或输入数据的结尾），将读取的内容存入 s 字符数组中，并用字符串结束符 '\0' 取代行尾的 '\n'。若读取错误或遇到输入结束则返回 NULL
  - ◆ 输入时，一定要确保数组的空间足够存储需要读入的字符串长度
- 行输出函数 `int puts (char s [ ]);`
  - ◆ 将字符数组 s 中的内容(以 '\0' 结束)输出到标准输出上，并在末尾添加一个换行符

```
char s[N];  
if (gets(s) != NULL)  
    puts(s);
```

puts 和 printf 输出的都必须是字符串('\0'结束)，否则可能会运行出错。

```
if (scanf("%s", s) != 0)  
    printf("%s", s);
```

输入结束标志为空格、制表符、回车等（不能读入空格）。

末尾不添加换行。如果输出后换行，通常写成 `printf("%s\n", s);`

# 字符串输入输出函数: gets, puts

puts 和 printf 等字符串操作的函数都必须是字符串 ( '\0' 结束 ), 否则可能会运行出错

```
char a[] = {'a', 'b', 'c'};
char b[] = "abc";

printf("%d, %d\n", sizeof(a), strlen(a) );
printf("%d, %d\n", sizeof(b), strlen(b) );

printf("%s\n", a );
printf("%s\n", b );

puts(a);
puts(b);
```

本代码片段输出什么?  
哪些地方有错?

# 字符串输入输出函数: gets, puts

puts 和 printf 等字符串操作的函数都必须是字符串 ( '\0' 结束 ), 否则可能会运行出错

```
char a[] = {'a', 'b', 'c'};  
char b[] = "abc";  
  
printf("%d, %d\n", sizeof(a), strlen(a) );  
printf("%d, %d\n", sizeof(b), strlen(b) );  
  
printf("%s\n", a );  
printf("%s\n", b );  
  
puts(a);  
puts(b);
```

本代码片段输出什么?  
哪些地方有错?

输出

```
3, 4  
4, 3  
abc  
abc  
abc  
abc
```

使用字符串处理函数处理非字符串, 得到结果是不可信的  
(有时可能碰巧结果无误, 但不表示程序正确)

# 字符串输入输出函数：scanf, printf

```
char a[N], b[N], c[N], d[N];  
double v;  
  
scanf("%s%s%s%lf", a, b, c, &v);  
printf("%s %s %s %f", a, b, c, v);  
  
scanf("%s%s%s%s", a, b, c, d);  
printf("%s %s %s %s", a, b, c, d);
```

输入：the num is 1.23  
输出：the num is 1.230000

输入：the num is 1.23  
输出：the num is 1.23

输入“相同”


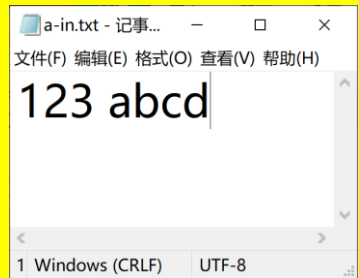
输出不同

这里的1.23是  
字符串

# 字符串IO: scanf vs gets, printf vs puts

	不同点	相同点
<code>int scanf("%s", a)</code>	返回值: int 输入结束标记: 空白符	输入字符串的中间没有空白符, 且在数据范围时, 两者一致
<code>char *gets(a)</code>	返回值: char * 输入结束标记: \n或EOF	
<code>int printf("%s", a)</code>	原样输出	输出数据合法时, 两者相似 (除了puts会额外添加\n以外)
<code>int puts(a)</code>	在输出末尾添加一个换行符	

输入:



123 abcd

```
char a[9] = "", b[9] = "";
gets(a);
puts(a);

scanf("%s", b);
printf("%s", b);
```

输出: ?

123 abcd

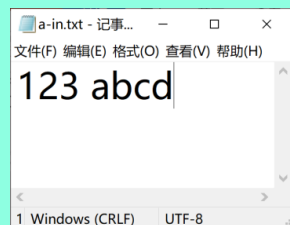
123  
abcd

针对第一组输入"123 abcd":  
"123 abcd" 是一行, 读入a。puts  
输出a, 然后加 \n。  
读到EOF, b没有读入任何数据,  
输出空串 (初始化为空串)。

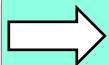
针对第二组输入:  
两行分别读入a和b, 并进行相应  
输出, 输出a时自动添加 \n。

# 字符串IO: scanf vs gets, printf vs puts

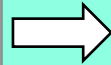
输入:



a-in.txt - 记事...  
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)  
123 abcd



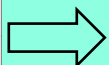
```
char a[9] = "", b[9] = "";  
scanf("%s", b);  
printf("%s", b);
```



123 abcd



a-in.txt - 记事...  
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)  
123  
abcd



```
gets(a);  
puts(a);
```



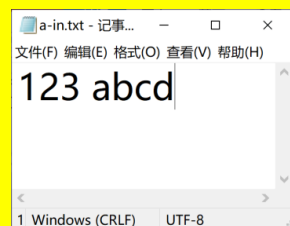
123

输出: ?

