
step further

专题

起步：

认知与体验（硬件、软件、程序与C语言）

进阶：

判断与推理（流程控制方法、语句）

抽象与联系（模块设计方法、函数）

表达与转换（基本操作、数据类型）

提高：

构造与访问（数组、结构体、指针）

归纳与推广（程序设计的本质）

数组及其应用

● 一维数组

● 二维数组

● 多维数组

● 数组的应用

➤ 可以利用数组存储一组有序数据

- 基于数组的排序
- 斐波那契数求解
- 矩阵相乘（二维数组）
- 杨辉三角（二维数组）
- ...

➤ 可以利用数组存储一组有序标志，以对应一组数据的状态

- 用筛法求素数
- 约瑟夫问题求解
- 鞍点（二维数组）
- ...

● 向量：线性代数中，指 n 个实数组成的有序数据群体

$\alpha = (a_1, a_2, \dots, a_n)$ ，其中 a_i 称为向量 α 的第 i 个分量， i 为下标
(物理中，只有大小没有方向的量称为标量；既有大小又有方向的量叫做向量/矢量)

● 矩阵：用来表示具有位置关系的同类数据，一般是由方程组的系数及常数所构成的方阵，通过矩阵变换来解线性方程组既方便，又直观，例如对于方程组：

➤ $a_1x + b_1y + c_1z = d_1$ 可构造一个矩阵：	a_1	b_1	c_1	d_1
➤ $a_2x + b_2y + c_2z = d_2$	a_2	b_2	c_2	d_2
➤ $a_3x + b_3y + c_3z = d_3$	a_3	b_3	c_3	d_3

● 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, s = 0;
scanf("%d...", &u, &v, &w, &x, &y);
s = u + v + w + x + y;
printf("%.2f \n", s / 5.0);
```

- 不能保留各个分量

```
int z, s = 0;
for(int i = 0; i < 5; ++i)
{
    scanf("%d", &z);
    s += z;
}
printf("%.2f \n", s / 5.0);
```

- C语言用数组类型（简称数组，array）描述这种由多个同类分量构成的数据群体。

```
int a_group[5], s = 0;
for(int i=0; i < 5; ++i)
{
    scanf("%d", &a_group[i]);
    s += a_group[i];
}
printf("%.2f \n", s / 5.0);
```

```
#define N 5

int a_group[N], s = 0;
for(int i=0; i < N; ++i)
{
    scanf("%d", &a_group[i]);
    s += a_group[i];
}
printf("%.2f \n", (float)s / N);
```

- 它是一种由程序员构造出来的派生类型，具体包括一维数组、二维数组和 multidimensional arrays，可用来定义相应的数组变量（通常也简称为数组）。
- 一个数组包括多个 **类型相同的元素**。元素的类型为数组的 **基类型**。每个元素由一个或多个索引（index，即 **下标**）唯一标识。
- 一个数组中的所有元素在内存中占据 **连续的空间**。
- 元素的个数为 **数组的长度**，一般为大于1的常量。从C99标准开始允许数组的长度为变量，不过必须在定义数组前确定该变量的值，以便分配空间。VC6、VS2008、TC等开发环境中，数组长度需为常量，DevC++等开发环境允许数组长度为变量。
- 通常情况下，对数组 **不能进行整体操作**。

一维数组

一维数组是常见的数组形式，用来表示向量这类数据。

一维数组类型的构造

- 一维数组类型由元素类型关键字、一个中括号和一个整数（表示数组的长度）构造而成。
- 可以给构造好的数组类型取一个别名，作为数组类型标识符，C语言规定该别名写在中括号前边。
- 比如，

```
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

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

- 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; // 定义了一个一维数组a
```

- 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
```

