



北京航空航天大学
BEIHANG UNIVERSITY



程序设计基础

Fundamentals of Programming

北京航空航天大学 程序设计课程组

软件学院 谭火彬

2022年



北京航空航天大学
BEIHANG UNIVERSITY



第六讲 数组

Array

- ◆ 数组使用常见的错误
- ◆ 数组的应用：参数、排序和查找
- ◆ 字符串与字符数组
- ◆ 二维数组以及多维数组



提纲：数组

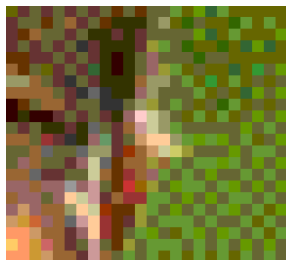
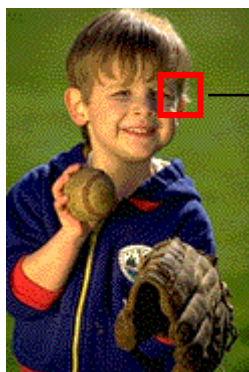
- ◆ 6.1 数组使用常见的错误
- ◆ 6.2 数组作为函数参数
- ◆ 6.3 数组的排序和查找
- ◆ 6.4 字符串与字符数组
- ◆ 6.5 字符串处理函数
- ◆ 6.6 二维数组以及多维数组



6.1 数组使用常见的错误

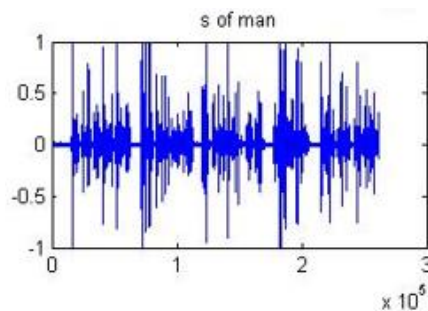
◆数组(array)

- ✓用于保存同类型的多个数据
- ✓数组中的每个数据称为数组元素



W: 122 pixels

H: 182 pixels



	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	



复习：一维数组的定义

◆ 一维数组是最基本数组类型，可以看成是一个逻辑上的一维序列

- ✓ 序列中的每个元素具有相同的数据类型，称为数组的**元素**
- ✓ 数组元素通过数组名和一个**整数下标**（偏移量） 进行访问

◆ 语法格式：`<类型> <数组名>[<数组长度>];`

- ✓ `<类型>` 是数组元素的数据类型，可以是任何合法的基本数据类型
- ✓ `<数组名>` 是 C 语言合法的标识符，表示数组变量的名字
- ✓ `<数组长度>` 是一个整型表达式，表示该数组的元素个数

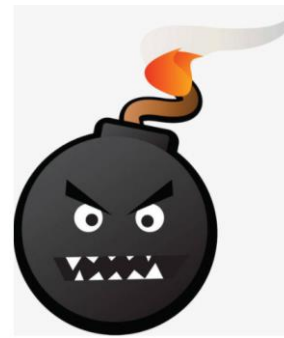


数组的使用存在各种问题

在前面很简单的数组练习中，也经常掉坑里，或踩地雷

复杂的数组应用，风险更大！

接下来，怎么办？





常见问题1：数组越界访问

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

◆数组是在内存中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[6] = {1, 2, 3, 4, 5, 6};
int b[6] = {20, 21};
int c[6] = {-1, -2, -3, -4, -5, -6, -7};
int d[] = {11, 12, 13, 14};
int e[6] = {0}; //int e[6] = {};
int f[6];
for (i = 0; i < 6; i++)
    printf("%d ", a[i]); printf("\n");
for (i = 0; i < 6; i++)
    printf("%d ", b[i]); printf("\n");
for (i = 0; i <= 6; i++)
    printf("%d ", c[i]); printf("\n");
for (i = 0; i < 5; i++)
    printf("%d ", d[i]); printf("\n");
for (i = 0; i < 6; i++)
    printf("%d ", e[i]); printf("\n");
for (i = 0; i < 6; i++) printf("%d ", f[i]);
```

不同的初始化方式（例程中定义的都是局部数组）

- 定义数组a[6]，并按序对每一个数组元素初始化赋值
- 定义数组b[6]，并初始化，其余元素**隐式初始化为0**
- 定义数组c[6]，注意：定义数组，但初始化时元素个数越界，产生**逻辑错误**，编译器不报错。该错误隐蔽，可能会导致严重问题
- 自动定义d[]，并初始化，未显示定义数组个数，以{}中赋值个数为准，**但代码中访问数组d时越界**
- 定义e[6]，统一初始化。
- 定义数组f[6]，但未初始化，其值“特不靠谱”

输出结果：

1 2 3 4 5 6

20 21 0 0 0 0

~~X~~-1 -2 -3 -4 -5 -6 20

~~X~~11 12 13 14 -1

0 0 0 0 0 0

~~X~~6422092 12651628 6422148 4199040 4201296 0

错误的“尽早发现原则”：能在编写时发现的，就不要在编译时发现；能在编译时发现的，就不要在运行时发现



数组越界访问

```
int a[6] = {1, 2, 3, 4, 5, 6};
int b[6] = {20, 21};
int c[6] = {-1, -2, -3, -4, -5, -6, -7};
int d[] = {11, 12, 13, 14};
int e[6] = {0}; //int e[6] = {};
int f[6];
for (i = 0; i < 6; i++)
    printf("%d ", a[i]); printf("\n");
for (i = 0; i < 6; i++)
    printf("%d ", b[i]); printf("\n");
for (i = 0; i <= 6; i++)
    printf("%d ", c[i]); printf("\n");
for (i = 0; i < 5; i++)
    printf("%d ", d[i]); printf("\n");
for (i = 0; i < 6; i++)
    printf("%d ", e[i]); printf("\n");
for (i = 0; i < 6; i++) printf("%d ", f[i]),
```

内存中的数组存放示意 (每个方格占4个字节)

	..	f[0]	f[1]	f[5]	e[0]	e[1]
..	e[5]	d[0]	d[3]	c[0]	c[1]
..	c[5]	b[0]	b[5]
a[0]	a[5]	..			

c[6]

输出结果:

1 2 3 4 5 6

20 21 0 0 0 0

-1 -2 -3 -4 -5 -6

11 12 13 14 -1

0 0 0 0 0 0

6422092 12651628 6422148 4199040 4201296 0

跑到别人家的地,
摘了别人家的菜!

20?



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 a_0, a_1, \dots, a_n 都是数域 F 中的数, n 是非负整数, 那么表达式

$$a_n \times x^n + a_{n-1} \times x^{n-1} + \dots + a_2 \times x^2 + a_1 \times x + a_0$$

就是数域 F 上关于变量 x 的多项式或一元多项式。

- 其中, $a_i \times x^i$ ($1 \leq i \leq n$) 代表该一元多项式中的一个项, a_i 是该项的系数, i 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 $g(x)$, 请你求出二者加和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。

```
int af[100020] = {};  
int ag[100020] = {};
```

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

程序出现运行时
错误, 问题何在?

提示：OJ上REG、OE等错误
大部分都是数组越界造成的！！

指数S的值的取值范围为
int类型, 可能越界



数组越界访问

◆为什么数组没有越界检查？

学编程：让我们学会遵守规则！



德国Füssen小镇火车站

没有检票口，直接上车！

是因为德国人买不起闸机吗？

当然不是！没有检票口是为了进站高效！

数组没有界限检查，是因为C语言能力不行吗？

同理，C语言没有越界检查也是为了高效！

但是，如果违规越界/逃票，后果很严重！

同样，编程违规，后果更严重！

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!



常见问题2：数组的整体处理

◆数据名代表**数组的首地址**，**不代表**具体的内容值，因此无法通过数组名进行赋值和比较；应通过以下方法复制数组

✓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n个字节用 ch 替换并返回 s, 常用于清零等



常见问题3：用变量定义数组大小

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

早期的C语言中，长度必须是字面量或字面量表达式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贮函数，利用它可实现动态申请空间[*]

先定义符号宏，以符号宏作为数组长度，这种用法比较常见

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：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

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



补充：关于变长数组

◆变长数组是指在数组定义时，数组长度可以是整型变量，实际的数组长度在运行时由该整型变量的值确定

```
#include <stdio.h>
int main(){
    int n, i, sum = 0;
    scanf("%d", &n);
    int a[n];
    for (i = 0; i < n; i++) {
        scanf("%d", &a[i]);
        sum += a[i];
    }
    printf("%.2f", sum * 1.0 / n);
    return 0;
}
```

变长数组的隐患

- 引起爆栈现象：变长数组的空间分配在函数调用栈上（变长数组只能是局部数组，不能定义成全局数组），而函数调用栈的长度有限（一般小于 2M），但在运行时 n 的大小是无法预料的，一旦 n 过大，将导致在函数调用栈上开辟了过大的空间，程序会崩溃退出
- 使用条件：在支持 C99 标准的编译器里使用变长数组时，要预估数组长度，确保不会发生“爆栈”的情况
- 为了避免运行时的错误，尽量避免使用变长数组



常见问题4：数组定义时的大小问题

◆实际问题中数据可能很大，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，但将数组移到main函数外面，变为全局数组时则ok，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）。C程序占用内存分为几个部分：

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 $2 \times 1024 \times 1024 = 2097152$ 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[1000 000]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。



常见问题小结

