

# 思考

---

- ▶ 对于用户输入的一个矩阵，计算其特征值和特征向量，进而求其最大和第二大的特征值：

$$\begin{bmatrix} 5 & 2 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 0 & 0 & 1 & -2 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

double a11, a12, a13, a14, ..., a41, a42, a43, a44; ?

1. 4\*4矩阵这样定义，5\*5呢，100\*100，10000\*10000呢？
  2. 如果是N\*N呢，N由用户指定或程序计算确定
-

# 思考

---

- ▶ 对于用户输入的任意一个矩阵，计算其特征值和特征向量，进而求其最大和第二大的特征值
- ▶ 写一个通用的程序！
- ▶ 程序接受用户输入的任意大小的矩阵！



# 思考

---

▶ 如何表示矩阵，从而计算（大规模、稀疏）方程组的解？

▶ 如何表示线性代数中的向量：

$$\overrightarrow{a} = (a_1, a_2, \dots, a_n)$$

▶ 如何表示一个字符串

`str = "I am the king" !`

▶ 如何表示一个乃至全班同学(student)的：姓名、学号、年龄、籍贯、成绩等信息？

---



# 6 复杂数据的描述—构造数据类型

郭延文

2019级计算机科学与技术系

- 
- ▶ C语言供了用基本类型构造复杂类型的机制，以便派生出一些数据类型来描述复杂数据，包括数组、指针、结构体及联合等派生类型
  - ▶ 需要在程序中用系统提供的关键词或符号（比如[]，\*，struct等）先构造类型，然后才能定义相应的变量
  - ▶ 派生类型变量的存储方式和可取的值往往比较复杂，一般不直接参与基本（C/C++标准内置函数）操作，需要设计特别的算法来处理
-

# 复杂数据的描述—构造数据类型

---

- ▶ 数组及其应用
- ▶ 指针及其应用
- ▶ 字符串
- ▶ 结构体/联合及其应用

对应教材第**5**章的内容

# 思考

---

▶ 如何表示矩阵，从而计算（大规模、稀疏）方程组的解？

▶ 如何表示线性代数中的向量：

$$= (a_1, \overrightarrow{a}, \dots, a_n)$$

▶ 如何表示一个字符串

`str = "I am the king" !`

▶ 如何表示一个全班同学(student)的：姓名、学号、年龄、籍贯、成绩等信息？（先定义结构体，然后定义结构体数组或链表）

---



# 6 复杂数据的描述—构造数据类型

## 6.1 数组及其应用

郭延文

2019级计算机科学与技术系



# 数组及其应用

---

- ▶ 一维数组

- ▶ 可用来表示向量...

- ▶ 二维数组

- ▶ 可用来表示矩阵...

- ▶ 多维数组

- ▶ 数组的应用

- ▶ 基于数组的排序程序

- 
- ▶ 如何表示向量和矩阵这样的数据？用基本类型来表示数据的各个分量？
  - ▶ 变量数量太多；
  - ▶ 独立的变量 (`a11, a12, a13 ...`) 之间缺乏显式的联系，从而降低了程序的可读性和可维护性，不利于数据操作流程的设计

```
int u, v, w, x, y, z, s = 0;
scanf("%d%d%d%d%d%d", &u, &v, &w, &x, &y, &z);
s = u + v + w + x + y + z;
printf("%.2f\n", s / 6.0);
```

OR

```
int a1, a2, a3, a4, a5, a6, s = 0;
scanf("%d%d%d%d%d%d", &a1, &a2, &a3, &a4, &a5, &a6);
s = (a1 + a2 + a3 + a4 + a5 + a6) / 6.0;
...
```



用数组表示!

```
int a[6], s = 0;
for(int i=0; i < 6; i++)
{
    cin >> a[i];
    s += a[i];
}
cout << s / 6.0;
```

# C语言数组 (array)

---

- ▶ 数组用以表示**确定个数**的**同类型**数据按**一定次序**构成的数据群体
- ▶ 数据群体中的每个个体即数组中的每个**元素**
- ▶ 对于数组类型的数据，编译器将在内存中分配**连续的空间**来存储数组元素
- ▶ 通常情况下，对数组**不能进行整体操作**
- ▶ 关于数组长度
  - ▶ 数组长度（元素的个数， $>1$ ）一般为常量，以便在编译期间为数组分配内存空间
  - ▶ 从C99标准开始允许数组的长度为变量，但须在定义数组前确定该变量的值，以便编译器分配空间（DevC++(GCC)等开发环境允许数组长度为变量）

# 一维数组

---

- ▶ 一维数组是常见的数组形式，用来表示“向量”等这类元素同类型的数据。
- ▶ 一维数组 **类型的构造**
  - ▶ 一维数组类型由元素类型关键词、一个中括号和一个整数（表示数组的长度）构造而成：

```
typedef int A[6];
```

//A是由6个int型元素所构成的一维数组类型标识符

# 一维数组变量的定义

---

两种方式：

1. 先构造类型，再定义变量，便于定义多个同类型变量

```
typedef int A[6];
```

```
A a, b;    //定义了一个一维数组a和一个一维数组b
```

2. 构造类型的同时定义变量

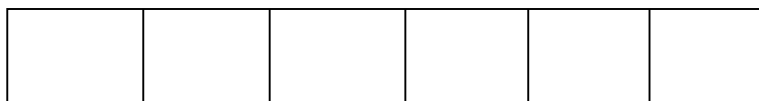
```
int a[6];    // int、[]和6构造了一个一维数组类型
```

```
    //并用该类型定义了一个一维数组a
```



- 
- ▶ 程序执行到数组定义处，系统内存即为该数组分配一定大小的空间，以存储其各元素的值。

`int a[6];`



- ▶ 一维数组名（标识符a），表示一维数组第一个元素在内存的起始位置，即`&a[0]`，是一个地址常量

- ▶ 数组所占内存空间的大小可以用sizeof操作符来计算：

```
int a[6];
```

```
printf(“%d \n”, sizeof(a));    //输出数组a所占内存字节数24
```



# 一维数组的初始化

---

- ▶ 用一对大括号把元素的初始值括起来
  - ▶ 比如, `int a[6] = {1, 2, 3, 4, 5, 6};`
- ▶ 如果每个元素都进行了初始化, 则数组长度可以省略
  - ▶ 比如, `int a[] = {1, 2, 3, 4, 5, 6};`
- ▶ 初始化表中的初始值个数可以少于数组长度, 这种情况下, 后部分数组元素初始化成0
  - ▶ 比如, `int a[6] = {1, 2, 3};`  
// 前三个元素为1、2、3, 后三个元素均为0



# 一维数组的操作

## ▶ 不能进行整体操作

▶ `s += a;` //不能这样求和

## ▶ 一般采用循环流程依次操作数组的元素。

▶ 访问数组元素的格式为：<数组名>**[下标]**

▶ 比如，`a[i]`表示自`a[0]`开始的第`i+1`个元素 (`i`: 0~5)

`int a[6];`

`a[0]`、`a[1]`、`a[2]`、`a[3]`、`a[4]`、`a[5]` // 注意：第一个元素为`a[0]`

▶ **[ ]** 为下标操作符，不表示数组的长度

▶ “**下标**”为整型表达式，常常表现为循环变量，下标为**0**时表示第一个元素，下标为`N-1`时表示长度为`N`的数组的最后一个元素。

▶ 下标`N`不表示数组中的任一元素 (`a[6]`**越界**)。C语言一般不对下标是否越界进行检查，要“小心”处置这个问题。

## ▶ 对于数组元素的依次访问通常又叫对数组的遍历

# 例：求某班同学的平均成绩

```
#include <iostream>
using namespace std;
const int N = 50;
int main()
{
    int sum = 0, score[N];
    float average;
    for(int i = 0; i < N; i++)
    {
        cout<<"Input a score: ";
        cin>>score[i];
        sum += score[i];
    }
    average = (float)sum / N;
    cout<< "The average score is" << average);
    return 0;
}
```

用数组存储同学们的成绩，后续程序可以对每个成绩进行处理，其中，`score[i]`表示数组的任一元素，`i`是数组的下标，取值范围是0~49，即数组`score`有50个元素：`score[0]`、`score[1]`、...、`score[49]`

# 思考:

---

## ► 可行?

```
int a[10], b[10];
```

```
.....
```

```
a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

```
b = a;
```

```
scanf("%d", &a);
```

```
printf("%d", a);
```

**不能整体操作!** 可以逐个元素循环赋值

```
for (int i=0; i<10; i++)
```

```
    a[i] = b[i];
```



## 作为函数参数的数组

- ▶ 当一维数组作为函数的参数在函数之间传递数据时，一般的做法是用一维数组的声明（不必指定长度）和一个整型变量的定义作为被调用函数的形参，调用者需要把一个一维数组的名称以及数组元素的个数传给被调用函数。

## 例：求一维数组的最大值

... ..