- 一维数组名（标识符a），可以代表第一个元素在内存的地址

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

2293416

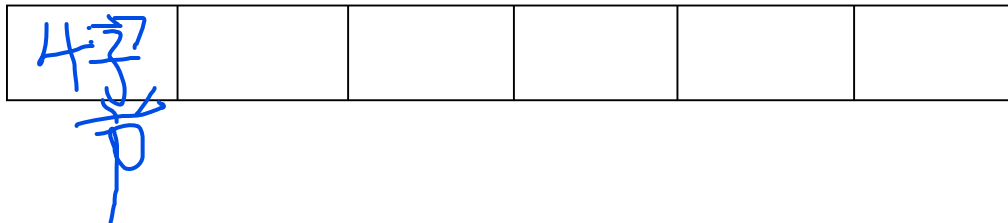
```
printf("%x \n", a);
```

22fea8

16

- 程序执行到数组定义处，即意味着系统要在内存为该数组分配一定大小的空间，以准备存储其各元素的值。

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



- 数组所占内存空间的大小可以用sizeof操作符来计算。比如，

```
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

- 用一对花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`

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

● 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`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]`

– `[]` 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。

– 如果下标为**`N`**则表示长度为**`N`**的数组中的某个元素，即**越界**。C语言一般不对下标是否越界进行检查，程序员必须仔细处置这个问题。

● 对于数组元素的依次访问通常又叫数组的遍历 (`travel`)

例1 求某班50位同学某门课的平均成绩。

...

```
#define N 50

int main( )
{
    int sum = 0, score[N];
    float average;
    for(int i = 0; i < N; ++i)
    { printf("Input a score: ");
        scanf("%d", &score[i]);
        sum += score[i];
    }
    average = (float)sum / N;
    printf("%f \n", average);
    return 0; }
```

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]

用符号常量表示数组的长度，易于维护，调试程序时可以将其值修改为5，以便减少输入的工作量。

用数组存储多个成绩，不仅可以使使用循环流程实现每个元素的输入与累加，还可以保证后续程序可以继续访问每个成绩

- 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2 求一维数组的最大值。

...

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

```
{
```

```
    int a1[10] = {12,1,34}, a2[20] = {23,465,34}, index_max;
```

```
    index_max = Max(a1, 10);
```

```
    printf("数组a1的最大元素是: %d \n", a1[index_max]);
```

```
    index_max = Max(a2, 20);
```

```
    printf("数组a2的最大元素是: %d \n", a2[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;  
}
```

↓ 下标

```
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 a1[10]={12,1,34,1,2,3,4,5,6,7}, a2[20]={23,465,34}, index_max;
  index_max = Max(a1, 10);
  printf("数组a1的最大元素是: %d \n", a1[index_max]);
  index_max = Max(a2, 20);
  printf("数组a2的最大元素是: %d \n", a2[index_max]);
  return 0;
}
```

index_max	?	2
	...	
	0	
	34	
	465	
	23	
	...	
	0	
	34	
	1	
0x20000000	a1	12

j	?	2
i	?	num
Max返址	0x80000003	
x	0x20000000	
num	10	

0x80000003
0x80000013


```

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 a1[10]={12,1,34}, a2[20]={23,465,34}, index_max;
  index_max = Max(a1, 10);
  printf("数组a1的最大元素是: %d \n", a1[index_max]);
  index_max = Max(a2, 20);
  printf("数组a2的最大元素是: %d \n", a2[index_max]);
  return 0;
}

```

index_max	?	1
	...	
	0	
	34	
	465	
	23	
	...	
	0	
	34	
	1	
0x20000000	a1	12

Max返回值

1

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

- **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**，而不是实参数组的所有元素，函数的形参并不会得到足够的存储空间存放所有数据，而只是接收、存储实参数组的起始位置，以便执行被调函数体时，可以到实参数组所在存储空间获取数据。这种方式**可提高程序的执行效率，节省存储空间**。

- **返回值类型不能是数组。**

- 可以返回数组的下标、数组的某个元素或数组的起始位置

特殊的做法（根据具体需求）

```
index_max = Max(a1, N);  
index_max = Max(a1, N/2);
```

```
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;  
}
```

```
if(n1+n2 <= N)  
    index_max = Max(a1, n1, n2);
```

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

二维数组

二维数组也是一种常见的数组类型，用来描述矩阵等具有二维结构的数据，第一维称为行，第二维称为列，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

- 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 b;    //定义了一个二维数组b
```

- 构造类型的同时定义变量

```
int b[3][2]; // int、[]和3、2构造了一个二维数组类型
```

```
//并用该类型定义了一个二维数组b
```

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

- C语言按行**优先策略**存储二维数组，即先存储第一行的元素，再存储第二行的元素，等等。



- 标识符**b**是二维数组名，**可以代表第一行元素在内存的地址**。

二维数组的初始化

自行上机验证

• `int b[3][2] = {0, 1, 2, 3, 4, 5};`

• 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,

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

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

二维数组的操作

❶ 不能进行整体操作

❷ 对二维数组的遍历，通常要采用**嵌套**的循环流程。

➤ 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**开始的第*i*+1行的第*j*+1个元素 (*i*: 0~2, *j*: 0~1)

```
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

– **<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。

– 如果行下标为*M*或列下标为*N*，则不表示数组中的某个元素，即越界了。

`int b[3][2];` // 共有6个元素:

`b[i][j]` (i: 0~2, j: 0~1)

`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)

```
for(int i = 0; i < 3; ++i)
    for(int j = 0; j < 2; ++j)
        scanf("%d", &b[i][j]);
```

例3 求矩阵的和。

...