```
int a[5] = {1, 2, 3, 4};
int b[5];
//整体操作错误
b = a;      X
if (a == b) X
    .....
//越界访问错误
b[5] = 1;
for(i=0; iX6; i++)
    b[i] = i;
//长度定义错误
int array[ ];X
int n;
scanf("%d", &n);
int c[n];X
```

正确做法：定义“符号替换宏”，以宏（字面量）作为数组长度

```
#include <stdio.h>
#define LENGTH 10
int main(){
    double s[LENGTH];
    .....
}
// LEN, buff, N, ...
```

正确做法：通过循环对数组中各元素逐一赋值或比较

```
for(i=0; i<12; i++)
    b[i] = a[i];
for(i=0; i<12; i++)
    if(a[i] == b[i]) ...
```

- ◆越界访问：不能超出下标范围进行读取
- ◆整体操作：不能将数组作为整体进行赋值、比较等操作
- ◆长度定义：长度必须是字面量或字面量表达式（C99支持变长数组），也不能定义长度为空的数组
- ◆数组大小位置：局部数组开得太大
- ◆访问未赋值的数组



6.2 数组作为函数参数

◆以数组作为函数的参数：传递数组

✓含义：数组作为参数传递给函数，实际上是传递数组的首地址，即数组第一个元素的地址（函数内部当指针使用！）

◆将实参数组首地址赋值给形参数组，函数内部直接使用通过该地址访问实参，它们共享同一个数据区域，是同一个数组

✓与普通的实参传递不同，函数内部修改数组元素的值时，直接修改实参数组的内容，这种修改在函数结束后依然有效

✓传递数组时，一般需要增加单独的整形参数，以告诉函数内部数组中实际存储的元素个数

◆定义：void f(int array[], int size) {...}

◆调用：函数调用时，用数组名作为实参



C06-01: 计算n维向量的点积

◆计算两个n维向量的点积dot_vec

✓形参中数组长度通常省略，实参传递时不同长度数据均可；如果定义了数组的长度，则只能是同长度的数组

```
//计算两个n维向量va和vb的点积
//可以省略数组的下标，以实参的长度为准
//如果定义了数组的下标，则必修是指定数组的长度
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];
    return s;
}
```



C06-01: 计算n维向量的点积

◆调用函数时，实参直接使用数组名

- ✓ “不能” 包括数组长度，如 `dot_vec(a[5], b[5],...)`，如果写 `a[5]` 传递的就是数组元素了

◆数组传递时，数组名作为实参，将所在空间首地址传递给形参，使得形参实际上操作实参的存储空间

◆函数参数本质上是值传递，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

```
#include <stdio.h>
#define LEN 5
//函数声明，声明计算点积的函数，原型与定义完全一致
double dot_vec(double va[], double vb[], int n);
//函数声明时，可以省略形参的名称
//double dot_vec(double [], double [], int);
int main()
{
    double a[LEN] = {1,2,3,4,5}, b[LEN];
    int i;
    printf("input five float number: ");
    for(i=0; i<LEN; i++)
        scanf("%lf", &b[i]);
    printf("dot_vec: %f\n", dot_vec(a, b, LEN));
    return 0;
}
```



6.3 数组的应用：排序和查找

- ◆数组中存储着多个同类型的数据，最常见的应用就是排序和查找
- ◆**排序 (sort)**：将一组原始数据按**递增 (升序)** 或**递减 (降序)** **顺序**进行重新排列的过程
 - ✓排序是计算机科学中大量研究问题的基础（先排序，再做后续处理），是一个非常基础、非常重要的问题，必须熟练掌握！
 - excel: 按学号排序, 按成绩, 按姓名拼音排序,
 - 如何在网上买评价高的东西? 如何找到学习好的学生?
 - ✓有很多种排序算法: **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.....

常用算法动态演示

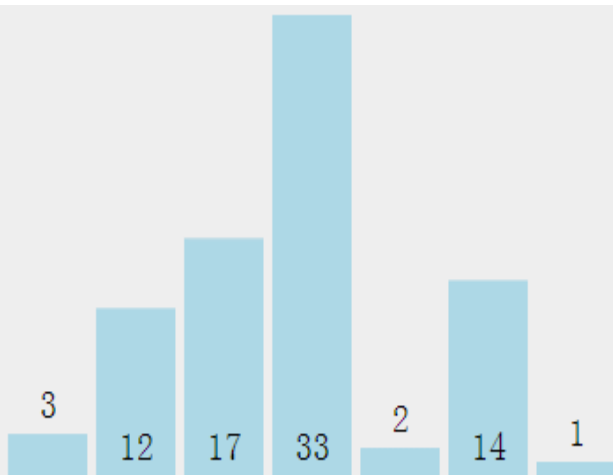
<http://runningls.com/demos/2016/sortAnimation/>
github: <https://github.com/liusaint/sortAnimation>



6.3.1 冒泡排序

◆冒泡排序(bubble sort, or sinking sort)

- ✓算法：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），则将数值保持不变；如果某一对为降序，则将数值交换（默认：从小到大的升序排序）
- ✓特点：使用冒泡排序法，（密度）较小的数值慢慢从下往上“冒”，就像水中的气泡一样，而较大的值则慢慢往下沉



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



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



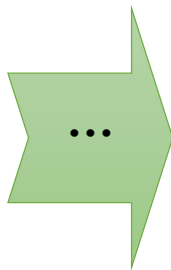
冒泡排序

◆升序冒泡排序过程

- ✓对数组元素进行连续比较（数组扫描）
- ✓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），则将数值保持不变
- ✓如果某一对为降序，则将数值交换。并将大值向下移动
- ✓第一遍扫描将最大值移动到数组最后。
- ✓第二遍扫描将次大值移动到数组倒数第二的位置，...

输入

72
66
80
63
51
78
92
35
98
86



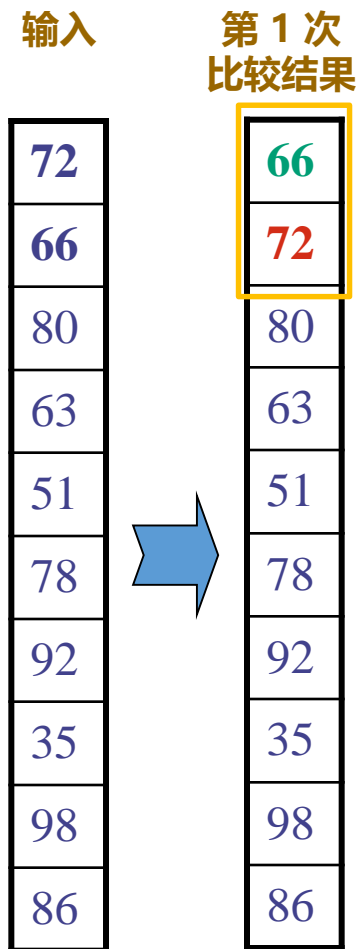
输出

35
51
63
66
72
78
80
86
92
98



冒泡排序过程

◆升序冒泡排序过程：第 1 次比较及其结果





冒泡排序过程

◆升序冒泡排序过程：第 2 次比较

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





冒泡排序过程

◆升序冒泡排序过程：第 2 次比较结果

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



冒泡排序过程

◆升序冒泡排序过程：第 4 次比较

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



冒泡排序过程

◆升序冒泡排序过程：第 4 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	第 4 次 比较结果
72	66	66	66	66
66	72	72	72	72
80	80	80	63	63
63	63	63	80	51
51	51	51	51	80
78	78	78	78	78
92	92	92	92	92
35	35	35	35	35
98	98	98	98	98
86	86	86	86	86



冒泡排序过程

◆升序冒泡排序过程：第 5 次比较

输入	第 1 次 比较结果	第 2 次 比较结果	第 3 次 比较结果	第 4 次 比较结果
72	66	66	66	66
66	72	72	72	72
80	80	80	63	63
63	63	63	80	51
51	51	51	51	80
78	78	78	78	78
92	92	92	92	92
35	35	35	35	35
98	98	98	98	98
86	86	86	86	86



冒泡排序过程

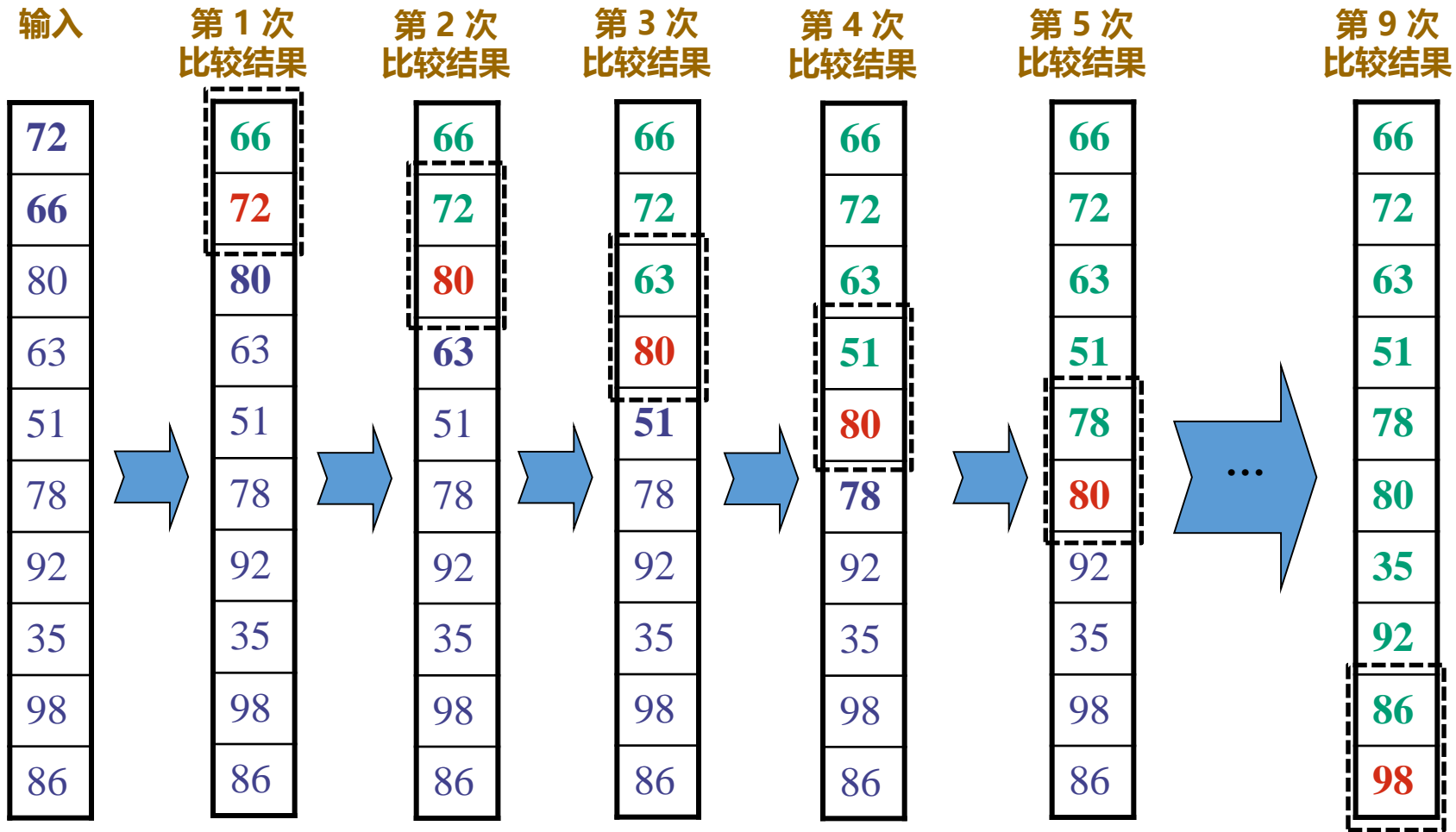
◆升序冒泡排序过程：第 5 次比较结果

输入	第 1 次 比较结果	第 2 次 比较结果	第 3 次 比较结果	第 4 次 比较结果	第 5 次 比较结果
72	66	66	66	66	66
66	72	72	72	72	72
80	80	80	63	63	63
63	63	63	80	51	51
51	51	51	51	80	78
78	78	78	78	78	80
92	92	92	92	92	92
35	35	35	35	35	35
98	98	98	98	98	98
86	86	86	86	86	86



冒泡排序过程

◆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 (C06-02)

◆升序冒泡排序过程（10个元素）：遍历第1遍9次比较结果
(1.9); 第2遍8次(2.8);

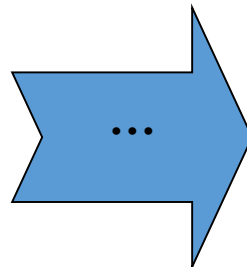
```
//整数排序：升序排列
void bubbleSort(int a[], int n)
{
    int i, j, hold;
    for(i = 0; i < n-1; i++)
        for(j = 0; j < n-1-i; j++)
            if(a[j] > a[j+1])
            {
                hold = a[j];
                a[j] = a[j+1];
                a[j+1] = hold;
            }
}
```

- 两两比较相邻数据，反序则交换
- 直到全部遍历结束

输入

72
66
80
63
51
78
92
35
98
86

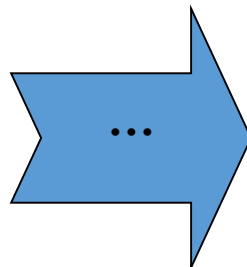
第 1 遍扫描



第 1.9 次比较结果

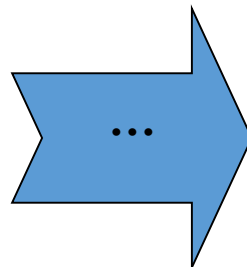
66
72
63
51
78
80
35
92
86
98

第 2 遍扫描



第 2.8 次比较结果

66
63
51
72
78
35
80
86
92
98





冒泡排序：优化 (C06-02b)

◆优化思路：在冒泡排序的某一遍过程中，如果没法发生任何元素交换，则表明数组已经有序，可以提前结束

//整数排序：升序排列

