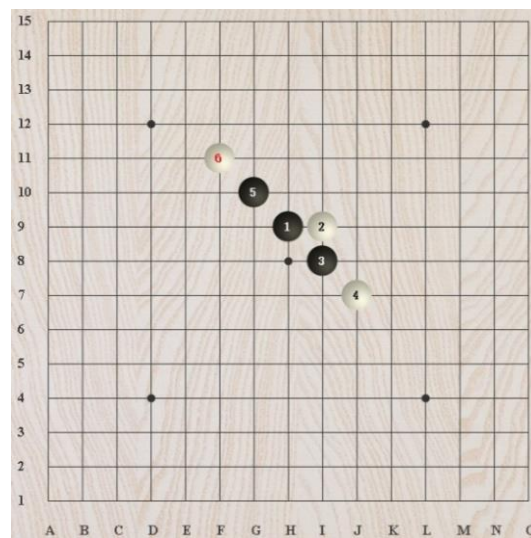


# 第六讲

## 数组 array



比赛排名

更新中, 上次更新于 2019-11-01 10:19:45

«

<

1

2

3

4

5

6

7

...

10

>

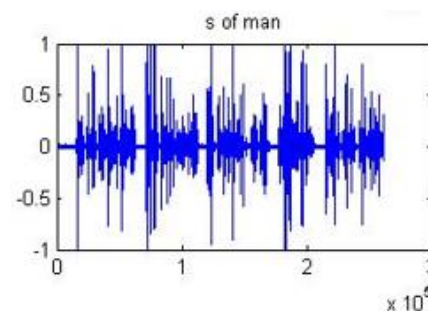
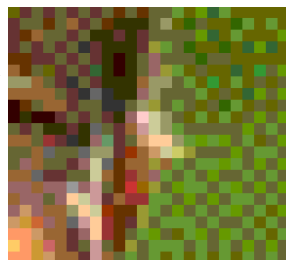
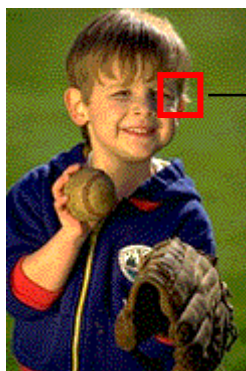
»

排名	用户	得分	罚时	A 864/904	B 686/707	C 327/65
	<div></div>					
101	高野淇	890	645:44:35	57:18:16(+2)	57:49:35	85:20:32
102	廖起湘	890	667:37:32	10:26:56(+1)	25:13:18(+2)	100:13:52
103	孙一凡	870	540:08:05	7:38:27(+1)	11:42:15(+2)	15:14:49
104	纪鹏亮	870	556:48:53	18:19:17(+7)	2:16:04	73:16:57
105	马柯杰	860	282:53:20	1:46:51	2:00:48(+1)	109:33:22
106	赵家璇	860	383:01:13	6:21:48	102:56:32(+1)	8:15:26
107	彭律章	860	880:59:30	102:43:19	58:36:27	102:51:46
108	吴凌轩	850	419:28:22	11:20:24	11:50:12(+3)	77:06:27

# 第6章 数组

## 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



	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	

W: 122 pixels

H: 182 pixels

# 第6章 数组

---

## 学习要点

1. 数组的结构、存储方式
2. 一维数组、二维数组的定义、初始化、访问
3. sizeof 的用法
4. 字符串与字符数组的关系
5. 标准库字符串处理函数
6. 数组作为函数参数
7. 一些基本的算法设计：基本的查找和排序方法等
8. 基于数组的简单数据结构（队、栈、散列表）
9. 多维数组简介

# 数组存储与访问【复习】

- 含义：具有**相同名称和相同类型**的一组**连续内存地址**
- 定义：告诉编译器元素个数、每个元素数据类型、数组首地址

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

类型 数组名[常量表达式];

- 初始化：在数组定义时通过{逗号分隔的值列表}实现显示或隐式赋值

```
int a[6] = {1, 3, 5, -2, -4, 6};
int b[6] = {1, 3};
int c[6] = {1, 2, 3, 4, 5, 6, 7};
int d[ ] = {1, 2, 3, 4, 5};
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<7; i++ )
    printf("%d ", c[i]); printf("\n");
for ( i=0; i<6; 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]);
```

- 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 3 5 -2 -4 6
1 3 0 0 0 0
1 2 3 4 5 6 1
1 2 3 4 5 1
0 0 0 0 0 0
6356628 1951818084 12061788 4199040 4201024 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，就不要在链接时发现；能在链接时发现的，就不要在运行时发现。**

# 关于数组越界访问【复习】

## 为什么数组没有界限检查？



德国Füssen小镇火车站  
(新天鹅堡的小镇)



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语言没有界限检查也是为了高效！

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!

# 数组的访问、越界等【复习】

- 读取：对n个元素的数组，下标读取，下标范围从 0 ~ (n-1)
- 赋值：下标赋值
- 长度： **#define ArrayNum(x) (sizeof(x)/sizeof(x[0]))**

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

- $b = a$ ，语法错误，不能把数组整体赋值给另一个数组
- $b[5] = a[5]$ ，逻辑错误，地址越界，但编译器不报错
- $b[5] = \{1, 2, 3, 4\}$ ，语法错误，数组除初始化外，不能用{值列表}
- 两个数组赋值需要通过循环逐一赋值数组元素

注意：数组元素访问时，C语言也不进行下标的越界检查。下标越界是逻辑错误（不是语法错误），可能丢失数据或运行错误。隐蔽，严重问题！

```
#define ArrayNum(x) (sizeof(x)/sizeof(x[0]))
int main(){
    int i; float f[10];
    for ( i=0; i<ArrayNum(f); i++ ){
        f[i] = i*i;    printf("%f\n", f[i]);
    }
    printf("%d, %d\n", sizeof(i), sizeof(int));
    printf("%d, %d\n", sizeof(f), sizeof(f[0]));
}
```

**sizeof(para)**：计算参数para所占的字节数，参数可以是变量、数组、类型名称

# 数组的拷贝【复习】

## 数组复制

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

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

```
void *memcpy(void *dest, void *src, size_t count);
```