"123"和 " abcd"分别读给b和a并进行相应输出, a后加 \n 。 (注意: " abcd"前面有一个空格也是字符串的有效元素)

"123"读入b, gets(a)遇到"123"后的 \n, 读入结束, puts()输出空串, 但输出puts自动添加 \n 。

输入:



a-in.txt - 记事...  
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)  
123 abcd



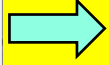
```
char a[9] = "", b[9] = "";  
gets(a);  
puts(a);
```



123 abcd



a-in.txt - 记事...  
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)  
123  
abcd



```
scanf("%s", b);  
printf("%s", b);
```



123  
abcd

输出:

针对第一组输入"123 abcd" :

"123 abcd" 是一行, 读入a。 puts输出a , 然后加 \n。  
读到EOF, b没有读入任何数据, 输出空串 (初始化为空串) 。

针对第二组输入: 两行分别读入a和b, 并进行相应输出, 输出a时自动添加 \n 。

gets  
puts  
放后



# 字符串IO: scanf vs gets, printf vs puts

输入:



a-in.txt - 记事...  
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)  
123 abcd



a-in.txt - 记事...  
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)  
123  
abcd

```
char a[9] = "", b[9] = "";  
scanf("%s", b);  
printf("%s", b);
```

```
gets(a);  
puts(a);
```

输出: ?

123 abcd

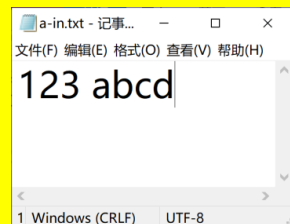
123

"123"和 " abcd"分别读给b和a并进行相应输出, a后加 \n。(注意: " abcd"前面有一个空格也是字符串的有效元素)

输出显示效果一样, 但意义完全不一样!

"123"读入b, gets(a)遇到"123"后的\n, 读入结束, puts()输出空串, 但输出puts自动添加\n。

输入:



a-in.txt - 记事...  
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)  
123 abcd



a-in.txt - 记事...  
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)  
123  
abcd

```
char a[9] = "", b[9] = "";  
gets(a);  
puts(a);
```

```
scanf("%s", b);  
printf("%s", b);
```

输出:

123 abcd

123  
abcd

gets  
puts  
放后

针对第一组输入"123 abcd":

"123 abcd" 是一行, 读入a。puts输出a, 然后加\n。

读到EOF, b没有读入任何数据, 输出未初始化的b, 值不确定。

针对第二组输入: 两行分别读入a和b, 并进行相应输出, 输出a时自动添加\n。

# 字符串输入函数：fgets

gets先天有缺陷！gets的漏洞，它不做地址越界检查。如定义char s[5], 调用gets(s)后输入 abcdef ? fgets采取了弥补措施！

- 行输入函数 `char * gets(char s[ ]);`

若读入的字符串大于数组s的长度（C语言不进行数组的越界检查！），程序运行会出现难以预期的错误。早些时候，这种错误成为一些黑客的攻击要点。一种普遍采用的解决方案：

```
char * fgets(char *s, int n, FILE *fp);
```

↑  
fgets从fp所指文件最多读n-1个字符送入s指向的内存区，并在最后加一个 '\0'（若读入n-1个字符前遇换行符或文件尾（EOF）即结束）

- 最多读入 n-1 个字符，保证输入不造成对数组的越界
- 如果数组 s 足够大，输入中包括换行符，则换行符也被读入数组
- 需指定输入来源文件 fp（stdin表示标准输入，即键盘）

# 字符串输出函数: fputs

- `char * fgets(char *s, int n, FILE *fp);`

`fgets`从 `fp` 所指文件读`n-1`个字符送入`s`指向的内存区，并在最后加一个 `'\0'` (若读入`n-1`个字符前遇换行符或文件尾 (EOF) 即结束)

- 最多读入`n-1`个字符，保证输入不造成对数组的越界
- 如果数组`s`足够大，输入中包括换行符，则换行符也被读入数组
- 需指定输入来源文件`fp` (`stdin`表示标准输入，即键盘)

- `int fputs(char *s, FILE *fp);`

把 `s` 指向的字符串写入 `fp` 指向的文件

- 返回值

- `fgets`正常时返回读取字符串的首地址，出错或文件尾时返回`NULL`
- `fputs`正常时返回写入的最后一个字符，出错为`EOF`

# 字符串输入输出函数实例

- `char * fgets(char *s, int n, FILE *fp);`
- `int fputs(char *s, FILE *fp);`

## 【例6-6】 puts, fputs, printf 对比

```
char s[N];  
if(fgets(s, N, stdin) != NULL)  
    puts(s);
```

VS

```
char s[N];  
if(fgets(s, N, stdin) != NULL)  
    printf("%s", s);
```

左边输出的最后多一个空行 (puts会在输出后加一个换行符'\n') 。

VS

上下两个程序的输出一样。

```
char s[N];  
if(fgets(s, N, stdin) != NULL)  
    fputs(s, stdout);
```

# 字符串输出（构造）函数：sprintf

【例6-7】由参数确定输出的小数位数 从标准输入读入浮点数 $x$  ( $-10 < x < 10$ )和整数 $m$  ( $0 < m < 13$ ), 在标准输出上输出 $\sin(x)$ 的值, 保留到小数点后 $m$ 位数字。

如：输入 3.14 3, 输出 0.002; 输入 3.14 10, 输出 0.0015926529

分析：如果直接用`printf("%.#f", sin(x))`, 则需要用`switch`或`if`语句, 有很多判断条件 (这里#是常数, 值跟输入的 $m$ 相同)。

**字符串构造函数** `int sprintf(char *buf, char *format [, argument]...);`

```
int m;    double x;
char buf[32];
scanf("%lf%d", &x, &m);
sprintf(buf, "%%.%df\n", m);
printf(buf, sin(x));
```

- `printf`是按格式把输出送到屏幕上, `sprintf`是按格式把输出送到 `buf` 对应的字符数组
- 注意: `buf` 对应的字符数组应足够大
- `sprintf` 常用于需要动态生成字符串的场合