```
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;
            }
}
```

//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; //默认没有元素交换
        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; //发生交换
            }
        }
        if (flag == 0) break;
    }
}
```

原始	第一遍
1	1
5	3
3	5
7	7
8	8

还需要再继续比较吗？



排序算法实现中的两个问题

◆数组作为形参中，实参数组中元素的值可以通过函数改变

✓ `void bubbleSort(int a[], int n)`

✓ 在冒泡排序函数中，修改形参数组a中的值，实参的值同步修改

✓ 不同于普通变量，数组元素传递实参时，实际上传递是元素的地址，函数内部直接通过地址修改数组中元素的值

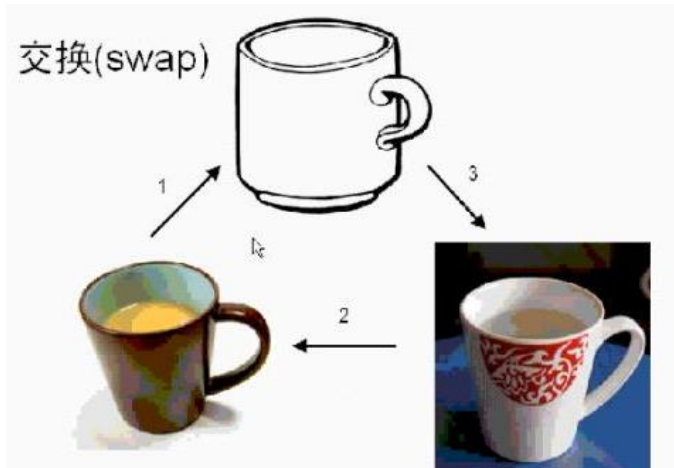
◆两个元素的交换：需要借助第3个元素实现

✓ `a[i]` 和 `a[i+1]` 的值互换

➤ `hold = a[i];`

➤ `a[i] = a[i+1];`

➤ `a[i+1] = hold;`





补充：冒泡排序的性能分析

◆比较次数

- ✓最好情况：改进前 $O(n^2)$ ，改进后 $O(n)$
- ✓最坏情况：改进前 $O(n^2)$ ，改进后 $O(n^2)$

◆交换次数

- ✓最好情况：0次
- ✓最坏情况： $n-1+n-2+\dots+2+1$
 $=n(n-1)/2$

◆时间复杂度

- ✓ $O(n^2)$

//优化的冒泡算法

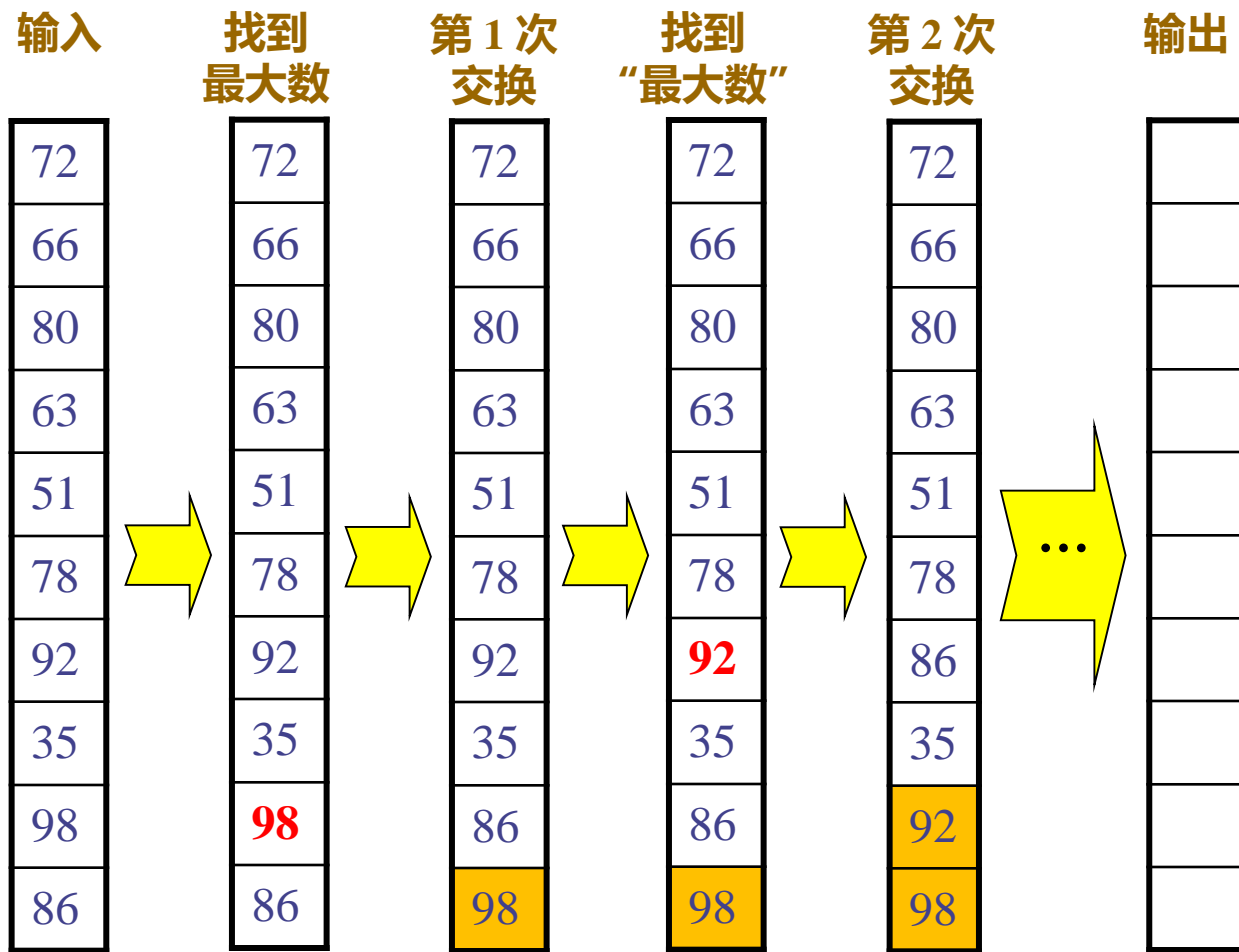
```
void bubbleSort(int a[], int n){
    int i, j, hold, flag;
    for(i = 0; i < n-1; i++){
        flag = 0; //默认没有元素交换
        for( j = 0; j < n-1-i; j++){
            if(a[j] > a[j+1]){
                hold = a[j];
                a[j] = a[j+1];
                a[j+1] = hold;
                flag = 1; //发生交换
            }
        }
        if (flag == 0) break;
    }
}
```