```
int Max(int x[ ], int num);
```

```
int main( )
```

```
{
```

```
    int a[10] = {12,1,34}, b[20] = {23,465,34}, index_max = 0;
```

```
    index_max = Max(a, 10);
```

```
    cout<<"数组a的最大元素是"<< a[index_max]);
```

```
    index_max = Max(b, 20);
```

```
    cout<<"数组b的最大元素是"<< b[index_max]);
```

```
    return 0;
```

```
}
```

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

```

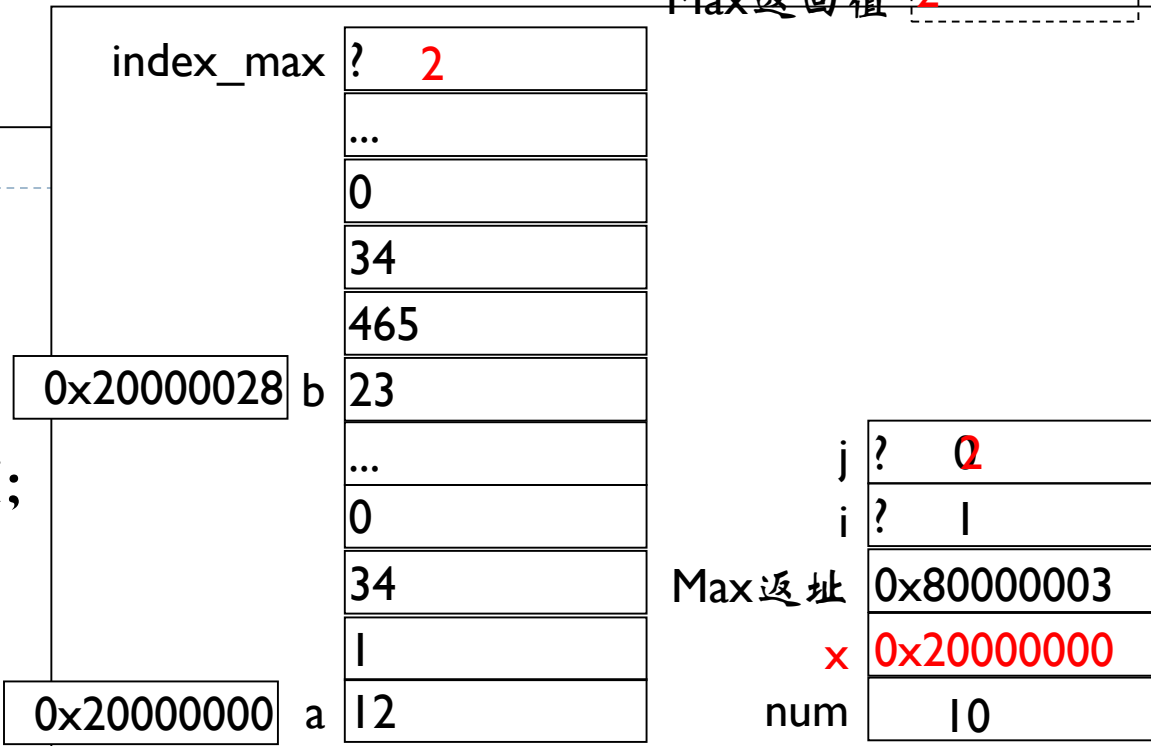
int Max(int x[ ], int num)
{
    int i, j;
    j = 0;
    for (i=1; i<num; i++)
        if (x[i] > x[j]) j = i;
    return j;
}

```

```

int main( )
{
    int a[10]={12,1,34}, b[20]={23,465,34}, index_max;
    index_max = Max(a,10);
    cout<<"数组a的最大元素是"<< a[index_max];
    index_max = Max(b,20);
    cout<<"数组b的最大元素是" <<b[index_max];
    return 0;
}

```



```

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

```

```

int main()
{
    int a[10]={12,1,34}, b[20]={23,465,34}, index_max;
    index_max = Max(a,10);
    cout<<"数组a的最大元素是"<< a[index_max];
    index_max = Max(b,20);
    cout<<"数组b的最大元素是"<<b[index_max];
    return 0;
}

```

0x20000028 b	index_max	?	1
	...		
	0		
	34		
	465		
	23		
	...		
	0		
	34		
	1		
0x20000000 a		12	

Max返回值

1

j	?	0
i	?	1
Max返址	0x80000013	
x	0x20000028	
num	20	

# 一维数组作为函数参数

---

- ▶ 数组作为函数参数传递数据时，实际传递的是数组在内存的起始位置，函数的形参数组不再分配内存空间，它共享实参数组的内存空间。
- ▶ 返回值不能是数组类型，因为数组不能整体操作。



# 例：筛法（Eratosthenes）求素数

举例：用筛法求20之前的素数

$s[i]$

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

2 3 5 7 9 11 13 15 17 19

2 3 5 7 11 13 17 19

2 3 5 7 11 13 17 19

2 3 5 7 11 13 17 19

2 3 5 7 11 13 17 19

2 3 5 7 11 13 17 19

2 3 5 7 11 13 17 19



## 例：筛法（Eratosthenes）求素数

---

分析：

假设所有的数从小到大排列在筛子中，

每个数对应一个**标志**，数 $i$ 对应的标志是 $s[i]$ ， $i$ 从2开始，

$s[i]$ 为true表示在筛子中，false表示不在筛子中，

依次将最小的数的倍数从筛子中筛掉，

最终筛子中剩下的数均为素数。

```
#define MAX 21
_Bool s[MAX];

for (int i=2; i < MAX; i++)
    s[i] = true;           // 初始化, 所有数都在筛子中
for (int i=2; i<MAX; i++)
    if (s[i])              // 值为true的元素下标为素数
        for (int m=i+1; m < MAX; m++)
            if (m%i == 0)
                s[m] = false;
                // 从筛子中筛去当前素数的倍数
```

## 再分析这个例题：

每次需要从i+1开始循环吗？每次+1有必要吗？

```
#define MAX 101
_Bool s[MAX];

for (int i=2; i < MAX; i++)
    s[i] = true;           // 初始化，所有数都在筛子中
for (int i=2; i<MAX; i++)
    if (s[i])              // 值为true的元素下标为素数
    {
        printf("%d\t", i);    // 打印素数i
        for (int m=i+1; m < MAX; m++)
            if (m%i == 0)
                s[m] = false;
                                // 从筛子中筛去当前素数的倍数
    }
```

2	3	5	7	11	13	17	19	23	29
31	37	41	43	47	53	59	61	67	71
73	79	83	89	97					

进行更深入的分析，可能发掘更高效的算法！

```
#define MAX 101
_Bool s[MAX];

for (int i=2; i < MAX; i++)
    s[i] = true;           // 初始化，所有数都在筛子中
for (int i=2; i<MAX; i++)
    if (s[i])              // 值为true的元素下标为素数
    {
        printf("%d\t", i); // 打印素数i
        for (int m = 2*i; m < MAX; m += i)
            s[m] = false;
            // 从筛子中筛去当前素数的倍数
    }
```

2	3	5	7	11	13	17	19	23	29
31	37	41	43	47	53	59	61	67	71
73	79	83	89	97					

## 一维数组

用以表示确定个数的同类型数据按一定次序构成的  
具有一维结构的数据群体

定义一维数组变量

```
const int N = ...
```

```
int a[N];    // int、[]和N构造了一个一维数组类型
```

```
            //并用该类型定义了一个一维数组a
```

- ▶ a是数组名，同时代表&a[0]
- ▶ 有效数组元素a[0]、a[1]、... a[N-1]
- ▶ 对数组不能直接进行整体操作

## 一维数组作为函数参数

---

- ▶ 数组作为函数参数传递数据时，实际传递的是数组在内存的起始位置，函数的形参数组不再分配内存空间，它共享实参数组的内存空间。
- ▶ 返回值不能是数组类型，因为数组不能整体操作。

# 内容回顾

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

int main( )
{
    int a[10]={12,1,34}, b[20]={23,465,34}, index_max;
    index_max = Max(a,10);
    printf("数组a的最大元素是: %d\n", a[index_max]);
    index_max = Max(b,20);
    printf("数组b的最大元素是: %d\n", b[index_max]);
    return 0;
}
```



## 例：筛法（Eratosthenes）求素数

```
#define MAX 101
_Bool s[MAX]; // 用数组表示一组数据的“状态”

for (int i=2; i < MAX; i++)
    s[i] = true;           // 初始化，所有数都在筛子中
for (int i=2; i<MAX; i++)
    if (s[i])              // 值为true的元素下标为素数
    {
        printf("%d\t", i); // 打印素数i
        for (int m = 2*i; m < MAX; m += i)
            s[m] = false;
            // 从筛子中筛去当前素数的倍数
    }
```

2	3	5	7	11	13	17	19	23	29
31	37	41	43	47	53	59	61	67	71
73	79	83	89	97					

# 二维数组

- ▶ 二维数组用以表示矩阵这类具有二维结构的数据，第一维称为二维数组的**行**，第二维称为二维数组的**列**，二维数组的每个元素由其所在的行和列唯一确定
- ▶ 二维数组类型的构造
  - ▶ 二维数组类型由元素类型标识符、两个中括号和两个整数（分别表示二维数组的行数与列数，行数与列数的乘积为二维数组的长度）构造而成。
  - ▶ 比如，  
`typedef int B[3][2];`  
//B是由6个int型元素所构成的二维数组类型标识符

# 二维数组变量的定义

---

两种方法:

1. 先构造类型，再定义变量，便于定义多个同类型变量

```
typedef int B[3][2];
```

```
B a, b;    //定义了一个二维数组a和一个二维数组b
```

2. 构造类型的同时定义变量

```
int b[3][2]; // 定义了一个3行2列的二维数组b
```



---

```
int b[3][2];
```

- ▶ C语言的二维数组一般是按行存储的，即先是第一行的元素，再是第二行的元素...