若输入 $m$  为 3, 则"`%%%.%df\n`"变为"`%.3f\n`", 且该字符串存入字符数组 `buf`

斜杠是编译器级别的转义, `%`是`printf`内部的解析特殊符号, 因此斜杠是不行的, 只能是`%%`, 不能是`\%`

**sprintf 函数非常好用, 应掌握。sprintf 与 printf 类似。**

# 字符串输入（构造）函数：sscanf

字符串构造函数：int sscanf(const char \*buf, char \*format [, arg]...);

【例6-8】分析日期和时间 计算机显示的时间通常有特殊的格式，比如计算机给出的格式

12/Nov/2020:12:15:00 +0800

重新输出

表示北京时间2020年11月12日12时15分0秒。给出一个这种格式表示的字符串，提取其中的每一项，并在屏幕上按行单独显示出来。如，红框的数据应显示为右边格式：

2020  
Nov  
12  
12  
15  
0  
+0800

```
int day, year, h, m, s;  
char mon[4], zone[6];  
char buf[] = "12/Nov/2020:12:15:00 +0800";  
sscanf(buf, "%d/%3c/%d:%d:%d:%d %s", &day, mon, &year, &h, &m, &s, zone);  
mon[3] = '\\0'; // 什么作用？  
printf("%d\\n%s\\n%d\\n%d\\n%d\\n%d\\n%s", year, mon, day, h, m, s, zone);
```

- scanf 是按要求的格式从键盘输入数据到对应的地址（变量地址或数组）
- sscanf 是按要求的格式从 buf 读入数据（也是在<stdio.h>里定义）
- 返回值也是成功读入的字段数，一般弃之不用

# 字符串复制函数：strcpy, strncpy

```
char *strcpy(char dest[ ], const char src[ ]);
```

将字符串 `src` 复制到字符数组 `dest` 中，返回 `dest[]`。`dest` 的长度应足够长以能够放下 `src` （应用：ctrl+c then ctrl+v）。

```
char *strncpy(char dest[ ], const char src[ ], size_t n);
```

拷贝 `src` 的前 `n` 个字符到 `dest` 数组中，如果提前遇到字符串结束符，则在 `dest` 中写入 0，直到完成 `n` 个字符写入。

在实际使用中，`n` 一般是字符数组 `dest` 的长度，这样可以防止数组越界。

注意如果 `src` 的字符个数大于 `n`，拷贝后的 `dest` 可能不是一个合法字符串（缺少结束符 0）。

# 字符串复制函数: strcpy, strncpy

```
char x[] = "1234 abcd ABC123";
char y[25], z[25]; // teststrncpy.c

printf("Source: %s\n", x);
printf("Dest_1: %s\n", strcpy(y, x));

strncpy(z, x, 11); //does not copy null character
z[11] = '\0';
printf("Dest_2: %s\n", z);

strncpy(z, "abcdefg hijklmn", 6);
printf("Dest_3: %s\n", z);
```

输出:

Source: 1234 abcd ABC123

Dest\_1: 1234 abcd ABC123

Dest\_2: 1234 abcd A

Dest\_3: abcdefbcd A

- z未初始化, 把x的前11个字符拷贝到z, z的最后必须添加'\0'
- z是字符串, 其元素超过6个, z的前6个被替换, 此时无需添加'\0'

```
strncpy(z, "abc", 6);
printf("Dest_4: %s\n", z);
```

Dest\_4: abc



# \*strncpy的几个实例

```
char x[] = "1234 abcd ABC123";
char z1[25], z2[25] = "", z3[25] = "";

printf("\nZ1_ini: %s\n", z1);
printf("Z2_ini: %s\n", z2);

strncpy(z1, x, 11);
// z1[11] = '\0';
printf("Z1_cpy: %s\n", z1);

strncpy(z2, x, 11);
z2[11] = '\0';
printf("Z2_cpy: %s\n", z2);

strncpy(z3, x, 11);
// z3[11] = '\0';
printf("Z3_cpy: %s\n", z3);
```



```
命令提示符
C:\alac\example\chap6>teststrncpy2
Z1_ini: H簋      棹a
Z2_ini:
Z1_cpy: 1234 abcd Ao俗  ☐@
Z2_cpy: 1234 abcd A
Z3_cpy: 1234 abcd A

C:\alac\example\chap6>teststrncpy2
Z1_ini: ☐?      棹a
Z2_ini:
Z1_cpy: 1234 abcd Ao俗  ☐@
Z2_cpy: 1234 abcd A
Z3_cpy: 1234 abcd A

C:\alac\example\chap6>teststrncpy2
Z1_ini: /M?      棹a
Z2_ini:
Z1_cpy: 1234 abcd Ao俗  ☐@
Z2_cpy: 1234 abcd A
Z3_cpy: 1234 abcd A
```

连续运行三次的结果

z1没有初始化，里面的内容是随机的（实际输出时当成字符串处理，遇到内存中字符串结束标志时结束，但可能数组越界了）。

z2和z3初始化全部为\0，因此输出内容一样。

# 字符串追加函数: strcat, strncat

(cat, concatenate, 连接)

```
char *strcat(char dest[ ], const char src[ ]);
```

将字符串 src 添加到字符串 dest 后面（追加），src 的第一个字符覆盖dest的\0结束符，并在最终字符串后面添加\0结束符，函数返回 dest。

```
char *strncat(char dest[ ], const char src[ ], size_t n);
```

将字符串 src 中最多前 n 个字符添加到字符串 dest 后面，src 的第一个字符覆盖dest的\0结束符，并在最终字符串后面添加\0结束符，函数返回 dest。