```
#define M 10
#define N 5
int main( )
{
    int sum = 0, mtrx[M][N];
    for(int i = 0; i < M; ++i)
        for(int j = 0; j < N; ++j)
        {
            printf("Input a number: ");
            scanf("%d", &mtrx[i][j]);
            sum += mtrx[i][j];
        }
    printf("sum = %d \n", sum);
    return 0;
}
```

eg.

```
int MonthDays[][13] = {{0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},  
                        {1, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}};
```

```
MonthDays[leap][month]
```

- 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4 用函数实现求矩阵的和。

...

```
#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**，否则，与实参类型不匹配。

实参：第一行（N个）元素在内存的地址；

形参：一个可以存放某行（N个元素）地址的空间。

```
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;
```

```
}
```

实参：第一行（N个）元素在内存的地址；
形参：？

...

```
#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;
```

```
}
```

```
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;
```

```
}
```

...

```
#define N 5

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

int main( )
{   int m[40][20] = {.....};
    printf("sum = %d \n", Sum(m, 40)); // X
    return 0;
}
```

```
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;
}
```

- 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为整个数组分配内存空间，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

- 多维数组可以用来表示数据立方体等高维数据，应用于图像处理、销售记录分析等领域。

- 多维数组类型由元素类型关键字、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。

- 比如，

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

- //定义了一个三维数组，元素个数为24 ( $= 2 \times 3 \times 4$ )

# 数组及其应用

---

- 一维数组

- 二维数组

- 多维数组

- 数组的应用

- ◆ 可以利用数组存储一组有序数据

- 基于数组的排序
- 斐波那契数求解
- 矩阵相乘（二维数组）
- 杨辉三角（二维数组）
- ...

- ◆ 可以利用数组存储一组有序标志，以对应一组数据的状态

- 用筛法求素数
- 约瑟夫问题求解
- 鞍点（二维数组）
- ...

# 基于一维数组的排序程序

---

- 排序问题是一种常见的非数值计算问题。
- 解决排序问题的算法有很多，比如冒泡排序、选择排序、插入排序、快速排序等等，以及它们的变种。
- 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，需要实现
  - 数组的遍历、两个元素的比较、交换、一个元素的插入等操作，
  - 这些操作往往是通过综合运用循环、分支流程控制及赋值操作等技术来完成的。

## ● 例5 基于数组的排序。

...

```
#define N 4
```

```
void Sort(int sdata[], int count);
```

```
int main()
```

```
{
```

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

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

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

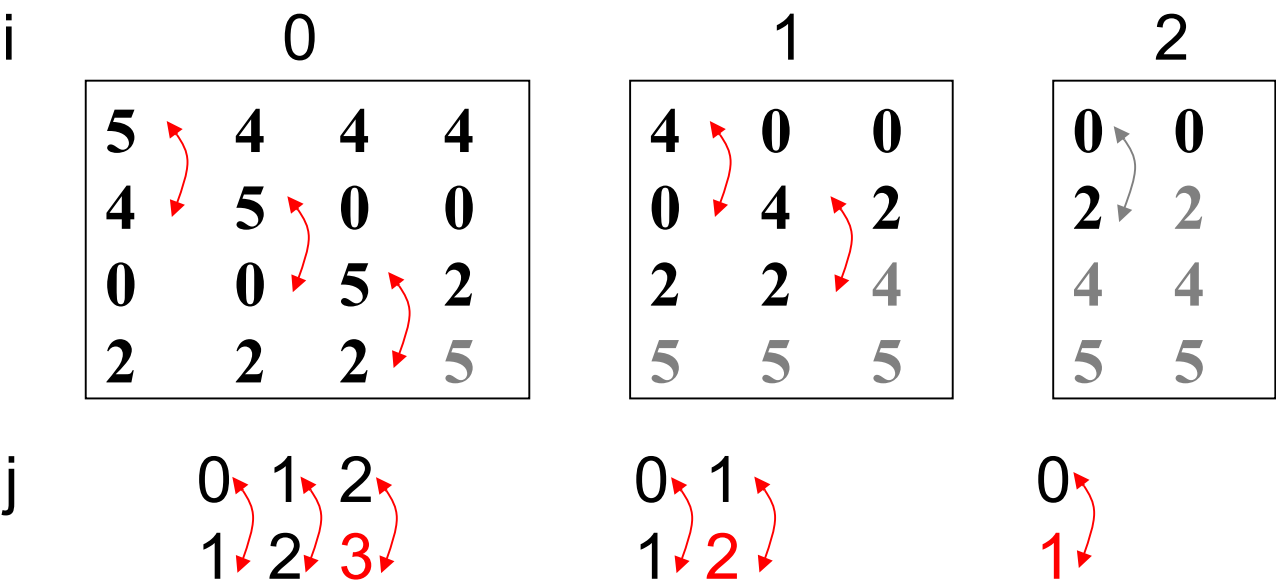
```
 printf("%d \t", a[i]);
```

```
 return 0;
```

```
}
```