- ▶ 标识符**b**是二维数组名，同时表示二维数组第一行元素在内存的起始位置，即**&b[0]**

# 二维数组的初始化

- ▶ `int b[3][2] = {0, 1, 2, 3, 4, 5};` // 行优先赋值
- ▶ 分行初始化: `int b[3][2] = {{0, 1}, {2, 3}, {4, 5}};`
- ▶ 可以省略行数:
  - ▶ 比如, `int b[][2] = {{0, 1}, {2, 3}, {4, 5}};`
- ▶ 如果二维数组初始化表中的初始值个数少于二维数组的长度, 或者某一行初始值的个数少于列数, 则未指定的部分元素被初始化成0.
  - ▶ 比如,  

<code>int b[3][2] = {0, 1, 2, 4, 5};</code>	// 第3行第2列元素为0
<code>int b[3][2] = {{0, 1}, {2}, {4, 5}};</code>	// 第2行第2列元素为0

# 二维数组的操作

- ▶ 不能进行整体操作
  - ▶ 对二维数组的操作，通常要采用嵌套的循环流程依次操作数组的元素。
  - ▶ 访问数组元素的格式为：<数组名>[行下标][列下标]
    - ▶ 比如， $b[i][j]$ 表示自 $b[0][0]$ 开始的第 $i+1$ 行的第 $j+1$ 个元素 ( $i:0\sim 2, j:0\sim 1$ )
- int b[3][2];**
- b[0][0]、b[0][1]、b[1][0]、b[1][1]、b[2][0]、b[2][1]**
- ▶ **[ ]** 为下标操作符，不表示数组的长度
  - ▶ 对**int b[M][N]**这个二维数组：
    - “行下标”和“列下标”均为整型表达式，常常表现为循环变量。行下标、列下标均为0时表示第一行的第一个元素；行下标为0、列下标为**N-1**时表示列数为N的二维数组第一行的最后一个元素；行下标为**M-1**、列下标为N-1时表示行数为M、列数为N的二维数组的最后一行的最后一个元素。
    - 对于M行N列的二维数组：不存在行下标为M或列下标为N的元素 ( **$b[M][N]$ 越界**)!

---

▶ 二维数组可以看成是一个特殊的一维数组。

▶ 比如， `int b[3][2]`；这个二维数组看成一维数组时，有3个元素，

每个元素又是一个一维数组（对应二维数组的一行元素），名为**`b[i]`**；**`b[i]`**又表示自**`b[0]`**开始的第*i*+1行第一个元素在内存的起始位置（*i*：0~2），即**`&b[i][0]`**

`b[i][j]`表示自**`b[i][0]`**开始的第*j*+1个元素（*j*：0~1），也即第*i*+1行第*j*+1个元素。

int b[3][2] 共有6个元素:

$b[i][j]$  ( $i: 0 \sim 2, j: 0 \sim 1$ )

$b[0][0]$  ( $i=0, j=0$ )  $b[0][1]$  ( $i=0, j=1$ )

$b[1][0]$  ( $i=1, j=0$ )  $b[1][1]$  ( $i=1, j=1$ )

$b[2][0]$  ( $i=2, j=0$ )  $b[2][1]$  ( $i=2, j=1$ )

第  $2*i + j$  个元素 ( $i: 0 \sim 2, j: 0 \sim 1$ )

第0个元素 ( $i=0, j=0$ )

第1个元素 ( $i=0, j=1$ )

第2个元素 ( $i=1, j=0$ )

第3个元素 ( $i=1, j=1$ )

第4个元素 ( $i=2, j=0$ )

第5个元素 ( $i=2, j=1$ )

还可以看成一个特殊的一维数组，这个一维数组有3个元素：

$b[0]$ 、 $b[1]$ 、 $b[2]$

每个元素又是一个一维数组，含有2个元素：

$b[0][0]$   $b[0][1]$   $b[1][0]$   $b[1][1]$   $b[2][0]$   $b[2][1]$



## 例：求矩阵元素的和

---

... ..

```
const int M = 10;
```

```
const int N = 5;
```

```
Int main()
```

```
{
```

```
    int sum = 0, mtrx[M][N];
```

```
    for(int i= 0; i < M; i++)    // 0 开始, M-1结束
```

```
        for(int j = 0; j < N; j++)    // 0 开始, N-1结束
```

```
        {
```

```
            cout<< "Input a number: ";
```

```
            cin>> mtrx[i][j];
```

```
            sum += mtrx[i][j];
```

```
        }
```

```
    cout<<"sum = " << sum;
```

```
    return 0;
```

```
}  
▶
```

mtrx[0][N \* i + j]

## 二维数组作为函数参数

- ▶ 当二维数组作为函数的参数在函数之间传递数据时，通常用二维数组的声明（不必指定行数）和一个整型变量的定义作为被调用函数的形参，调用者需要把一个二维数组的名称以及数组的行数传给被调用函数。

## 例：用函数实现求矩阵元素求和

... ..

```
const int N=5;
int Sum(int x[ ][N], int lin);
int main( )
{
    int mtrx[10][N] = {.....};
    cout<<"sum = "<<Sum(mtrx, 10));
    return 0;
}
```

形参二维数组的列数必须确定：

实参：第一行（N个）元素在内存的起始位置

形参：可以存放某行（N个元素）位置的空间

```
int Sum(int x[ ][N], int lin)
{
    int s = 0;
    for(int i = 0; i < lin; i++)
        for(int j = 0; j < N; j++)
            s += x[i][j];
    return s;
}
```

- ▶ 如果要提高上述Sum函数的通用性（形参二维数组的列数必须确定），可以将二维数组降为一维数组处理：

```
int Sum(int x[], int num)
{
    int s = 0;
    for(int i = 0; i < num; i++)
        s += x[i];
    return s;
}

.....
int m1[10][5], m2[20][5], m3[40][20];
.....
cout<<" sum = "<< Sum(m1[0], 10 * 5));
cout<<" sum = "<< Sum(m2[0], 20 * 5));
cout<<" sum = "<< Sum(m3[0], 40 * 20));
```

## 二维数组作为函数参数

---

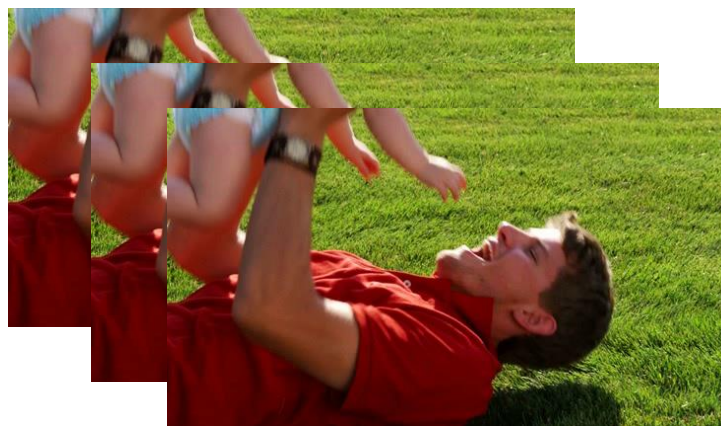
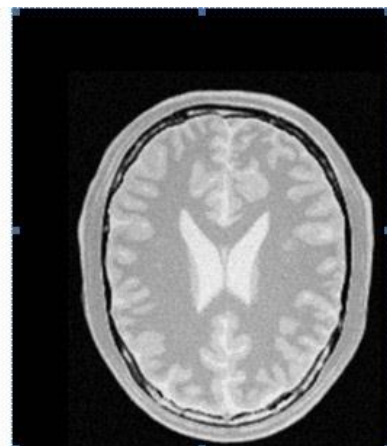
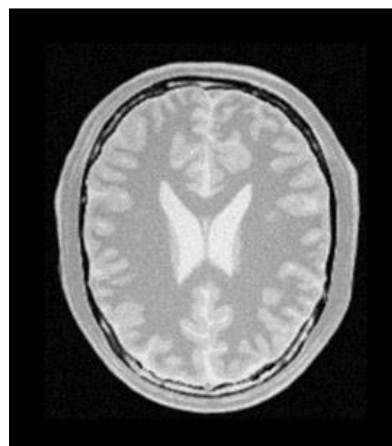
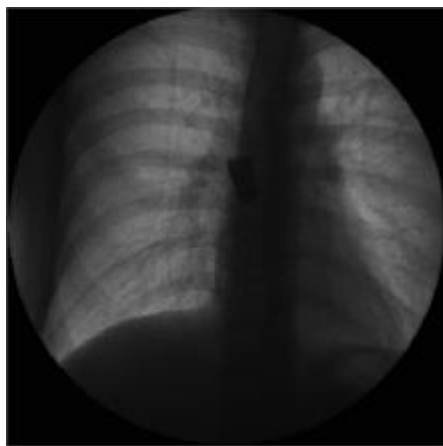
- ▶ 二维数组作为函数参数传递数据时，实际传递的也是数组在内存的起始位置，函数的形参数组不再分配内存空间，它共享实参数组的内存空间。
- ▶ 返回值不能是二维数组类型，因为数组不能整体操作。