6.3.2 选择排序

◆ 算法原理

- ✓ 从 n 个数中找出最大（小）者，与第 n 个数交换位置
- ✓ 从剩余的 $n-1$ 个数中再找出最大者，与第 $n-1$ 个数交换位置
- ✓
- ✓ 一直到剩下最后一个数





选择排序：算法实现 (C06-03)

//对数组中的n个元素进行选择排序

```
void selectSort(int x[], int n) {  
    int i, j, temp;  
    for(i = n; i > 1; i--) {  
        j = max(x, i); // i 个数中找最大  
        temp = x[j]; //记录最大数  
        x[j] = x[i-1];  
        x[i-1] = temp; //最大数挪到最后  
    }  
}
```

//找出数组前n个元素中最大值，返回最大元素下标

```
int max(int x[], int n) {  
    int i, j=0; //默认第0个元素最大  
    for(i=1; i<n; i++) //跟后面的元素比较  
        if(x[i] > x[j]) j = i;  
    return j;  
}
```

输入

72
66
80
63
51
78
92
35
98
86

找到
最大数

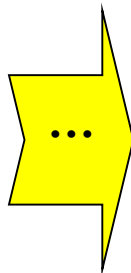
72
66
80
63
51
78
92
35
98
86

第1次
交换

72
66
80
63
51
78
92
35
86
98

找到
最大数

72
66
80
63
51
78
92
35
86
98



- 每次从有 i 个数据的数组 x 中选出最大 (小) 的数 $x[j]$
- $x[j]$ 和 $x[i-1]$ 进行交换
- 书上的实现本质一样，读者自行体会



选择排序：算法实现（小优化）

```
//对数组中的n个元素进行选择排序
void selectSort(int x[], int n) {
    int i, j, temp;
    for(i = n; i > 1; i--) {
        j = max(x, i); // i 个数中找最大
        temp = x[j]; //记录最大数
        x[j] = x[i-1];
        x[i-1] = temp; //最大数挪到最后
    }
}

//找出数组前n个元素中最大值，返回最大元素下标
int max(int x[], int n) {
    int i, j=0; //默认第0个元素最大
    for(i=1; i<n; i++) //跟后面的元素比较
        if(x[i] > x[j]) j = i;
    return j;
}
```



```
void selectSort(int x[], int n) {
    int i, j, temp;
    for(i = n; i > 1; i--) {
        j = max(x, i); // i 个数中找最大
        //最后一个元素最大，不交换
        if(j != i-1)
        {
            temp = x[j];
            x[j] = x[i-1];
            x[i-1] = temp;
        }
    }
}

int max(int x[], int n) {
    int i, j=0;
    for(i=1; i<n; i++)
        if(x[i] > x[j]) j = i;
    return j;
}
```



补充：选择排序的性能分析

◆比较次数

✓最好情况： $n(n-1)/2$

✓最坏情况： $n(n-1)/2$

◆交换次数

✓最好情况：0次

✓最坏情况： $n-1$ 次

◆时间复杂度

✓ $O(n^2)$

//对数组中的n个元素进行选择排序

```
void selectSort(int x[], int n) {  
    int i, j, temp;  
    for(i = n; i > 1; i--) {  
        j = max(x, i); // i 个数中找最大  
        temp = x[j]; //记录最大数  
        x[j] = x[i-1];  
        x[i-1] = temp; //最大数挪到最后  
    }  
}
```

//找出数组前n个元素中最大值，返回最大元素下标

```
int max(int x[], int n) {  
    int i, j=0; //默认第0个元素最大  
    for(i=1; i<n; i++) //跟后面的元素比较  
        if(x[i] > x[j]) j = i;  
    return j;  
}
```



使用排序算法

```
#include <stdio.h>
#define Num(x) (sizeof(x)/sizeof(x[0]))
int max(int[], int);
void selectSort(int[], int);
void bubbleSort(int[], int);
int main()
{
    int i, a[] = {21, 10, 4, 8, 12, 6,
                  86, 68, 45, 39};
    for (i = 0; i < Num(a); i++)
        printf("%4d", a[i]);
    printf("\n");
    selectSort(a, Num(a));
    //bubbleSort(a, Num(a));
    for (i = 0; i < Num(a); i++)
        printf("%4d", a[i]);
    printf("\n");
    return 0;
}
```

```
void selectSort(int x[], int n){
    int i, j, temp;
    for (i = n; i > 1; i--){
        j = max(x, i);
        temp = x[j];
        x[j] = x[i - 1];
        x[i - 1] = temp;
    }
}

int max(int x[], int n){
    int i, j;
    j = 0;
    for (i = 1; i < n; i++)
        if (x[i] > x[j])
            j = i;
    return j;
}
```

```
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 (flag == 0) break;
    }
}
```

提示：不同于简单变量，数组作为参数传递时，并不是把数组中每个元素的值传递给形参，而是**直接传递数组的首地址**，通过该地址函数内部直接访问数组实际的值



冒泡排序 VS 选择排序

冒泡排序

◆比较次数

- ✓最好情况：改进前 $O(n^2)$, 改进后 $O(n)$
- ✓最坏情况：改进前 $O(n^2)$, 改进后 $O(n^2)$

◆交换次数

- ✓最好情况：0次
- ✓最坏情况： $n(n-1)/2$

◆时间复杂度

- ✓ $O(n^2)$

选择排序

◆比较次数

- ✓最好情况： $n(n-1)/2$
- ✓最坏情况： $n(n-1)/2$

◆交换次数

- ✓最好情况：0次
- ✓最坏情况： $n-1$ 次

◆时间复杂度

- ✓ $O(n^2)$

思考1：哪个算法更“快”些？

思考2：算法稳定性？
(稳定排序算法：若两个数 $a[i]$ 与 $a[j]$ 相等，排序完成后这两个数的**相对顺序不变**，则这个排序算法称为稳定的)



冒泡排序 VS 选择排序：稳定性问题

3	2^1	2^1	2^1	1	1
2^1	2^2	2^2	1	2^1	2^1
2^2	3	1	2^2	2^2	2^2
5	1	2^3	2^3	2^3	2^3
1	2^3	3	3	3	3
2^3	5	5	5	5	5

冒泡

- 交换次数多, “慢”
- 稳定的排序

3	3	1	1	1	1
2^1	2^1	2^1	2^3	2^2	2^2
2^2	2^2	2^2	2^2	2^3	2^3
5	2^3	2^3	2^1	2^1	2^1
1	1	3	3	3	3
2^3	5	5	5	5	5

选择

- 交换n-1次, “快”
- 不稳定的排序



6.3.3 查找

◆查找

✓寻找数组中某个元素的过程，即：确定数组中是否包含其关键字
等于给定值的数据元素

◆方法：顺序查找、折半查找

◆应用：根据学号查找学生，根据成绩挑学生， ...