# 冒泡法排序

- 比较相邻两个数，大的调到后头
- N个数排N-1趟，每趟内比较的次数随趟数递减。



int a[4]

|                      |   |  |
|----------------------|---|--|
| 输入4个数给a[0]到a[3]      |   |  |
| for(i=0; i<3; ++i)   |   |  |
| for(j=0; j<3-i; ++j) |   |  |
| a[j]>a[j+1]          |   |  |
| T                    | F |  |
| a[j]↔a[j+1]          |   |  |
| 输出a[0]到a[3]          |   |  |

推广到N个数

交换

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

交换

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

交换

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

越界问题！

越界问题！  
不同的变量？

int a[N]

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] ↔ a[j+1] |   |

输出a[0]到a[N-1]

```
#define N 4
int main()
{
 int a[N];
 for(int i = 0; i < N; ++i)
 scanf("%d", &a[i]);

 for(int i = 0; i < N; ++i)
 printf("%d \t", 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])
 {
 int temp=a[j];
 a[j]=a[j+1];
 a[j+1]=temp;
 }
```

```
#define N 4
```

```
int main()
```

```
{ int a[N];
 for(int i=0; i<N; ++i)
 scanf("%d", &a[i]);
```

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

```
...
```

```
}
```

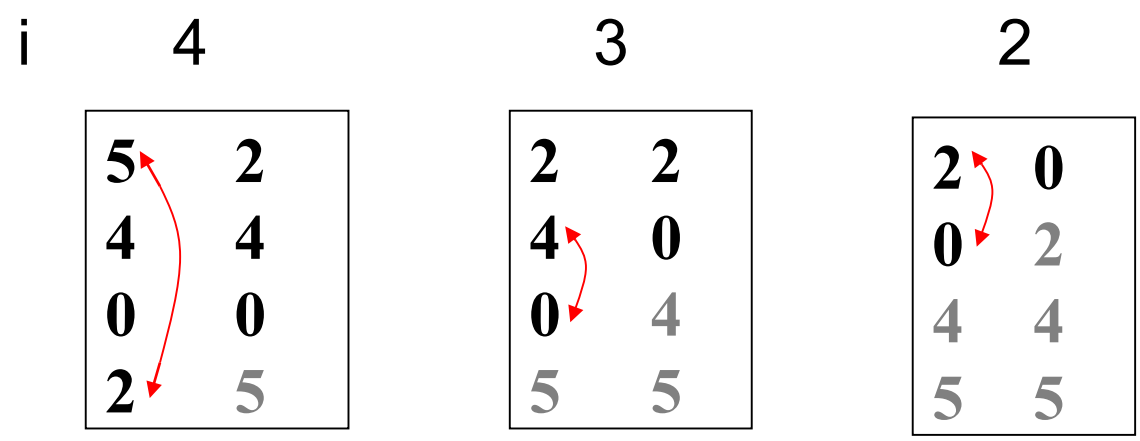
函数名改为**Sort**，再与前面的**main**函数一起调试

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



# 选择法排序

- 从N个数中找出最大者，与第N个数交换位置；然后从剩余的N-1个数中再找出最大者，与第N-1个数交换位置；…；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。



if(max != i-1)

|                      |   |
|----------------------|---|
| 输入N个数给a[0]到a[N-1]    |   |
| for(i=N; i>1; --i)   |   |
| 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 count)
```