将数组src中count个元素逐一拷贝到数组dest


```
void *memset(void *s, int ch, size_t n);
```

将s中当前位置后面的n个字节用 ch 替换并返回 s，常用于清零等。

# 数组常见错误【复习】

- 整体操作：不能将数组作为整体进行赋值等操作
- 地址越界：不能超出下标范围进行读取
- 长度定义：长度必须是常量或常量表达式，也不能定义长度为空的数组

```
int a[5] = {1, 2, 3, 4};
int b[5];
//整体操作错误
b = a;
if (a == b)
    .....
//地址越界错误
b[5] = 1;
for(i=0;i<6;i++)
    b[i] = i;
//长度定义错误
int length = 10;
double s[length];
int arry[ ];
int n;
scanf("%d", n);
int c[n];
```



正确做法：先定义常量，以常量作为数组长度

```
#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]) ...
```



# \*数组定义时下标应为常数

```
int length = 10;  
double s[length];  
int array[ ];
```

长度必须是常量或常量表达式，不能是变量。  
也不能定义长度为空的数组。

用变量定义数组长度，可能编译通过。书上本身没有错。是我们的dev或cb（或其他IDE等），所采用的编译器有很多扩展的缘故造成跟C标准并不完全一致。

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

先定义常量，以常量作为数组长度，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：这里的变长指的是数组的长度是在运行时才能决定，但一旦决定，在数组的生命周期内就不会再变。） 2) 在 GCC 标准规范的 6.19 Arrays of Variable Length 中指出，作为编译器扩展，GCC 在 C90 模式和 C++ 编译器下遵守 ISO C99 关于变长数组的规范。

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

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

# \*数组定义的大小问题

- 实际处理的问题可能很大，如淘宝数据几亿用户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（但也不宜上百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[1 000 000]的数组会提示stack overflow，但是将数组移到main函数外面，变为全局数组的时候则ok，就感到很迷惑，然后上网查了些资料，才得以理解。

对于全局变量和局部变量，这两种变量存储的位置不一样。对于全局变量，是存储在内存中的静态区（static），而局部变量，则是存储在栈区（stack）。这里，顺便普及一下程序的内存分配知识：

C语言程序占用的内存分为几个部分：

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

在C语言中，一个静态数组能开多大，决定于剩余内存的空间，在语法上没有规定。所以，能开多大的数组，就决定于它所在区的大小了。

在WINDOWS下，栈区的大小为2M，也就是 $2 \times 1024 \times 1024 = 2097152$ 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（我没有在网上找到它具体大小的数据，但是可以肯定比栈区大），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。

课后读物

## 6.1 数组作为函数参数

### 【例6-1】计算n维向量的点积

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

- 函数原型中数组名可省略。
- 对数组元素的访问跟单个变量一样。
- 调用函数实参中直接使用数组名（“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。
- 函数定义形参中数组长度通常省略。
- 数组传递时，数组名作为实参，它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。

```
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;
}
```