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

$\text{key} == a[i]$



顺序查找

◆顺序查找（线性查找）算法

✓从下标0开始顺序扫描数组，**依次**将数组中每个元素与查找关键字相比较，若当前扫描的元素与查找键相等，则查找成功，返回索引；否则，查找失败，即查找表中没有要查找的记录

◆时间复杂度： $O(n)$

◆适用：小型数据查找，对大数组，顺序查找的**效率较低**（尤其对需要频繁查找的情况）

◆优点：算法简单，对查找对象没有要求

key: 3

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

顺序查找法

key list

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

00:00 / 00:17

2x 标清 字幕



顺序查找：算法实现

◆C06-05：顺序查找

- ✓被查找的数据范围：int x[]
- ✓需要查找的关键字：int key
- ✓查找方法：顺序查找，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

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

key == a[i]

```
//在长度为SIZE的数组x中查找key，返回元素的位置
//查找失败，则返回-1
int find(int x[], int key, int SIZE)
{
    int j;
    for ( j = 0; j < SIZE; j++ )
        if ( x[j] == key )
            return j;
    return -1;
}
```



折半查找（二分查找）

◆前提条件：查找的数组是有序的（升序或降序）

◆折半查找算法（以升序数组为例）

✓将待查找的关键字key，首先和查找表中间位置记录的关键字相比较，如果相等，查找成功；如果key较小，则在查找表的前半个子表中继续折半查找；如果key较大，则在查找表的后面半个子表中继续折半查找。不断重复上述过程，直到查找成功或失败

◆时间复杂度： $O(\log_2 n)$

◆优点：查找效率高





折半查找

◆过程解析（升序数组）

- ✓(1) 找到数组的中间位置 $a[middle]$, $middle = (low+high)/2$, 将其与查找键 key 比较, 若相等, 则已找到查找键, 返回该元素的数组下标, 否则将问题简化为**查找一半数组**, 执行下一个步骤;
- ✓(2) 如果查找键小于数组的中间元素, 则查找数组的前半部分 $0 \sim middle-1$, 否则查找数组的后半部分 $middle+1 \sim high$;
- ✓(3) 如果查找键不是指定子数组的中间元素, 则对原数组的四分之一**重复如上步骤**, 直到查找键等于指定子数组的中间元素或子数组只剩下一个元素且不等于查找键（表示找不到这个查找键）为止。

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]	a[14]	a[15]
1	3	4	7	9	10	13	34	35	37	46	49	50	53	59	80



折半查找效率分析

◆最糟糕的情况下，查找1024个元素的数组只要进行10次比较

0趟：(1024=2¹⁰): a[0] a[1] a[2] a[3] a[510] a[511] a[512] a[1023]
1趟：(512=2¹⁰-1): a[0] a[1] a[2] a[3] ... a[255] ... a[510] a[512] a[1023]
2趟：(256=2¹⁰-2): ...
.....
9趟：(2=2¹⁰-9): ...
10趟：(1=2¹⁰-10): ...

◆每查找一次，排除所查找数组的一半元素，即除以2

◆查找10亿个元素的数组：

✓顺序查找，10亿次比较；折半查找：30次比较

◆折半查找需要有序数组，排序成本比单个的数组元素查找要高得多，如果数组需要多次高速查找，则先对数组进行排序再查找

◆查找次数： $\log_2 n$



折半查找：算法实现（假设数组元素是升序排列）

```
//折半查找，递归实现：数组b的low~high位置查找key
int recBinFind(int b[], int key,
               int low, int high){
    int mid;
    if( low > high )//递归出口，没有可查找的元素
        return -1;

    countFind++; //全局变量，记录查询次数
    mid = (low + high)/2;//中间位置
    if( key == b[mid] ) //找到了，退出
        return mid;
    else if( key > b[mid] )//递归查后半部分
        return recBinFind(b,key,mid+1, high);
    else //递归查前半部分
        return recBinFind(b,key,low, mid-1);
}
```

折半查找：递归实现

```
//折半查找，循环实现：数组b的low~high位置查key
int binFind(int b[], int key,
            int low, int high) {
    int mid;
    while( low <= high ){//还有要查找的元素
        countFind++;//全局变量，记录查询次数
        mid = (low + high)/2;//中间位置
        if( key == b[mid] )//找到了，退出
            return mid;
        else if (key < b[mid])//前半部分查找
            high = mid-1;
        else //后半部分查找
            low = mid+1;
    }
    return -1;//没找到
}
```

折半查找：非递归实现（循环）



折半查找：算法实现（应用）

```
#include <stdio.h>
#define LEN 1000
int binFind(int b[], int key, int low, int high); //折半查找，循环实现
int recBinFind(int b[], int key, int low, int high); //折半查找，递归实现
int countFind = 0; //全局变量，统计查找次数
int main(){
    int a[LEN], key, result, i;
    //构造查询数组
    for( i=0; i < LEN; i++) a[i] = 2*i;
    printf("Enter integer search key: ");
    scanf("%d", &key);
    //进行折半查找
    result = recBinFind(a, key, 0, LEN-1);
    //result = binFind(a, key, 0, LEN-1);
    if(result != -1)
        printf("Found, it's a[%d]", result);
    else
        printf("Key not found");
    printf("\nfind times: %d", countFind);
    return 0;
}
```

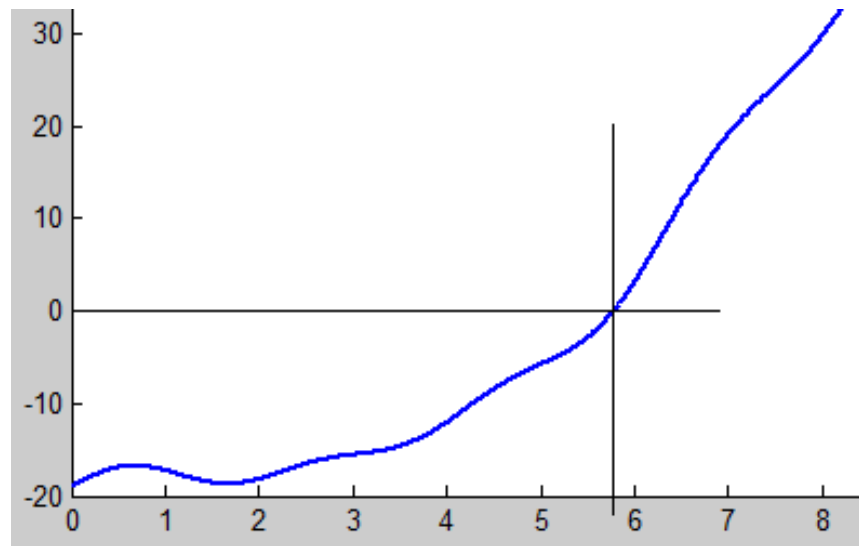
折半查找非常重要，一定掌握！

“折半查找都不会？！不招！”
(来自某知名互联网企业的技术主管)



折半查找的应用：方程求解

- ◆应用：利用折半查找对单调函数方程求根
- ◆当无法直接求出解析解时，可以使用折半查找，遍历方程可能的根
- ◆C06-07：方程 $f(x)$ 定义如下，求该方程的根
 - ✓ $f(x) = 2\sin(x) + \sin(2x) + \sin(3x) + (x-1)^2 - 20 = 0$
 - ✓已知： $f(4) < 0$, $f(8) > 0$ ，且 $f(x)$ 在 $x \in [4, 8]$ 之间是单调递增的





C06-07: 利用折半查找求解方程

```
#include <stdio.h>
#include <math.h>
//方程函数, 计算方程的值
double f(double x);
//double判0函数, x的是否为0
int isZero(double x);
//折半查找函数, 在low~high区间查找方程的根
double binFind(double low, double high);
int main(){
    double ans = binFind(4.0, 8.0);
    printf("%.4f\n", ans);
    return 0;
}
double f(double x){
    return 2*sin(x)+sin(2*x)
        +sin(3*x)+(x-1)*(x-1)-20;
}
int isZero(double x){
    double eps = 1e-12;
    if(fabs(x)<=eps) return 1;
    else return 0;
}
```

```
double binFind(double low, double high)
{
    double mid;
    double value;
    while(low<=high){
        mid = (low+high)/2;
        value = f(mid);
        if(isZero(value)) return mid;
        else if(value > 0) high = mid;
        else low = mid;
    }
    return 0; //找不到解, 本题应该不会执行到此处
}
```




6.4 字符串与字符数组

- ◆字符串类型：以双引号括起来的零个或多个字符组成的有限序列
 - ✓如： "Hello, world", "A", "123456"等
- ◆不同于基本类型，C语言并没有提供字符串数据类型，而是通过字符数组来实现字符串
- ◆问题：采用数组存储元素时，如何记录实际存储的元素个数？
 - ✓程序必须准确获知数组中实际存储的元素个数，普通数组是由程序员单独维护一个整型变量（如：n），随时更新为实际元素个数
 - ✓在字符串类型中，这种方式显然是不合适的！



C语言中的字符串

- ◆ C语言约定：以**字符串结束标志符'\0'**，表示字符串结束，
- ◆ C语言在进行字符串处理时，在有效字符串最后，设置'\0'表示字符串结束
- ◆ C语言中各类字符串处理会自动根据\0来处理和显示字符串

"Hell, world"

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

"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



利用字符数组实现字符串

◆利用字符数组实现字符串

✓(1) 显式定义字符串长度, `char s1[64];`

➤最多可以存储63个字符, 最后至少要留一个位置存储\0

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

➤度由编译器根据字符串中实际数组长度, 共11个字符+结尾隐藏'\0',

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

◆字符串数组**最后位置补充\0**后, 就可以当字符串使用

◆字符串也可以**直接按照字符数组**使用, 注意最后有\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