```
{
```

```
 for(int i = count; i > 1; --i)
```

```
 {
```

```
 int max = 0;
```

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

```
 if(sdata[max] < sdata[j])
```

```
 max = j;
```

---

```
 if(max != i-1)
```

```
 {
```

```
 int temp = sdata[max];
```

```
 sdata[max] = sdata[i-1];
```

```
 sdata[i-1] = temp;
```

```
 }
```

```
 }
```

```
}
```

若改为“for(int i = 1; i < count; ++i)”，  
程序中的其他代码应做哪些相应修改，  
可以实现原功能？

# 快速法排序\*

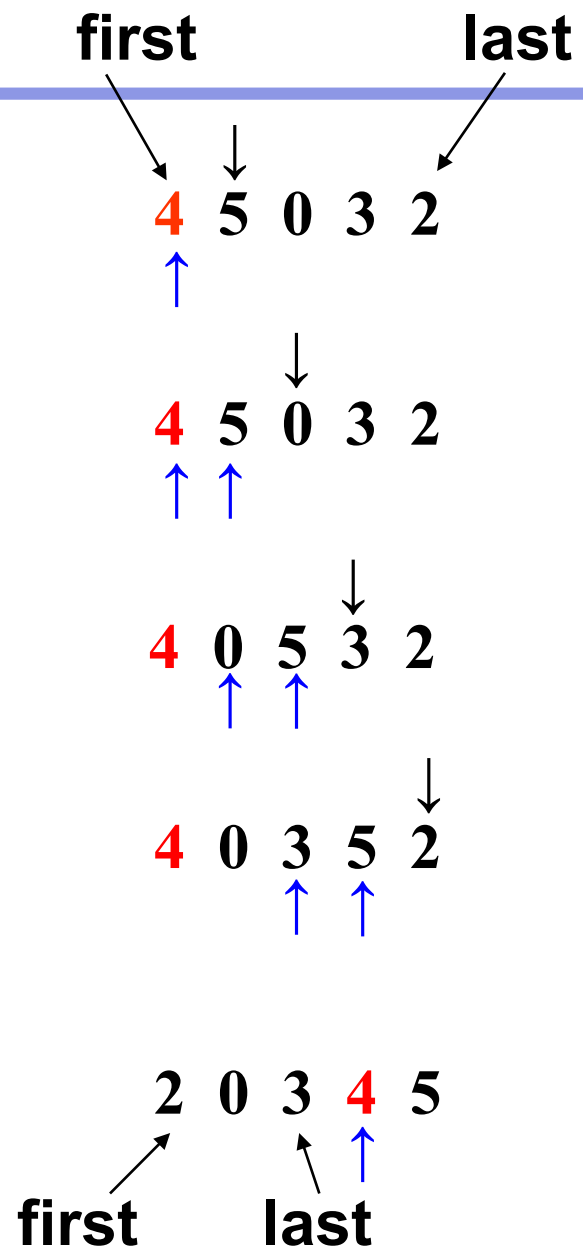
---

- 是对冒泡法的一种改进
- 基本思想：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，整个排序过程可以递归进行，以达到整个序列有序
- 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。

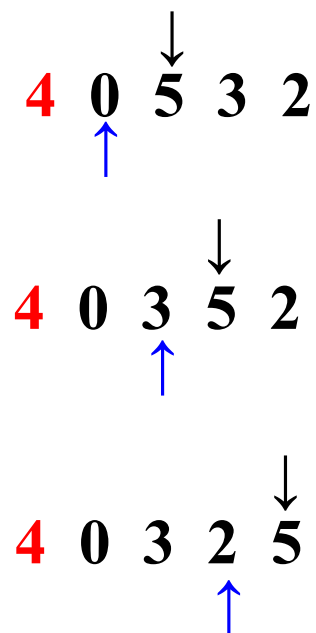
## ● 一趟快速排序的思想：

首先任意选取一个数据（通常选第一个数据）作为关键数据（转换值），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split\_point的初始值为0，其对应元素推选为pivot（转换值），从first所指位置后起，向后搜索，每遇到小于转换值的元素，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。



**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)
{
 if(first < last)
 {
 int split_point = Split(sdata, first, last);
 //寻找分割点
 QuickSort(sdata, first, split_point-1);
 //对分割点以左的部分进行排序
 QuickSort(sdata, split_point+1, last);
 //对分割点以右的部分进行排序
 }
}
```