# 多维数组

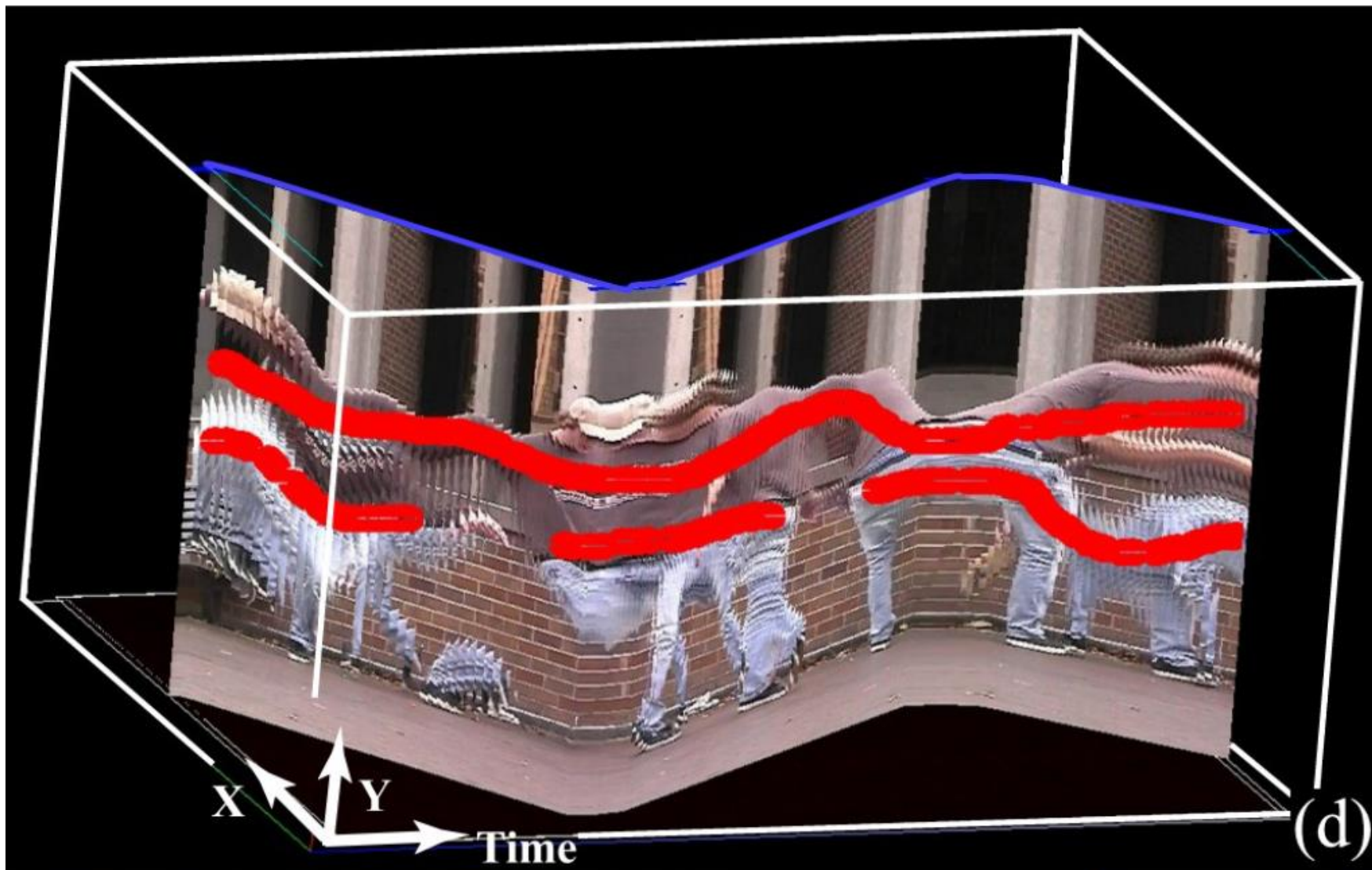
- ▶ 可以用来表示  
数据立方体

等高维数据:

- ▶ 医学图像处理
- ▶ 视频处理
- ▶ 时空数据分析  
等领域



# 视频体



# 多维数组的定义

---

- ▶ 多维数组类型由元素类型关键词、多个中括号和多个整数（分别表示每一维的元素个数）构造而成。可以用构造好的多维数组类型定义多维数组，也可以在构造多维数组类型的同时直接定义多维数组：

```
int d[2][3][4];
```

```
//定义了一个三维数组，元素个数为24 (=2×3×4)
```



# 访问多维数组

```
int d[2][3][4];
```

► 访问多维数组元素的格式为：

<数组名> [<第1维下标>][第2维下标]...[第n维下标]

d[0][0][0]

d[0][0][1]

d[0][0][2]

d[0][0][3]

d[0][1][0]

d[0][1][1]

d[0][1][2]

d[0][1][3]

d[0][2][0]

d[0][2][1]

d[0][2][2]

d[0][2][3]

d[1][0][0]

d[1][0][1]

d[1][0][2]

d[1][0][3]

d[1][1][0]

d[1][1][1]

d[1][1][2]

d[1][1][3]

d[1][2][0]

d[1][2][1]

d[1][2][2]

d[1][2][3]

# 数组及其应用

---

- ▶ 一维数组
  - ▶ 可用来表示向量...
- ▶ 二维数组
  - ▶ 可用来表示矩阵...
- ▶ 多维数组
- ▶ 数组的应用
- ▶ 基于数组的排序程序



# 数组的应用

---

## ▶ 实际应用中:

- ▶ 表示有序的（相同属性）一组数据

  - ▶ 向量、数列、同学们的成绩...

- ▶ 还可以利用数组存储一组有序**标志位**，以对应一组（有相同属性）数据的状态

## 例：求解约瑟夫斯（Josephus）问题

- ▶ 有n个小孩围成一圈，做游戏。从某个小孩开始顺时针报数，报到k的小孩从圈子离开；然后，从下一个小孩开始重新报数，每报到k，相应的小孩从圈子离开；最后只剩下一个小孩，该小孩是幸运儿；这位幸运的小孩一开始站在什么位置？
- ▶ **分析：**采用一维数组`in_circle[n]`表示n个小孩围成一圈，`in_circle[index]`为true表示编号为index的小孩在圈子里，从index为0的小孩开始报数，圈子中index的下一个位置为 $(index+1)\%n$ （下一圈连续报数）。

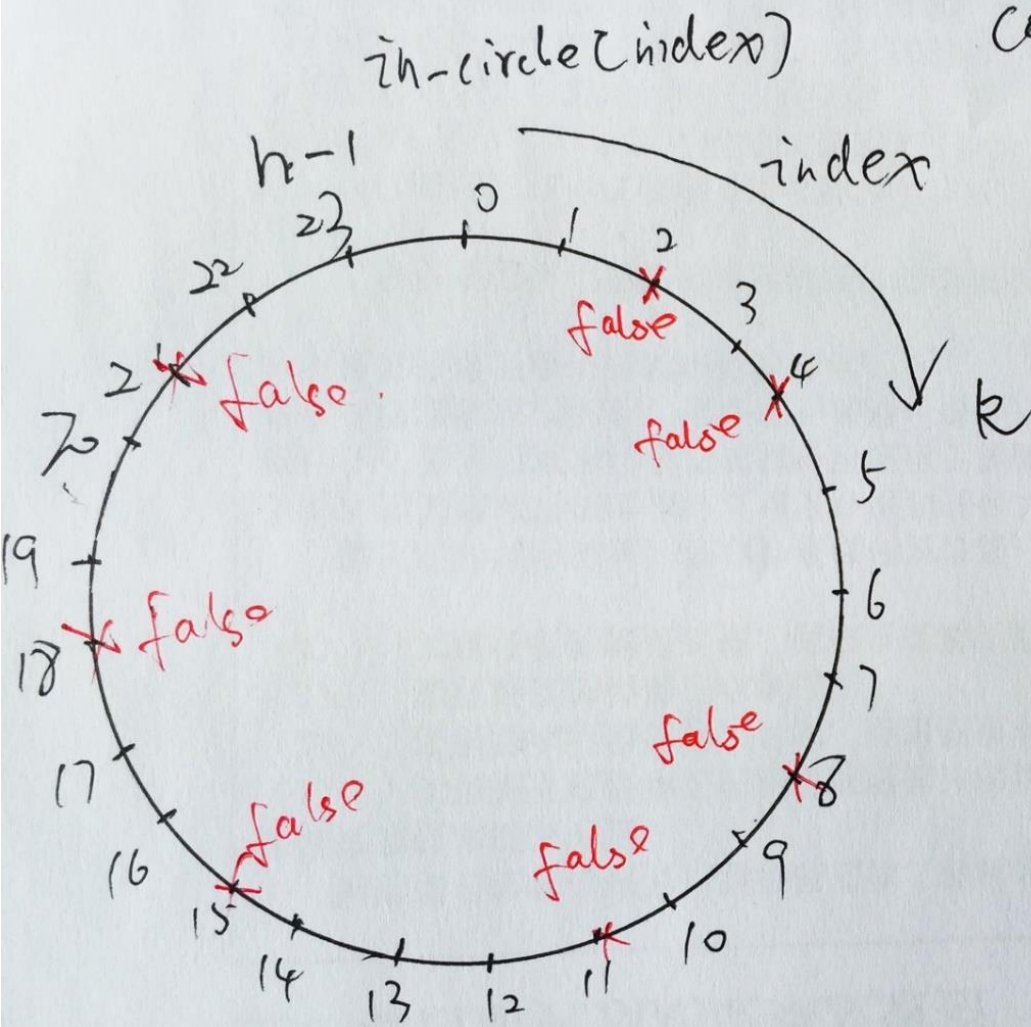
---

```
#include <stdio.h>
#define N 20    // C++: const N = 20;
#define K 5     // C++: const K = 5;
int Josephus(int n, int k);
int main( )
{
    printf("The lucky boy is No.%d. \n", Josephus(N, K));
    return 0;
}
```