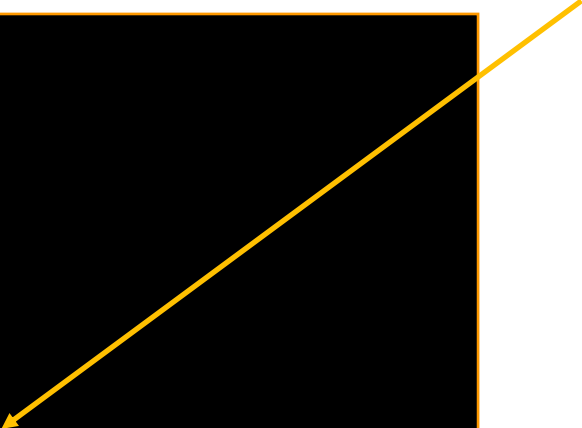
# 数组作为函数参数

## 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。
- 理解：将实参数组的首地址赋值给形参数组，就是它们共享同一个数据区域，即它们是同一个数组
- 定义：**void f( int array[ ], int size) {...}**
- 调用：函数调用时，用数组名作为实参

```
int main()
{
    int a[12];
    ...
    f(a, 12);
    ...
}

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



### 数组作为参数：

- 形参为数组时，数组长度可省略
- 通常需要额外定义形参传递数组长度

\*说明：把数组作为参数传递给函数，实际上只有数组的首地址作为指针传递给了函数中的形参数组。编译器无法获知数组长度，因此，需要额外定义形参来传递数组长度。

## 6.2 数组元素的排序和查找

### 排序 (sort)

- 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?
- 功能: 将一组原始数据按递增或递减的规律进行重新排列的过程
- 种类: **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、加速堆排序
- 排序是计算机科学中大量研究的问题, 是一个非常基础、非常重要的问题, **必须熟练掌握!**

常用算法动态演示

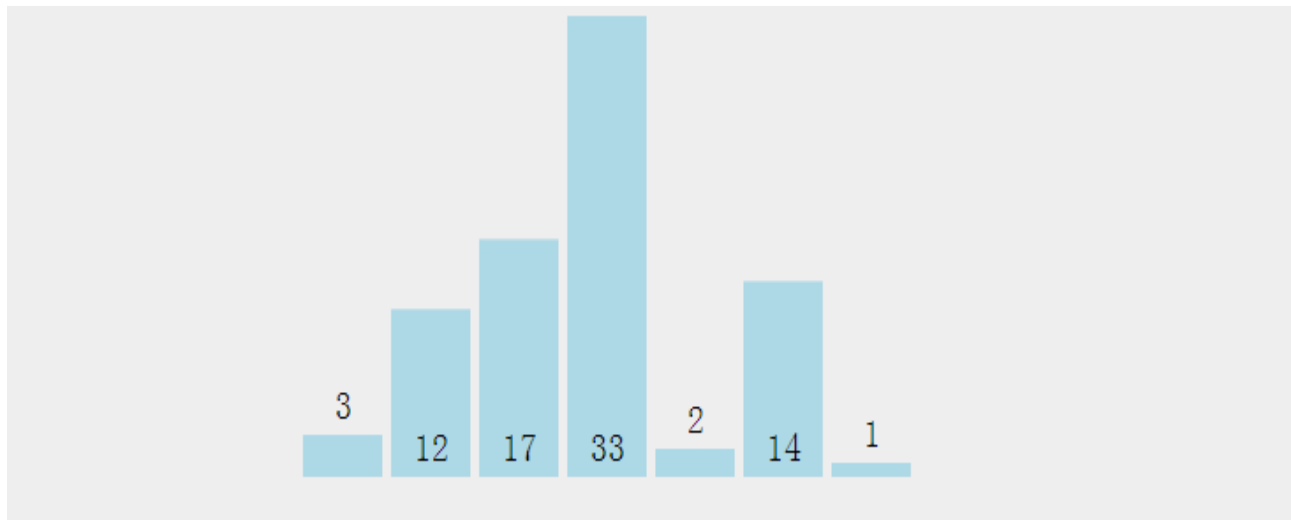
<http://runningls.com/demos/2016/sortAnimation/>

github: <https://github.com/liusaint/sortAnimation>

## 6.2.1 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。
- 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
---	---	---	----	----	----	----

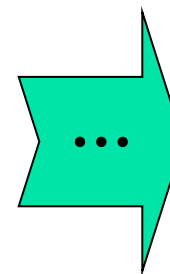
# 冒泡排序

## 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

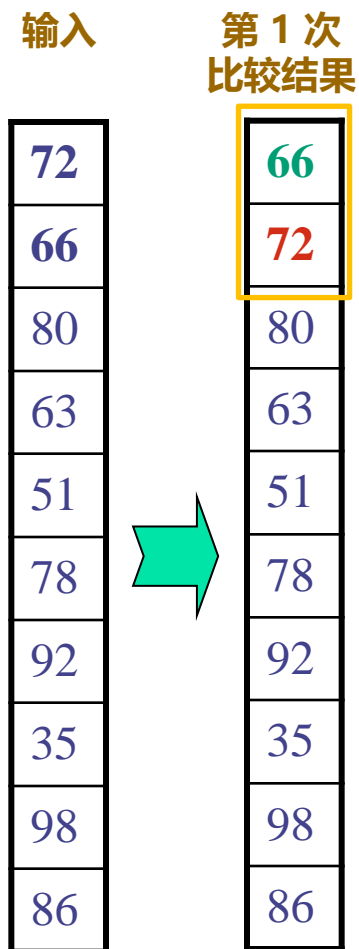


输出

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

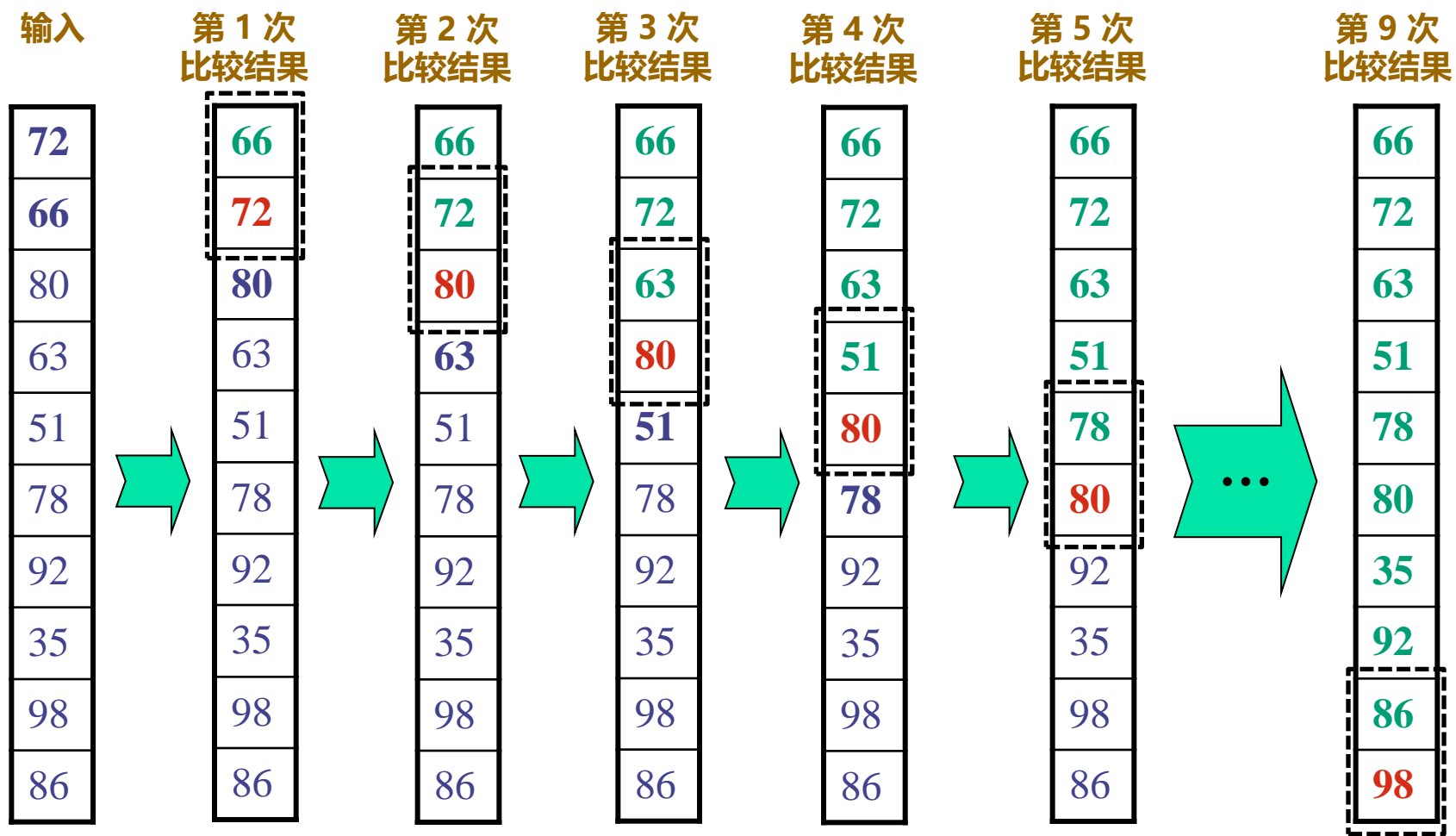
- 升序冒泡排序过程：第 5 次比较结果

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）



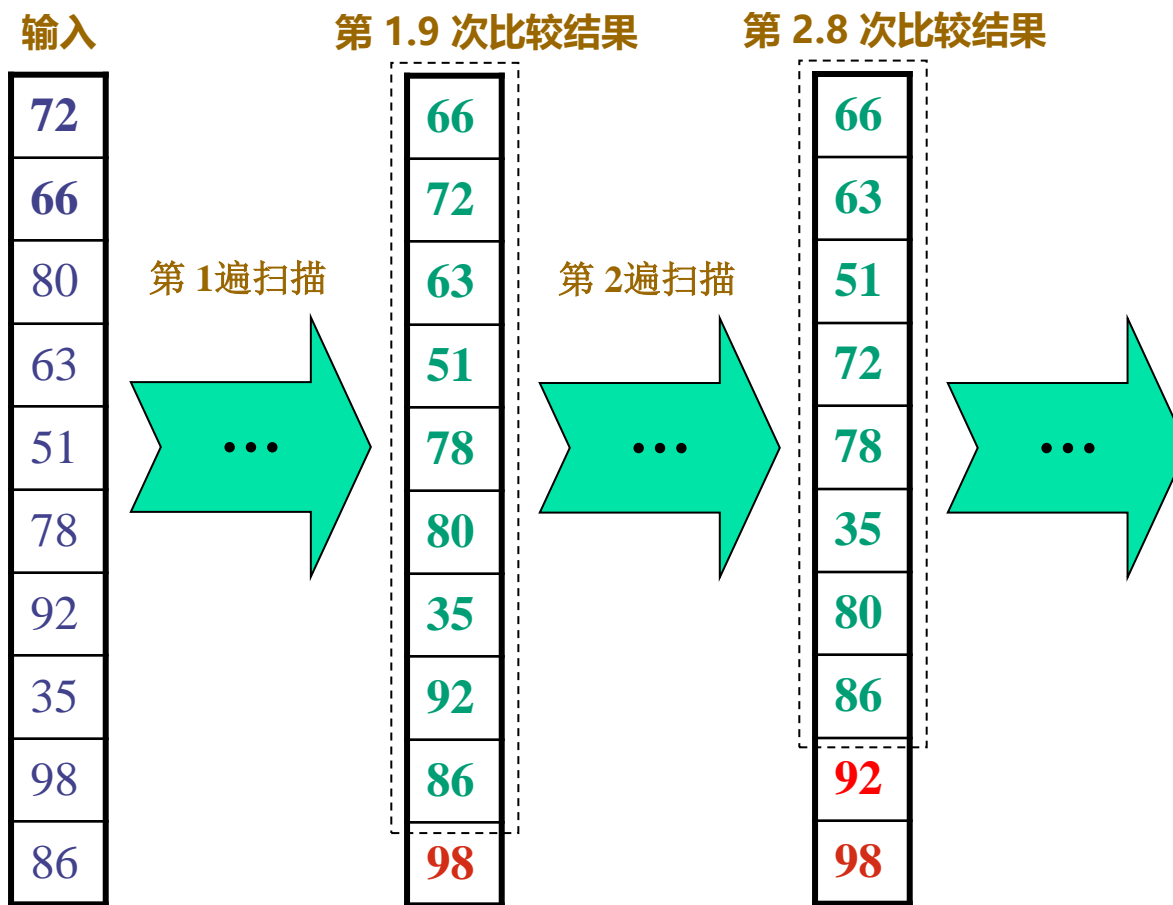
# 冒泡排序原理

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

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

算法实现：

- 两两比较相邻数据
- 反序则交换
- 直到全部遍历结束



算法过程：

- 对当前还未排好序的范围内的全部数，自上而下对相邻的两个数依次进行比较
- 让较大的数往下沉，较小的往上冒

# 冒泡排序的优化

## • 优化的冒泡算法 (\*)

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

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

# 排序问题中两个变量的互换

- 两个变量的互换需要借助第3个变量

例： $a[i]$  和  $a[i+1]$  的值互换