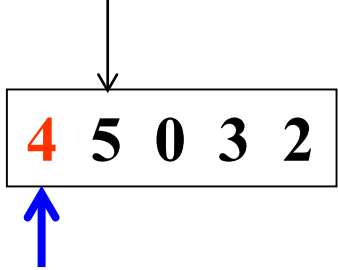
```
int Split(int sdata[], int first, int last)
{
 int pivot = sdata[first];
 int split_point = first;
 for(int unknown = first+1; unknown <= last; ++unknown)
 if(sdata[unknown] < pivot)
 {
 ++split_point;
 int t = sdata[split_point];
 sdata[split_point] = sdata[unknown];
 sdata[unknown] = t;
 }
 sdata[first]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

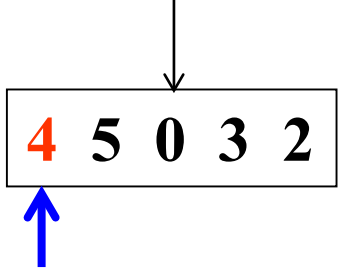
```
int Split(int sdata[], int first, int last)
{
 int pivot = sdata[first];
 int split_point = first;
 for(int unknown = first+1; unknown <= last; ++unknown)
 if(sdata[unknown] < pivot)
 {
 ++split_point;
 int t = sdata[split_point];
 sdata[split_point] = sdata[unknown];
 sdata[unknown] = t;
 }
 sdata[first]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

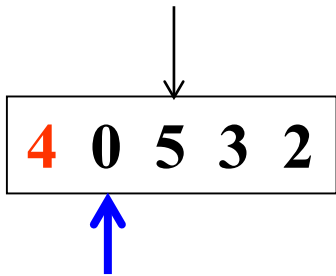


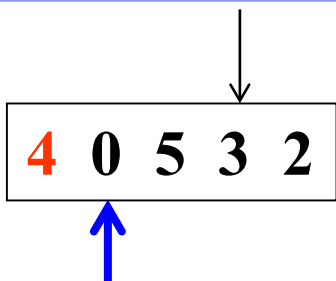
# 令第一个元素为 pivot

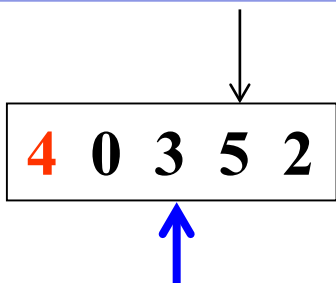
---

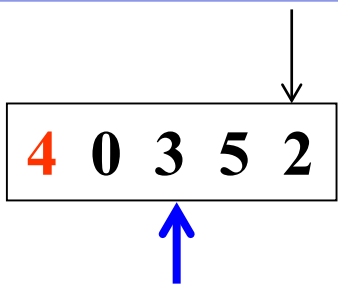


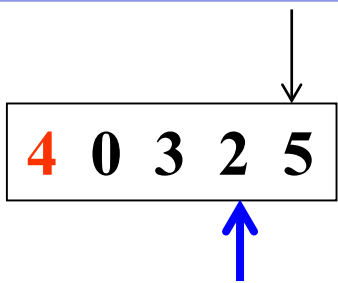










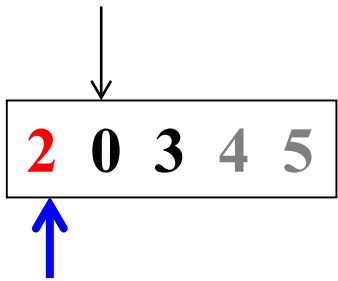
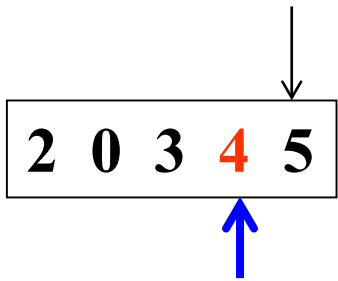