---

```
int Josephus(int n, int k)
{
    _Bool in_circle[n];
    int index, numRemained = n;

    for(index = 0; index < n; index++)
        in_circle[index] = true;    //初始化数组in_circle
```



count: 当前计数器  
初始为0  
=k 则离开.

```
while ( ) // 当 count=k
{
    if ( )
    {
        count++;
        index++;
    }
}
```

//报数

index = 0;

int count = 0;

while(count < k)           // 确定报k的小孩离开的循环

{

    if(in\_circle[index])

        count++;

    index = index+1;

}



//报数

index = 0;

int count = 0;

while(count < k)           // 确定报k的小孩离开的循环

{

    if(in\_circle[index])

        count++;

    index = (index+1)%n;

}

//报数

index = n-1;            //n-1的下一位置为0

int count = 0;

while(count < k)            // 确定报k的小孩离开的循环

{

    index = (index+1)%n;

    if(in\_circle[index])

        count++;

}

//报数，离开

index = n-1;            //n-1的下一位置为0

int count = 0;

while(count < k)            // 确定报k的小孩离开的循环

{

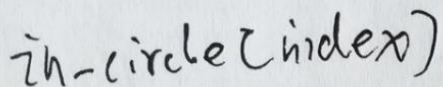
    index = (index+1)%n;

    if(in\_circle[index])

        count++;

}

in\_circle[index] = false; //某小孩沮丧地离开圈子



```
while ( ) // i.e: count = k
{
    if ( )
    {
        count ++ ;
        index ++ ;
    }
}
```

```
//报数，离开，人数减1
index = n-1;          //n-1的下一位置为0
while(numRemained > 1)
{
    int count = 0;
    while(count < k)    // 确定报k的小孩离开的循环
    {
        index = (index+1)%n;
        if(in_circle[index])
            count++;

    }
    in_circle[index] = false; //小孩离开圈子
    numRemained--; //圈中人数减1
}
```

//找出最后一个囚犯

```
for (index = 0; index < n; index++)  
    if (in_circle[index])  
        break;
```

```
//printf("The lucky boy is No.%d.\n", index);  
return index;  
}
```

# 数组的应用

---

## ▶ 实际应用中:

- ▶ 表示有序的（相同属性）一组数据

  - ▶ 向量、数列、同学们的成绩...

- ▶ 还可以利用数组存储一组有序**标志位**，以对应一组（有相同属性）数据的状态

(以上两个例子)

# 例：求矩阵的乘积

▶ 设有两个矩阵  $A_{mn}$ 、 $B_{np}$ ,

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \end{pmatrix}$$

$$\begin{pmatrix} 4 & 4 \\ 5 & 5 \\ 6 & 6 \\ 7 & 7 \end{pmatrix}$$

乘积矩阵  $C_{mp} = A_{mn} \times B_{np}$ ,  $C_{mp}$  中每一项的计算公式为:

$$C_{ij} = \sum_{k=0}^{n-1} A_{ik} \times B_{kj}$$

$$i=0, \dots, m-1, j=0, \dots, p-1$$

$$\begin{pmatrix} 4+5+6+7 & 4+5+6+7 \\ 8+10+12+14 & 8+10+12+14 \\ 12+15+18+21 & 12+15+18+21 \end{pmatrix}$$



```

#include <stdio.h>
#define N 4
#define M 3
-----
#define P 2
void prod(int ma[ ][N], int mb[ ][P], int mc[ ][P], int m);
int main( )
{
    int ma[M][N] = {{1, 1, 1, 1}, {2, 2, 2, 2}, {3, 3, 3, 3}};
    int mb[N][P] = {{4, 4}, {5, 5}, {6, 6}, {7, 7}};
    int mc[M][P];
    prod(ma, mb, mc, M);
    for(int i = 0; i < M; i++)
        for(int j = 0; j < P; j++)
        {
            printf(" %d ", mc[i][j]);
            if((j + 1) % P == 0) printf("\n");
        }
    return 0;
}
-----

```

22	22
44	44
66	66



# 例：求矩阵的乘积

▶ 设有两个矩阵  $A_{mn}$ 、 $B_{np}$ ,

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \end{pmatrix}$$

$$\begin{pmatrix} 4 & 4 \\ 5 & 5 \\ 6 & 6 \\ 7 & 7 \end{pmatrix}$$

乘积矩阵  $C_{mp} = A_{mn} \times B_{np}$ ,  $C_{mp}$  中每一项的计算公式为:

$$C_{ij} = \sum_{k=0}^{n-1} A_{ik} \times B_{kj}$$

$$i=0, \dots, m-1, j=0, \dots, p-1$$

$$\begin{pmatrix} 4+5+6+7 & 4+5+6+7 \\ 8+10+12+14 & 8+10+12+14 \\ 12+15+18+21 & 12+15+18+21 \end{pmatrix}$$

设有两个矩阵  $A_{mn}$ 、 $B_{np}$ ,

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \end{pmatrix}$$

$$\begin{pmatrix} 4 & 4 \\ 5 & 5 \\ 6 & 6 \\ 7 & 7 \end{pmatrix}$$

```
void prod(int ma[ ][N], int mb[ ][P], int mc[ ][P], int m)
{
    for(int i = 0; i < m; i++)
        for(int j = 0; j < P; j++)
        {
            int s = 0;
            for(int k = 0; k < N; k++)
                s += ma[i][k] * mb[k][j];
            mc[i][j] = s;
        }
}
```

# 基于数组的排序

---

- ▶ 排序是一种常见、非常重要的问题
- ▶ 解决排序问题的算法有很多：
  - ▶ 起泡排序
  - ▶ 选择排序
  - ▶ 插入排序
  - ▶ 快速排序等
  - ▶ 以及它们的变种

# 基于数组的排序

---

- ▶ 如果待排序数据存在数组中，则用程序实现这些排序算法时涉及到
  - ▶ 数组的遍历
  - ▶ 两个元素的比较、交换
  - ▶ 一个元素的插入等操作
  - ▶ ...
- ▶ 需要综合运用循环、分支流程控制及赋值操作等完成

## 例：数组元素的排序

---

```
#include <iostream>
using namespace std;
const int N=4;
void Sort(int sdata[ ], int count);
int main( )
{
    int a[N];
    for(int i = 0; i < N; i++)
        cin>>a[i];
    Sort(a, N);    //对4个数(5、4、0、2)从小到大排序
    for(int i = 0; i < N; i++)
        cout<<a[i];
    return 0;
}
```

---

# 起泡法排序

- ▶ 比较相邻两个数，小的调到前头或大的“掉”到后面
- ▶ N个数排N-1趟，每趟内比较的次数随趟数递减。

5	4	4	4
4	5	0	0
0	0	5	2
2	2	2	5

第i=0趟

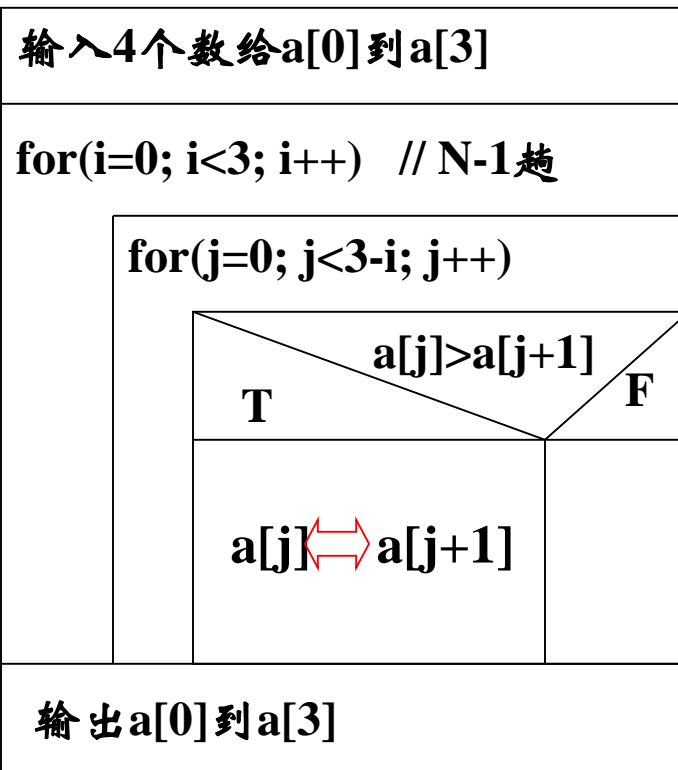
4	0	0
0	4	2
2	2	4
5	5	5

第i=1趟

0	0
2	2
4	4
5	5

第i=2趟

int a[4]



► 推广到N个数

int a[N]

交换两个数

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

输入N个数给a[0]到a[N-1]

```
for(i=0; i<N-1; i++)
```

```
for(j=0; j<N-1-i; j++)
```

$a[j] > a[j+1]$

T

F

$a[j] \leftrightarrow a[j+1]$

输出a[0]到a[N-1]



```
#include <iostream>
using namespace std;
const int N=4;
int main( )
{
    int a[N];
    for(int i = 0; i < N; i++)
        cin>>a[i];

    for(int i = 0; i < N; i++)
        cout<<a[i];
    return 0;
}
```

```
for(i=0; i<N-1; i++)
    for(j=0; j<N-1-i; j++)
        if(a[j]>a[j+1])
        {
            temp=a[j];
            a[j]=a[j+1];
            a[j+1]=temp;
        }
```

# 上机遇到的问题

---

...

```
for(i=0; i<N-1; i++)
```

```
    for(j=0; j<N-i; j++)
```

```
        if(a[j]>a[j+1]) // 有越界的元素吗?
```

```
        {
```

```
            temp=a[j];
```

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

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

```
        }
```

...