```
hold = a[i];
```

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

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

- 只有两个赋值语句无法进行数值交换（会有数据丢失）

例：

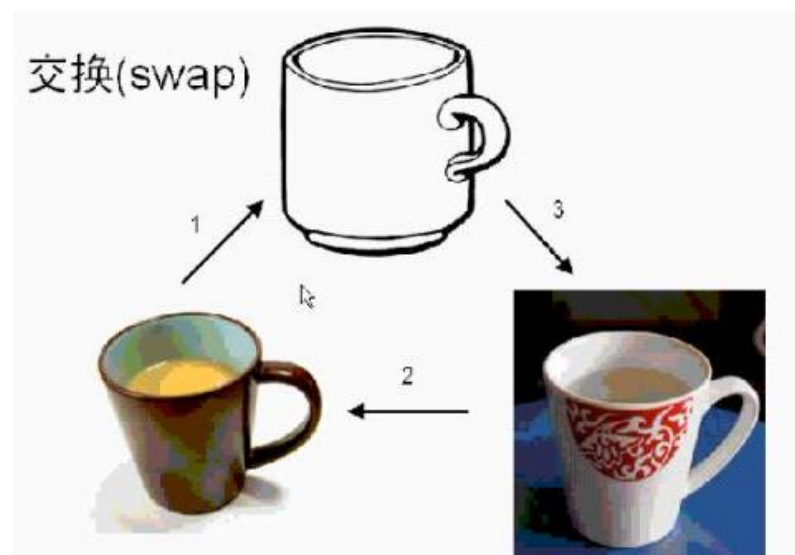
```
a[i] = 5;
```