常见错误：利用字符数组实现字符串

- ◆字符串本质上是用数组来实现，要满足数组使用要求
 - ✓无法直接使用运算符整体赋值、比较等操作字符串
 - ✓作为字符串使用时，最后必须有冗余的空间存储\0
- ◆C语言针对字符串提供了各类辅助操作（字符串处理函数）
 - ✓使用%s，可以整体输入（scanf）、输出（printf）字符串，输入的字符串会自动补\0，输出时自动以\0作为结束符
 - ✓初始化的时候，可以直接用字符串字面量初始化数组

```
char s1[3] = "first"; //逻辑错误，编译器不报错，但空间不足以存储字符串，也无法存储\0
printf("%s", s1);      //s1中没有有效的\0，输出结果不确定
s1 = "second";         //s1本质上是数组，不能对数组整体赋值
char s2[10];
s2= "first";           //s2也是数组，不能整体赋值
s2[0] = "first";       //s2[0]是s2数组的一个成员，是字符类型，不能整体赋值字符串
```



示例：利用字符数组实现字符串

```
//6个全局字符数组
char G1[10]; //定义长度10的全局字符数组G1
char G2[10] = "Hi"; //定义长度10的全局字符数组G2,初始化为字符串"Hi"
char G3[10] = {'H', 'i'}; //定义长度10的全局字符数组G3,初始化2个字符'H','i'
//定义长度10的全局字符数组G4,初始化3个字符'H','i','\0',可做字符串使用
char G4[10] = {'H', 'i', '\0'};
char G5[] = "Hi"; //定义全局字符数组G5,初始化为字符串"Hi",长度为3
char G6[] = {'H', 'i'}; //定义全局字符数组G6,初始化为2个字符'H','i',长度为2
int main(){
    //6个局部字符数组
    char La[10]; //定义长度为10的局部字符数组La
    char Lb[10] = "Hi"; //定义长度为10的局部字符数组Lb,初始化为字符串"Hi"
    char Lc[10] = {'H', 'i'}; //定义长度为10的局部字符数组Lc,初始化2个字符
    // 'H','i',后面的自动补\0,可做字符串使用
    //定义长度10的局部字符数组Ld,初始化3个字符'H','i','\0',可做字符串使用
    char Ld[10] = {'H', 'i', '\0'};
    char Le[] = "Hi"; //定义局部字符数组Le,初始化为字符串"Hi",长度为3
    char Lf[] = {'H', 'i'}; //定义局部字符数组Lf,初始化为2个字符'H','i',长度为2
    ...
}
```

思考：利用sizeof测试各个变量的大小！



利用字符数组实现字符串

◆字符串定义的三大条件:

✓数组

✓char类型

✓'\0' 结尾 (没有\0结尾的字符数组不能作为字符串使用)

◆1. 直接按照字符数组的方式操作字符串

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

判断字符串结
束的通常用法

◆2. 使用专门的字符串处理函数



6.5 字符串处理函数

- ◆ C语言提供了一系列字符串处理功能函数，所有这些函数只能处理字符串（即一定要有**\0结尾**）
- ◆ `<stdio.h>`，提供各类字符串输入输出转换函数
 - ✓ `scanf`、`printf`中提供%s操作字符串
 - ✓ `puts`、`gets`字符串输入和输出函数
 - ✓ `sscanf`、`sprintf`提供字符串和普通数据类型的转换
- ◆ `<string.h>`，以str开头，提供各类操作函数
 - ✓ 字符串长度：`strlen`
 - ✓ 字符串拷贝：`strcpy`、`strncpy`
 - ✓ 字符串连接：`strcat`、`strncat`
 - ✓ 字符串比较：`strcmp`、`strncmp`
 - ✓ 字符串查找：`strstr`、`strchr`、`strrchr`

- ✓ 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，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！



字符串的输出输入函数

◆scanf和printf的%s

- ✓ 输入时以空白字符分隔，不读入空白字符，存储空间足够时，自动补\0
- ✓ 注意scanf里面的变量不需要取地址（&），因为数组本身就是地址

◆gets和puts

- ✓ gets输入时以换行符分隔，读入整个一行（换行符也读入，但会自动转换为\0），存储空间足够时，自动将换行符转换为\0
- ✓ puts输出时，自动在后面补充换行符\n

◆区别

- ✓ scanf读入串中无法含空白字符；gets读一行（含换行，并转换为\0），串中可以有空白字符
- ✓ 输入结束方式不一致，scanf是判断EOF，gets是判读NULL
- ✓ 输出时puts自动补回车，printf原样输出

```
char s[100];  
//使用scanf输入字符串  
while(scanf("%s",s) != EOF)  
    printf("%s\n", s);
```



```
char s[100];  
//gets输入一行字符串，存储到s中，NULL表示输入结束  
while (gets(s) != NULL)  
    puts(s); //会自动添加换行，不需要换行
```




C06-08: 字符串倒置

```
#include <stdio.h>
#include <string.h>
//字符串逆序函数
void strrev(char[]);
int main()
{
    char a[100];
    int i, hi = 0, low = 0;
    gets(a); //输入, 长度不超过99
    puts(a); //输出字符串
    strRev(a); //逆序字符串
    puts(a); //输出逆序后的字符串

    return 0;
}
```

```
//字符串逆序函数
void strRev(char s[])
{
    int high = 0, low = 0;
    char temp;
    while (s[high] != '\0') //计算字符串长度
        high++;
    //将字符串前后字符互相交换
    //hi从最后一个有效字符开始, low从第一个有效字符开始
    for (high--; high > low; low++, high--)
    {
        temp = s[low];
        s[low] = s[high];
        s[high] = temp;
    }
}
```



字符串处理函数只能处理字符串

◆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";  
//a不是字符串，相关操作会有问题  
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);
```

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)



作为字符串输入

◆标准输入都可以作为字符串输入，字符串输入可以保证输入和输出内容完全一致

```
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

输入相同

输入：the num is 1.23

输出：the num is 1.23

输出不同

这两个1.23是
否一样？

这里的1.23是
字符串



字符串的输入输出函数

输入:



```
char a[5], b[5];
```

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

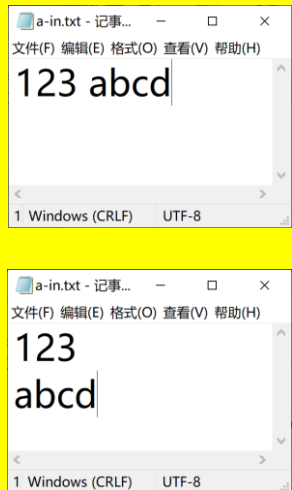
输出:

123 abcd

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

123
abcd

输入:



```
char a[5], b[5];
```

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

输出: ?

123 abcd

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

123

“123 abcd”是一行，读入a，a越界访问，此处运气好，结果正确，但有隐患。puts输出a，然后加 \n
读到EOF，b没有读入任何数据，保持原样

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。

“123”和“abcd”分别读给b和a并进行相应输出，a后加 \n

“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自动添加的 \n。



C06-09: 字符串输入越界问题

◆scanf和gets输入字符串时，并不做越界检查，存在安全隐患

```
#include <stdio.h>
#include <string.h>
int main(){
    char key[] = "hello"; //密码
    char str[10] = {0};
    printf("&key = %x, &str = %x\n\n", &key, &str);
    while (1) {
        printf("Password(hint: %s):", key);
        gets(str); //通过越界破坏密码
        if (!strcmp(key, str)) {
            printf("Correct password!\n"); break;
        } else printf("Incorrect, try again!\n\n");
    } return 0;
}
```

&key = 65fe1a, &str = 65fe10

Password(hint: hello):123456
Incorrect, try again!

第一次尝试错误

Password(hint: hello):1234567890123456
Incorrect, try again!

第二次修改了key

Password(hint: 123456):123456
Correct password!

第三次成功破解

0x61fe1a

h	e	l	l	o	\0
---	---	---	---	---	----

gets函数读入第二次输入的长字符串，由于不检查str的大小，使得str的内容冲掉了原来key所在的内存，修改了key；于是，黑客，就“破解”（修改）了密码！！

0x61fe1a

key

0x61fe10

str

1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	\0		
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	----	--	--



字符串转换函数：sprintf

◆将多个普通变量输出为sprintf

- ✓printf将不同变量输出到屏幕上，屏幕输出的本质上就是字符串
- ✓sprintf用法与printf一样，只不过不是输出到屏幕，而是**直接保存到字符串**中
- ✓比printf多一个参数（第一个参数），就是要存储的字符串数组，后面参数的使用与printf完全

//字符串转换函数，将其他变量转换为字符串

```
int sprintf(char *buf, char *format [, argument]...);
```



C06-10: 根据输入整数决定输出小数点位数

◆C06-10: 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

◆问题分析

✓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)

✓利用`sprintf`函数, 根据输入的 m 构造格式字符串



使用sprintf生成字符串

```
#include <stdio.h>
int main()
{
    int m; // 小数点位数
    double x; // 输入x
    char format[32]; // 要存储的格式串
    scanf("%lf%d", &x, &m);
    // 利用sprintf生成格式串
    sprintf(format, "%%.%df\n", m);
    // 利用生成的格式串控制输出格式
    printf(format, sin(x));
}
```

若输入m位3，则"%%.%df\n"变为"**%.3f\n**"，且该字符串存入字符数组format，利用该字符串控制输出格式

说明：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，任何可以输出到屏幕的数据都可以转换为字符串



字符串转换函数：sscanf

- ◆从字符串中转换出各种类型的变量，与sprintf对应
 - ✓scanf从键盘读入一串符号，将其存入到不同变量中
 - ✓sscanf用法与scanf一样，只不过不是从键盘输入，而是直接从一个给定的字符串中读入
 - ✓比scanf多一个参数（第一个参数），就是要存储的要读取的字符串，后面参数的使用与printf完全