## “冒泡” 改写为函数的形式

...

```
int main()
```

```
{
```

```
    int a[N];
```

```
    for (int i=0; i<N; i++)
```

```
        cin >> a[i];
```

```
    BubbleSort(a, N);
```

```
    ...
```

```
}
```

```
void BubbleSort(int sdata[ ], int count)
```

```
{
```

```
    int temp;
```

```
    for(int i = 0; i < count-1; i++)
```

```
        for(int j = 0; j < count-1-i; j++)
```

```
            if(sdata[j] > sdata[j+1])
```

```
            {
```

```
                temp = sdata[j];
```

```
                sdata[j] = sdata[j+1];
```

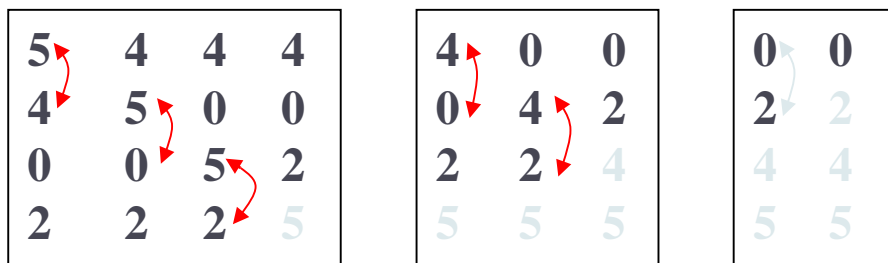
```
                sdata[j+1] = temp;
```

```
            }
```

```
}
```

# 冒泡排序法

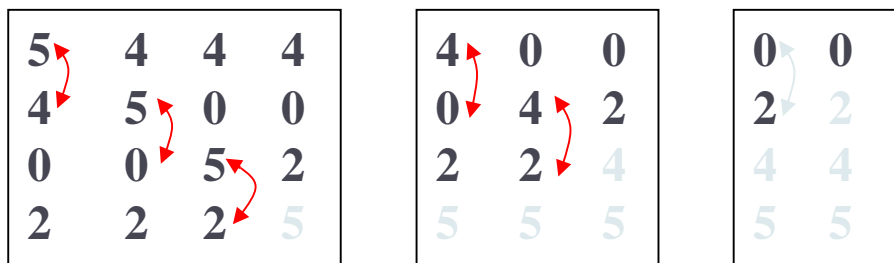
- ▶ 以上算法：每次大循环把最大的元素调整到最后
  - ▶ 每次小循环把两个元素中大的向后调整



- ▶ **Another option:** 每次大循环把最小的元素调整到最前
  - ▶ 像气泡飘向最上面（冒泡）
  - ▶ 请自己改写！作为上机题目！

# 再审视：冒泡排序法

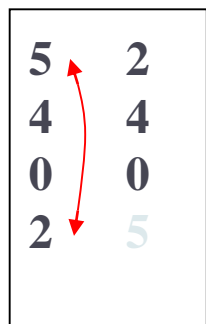
- ▶ 以上算法：每次大循环把最大的元素调整到最后
- ▶ 每次小循环把两个元素中大的向后调整



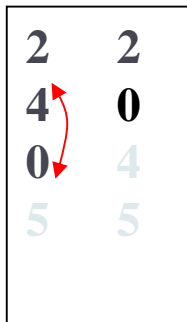
- ▶ 每次小循环都可能交换，通过不断交换，把最大的调整到相应位置
- ▶ 能否只先比较，找到最大数的位置，最后再和相应位置的数字交换

# 选择法排序

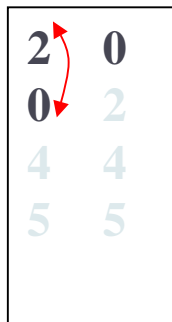
- 从N个数中选择最大者，与第N个数交换位置；然后从剩余的N-1个数中再选择最大者，与第N-1个数交换位置；...；直到只剩下一个数为止
- N个数选择N-1趟，每趟内比较的次数随趟数递减，且不是每次比较都进行元素交换操作。



前N个选



前N-1个选



前2个选

if(max != i-1)

输入N个数给a[0]到a[N-1]

for(i=N; i>1; i--) //从前i个数中选

max=0

for(j=1; j < i; j++)

a[max]<a[j]

T

F

max=j

a[max] ↔ a[i-1]

输出a[0]到a[N-1]

```

void SelSort(int sdata[ ], int N)
{
    for(int i = N; i > 1; i--)           // 大循环 N-1次
    {
        int max = 0;
        for(int j = 1; j < i; j++)       // 小循环：“选择”（找到）当
                                          // 前大循环最大的
            if(sdata[max] < sdata[j])
                max = j;
        if(max != i-1)                   // 交换
        {
            int temp = sdata[max];
            sdata[max] = sdata[i-1];
            sdata[i-1] = temp;
        }
    }
}

```

## 二维数组

---

二维数组用以表示矩阵这类具有二维结构的数据

两种方法：

1. 先构造类型，再定义变量，便于定义多个同类型变量

```
typedef int B[3][2];
```

```
B a, b;    //定义了一个二维数组a和一个二维数组b
```

2. 构造类型的同时定义变量

```
int b[3][2]; // 定义了一个3行2列的二维数组b
```



## 二维数组

---

```
int b[3][2] = {0, 1, 2, 3, 4, 5};
```

- b同时表示b[0]在内存的起始位置
- 下标从0开始, b[i][j]是第i+1行的第j+1个元素
- 可以看作含b[0], b[1]和b[2]三个一维数组元素
- 对元素遍历操作, 双循环!



## 例：求矩阵元素的和

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

```
#define M 10
```

```
#define N 5
```

```
int main()
```

```
{
```

双循环！

```
    int sum = 0, mtrx[M][N];
```

```
    for(int i = 0; i < M; i++)    // 0 开始, M-1 结束
```

```
        for(int j = 0; j < N; j++)    // 0 开始, N-1 结束
```

```
        {
```

```
            printf("Input a number: ");
```

```
            scanf("%d", &mtrx[i][j]);
```

$mtrx[0][N * i + j]$

```
            sum += mtrx[i][j];
```

```
        }
```

```
    printf("sum = %d\n", sum);
```

```
    return 0;
```

```
}  
▶
```

## 二维数组作为函数参数

---

- ▶ 当二维数组作为函数的参数在函数之间传递数据时，通常用**二维数组的声明**（不必指定行数）和一个**整型变量**的定义作为被调用函数的形参，调用者需要把一个**二维数组的名称**以及**数组的行数**传给被调用函数。

## 例：用函数实现求矩阵元素求和

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

```
#define N 5
```

```
int Sum(int x[ ][N], int lin);
```

```
int main( )
```

```
{
```

```
    int mtrx[10][N] = {·····};
```

```
    printf("sum = %d\n", Sum(mtrx, 10));
```

```
    return 0;
```

```
}
```

形参二维数组的列数必须确定：

实参：第一行（N个）元素在内存的起始位置

形参：可以存放某行（N个元素）位置的空间

```
int Sum(int x[ ][N], int lin)
```

```
{
```

```
    int s = 0;
```

```
    for(int i = 0; i < lin; i++)
```

```
        for(int j = 0; j < N; j++)
```

```
            s += x[i][j];
```

```
    return s;
```

```
}
```

- ▶ 如果要提高上述Sum函数的通用性（形参二维数组的列数必须确定），可以将二维数组降为一维数组处理：

## 内容回顾

```
int Sum(int x[], int num)
{
    int s = 0;
    for(int i = 0; i < num; i++)
        s += x[i];
    return s;
}

.....
int m1[10][5], m2[20][5], m3[40][20];
.....
printf("sum = %d\n", Sum(m1[0], 10 * 5));
printf("sum = %d\n", Sum(m2[0], 20 * 5));
printf("sum = %d\n", Sum(m3[0], 40 * 20));
```