```
a[i+1] = 7;
```

```
a[i] = a[i+1]; // 执行后 a[i] = 7
```

```
a[i+1] = a[i]; // 执行后 a[i+1] = 7
```

可见，原  $a[i]$  的值 5 丢失了。



# \*冒泡排序性能分析

- 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： $n - 1 + n - 2 + \dots + 2 + 1 = \frac{n(n-1)}{2}$

- 时间复杂度

$O(n^2)$

```
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;
            }
}
```

改进前

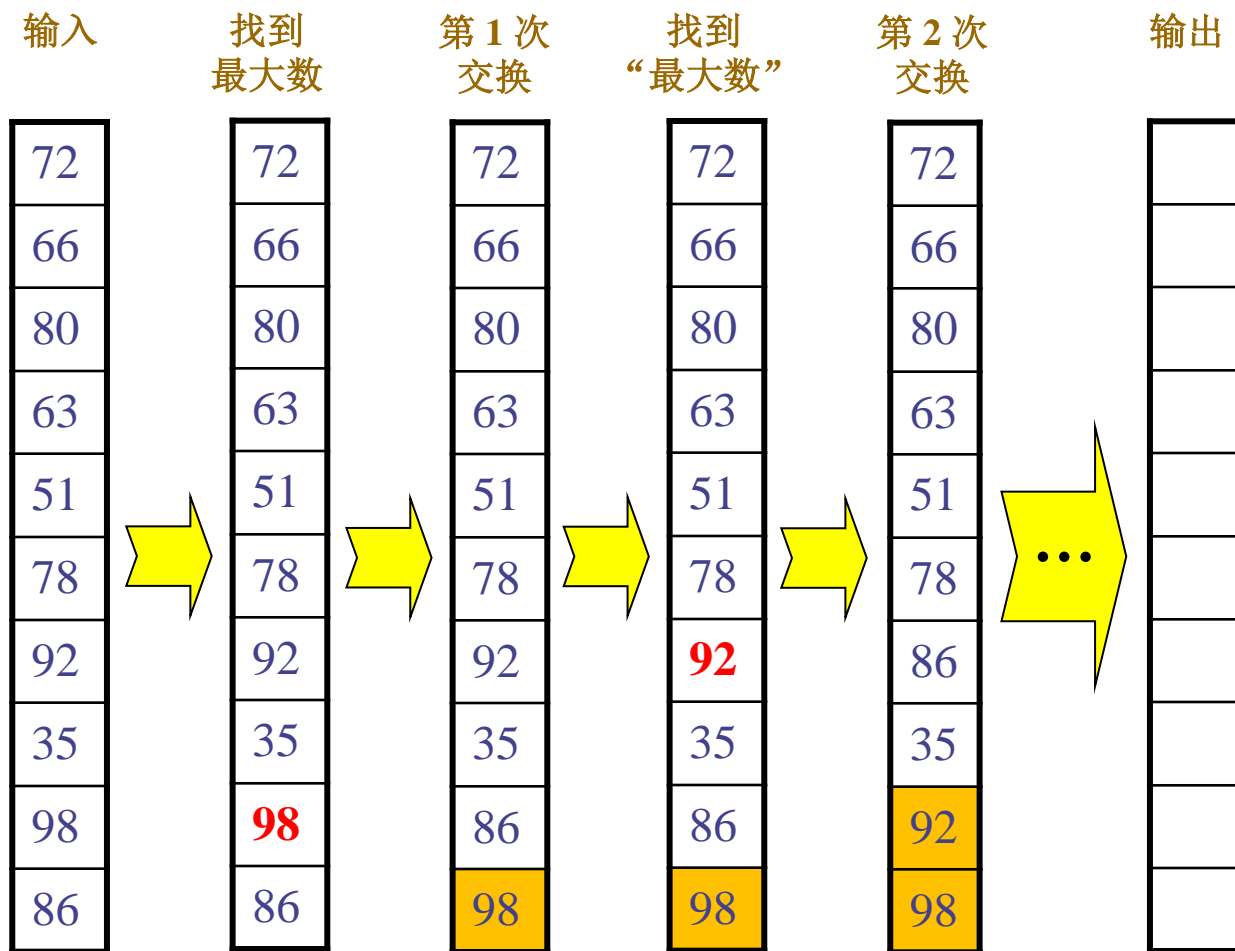
```
void bubbleSort(int a[], int n)
{
    int i, j, hold, flag;
    for( i = 0; i < n-1; i++){
        flag = 0;
        for(j = 0; j < n-1-i; j++)
            if(a[j] > a[j+1]){
                hold = a[j];
                a[j] = a[j+1];
                a[j+1] = hold;
                flag = 1;
    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