---

2 0 3 4 5

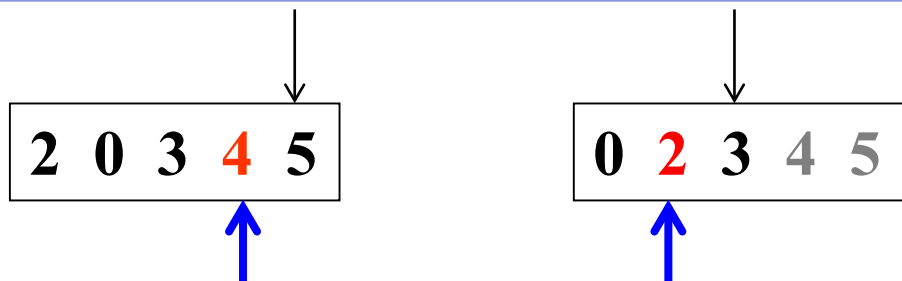












```
int pivot = a[first]; //以首元素为基准
int i = first;
for(int unknown = first+1; unknown <= last; ++unknown)
 if(a[unknown] < pivot)
 swap(a[++i], a[unknown]);
swap(a[first], a[i]);
return i;
```

## ● 实现一维数组上的排序功能涉及哪些程序设计要素？

- 循环流程控制
- 分支流程控制
- 赋值操作
- 比较操作
- 算术操作

## 例6 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:

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

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

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

...

```
#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 \t", mc[i][j]);
 if((j + 1) % P == 0) printf("\n"); //换行
 }
 return 0; }
```

|    |    |
|----|----|
| 22 | 22 |
| 44 | 44 |
| 66 | 66 |

---

```
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;
 }
}
```



❁ **例7 求解约瑟夫斯 (Josephus) 问题：**有 $n$ 个囚犯站成一圈，准备被处决。从某个人开始顺时针报数，报到 $k$ 的囚犯从圈子离开，被处决；然后，从下一个人开始重新报数，每报到 $k$ ，相应的囚犯从圈子离开，被处决；最后只剩下一个人，该囚犯可以继续活着；这位幸运的囚犯一开始站在什么位置？

❁ **分析：**

- 采用一维数组`in_circle[n]`表示 $n$ 个囚犯围成一圈
- `in_circle[index]`为`true`表示编号为`index`的囚犯在圈子里
- 剩下的人数 `numRemained:  $n \rightarrow 1$`
- 从`index`为0的囚犯开始报数，圈子中`index`的下一个位置为 $(index+1) \% n$ （下一圈连续报数）

---

...

```
#define N 20
```

```
#define K 5
```

```
int Josephus(int n, int k);
```

```
int main()
```

```
{
```

```
 printf("The survival is No.%d \n", Josephus(N, K));
```

```
 return 0;
```

```
}
```

---

```
int Josephus(int n, int k)
{
 bool in_circle[n]; //bool in_circle[N];
 int index;

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

```
int numRemained = n;
index = 0; n-1
while (numRemained > 1)
{
 int count = 0;
 while (count < k)
 {
 index = (index+1)%n;
 if (in_circle[index])
 ++count;
 index = (index+1)%n;
 }
 in_circle[index] = false; //囚犯离开圈子
 numRemained--; //圈中人数减1
}
```

`in_circle[index-1] = false;`  
//可能会出现-1

---

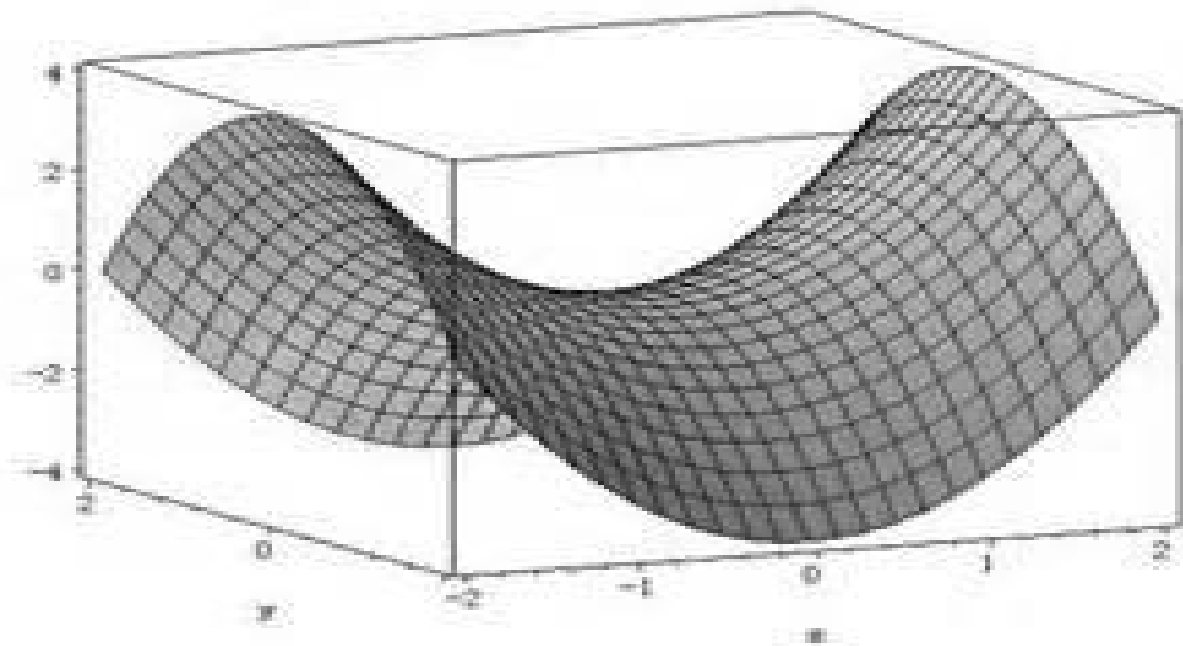
```
//找最后一个囚犯
for(index = 0; index < n; ++index)
 if(in_circle[index])
 break;