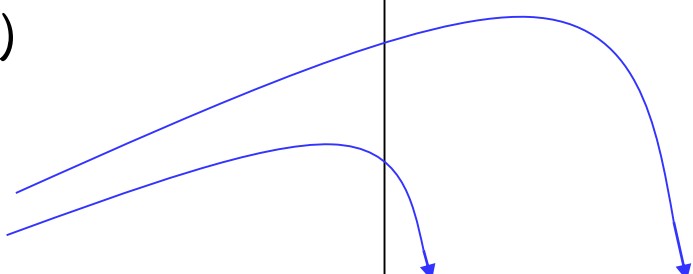
# 内容回顾

## 冒泡排序法

```
...  
int main()  
{  
    int a[N];  
    for (int i=0; i<N; i++)  
        cin >> a[i];
```

BubbleSort(a, N);

```
...  
}
```

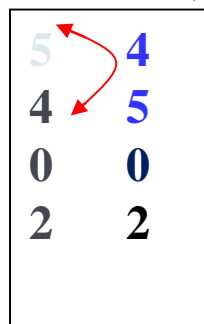


```
void BubbleSort(int sdata[ ], int count)  
{  
    int temp;  
    for(int i = 0; i < count-1; i++)  
        for(int j = 0; j < count-1-i; j++)  
            if(sdata[j] > sdata[j+1])  
            {  
                temp = sdata[j];  
                sdata[j] = sdata[j+1];  
                sdata[j+1] = temp;  
            }  
}
```

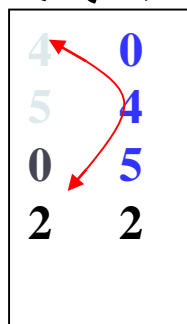
```
void SelSort(int sdata[ ], int N)
{
    for(int i = N; i > 1; i--)           // 大循环 N-1次
    {
        int max = 0;
        for(int j = 1; j < i; j++)       // 小循环：“选择”（找到）当
                                          // 前大循环最大的
            if(sdata[max] < sdata[j])
                max = j;
        if(max != i-1)                   // 交换
        {
            int temp = sdata[max];
            sdata[max] = sdata[i-1];
            sdata[i-1] = temp;
        }
    }
}
```

# 直接插入法排序

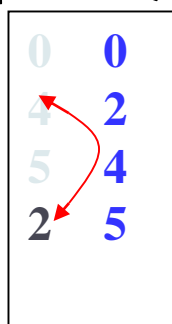
- 在已排好序的*i*个数中寻找合适的位置，将下标为*i*的元素插入，插入点后的数往后移
- N*个数考察*N-1*趟，每趟内比较的次数随趟数递增，一旦发现需要进行插入操作，则需要对部分元素进行移动操作。



插入4



插入0

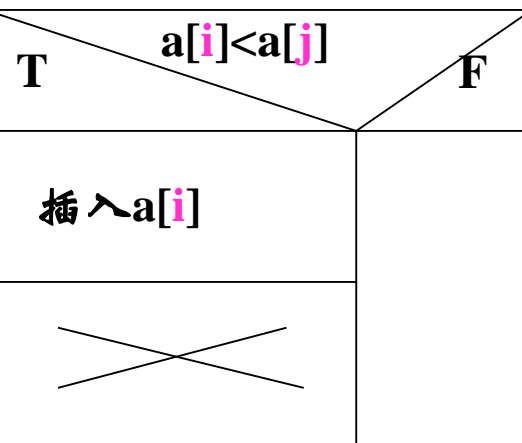


插入2

输入*N*个数给*a*[0]到*a*[*N*-1]

for(*i*=1; *i*<*N*; *i*++) // 插入第*i*个数

for(*j*=0; *j*<*i*; *j*++)



输出*a*[0]到*a*[*N*-1]



## 将 $a[i]$ 插入到 $a[j]$ 前

插入前:  $a[0] \ a[1] \ \dots \ a[j-1] \ a[j] \ \dots \ a[i-1]$

后:  $a[0] \ a[1] \ \dots \ a[j-1] \ a[i] \ a[j] \ \dots \ a[i-1]$

移位后:  $a[j] \ a[j+1] \ \dots \ a[i]$

```
temp = a[j];
```

```
a[j] = a[i];
```

```
for(k = i-1; k >= j+1; k--)
```

```
    a[k+1] = a[k];
```

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

```
void InsSort(int sdata[ ], int N)
{
    int temp;
    for (int i = 1; i < N; i++)
        for (int j = 0; j < i; j++)
            if(sdata[i] < sdata[j])
            {
                temp = sdata[j];
                sdata[j] = sdata[i];          // 先插入sdata[i]到相应位置
                for (int k = i-1; k >= j+1; k--) // 挪动j+1及后面的元素
                    sdata[k+1] = sdata[k];
                sdata[j+1] = temp; // 原来的sdata[j] 后挪1位
                break;
            }
}
```

## 将 $a[i]$ 插入到 $a[j]$ 前

插入前:  $a[0] \ a[1] \ \dots \ a[j-1] \ a[j] \ \dots \ a[i-1]$

后:  $a[0] \ a[1] \ \dots \ a[j-1] \ "a[i]" \ "a[j]" \ \dots \ "a[i-1]"$

插入后:  $a[j] \ a[j+1] \ \dots \ a[i]$

```
temp = a[i];  
for (int k = i-1; k >= j; k--)  
    a[k+1] = a[k];  
a[j] = temp;
```

```
void InsSort(int ndata[ ], int N)
{
    int temp;
    for (int i = 1; i < N; i++)
        for (int j = 0; j < i; j++)
            if(ndata[i] < ndata[j])
            {
                temp = ndata[i];
                for (int k = i-1; k >= j; k--)    // 先挪动从j开始的元素
                    ndata[k+1] = ndata[k];
                ndata[j] = temp; // 插入原来的sdata[i]到相应位置
                break;
            }
}
```

# 快速法排序\*

---

1. 是对起泡法的一种改进
2. 基本思想：(1)通过一趟排序将待排数据分隔成独立的两部分，其中一部分数据均比另一部分数据小，然后(2)对这两部分数据分别进行快速排序。整个排序过程可以递归进行，以达到整个序列有序
3. 即，设两个下标变量first和last，它们的初始值分别为0和N-1，然后寻找分割点split\_point，再将分割点±1分别作为右半部分的first和左半部分的last，对两部分重复上述步骤，每部分直至first==last为止。

---

## ► 一趟快速排序的思想：

首先任意选取一个数据（通常选第一个数据）作为关键数据，然后将所有比它小的元素都放到它前面，所有比它大的元素都放到它后面。

即，设分割点（位置）`split_point`的初始值为0，其对应元素推选为（关键数据），从first所指位置后起，向后搜索，每遇到小于关键数据的元素，就将分割点右移一次，并将那个小元素和分割点元素互换，最后将分割点元素与first元素交换，此时以分割点`split_point`为分界线，将序列分隔成了符合要求的两个子序列。

first last

4 5 0 3 2



4 5 0 3 2



4 0 5 3 2



4 0 3 5 2



4 0 3 2 5



2 0 3 4 5



first last

pivot是4,  
其下标是split\_point

```
int Split(int sdata[ ], int first, int last);
```

---

```
void QuickSort(int sdata[ ], int first, int last)  // 采用递归!  
{  
    if(first < last)  
    {  
        int split_point;  
        split_point = Split(sdata, first, last);  
        //寻找分割点  
        QuickSort(sdata, first, split_point-1);  
        //对分割点左边的部分进行排序  
        QuickSort(sdata, split_point+1, last);  
        //对分割点右边的部分进行排序  
    }  
}
```



first last

4 0 3 2 5

4 5



split\_point

2 0 3 4 5

4 5



first last

```
int Split(int sdata[ ], int first, int last)
{
    int split_point, pivot;
    pivot = sdata[first];
    split_point = first;
    for (int unknown = first+1; unknown <= last; unknown++)
        if(sdata[unknown] < pivot)
        {
            split_point++;
            int t = sdata[split_point];
            sdata[split_point] = sdata[unknown];
            sdata[unknown] = t;
        }
    sdata[first] = sdata[split_point];
    sdata[split_point] = pivot;
    return split_point;
}
```

# 快速排序法 - 课外阅读

[http://www.sohu.com/a/246785807\\_684445](http://www.sohu.com/a/246785807_684445)

<https://segmentfault.com/a/1190000017314698>

在这种极端情况下，快速排序需要进行  $N$  轮，时间复杂度退化成了  $O(N^2)$ 。



我们该怎么避免这种情况发生呢？

其实很简单，我们可以不选择数列的第一个元素，而是**随机选择一个元素作为基准元素**。



# 排序算法总结

要求大家了解每种的基本方法，至少熟练掌握一种！

排序算法	冒泡排序	选择排序	直接插入排序	快速法排
特点	比较相邻两个数，小的调到前头，  像气泡不断的往上冒...	从i个数中选择最大者， 与第i个数交换位置 ...	在已排好序的i个数中寻找合适的位置，将下标为i的元素插入，插入点后的数往后移	通过一趟排序将待排数据分隔成独立的两部分，其中一部分数据均比另一部分数据小，然后对这两部分数据分别进行快速排序
算法结构	双层循环 for(i=0;i<N-1;i++)	双层循环 (for(i=N;i>1;i++))	三层循环 for(i=1;i<N;i++)	递归 + 循环