改进后

## 6.2.2 选择排序(select sorting)

### 算法

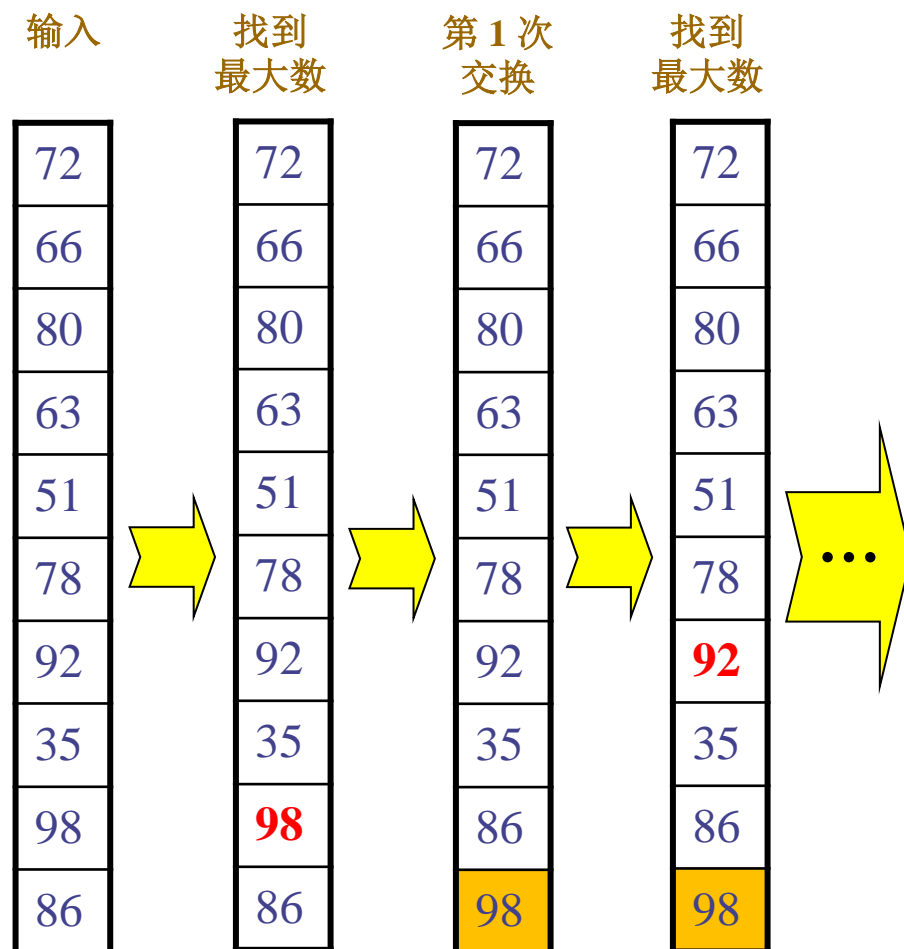
- 从n个数中找出最大（小）者，与第n个数交换位置
- 从剩余的n-1个数中再找出最大者，与第n-1个数交换位置
- .....
- 一直到剩下最后一个数



# 选择排序

```
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  
    }  
}  
  
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;  
}
```

- 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



书上的选择排序算法跟本例实质一样，读者自行体会。

# 选择排序的”小”优化

```
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
    }
}

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;
}
```



```
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
        if(j != i-1) // 最后一个元素最大, 不交换
        {
            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;
}
```



# \*选择排序的性能分析

- 比较次数

- ◆ 最好情况:  $\frac{n(n-1)}{2}$
- ◆ 最坏情况:  $\frac{n(n-1)}{2}$

- 交换次数

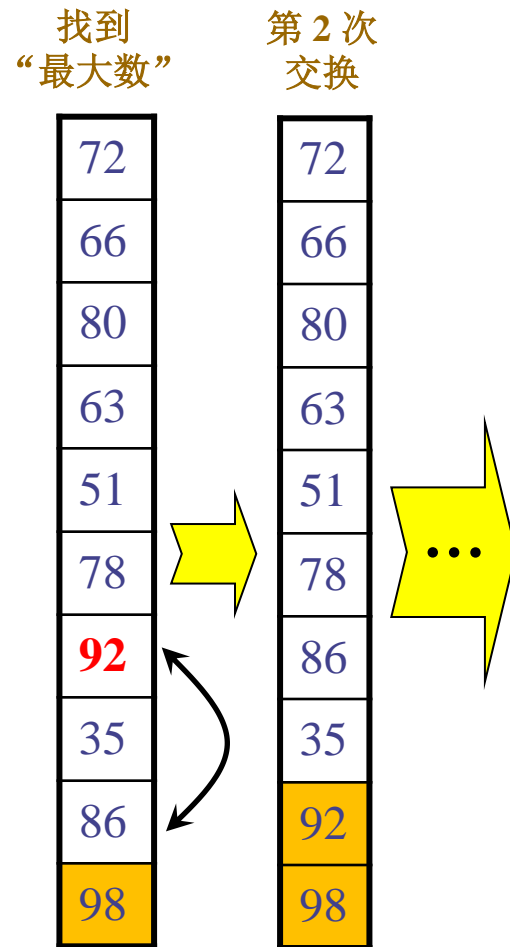
- ◆ 最好情况: 0次
- ◆ 最坏情况:  $n - 1$ 次

- 时间复杂度

$$O(n^2)$$

```
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
    }
}

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;
}
```



## 6.2.3 选择排序【例a6-3】和冒泡排序

```
#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;
}
```

实参 a 和形参 x 都是地址，指向同一片内存，函数中对 x 的改变，实际上就是对 a 的改变。

```
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;
            }
}
```

在如上代码的合适位置，分别写如下测试代码并观察输出，思考为什么？

```
printf("address a: a = %d, &a = %d\n", a, &a);
printf("\naddress x: x = %d, &x = %d\n", x, &x);
```

# \*思考：选择排序 vs 冒泡排序

## 选择排序性能分析

- 比较次数

- ◆ 最好情况： $\frac{n(n-1)}{2}$
- ◆ 最坏情况： $\frac{n(n-1)}{2}$

- 交换次数

- ◆ 最好情况：0次
- ◆ 最坏情况： **$n - 1$ 次**

- 时间复杂度

$$O(n^2)$$

## 冒泡排序性能分析

- 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： $(n - 1) + \dots + 2 + 1 = \frac{n(n-1)}{2}$

- 时间复杂度

$$O(n^2)$$

思考1：  
哪个算法更  
“快”些？

## \*思考：冒泡排序与选择排序的比较

```
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;
    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;
            }
}
```

思考2:

算法的稳定性分析?