return index;
}
```

```
The survival is No.6
```

注意：  
break;

- 例8：编写一个程序，计算一个矩阵的鞍点。（矩阵的鞍点是指矩阵中的一个位置，该位置上的元素是其所在行上的最大值且是一个极大值，所在列的最小值且是一个极小值。一个矩阵可能有多个鞍点，也可能没有鞍点。）



|   |   |   |   |   |
|---|---|---|---|---|
| 3 | 7 | 7 | 9 | 3 |
| 3 | 6 | 3 | 6 | 3 |
| 4 | 5 | 4 | 5 | 4 |
| 3 | 6 | 3 | 6 | 3 |
| 3 | 8 | 3 | 8 | 3 |

```
3 7 3 9 3
3 6 3 6 3
4 5 4 5 4
3 6 3 6 3
3 8 3 8 3
```

```
Saddle Point: a[2][1]: 5
Saddle Point: a[2][3]: 5
```

```
#define M 5
#define N 5
int i, j, l, k, a[M][N] =
{{3, 7, 3, 9, 3}, {3, 6, 3, 6, 3}, {4, 5, 4, 5, 4}, {3, 6, 3, 6, 3}, {3, 8, 3, 8, 3}};

for(i=0; i < M; ++i)
{
 for(j=0; j < N; ++j)
 printf("%d \t", a[i][j]);
 printf("\n");
}
printf("\n");
```

---

```
bool flag_max[M][N] = {false};

for(i=0; i < M; ++i)
{
 k = 0;
 for(j=1; j < N; ++j)
 if(a[i][j] > a[i][k])
 k = j; //第i行找最大数，下标存k

 for(j=0; j < N; ++j)
 if(a[i][j] == a[i][k])
 flag_max[i][j] = true;
 //第i行所有最大数对应true，处理多个鞍点
}
```



---

```
bool flag_min[M][N] = {false};

for(j=0; j < N; ++j)
{
 k = 0;
 for(i=1; i < M; ++i)
 if(a[i][j] < a[k][j])
 k = i; //第j列找最小数，下标存k

 for(i=0; i < M; ++i)
 if(a[i][j] == a[k][j])
 flag_min[i][j] = true;
 //第j列所有最小数对应true，处理多个鞍点
}
```

---

```
bool flag_extremum[M][N] = {false}; //初始化极值标志
```

```
for(i=1; i < M-1; ++i)
{
 for(j=1; j < N-1; ++j)
 if(a[i][j] != a[i][j-1] //左
 && a[i][j] != a[i][j+1] //右
 && a[i][j] != a[i-1][j] //上
 && a[i][j] != a[i+1][j]) //下
 flag_extremum[i][j] = true;
 //第i行第j列元素为极值
}
```

---

```
bool count = false; //初始化无鞍点标志
```

```
for(i=1; i < M-1; ++i)
{
 for(j=1; j < N-1; ++j)
 if(flag_max[i][j] && flag_min[i][j] && flag_extremum[i][j])
 {
 printf("Saddle Point: a[%d][%d]: %d \n", i, j, a[i][j]);
 count = true;
 }
}
if(!count)
 printf("not any Saddle Points. \n");
```

# 小结

---

## 数组：

- 一种派生数据类型，表示固定多个同类型的有序数据群体
- 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
  - 一个程序代码量 $\approx$ 100行
- 继续保持良好的编程习惯

# Thanks!

---