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	In-place	不稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
希尔排序	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(1)$	In-place	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Out-place	稳定
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	In-place	不稳定
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	In-place	不稳定
计数排序	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	Out-place	稳定
桶排序	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n + k)$	Out-place	稳定
基数排序	$O(n \times k)$	$O(n \times k)$	$O(n \times k)$	$O(n + k)$	Out-place	稳定



# 小 结

---

## ▶ 数组：

- ▶ 一种派生数据类型，表示固定多个同类型的数据群体
- ▶ 存储于内存中的连续空间
- ▶ 不能整体操作

## ▶ 要求：

- ▶ 掌握数组的定义、初始化和操作方法
- ▶ 掌握一维数组、二维数组的特征与应用方法
  - ▶ 作为函数的参数用法！
  - ▶ 至少掌握一种排序方法！
- ▶ 保持良好的编程习惯



## Q & A

---

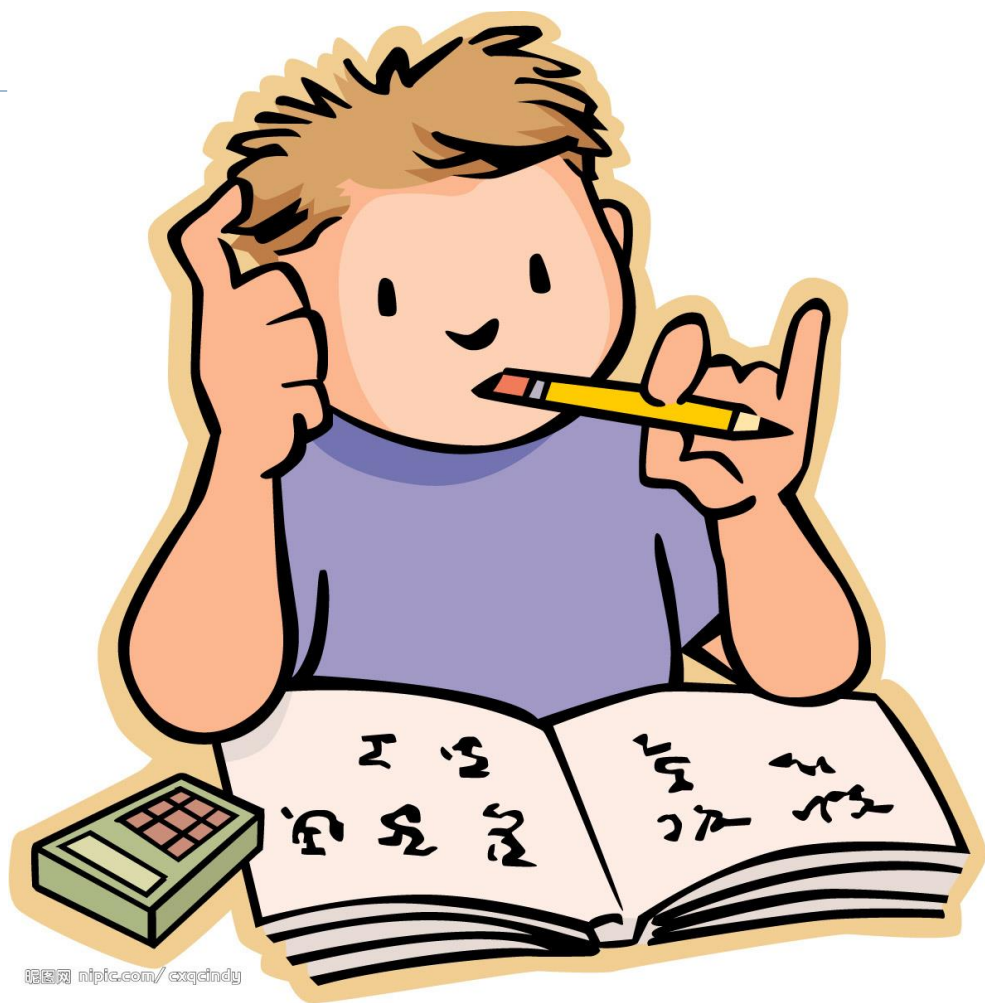


# 课程预告

---

► 下周：指针及其应用

► 希望大家提前预习！



## 操作符的目

### ▶ 一个操作符能连接的操作数的个数

#### ▶ 算术操作符

##### ▶ 取正/取负操作符

##### ▶ 自增/自减操作符

#### ▶ 关系操作符

#### ▶ 逻辑操作符

#### ▶ 位操作符

#### ▶ 赋值操作符

#### ▶ 条件操作符

双目

单目

单目

双目

双目/单目

双目/单目

双目

三目

连接两个操作数

连接一个操作数

连接三个操作数



## 操作符的优先级

- ▶ 是指操作符的优先处理级别
- ▶ C语言将基本操作符分成若干个级别
  - ▶ 第1级为最高级别，第2级次之，以此类推。
- ▶ C语言操作符的优先级一般按“单目、双目、三目、赋值”依次降低，其中双目操作符的优先级按“算术、移位、关系、逻辑位、逻辑”依次降低

( )	高
单目操作符	
* / %	
+ -	
<< >>	
> < >= <=	
== !=	
&	
^	
&&	
? :	
=	
,	低

## 表达式的操作顺序

- ▶ 一个表达式可以包含多个操作，先执行哪一个操作呢？
- ▶ C语言有以下规则：

对于相邻的两个操作，操作规则为：

- a) 判断两个操作符的优先级高低，然后先处理优先级高的操作符；
- b) 如果两个操作符的优先级相同，再判断两个操作符的结合性，结合性为左结合的先处理左边的操作符，结合性为右结合的先处理右边的操作符；
- c) 加圆括号的操作优先执行！

## 建议!!!

- ▶ 以上要掌握知识点，但写程序不建议写“难以理解”的表达式！

例如：

```
int a = 12;
```

```
a = a += a -= a*a; // 先优先级*, 然后+/- = 右结合性
```

建议写为：

```
a -= a*a;
```

```
a += a;
```

## 将一维数组作为函数形参

```
#include <stdio.h>
int Max(int x[ ], int num);
int main( )
{
    int a[10] = {12,1,34}, index_max = 0;
    index_max = Max(a, 10);
    printf("数组a的最大元素是: %d\n", a[index_max]);

    return 0;
}
```

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

## 例：用函数实现求矩阵元素求和

```
#include <stdio.h>

#define N 5

int Sum(int x[ ][ ], int lin, int col); // ✗

int main( )
{
    int mtrx[10][N] = {·····};
    printf("sum = %d\n", Sum(mtrx, 10, N));
    return 0;
}
```

形参二维数组的列数必须确定，  
否则，与实参类型不匹配。  
实参：第一行（N个）元素在内存  
的起始位置  
形参：？

```
int Sum(int x[ ][ ], int lin, int col) // ✗
{
    int s = 0;
    for(int i = 0; i < lin; i++)
        for(int j = 0; j < col; j++)
            s += x[i][j];
    return s;
}
```

## 例：用函数实现求矩阵元素求和

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

```
#define N 5
```

```
int Sum(int x[ ][N], int lin);
```

```
int main( )
```

```
{
```

```
    int m[40][20] = {·····};
```

```
    printf("sum = %d\n", Sum(m, 40)); // ×
```

```
    return 0;
```

```
}
```

实参二维数组的列数必须与形参的一致，  
否则，与形参类型不匹配。

```
int Sum(int x[ ][N], int lin)
```

```
{
```

```
    int s = 0;
```

```
    for(int i = 0; i < lin; i++)
```

```
        for(int j = 0; j < N; j++)
```

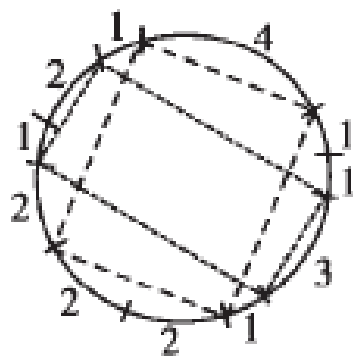
```
            s += x[i][j];
```

```
    return s;
```

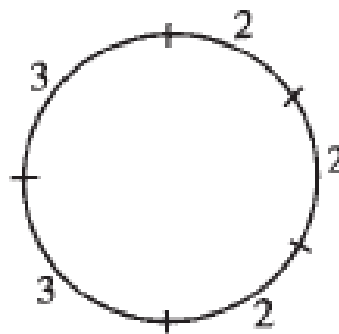
```
}
```

## 回顾：上机题目 - 寻找圆内接矩形

- 在圆周上有若干点，已经知道这些点与点之间的弧长，并且依圆弧顺序排列，请写一个程序，找出这些点中有没有4个可以围成长方形，程序返回是否有长方形即可。为了方便计算，弧长均为正整数。



a)



b)

## 上机题目 - 寻找圆内接矩形

在圆周上有若干点，已经知道这些点与点之间的弧长，并且依圆弧顺序排列，请写一个程序，找出这些点中有没有4个可以围成长方形，函数

```
int find_cirlces(double len[], int n, int idx[])
```

返回有几个长方形，double len[]为输入的弧长，n段弧，长方形顶点索引存放在idx中，例如idx[0]、idx[1]、idx[2]和idx[3]为一个长方形；在main函数中输入各段弧长并调用该函数，打印出长方形个数和每个长方形的顶点索引。  
(弧长为浮点数)