当字符串src长度小于等于n时，功能与strcat相同

应用：今天的作业没有写完，明天接着写

# 字符串追加函数: strcat, strncat

```
char x[] = "1234 abcd ABC123";  
char y[20] = "", z[20] = "";  
  
printf("    x: %s\n", x);  
printf("    y: %s\n", strcat(y, x));  
  
strncat(z, x, 11);  
printf("    z: %s\n", z);  
  
strncat(z, "abc", 6);  
printf("z+abc: %s\n", z);  
  
strncat(z, "123456789abcdef0", 12);  
z[19] = '\0';  
printf("z+new: %s\n", z);
```

输出:

 选择C:\a1ac\example\chap6\teststrcat.exe

```
x: 1234 abcd ABC123  
y: 1234 abcd ABC123  
z: 1234 abcd A  
z+abc: 1234 abcd Aabc  
z+new: 1234 abcd Aabc12345
```

- 把x的前11个字符追加到z, 无需手动在后面添加'\0', z初始化时20个字符都是'\0'。
- z已有14个元素, 追加12个元素后, 已超z的容量。**z数组越界! 可能会导致越界部分内容覆盖掉别的有用数据! 这种用法是错误的!!!**

# 字符串比较函数: strcmp, strncmp

按姓名拼音排序

Bill Gates

Song You

Zhou Jielun

```
int strcmp(char s1[ ], char s2[ ]);
```

比较字符串 s1 与 s2 , 函数在 s1 等于、小于或大于 s2 时分别返回 0、小于 0 或大于 0 的值。

```
int strncmp(char s1[ ], char s2[ ], size_t n);
```

比较字符串 s1 与 s2 的前 n 个字符, 函数在 s1 等于、小于或大于 s2 时分别返回 0、小于 0 或大于 0 的值。

# 字符串比较函数: strcmp, strncmp

```
char *s1 = "Happy New Year to you";  
char *s2 = "Happy New Year to you";  
char *s3 = "Happy Holidays";  
  
printf("s1-s2: %d\n", strcmp(s1, s2));  
printf("s1-s3: %d\n", strcmp(s1, s3));  
printf("s3-s1: %d\n", strcmp(s3, s1));  
printf("s1-s3, 6: %d\n", strncmp(s1, s3, 6));  
printf("s1-s3, 7: %d\n", strncmp(s1, s3, 7));  
printf("s3-s1, 7: %d\n", strncmp(s3, s1, 7));
```

输出

```
s1-s2: 0  
s1-s3: 1  
s3-s1: -1  
s1-s3, 6: 0  
s1-s3, 7: 6  
s3-s1, 7: -6
```

这里返回的是字符编码值的差。  
有些系统可能输出1和-1，这跟  
所用的编译系统有关。

# 字符串检查计算函数: strlen

```
int strlen(char s[ ]);
```

返回字符串s的字符个数，长度中不包括终止符 '\0' 。

```
char s1[] = "abcdefghijklmnopqrstuvwxyz";  
char s2[] = "just do it";  
char s3[] = {'w', 'e', '\0'};
```

```
printf("strlen(s1): %d\n", strlen(s1));  
printf("strlen(s2): %d\n", strlen(s2));  
printf("sizeof(s2): %d\n", sizeof(s2));  
printf("strlen(s3): %d\n", strlen(s3));  
printf("sizeof(s3): %d\n", sizeof(s3));
```

输出

```
strlen(s1): 26  
strlen(s2): 10  
sizeof(s2): 11  
strlen(s3): 2  
sizeof(s3): 3
```

```
//自己实现strlen的一种方法  
int strlen(char s[])  
{  
    int i = 0;  
    while (s[i] != '\0')  
        i++;  
    return i;  
}
```

## **\*\*字符串匹配函数： strchr、 strstr...**

```
char *strchr(char s[ ], int c);
```

```
char *strrchr(char s[ ], int c);
```

**返回字符 c 在字符串 s 中第一次或最后一次出现的**位置的指针**。如果 s 中没有 c，两个函数都返回NULL。**

```
char *strstr(char *s, char *sub_str);
```

**返回子字符串 sub\_str 在字符串 s 中第一次出现的位置的指针。如果 s 中没有 sub\_str，返回NULL。**

思考：返回子字符串 sub\_str 在字符串 s 中第二次、第三次出现的位置的指针，如何实现？

## 6.5 二维数组与多维数组

### 现实世界的数据类型

- 简单数据类型

- ◆ 整数
- ◆ 浮点
- ◆ 字符

- 复合数据类型

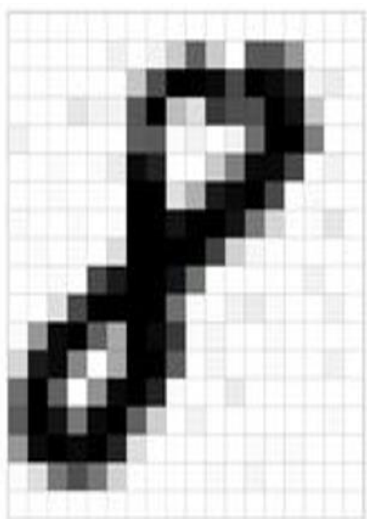
- ◆ 一维数组
- ◆ 多维数组：？

A word cloud of C data types. The word 'int' is the largest and most central, colored green. Other words include 'unsigned int' (green), 'long long' (cyan), 'double' (purple), 'float' (green), 'short' (brown), 'long' (brown), 'unsigned long' (grey), 'long int' (green), 'long double' (purple), 'char' (blue), 'char c[10]' (blue), 'unsigned short' (blue), and 'double d[4]' (grey). The words are arranged in a circular pattern around the central 'int'.