(稳定的排序算法: 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)

## \*冒泡与选择的比较示例（2<sup>#</sup>，2表示数值，#是出现顺序）

3	2 <sup>1</sup>	2 <sup>1</sup>	2 <sup>1</sup>	1	1
2 <sup>1</sup>	2 <sup>2</sup>	2 <sup>2</sup>	1	2 <sup>1</sup>	2 <sup>1</sup>
2 <sup>2</sup>	3	1	2 <sup>2</sup>	2 <sup>2</sup>	2 <sup>2</sup>
5	1	2 <sup>3</sup>	2 <sup>3</sup>	2 <sup>3</sup>	2 <sup>3</sup>
1	2 <sup>3</sup>	3	3	3	3
2 <sup>3</sup>	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 <sup>1</sup>	2 <sup>1</sup>	2 <sup>1</sup>	2 <sup>3</sup>	2 <sup>2</sup>	2 <sup>2</sup>
2 <sup>2</sup>	2 <sup>2</sup>	2 <sup>2</sup>	2 <sup>2</sup>	2 <sup>3</sup>	2 <sup>3</sup>
5	2 <sup>3</sup>	2 <sup>3</sup>	2 <sup>1</sup>	2 <sup>1</sup>	2 <sup>1</sup>
1	1	3	3	3	3
2 <sup>3</sup>	5	5	5	5	5

**选择**

- 交换n-1次，“快”
- 不稳定的排序

## 6.2.4 查找(finding, searching)

- 功能：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(key value)等于给定值的数据元素
- 种类：线性查找、折半查找
- 应用：根据学号查找学生，根据成绩挑学生，...

### 查找所需四要素

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

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

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

# 线性（顺序）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
- 前提条件：**无序**或有序数组的查找
- 时间复杂度：O(n)
- 适用：小型数据查找，对大数组，线性查找的效率不高（尤其对需要频繁查找的情况）
- 优点：算法简单，对查找对象没有要求
- 缺点：n很大时，效率低

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

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

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

key: 3

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



# 线性（顺序）查找

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

key 是要查找的值,  
SIZE 是数组大小

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

key == a[i]

## 线性查找

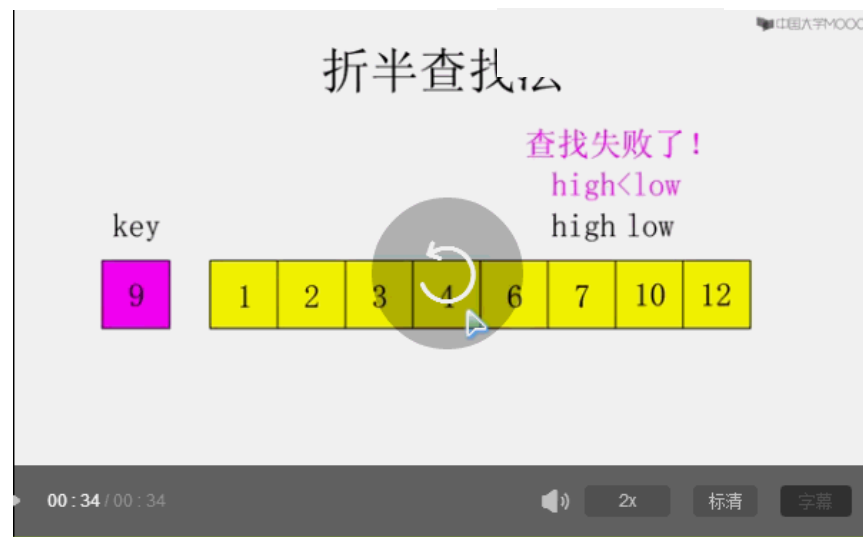
- 被查找的数据范围: int x[]
- 需要查找的关键字: int key
- 查找方法: 线性, 即遍历整个数组
- 查找结果:
  - ◆ 找到, 返回元素下标
  - ◆ 未找到, 返回-1 (当然, 也可以返回其他标记, 由程序员根据实际需要来决定)



# 折半查找（二分查找）

猜数字游戏（我的心里想了一个 1~1000 整数，猜猜是多少）

- 算法：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。
- 前提：数组有序（升序或降序）
- 时间复杂度： $O(\log_2 n)$
- 优点：查找效率高



# 折半查找（二分查找）

## 执行过程：

- (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（二分查找）

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

- 每查找一次，排除所查找数组的一半元素，即除以2
- 查找10亿个元素的数组：线性查找，10亿次比较；折半查找，30次比较
- 折半查找需要排序数组，而排序数组的成本比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，则排序数组是值得的
- 查找次数:  $\log n$        $n / \log n \rightarrow \infty$  ( when  $n \rightarrow \infty$  )

# 折半查找【例a6-5】

a[low]	a[1]	.....	a[mid]	.....	a[high]
--------	------	-------	--------	-------	---------