//字符串转换函数，从字符串中提取各种变量

```
int sscanf(const char *buf, char *format [, arg]...);
```



C06-11: 从一行字符串中提取变量

◆C06-11: 提取日期和时间

- ✓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
`17/Apr/2018:10:28:28 +0800`，表示北京时间2018年4月17日10时+28分28秒。给出一个这种格式表示的字符串，提取其中的每一项，并在屏幕上逐行显示出来

◆问题分析

- ✓定义各种类型的输入变量
- ✓编写格式控制符，从字符串中读入变量的值



使用sscanf从字符串中提取变量

```
#include <stdio.h>
int main()
{
    int day, year, h, m, s;
    char mon[4], zone[6];
    char buf[] = "17/Apr/2018:10:28:28 +0800"; //要转换的字符串
    //从字符串中, 按照指定的格式读入日期的各个部分
    sscanf(buf, "%d/%3c/%d:%d:%d:%d %s",
           &day, mon, &year, &h, &m, &s, zone);
    mon[3] = '\0'; //理解此句话的作用
    printf("%d\n%s\n%d\n%d\n%d\n%d\n%s",
           year, mon, day, h, m, s, zone); //按照格式输出
    return 0;
}
```



2018
Apr
4
17
28
28
+0800



字符串长度函数：strlen

◆返回字符串s的**实际字符个数**，不包括终止符 '\0'

✓仅支持\0结束的字符串

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

◆注意：与sizeof的不同

✓sizeof返回变量自身的长度（字节数），任何数据类型均可以

✓如果是数组，返回数组的长度（字节数）

```
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'};
```

思考：

代码中的变量，使用sizeof(G#)分别是多少？

哪些可以使用strlen(G#)，结果是多少？

(# 表示左边代码中的数字1~6)



字符串拷贝函数：strcpy和strncpy

- ◆字符串采用字符数组实现，无法直接使用赋值表达式（=）操作
- ◆使用库函数实现字符串的拷贝（赋值）

```
//将字符串src复制到字符数组dest 中，返回 dest  
//注意：dest的长度应足够长以能够放下src的内容  
char *strcpy(char dest[ ], const char src[ ]);
```

```
//将字符串src中最多n个字符复制到字符数组dest中，返回dest  
//注意：n<=strlen(src)时，只把src前n个元素复制到dest，  
//此时需要手动在dest末尾添加'\0'（即执行 dest[n] = '\0';）  
char *strncpy(char dest[ ], const char src[ ], size_t n);
```



字符串拷贝函数

```
char x[] = "1234 abcd ABC123";  
char y[25], z[25];  
printf("Source: %s\n", x);  
printf("Dest_1: %s\n", strcpy(y, x));  
strncpy(z, x, 11); //无法拷贝\0  
z[11] = '\0';      //手动补充\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, 后面的保留, 后面\0结束

```
strncpy(z, "abc", 6);  
printf("Dest_4: %s\n", z); //abc  
for (i = 0; i < 25; i++)  
    putchar(z[i]);
```

如果继续执行这几行, 输出什么?


abc bcd A 棹a ?a 捞



strncpy的应用实例

```
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);
```

连续运行三次的结果



```
命令提示符
C:\alac\example>a6-test
Z1_ini: x庾      棹a
Z2_ini:
Z1_cpy: 1234 abcd Ao鉕 @
Z2_cpy: 1234 abcd A
Z3_cpy: 1234 abcd A

C:\alac\example>a6-test
Z1_ini: ?      棹a
Z2_ini:
Z1_cpy: 1234 abcd Ao鉕 @
Z2_cpy: 1234 abcd A
Z3_cpy: 1234 abcd A

C:\alac\example>a6-test
Z1_ini: 踴?      棹a
Z2_ini:
Z1_cpy: 1234 abcd Ao鉕 @
Z2_cpy: 1234 abcd A
Z3_cpy: 1234 abcd A
```

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



字符串连接函数：strcat、strncat

- ◆将两个字符串连接成一个字符串，即将一个字符串的内容追加到另一个字符串的后面，构成一个新的字符串

- ✓strcat

```
//将字符串src连接到字符串dest已有字符串后面（追加）  
char *strcat(char dest[ ], const char src[ ]);
```

- ✓区别于strncpy，strncat会自动补\0，但要注意空间一定要足够

```
//将字符串src中最多前n个字符添加到字符串dest后面，并自动补\0  
char *strncat(char dest[ ], const char src[ ], size_t n);
```




strcat和strncat的应用

```
char x[] = "1234 abcd ABC123";
char y[20] = "", z[20] = "";
printf("Source: %s\n", x);
//将x连接到y的后面
printf("Dest_1: %s\n", strcat(y, x));
//将x的前11个字符连接到z的后面
strncat(z, x, 11);
printf("Dest_2: %s\n", z);
//在z的后面连接上字符串"abc",不超过6个字符
strncat(z, "abc", 6);
printf("Dest_3: %s\n", z);
//在z的后面连接上字符串"123456789abcdef0",不超过12个字符
//此处由于z中已有14个字符,追加12个后越界了,代码存在问题
strncat(z, "123456789abcdef0", 12);
z[19] = '\0';//在z的后面可手工补\0
printf("Dest_4: %s\n", z);
```

输出:

Source: 1234 abcd ABC123

Dest_1: 1234 abcd ABC123

Dest_2: 1234 abcd A

Dest_3: 1234 abcd Aabc

Dest_4: 1234 abcd Aabc12345



字符串比较函数：strcmp、strncmp

◆比较两个字符串的大小，按照ASCII的顺序比较

```
//比较字符串s1与s2，  
//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 strcmp(char s1[ ], char s2[ ]);
```

```
//比较字符串s1与s2的前n个字符  
//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);
```



strcmp和strncmp的应用

```
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

返回的是字符编码值的差!
有些系统可能输出1和-1, 与编译器的实现相关!



字符串查找函数：strchr、strrchr和strstr

◆在字符串中查询某个字符或子字符串的位置

✓具体使用方法学习指针后再理解

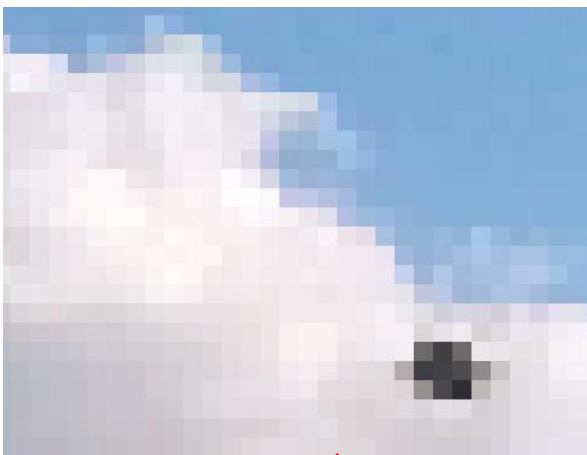
```
//从前往后，查询字符串s中是否存在字符c  
//返回第一次出现字符c的位置指针，如果不存在，则返回NULL  
char *strchr(char s[], int c);  
//从后往前，查询字符串s中是否存在字符c  
//返回最后出现字符c位置指针，如果不存在，则返回NULL  
char *strrchr(char s[], int c);
```

```
//在字符串中查询是否存在子字符串sub_str  
//返回中第一次出现的位置指针  
//如果不存在，则返回NULL。  
char *strstr(char *s, char *sub_str);
```



6.6 二维数组与多维数组初步

- ◆一维数组（单下标）：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，...

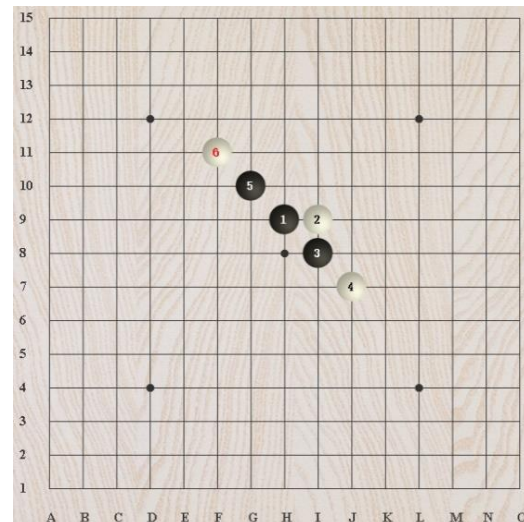


	A	B	C	D	E	F	G	H
1	排名	用户	学号	得分	罚时	A	B	C
2						922/1235	916/1033	828/927
3	1	刘	#####	800	13:57:51	0:51:09	1:05:29	0:57:33
4	2	刘	#####	800	15:31:30	0:27:08(+1)	1:58:11	1:49:46
5	3	校	#####	800	16:59:24	1:26:11(+3)	1:32:18	1:43:33
6	4	俊	#####	800	21:33:04	1:16:00	1:52:45	2:43:37(+1)
7	5	柯	#####	800	23:30:58	3:18:20(+4)	4:28:36(+1)	3:34:32
8	6	玮	#####	800	24:03:40	2:42:19	3:14:24	2:06:48(+1)
9	7	超	#####	800	24:32:52	3:24:27(+3)	1:29:11	2:56:31(+1)
10	8	昕	#####	800	25:43:39	0:52:54(+3)	0:56:25	1:48:14
11	9	一	#####	800	25:53:40	3:43:44(+1)	2:22:09	2:44:14
12	10	怀	#####	800	27:57:44	3:47:20(+2)	2:20:19	2:51:30