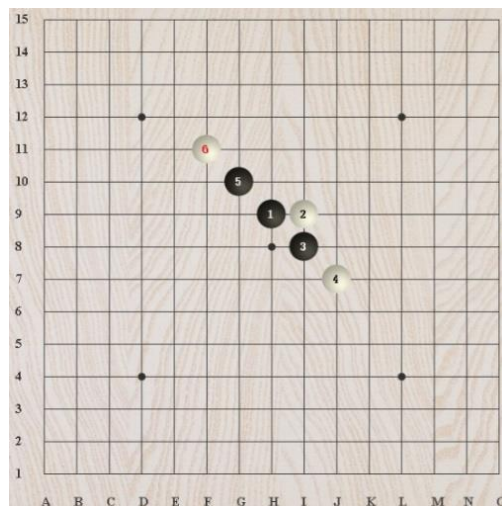


## 6.5 二维数组与多维数组

- 一维数组（单下标数组）：字符串，语音信号， $\sin(t_k)$ ，...
- 二维数组（双下标）应用更广泛：平面图像，excel表格，...

[illegible]

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

[illegible]

AC编程

主页

小组

题目

赛事

E4-2020级程序设计基础训练第四次练习

简介

题目

排名

我的提交

提问&&公告

服务器当前时间

2020-11-03 20:14:46

比赛结束时间

2020-11-07 12:00:00

比赛剩余时间

87:45:13

比赛排名

更新中, 上次更新于 2020-11-03 20:14:20

«

«

1

2

3

4

5

6

7

»

»

排名	用户	得分	罚时	A 475/532	B 375/441	C 159/260
1	蒋	1000	32:00:22	0:35:06(+1)	0:58:16	4:12:59(+2)
2	秦	1000	32:05:03	0:39:43	0:52:38(+1)	3:50:38(+2)
3	占	1000	32:17:25	0:22:45	0:46:42(+1)	2:22:27(+4)
4	景	1000	35:12:18	2:06:42	2:11:37	2:28:07(+1)
5	江	1000	45:05:47	0:55:31	1:28:28(+2)	3:48:57(+2)
6	潘	1000	53:26:50	24:01:39(+1)	0:07:15	4:18:10(+4)
7	李	1000	59:59:14	4:22:50	4:28:43	4:38:29(+1)
8	黄	1000	60:34:12	0:51:45	1:05:36	1:27:04(+3)
9	杨	1000	62:18:17	0:06:01	2:42:38(+1)	3:31:40(+1)

# 二维数组定义及初始化

- 二维数组定义：数据类型 数组名[行数][列数]；其中行数和列数是**常量表达式**
  - 例如：`int a[3][2];` // 3行2列， $3 \times 2 = 6$ 个数组元素
- 二维数组初始化：数据类型 数组名[行数][列数] = {初始化数据}

a[0][0]	a[0][1]
a[1][0]	a[1][1]
a[2][0]	a[2][1]

// 定义若干局部数组

```
int a[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

```
int b[2][3] = {1, 2, 3, 4, 5, 6};
```

```
int c[2][3] = {{1, 2}, {4}};
```

```
int d[ ][3] = {{1, 2}, {4}};
```

```
int e[2][3] = {1, 2, 3, 4};
```

```
int f[1][3];
```

数组 a, c, d 的定义方式是好的习惯

- 按序对每一个数组元素初始化赋值
- 按序对每一个数组元素初始化赋值
- 每一行初始化值用一对大括号括起来，初始化值不足时默认值为0
- 行数由初始化值中的行数决定。二维数组初始化时可省略行数，但不能省略列数
- 按行优先规则顺序初始化
- 无初始化，默认“随机值”

a	1 2 3 4 5 6	初始化后每个数组的值
b	1 2 3 4 5 6	
c	1 2 0 4 0 0	
d	1 2 0 4 0 0	
e	1 2 3 4 0 0	
f	200858566 208295927 -1	

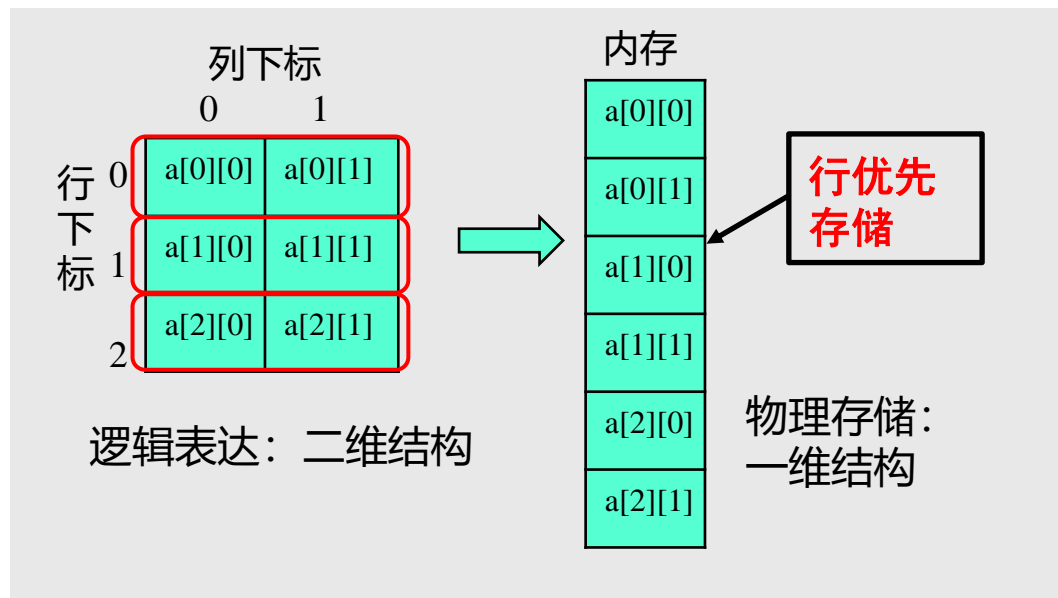
二维数组初始化的方法：

- 可以定义时初始化
- 定义时无初始化：全局二维数组默认为0，局部二维数组赋值为随机值
- 不完全初始化：初始化值的个数不超过二维数组范围时，其余部分默认为0

同样，二维数组也要避免越界！

# 二维数组的存储排列方式

- C语言的数组名都表示数组的首地址，数组元素从首地址开始，按顺序连续在地址中存储
- **二维数组的存储是行优先的**，例： `int a[3][2];`
- 二维数组应用：表示由行和列组成的二维表格，矩阵等
- 引用某个元素： `a[i][j]`；表示数组中第  $i*n+j$  个元素，第一个表示元素所在行下标(从0开始)，第二个表示元素所在列下标(从0开始)
- 通常用循环的for结构访问二维和 multidimensional 数组元素



```
int a[3][2]; //定义包含 3 行2 列的数组a
for(i = 0; i < 3; i++)
{
    for(j = 0; j < 2; j++)
    {
        // a[i][j] = i*2 + j;
        scanf("%d", &a[i][j]);
        printf("%d ", a[i][j]);
    }
    printf("\n");
}
```

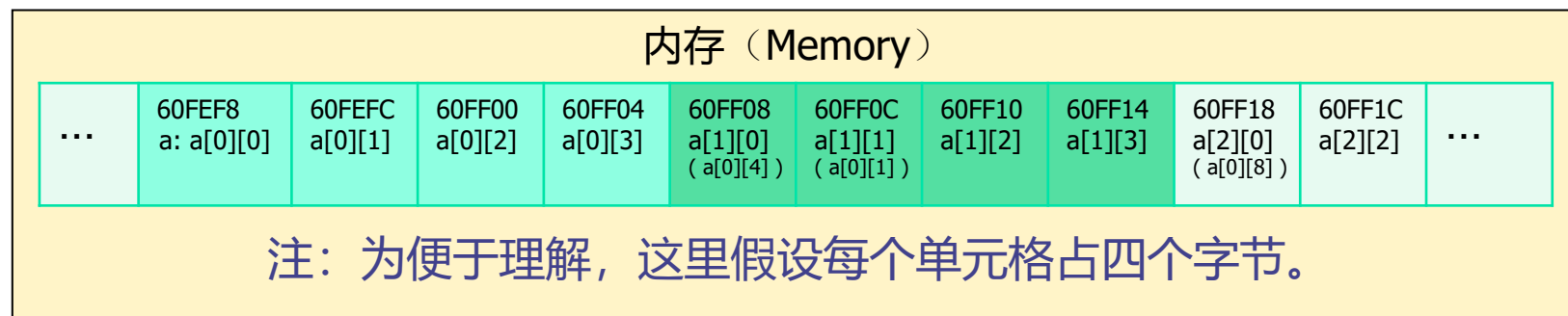
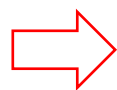
## 二维数组的存储排列方式

- 二维数组的存储是行优先的。设定义数组

`int a[5][4];` // 行  $m$ :  $0 \leq m \leq 4$ ; 列  $n$ :  $0 \leq n \leq 3$

- C语言不对数组的下标做严格的范围检查，访问数组元素 `a[1][1]` 也可以写为 `a[0][5]`，访问数组元素 `a[2][0]` 也可以写为 `a[1][4]` 或 `a[0][8]`。为了提高程序的可读性和维护性，建议访问数组元素时，行、列的编号都限制在定义时的行、列的范围内

a[0][0]	a[0][1]		
	a[1][1]		
a[2][0]		a[2][2]	
		a[3][2]	



**逻辑表达：二维数组的二维结构，是给人看的**

**物理存储：二维数组在计算机里还是一维结构，本质还是一维的**

# 二维数组的存储排列方式

二维数组可以看成是一个超级一维数组、或嵌套的一维数组（数组的每个元素为一个一维数组）。定义数组 `int a[5][4]`，则 `a` 相当于

```
a[ ] = { a[0],  
        a[1],  
        a[2],  
        a[3],  
        a[4] }
```

```
a[0][ ] = { a[0][0], a[0][1], a[0][2], a[0][3] }  
a[1][ ] = { a[1][0], a[1][1], a[1][2], a[1][3] }  
a[2][ ]  
...  
...
```

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

语法糖：

- 二维数组原来是**糖衣**，一维数组才是**药**！
- 药虽苦，但治病。药不好吃，糖衣帮助。



# 二维字符数组

- 二维字符数组初始化

```
char a[3][8]={"str1", "str2", "string3"};
```

```
char b[][6]={"s1", "st2", "str3"};
```

二维数组当作一维数组使用，这个二维数组中的每一个元素是个一维数组

- 二维字符数组的引用

a[0][0]

a[0]	s	t	r	1	\0	\0	\0	\0
a[1]	s	t	r	2	\0	\0	\0	\0
a[2]	s	t	r	i	n	g	3	\0