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



	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		





二维数组的定义和初始化

◆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

✓如：int a[2][3]; //2行3列，2x3=6个数组元素

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

◆二维数组初始化：数据类型 数组名[行数][列数]={初始化数据}

// 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[2][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”



二维数组的存储方式

◆内存本质上是一维结构，二维数组存储是需要按照行优先的顺序依次存储

✓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)

◆利用两重for循环，依次访问二维数组中的元素

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

逻辑结构：二维结构



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

物理实现：一维结构
(行优先)

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



二维数组的存储方式

- ◆ 二维数组的存储是行优先的。设定义数组

`int a[2][3];` // 行 i : $0 \leq i < 2$; 列 j : $0 \leq j < 3$

- ◆ 数组 a 的存储方式如下图所示

- ◆ 注意：采用行优先存储方式

✓ 由于C语言不对数组下标进行检查，访问数组元素 $a[1][0]$ 也可以写为 $a[0][3]$ (并不越界)，为了提高程序的可读性和维护性，建议访问数组元素时，行、列编号都限制在定义时的行、列的范围内

内存 (Memory)									
	60FEF8 a: a[0][0]	60FEFC a[0][1]	60FF00 a[0][2]	60FF04 a[1][0]	60FF08 a[1][1]	60FF0C a[1][2]
.....					

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



二维数组本质上可以理解为嵌套的一维数组

◆ 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][]`
...
...

<code>a[0][]</code>	<code>a[0][0]</code>	<code>a[0][1]</code>		
<code>a[1][]</code>				
<code>a[2][]</code>			<code>a[2][2]</code>	
<code>a[3][]</code>			<code>a[3][2]</code>	
<code>a[4][]</code>				

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

◆ 由于字符串本身就是用字符数组实现的，如果要建立字符串的数组，则需要用到二维字符数组（一维字符串数组）

◆ 二维字符数组初始化

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

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

◆ 二维字符数组的引用

`a[0][0]`

<code>a[0]</code>	s	t	r	1	\0	\0	\0	\0
<code>a[1]</code>	s	t	r	2	\0	\0	\0	\0
<code>a[2]</code>	s	t	r	i	n	g	3	\0

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



二维数组应用：C06-12 星期几

◆C06-12：已知本月有n天，第x天是星期y，求下月k日是星期几，输出星期几的英文单词

```
#include <stdio.h>
//二维数组存储星期的英文单词
//列的长度比最长的单词长度+1
char weekName[][12] =
{
    "Sunday",
    "Monday",
    "Tuesday",
    "Wednesday",
    "Thursday",
    "Friday",
    "Saturday"};
```

```
int weekday(int, int, int, int);
int main(){
    int x, y, n, k, m;
    x = 1;    y = 0;    n = 31;
    printf("day of next month: ");
    scanf("%d", &k);
    m = weekday(x, y, n, k);
    printf("%s\n", weekName[m]);
    return 0;
}
int weekday(int x, int y, int n, int k)
{
    return (n - x + k + y) % 7;
}
```



二维数组应用：C06-13-最长行

◆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 maxlen = 0, len, tmp;
while (gets(arr[in]) != NULL)
{
    len = strlen(arr[in]);
    if (len > maxlen)
    {
        maxlen = len;
        tmp = in;
        in = longest;
        longest = tmp;
    }
}
printf("%d: %s\n", maxlen, 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。



二维数组应用：C06-14 输出每月的天数

◆C06-14：输入年和月，请输出本年本月的总天数

```
#include <stdio.h>
int days[2][13]={//二维数组，存储每个月的天数
    {0,31,28,31,30,31,30,31,31,30,31,30,31},
    {0,31,29,31,30,31,30,31,31,30,31,30,31}
};
int isLeap(int year){//判断闰年
    return ((year%4==0 && year%100!=0) || year%400==0);
}
int main(){
    int year,month;
    scanf("%d%d",&year,&month); //输入年月
    //直接从二维数组中获取当月的天数
    printf("%d年%d月有%d天\n",year, month, days[isLeap(year)][month]);
    return 0;
}
```



二维数组作为函数的参数

◆数组参数形式: `int a[][4]`

- ✓二维数组: 可省略行数, 但不能省略列数
- ✓多维数组: 只能省略第一维下标, 后面维的下标不能省略

◆为使函数处理n行的矩阵, 一般将行数n作为参数传递给函数

◆C06-15: 打印二维数组的值

//打印n行3列的数组

```
void printArr(int[][3], int n);
```

```
int main(){
```

```
    int arr1[2][3] = {{1,2,3}, {4,5,6}};
```

```
    int arr2[5][3] = {{1,2,3}, {4,5,6},  
                      {7,8,9}, {10,11,12}, {13,14,15}};
```

```
    printArr(arr1, 2); //打印2行3列的数组
```

```
    printArr(arr2, 5); //打印5行3列的数组
```

```
    return 0;
```

```
}
```

//打印n行3列的数组

```
void printArr(int a[][3], int n){
```

```
    int i, j;
```

```
    for (i=0; i<n; i++) {
```

```
        for(j=0; j<3; j++)
```

```
            printf("%d ", a[i][j]);
```

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

```
    }
```

```
}
```



C06-16: 成绩处理

◆有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
- ✓ (2) 每门课的平均成绩
- ✓ (3) 每个同学的平均成绩

ID	C语言	高数	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] =  
{  
    { 87, 96, 70 },  
    { 68, 83, 90 },  
    { 95, 93, 90 },  
    { 100, 81, 82 },  
    { 88, 63, 81 },  
    { 78, 87, 66 }  
};
```

输出示例：

```
                C语言  高数  大语  
highest grade: 100    96    90  
                id:    3      0      1  
lowest grade:   68    63    66  
                id:    1      4      5  
course average: 86.0   83.8   79.8  
student average:  
84.3  
80.3  
92.7  
87.7  
77.3  
77.0
```

更多问题：

- 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）？
- 求每门课的方差？
- 画出每门课的成绩段分布（直方图）？
-



C06-16: 成绩处理

◆部分代码，请自行完成

```
#define stu 6
#define course 3
int grade[stu][course] = {
    { 87, 96, 70 },    { 68, 83, 90 },
    { 95, 93, 90 },    { 100, 81, 82 },
    { 88, 63, 81 },    { 78, 87, 66 } };
int mark_maxmin[2][course]; /*第一行记录每门课的最高成绩或最低成绩，第二行是该成绩的学生id */
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
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 main(){
    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;
    }
    print_maxmin(mark_maxmin[0], "highest grade");
    print_maxmin(mark_maxmin[1], "id");
}

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;
    }
    print_maxmin(mark_maxmin[0], "highest grade");
    print_maxmin(mark_maxmin[1], "id");
}
```




✓ a[5]

✓ a[5][4]

✓ a[5][4][3]

➤ (应用：三维制作)

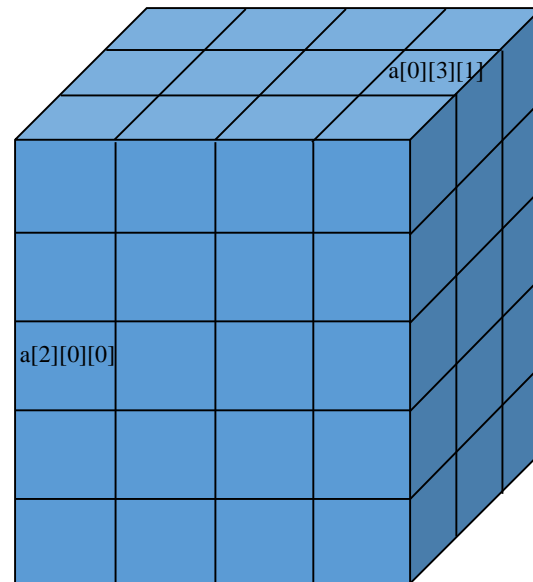
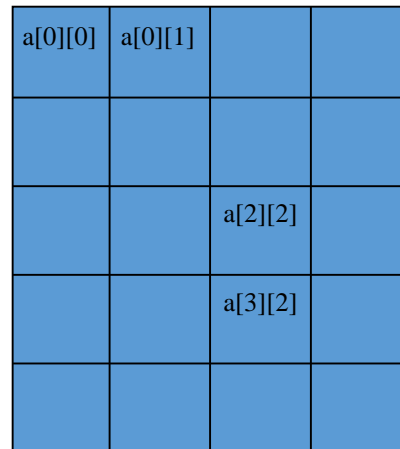
✓ a[X][Y][Z][?]

➤ (应用：三维动画制作， t 是第4维)

✓ a[M][N][K][L][P]

✓二维数组：一维数组，数组的每个成员都是一维数组

✓三维数组：一维数组，数组的每个成员都是二维数组





小结

◆数组中常见的错误

- ✓越界访问、整体操作、长度问题

◆数组作为函数参数

- ✓形参为数组的地址，函数内直接访问实参的值

◆数组的排序和查找

- ✓冒泡排序、选择排序、折半查找

◆字符串与字符数组

- ✓\0结束的字符串

◆字符串处理函数

- ✓输入输出、字符串拷贝、连接、比较、求长度、查找

◆二维数组以及多维数组

- ✓行优先的顺序存储、多维数组的访问使用