```
int i;  
char a[3][8] = {"str1", "str2", "string3"};  
  
for (i = 0; i < 3; i++)  
    printf(" %s\n", a[i]);  
    //输出第i行字符串  
  
for (i = 0; i < 3; i++)  
    printf(" %c\n", a[i][i]);  
    //输出第i行i列的字符  
  
for (i = 0; i < 3; i++)  
    printf(" %s\n", &a[i][i + 1]);  
    // 输出第i行i+1列字符开始的字符串
```

运行结果:

```
str1  
str2  
string3  
s  
t  
r  
tr1  
r2  
ing3
```

# 二维数组使用实例

【例6-9】星期几 已知本月有n天，第x天是星期y，求下月k日是星期几

- 定义并初始化一个7行12列的全局二维字符数组
- 确保最长的字符串可以被正确存储

S	u	n	d	a	y	\0	\0	\0	\0	\0	\0
M	o	n	d	a	y	\0	\0	\0	\0	\0	\0
T	u	e	s	d	a	y	\0	\0	\0	\0	\0
.	.	.									

```
#include <stdio.h>
char day_name[][12] =
{
    "Sunday",
    "Monday",
    "Tuesday",
    "Wednesday",
    "Thursday",
    "Friday",
    "Saturday"
};
int weekday(int, int, int, int);
int main()
{
    int n = 30, x = 8, y = 4;
    int k, m;
    scanf("%d", &k);
    m = weekday(x, y, n, k);
    printf("%s\n", day_name[m]);
    return 0;
}
int weekday(int x, int y, int n, int k)
{
    return (n - x + k + y) % 7;
}
```



## 二维数组使用实例

### 【例6-10】数据中的最长行 输入若干行字符串，输出最长行的长度和字符串

```
char arr[2][MAX_N] = {{""}};
int in = 1, longest = 0;
int max_len = 0, len, tmp;
while (gets(arr[in]) != NULL)
{
    len = strlen(arr[in]);
    if (len > max_len)
    {
        max_len = len;
        tmp = in;
        in = longest;
        longest = tmp;
    }
}

printf("%d: %s\n", max_len, arr[longest]);
```

程序算法分析（示例）：

目前最长：arr[longest = 0] = "abcd"

输入前：arr[in = 1] = "ab"

第2行更短，新的输入存入第2行，即下一条：

输入后：arr[in = 1] = "abcdefghi"

若 len > max\_len（新输入的字符串更长），

则 max\_len ← len, in 和 longest 交换数据，字符数组变为：

输入前：arr[in = 0] = "abcd"

目前最长：arr[longest = 1] = "abcdefghi"

保持长的第2行，下一次新输入存入第1行，即：

待输入：arr[in = 0] = "??"

若输入字符串比目前最长的字符串短，不做处理，

接着检查下一个输入字符串，存入当前行。

输入文件结束时，输出结果。

本设计比较巧妙。不需要拷贝数组（字符串）。longest记录已输入的最长行，in表示将要输入的行。



# 二维数组作为函数参数

- 数组参数形式：int a[][4]  
可省略行数，但不能省略列数。多维数组中可省略第一个下标，但不能省略其他下标。
- 访问二维数组元素：二维数组的每行是一个一维数组，要找到特定行中的元素，函数要知道每行有多少元素，以便在访问数组时跳过适当数量的内存地址。如定义int a[5][4]，当访问元素a[3][2]时，函数知道跳过内存中前 3 行的12个元素以访问第 4 行（行下标为3），然后访问这一行的第 3 个元素（列下标为2），即数组中该元素前面有 3\*4+2个元素。
- 实参是否有足够的行数，需要由编程人员来保证。
- 为使函数处理n行的矩阵，则需要将n作为一个参数传递给函数。

```
void print_arr(int[][4]);
int main()
{
    int arr[5][4] = {{1, 2, 3}, {4, 5, 6}};
    print_arr(arr);
    return 0;
}
void print_arr(int a[][4])
{
    int i, j;
    for (i = 0; i < 5; i++)
    {
        for (j = 0; j < 4; j++)
            printf("%d ", a[i][j]);
        printf("\n");
    }
}
```

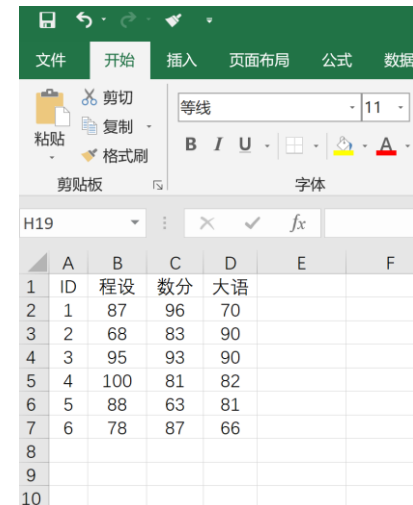
a[0][0]			
		a[2][2]	
		a[3][2]	

b[0]	b[1]	b[2]	b[3]
------	------	------	------

## 二维数组的应用：成绩处理

【例6-12】有一个m行n列的表格存放成绩，表示m个同学，n门课，求：

- (1) 每门课的最高（低）成绩以及获得该成绩的同学的学号（这里用数组行号表示）
- (2) 每门课的平均成绩
- (3) 每个同学的平均成绩




	A	B	C	D	E	F
1	ID	程设	数分	大语		
2	1	87	96	70		
3	2	68	83	90		
4	3	95	93	90		
5	4	100	81	82		
6	5	88	63	81		
7	6	78	87	66		
8						
9						
10						

ID	程设	数分	大语	...
1	87	96	70	
2	68	83	90	
3	95	93	90	
4	100	81	82	
5	88	63	81	
6	78	87	66	
...				

m行n列

数据示例：

```
int grade[stu][course];
```



	1	2	3	4
1	87	96	70	
2	68	83	90	
3	95	93	90	
4	100	81	82	
5	88	63	81	
6	78	87	66	

输出：

```
命令提示符
C:\alac\example\chap6>c6-12_grade < c6-12.in

highest grade:  程设  数分  大语
                id:    4    1    2
lowest grade:   程设  数分  大语
                id:    2    5    6

course average:  86.0  83.8  79.8

student average:
1      84.3
2      80.3
3      92.7
4      87.7
5      77.3
6      77.0
```

更多问题：

- (1) 学生平均成绩从高到低排序，并按这样顺序输出成绩（每行输出一个同学的各门课成绩、平均成绩）？
- (2) 按平均成绩排序，若平均成绩相等则按程设分数排序，...？
- (3) 求每门课的方差？
- (4) 画出每门课的成绩段分布（直方图）？

.....

## 二维数组的应用：成绩处理

【例6-12】 $m \times n$ 的成绩表格( $m$ 个同学,  $n$ 门课), 求: 每门课的最高(低)成绩; 每门课的平均成绩; 每个同学的平均成绩。(代码未完成, 请读者自行完成其余代码)

```
...
#define stu 6
#define course 3
void input_grade();
void max(); // 求课程的最高分, 并输出
void min(); // 求课程的最低分, 并输出
void print_maxmin(int[], char []); // 打印最高分或最低分
void stu_grade_aver(); // 求每个学生的平均成绩
void cou_grade_aver(); // 求每门课的平均成绩
int grade[stu][course];
int mark_maxmin[2][course];
double s_aver_g[stu]; // 每个学生的平均成绩
double c_aver_g[course]; // 每门课的平均成绩
int main()
{
    input_grade();
    printf("\n%36s", "程设  数分  大语");
    max();
    min();
    cou_grade_aver();
    stu_grade_aver();
    return 0;
}
```

```
void max()
{
    int i, j, h_grade, h_index;
    for(j=0; j<course; j++)
    {
        h_grade = 0;
        h_index = 0;
        for(i=0; i<stu; i++)
            if(grade[i][j] > h_grade)
            {
                h_index = i;
                h_grade = grade[i][j];
            }
        mark_maxmin[0][j] = h_grade;
        mark_maxmin[1][j] = h_index+1;
    }
    print_maxmin(mark_maxmin[0], "highest grade");
    print_maxmin(mark_maxmin[1], "id");
}
```

本例只是讲解二维数组应用的一个简单例子。写一个完整的成绩处理软件需要更多代码。这里并没有从软件工程的角度来认真考量。如果要编写一个实际应用的成绩处理软件, 在完成需求分析后, 必须认真考虑变量与函数命名规范、程序逻辑、数据结构、算法设计、数据管理、输入输出处理、等等。在学习完后面的指针、结构体、文件等知识后, 会发现类似表格处理等软件的实现较容易。

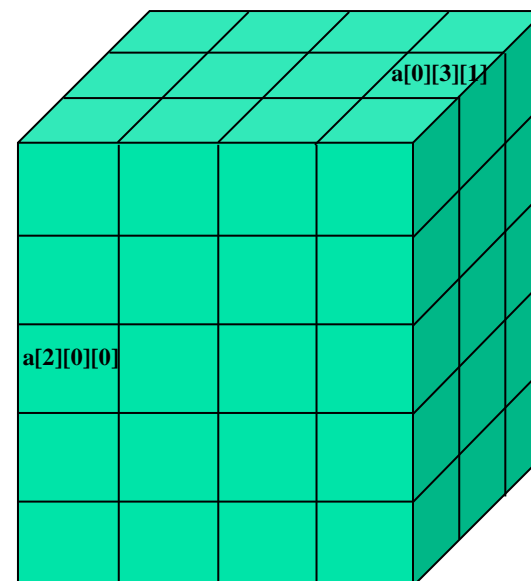
# 多维数组

## 多下标数组可以有多个下标

- ◆  $a[5]$
- ◆  $a[5][4]$
- ◆  $a[5][4][3]$   
(应用：三维制作)
- ◆  $a[X][Y][Z][?]$   
(应用：三维动画制作， $t$ 是第4维)
- ◆  $a[M][N][K][L][P]$

$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$
--------	--------	--------	--------	--------

$a[0][0]$	$a[0][1]$		
		$a[2][2]$	
		$a[3][2]$	



# 本讲总结

- 避免数组使用错误（理解）：数组越界，访问未正确初始化的数组
- 数组作为函数参数（理解）：传递的是数组的地址，确定访问的开始位置
- 数组应用：排序与查找（重点掌握）  
冒泡排序（掌握），其他排序（未来会进一步学习）；顺序查找（掌握），折半查找（掌握）
- 数组应用扩展（自学）：选择、归并、快速排序等，辅助读物《Data Structures and Algorithms》
- 字符串和字符数组的异同（重点掌握）
- 了解基本的标准库字符串处理函数（掌握常用库函数）
- 二维数组定义、结构和访问（掌握）
  - 一维数组是本质，二维数组是抽象，哪个顺手就用哪个
  - 实在没有顺手的，果断舍弃二维数组，直接采用一维数组也可以解决所有问题
- 更多维数组（三维及其以上）定义（了解）
- 使用数组的常用数据结构（将来继续学习）：队列、栈、散列表