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

int main()
{
    int a[LEN], key, result, i;
    for( i=0; i < LEN; i++)
        a[i] = 2*i;
    printf("Enter integer search key: ");
    scanf("%d", &key);
    result = rec_bin_find(a, key, 0, LEN-1);
    // result = bin_find(a, key, 0, LEN-1);
    if(result != -1)
        printf("Found, it's a[%d]", result);
    else
        printf("Key not found");
    printf("\nfind times: %d", count_find);
    return 0;
}
```

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

    if( low > high )
        return -1;

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

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

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

# 折半查找另一种实现（数据“固定”，外部数组）

## 函数中不再需要局部数组

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

int main()
{
    int key, result;
    for(i=0; i < LEN; i++)
        b[i] = 2*i;
    printf("Enter integer search key: ");
    scanf("%d", &key);
    result = rec_bin_find(key, 0, LEN-1);
    // result = bin_find(key, 0, LEN-1);

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

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

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

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

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)

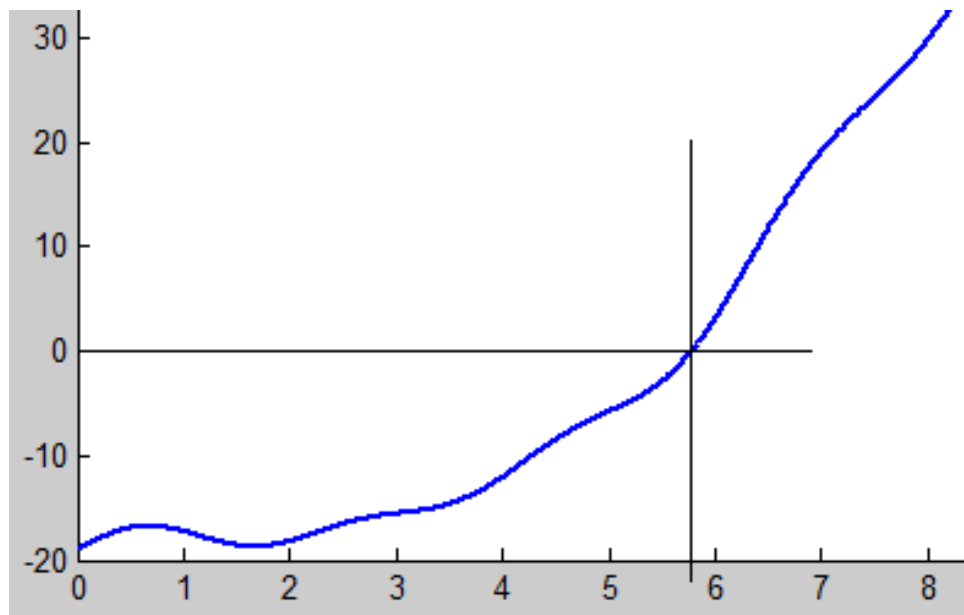
# 折半查找的应用示例

## 【例】单调函数的方程求根

$$f(x) = 2\sin(x) + \sin(2x) + \sin(3x) + (x-1)^2 - 20 = 0,$$

对方程求根（解析解很难求得）。

已知  $f(4) < 0$ ,  $f(8) > 0$ ,  $f(x)$  在  $x \in [4, 8]$  之间是单调递增的。



## 6.3 字符串和字符数组

- 字符串：由零个或多个字符组成且通过双引号括起来的有限序列，例如，"Hello, world", "A", "123456"等
- ◆ **字符串结束标志符：**'\0'，表示空字符 (null character)，是由编译器自动添加到字符串结尾处

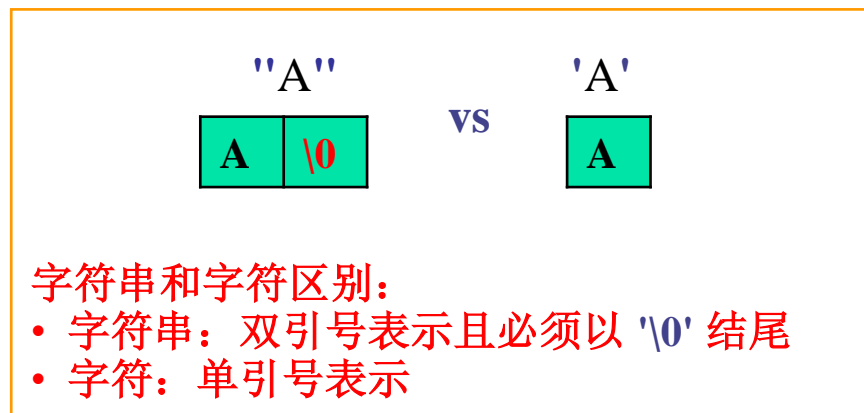
H	e	l	l	o	,	w	o	r	l	d	\0
---	---	---	---	---	---	---	---	---	---	---	----

- 字符串定义

- ◆ **方法1：字符数组，** `char s1[64];`

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

- ◆ **方法2：字符指针，**如 `char *s2;` (后面介绍)



说明：C语言没有专用的字符串数据类型，都是通过字符数组或字符指针实现

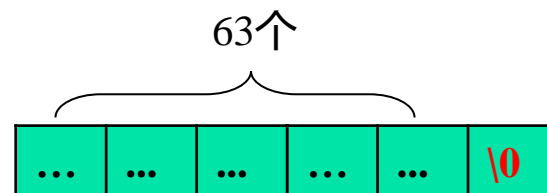
# 字符串和字符数组

## • 定义和初始化字符数组

(1) 显式定义字符串长度, `char s1[64];`

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

数组长度由编译器根据字符串中实际字符个数确定, 共11个字符, 结尾隐藏'\0', 实际长度12, 等价于`char a[] = { 'H', 'e', 'l', 'l', 'o', ',', 'w', 'o', 'r', 'l', 'd', '\0' };`



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

### 字符数组的长度问题:

- 显示定义中, 字符串最多只能存储“长度-1”个实际字符
- 隐式定义中, 字符串长度= “” 中的字符个数+1
- 用字符数组定义字符串时, 需提供足够空间存储字符串中的实际字符和空字符
- 已知字符串, 建议用隐式方法定义
- 数组一旦定义, 空间已确定, 只能修改存储内容, 不能修改存储大小 (回想一下, 为什么强烈不建议用变量定义数组大小)
- 长度错误为逻辑错误, 编译器不报错

### 字符数组的赋值问题:

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

```
char s1[3] = "first"; //逻辑错误, 编译器不报错
printf("%s", s1); //结果未知
s1[] = "second"; //错误
char s2[10];
s2[10] = "first"; //错误
```



# 字符数组定义与初始化

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

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

**【例a6-5-1】**  
sizeof(G#) is ?    sizeof(L#) is ?    G#[i] is ?    L#[i] is ?  
(# 表示左边代码中的数字1~6)

**计算字符串长度：**

- strlen(G)：返回字符串实际长度，不计算'\0'；
- sizeof(G)：返回字符数组所占字节数，计算'\0'，等价于sizeof(G)/sizeof(char)
- 请从实现原理上理解strlen()和sizeof()

# 字符数组访问

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

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

字符串定义的三大条件：

- 数组
- char类型
- '\0' 结尾

判断字符串结束的通常用法

三者缺一不可！

# 字符串和字符数组的关系

- 字符串与字符数组很相似但又相异。两者关系很密切，很多应用可以互相替换，但两者又有区别（串是“常量”，数组是“变量”），注意不能互换与混淆。
- 字符数组可用于定义字符串，但未必所有字符数组都是字符串，只有以 '\0' 结尾的字符数组才“被认为”是字符串，否则只能称之为字符序列。
- 数组可以是int, float, char等很多类型，字符串可以看成是一个char类型的数组。

## 字符数组：

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

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

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

# 字符串和字符数组的关系【示例】

```
#include <stdio.h>
#include <string.h>
int main()
{
    char a[] = "hi";
    char b[] = {'h', 'i'};
    char *aPtr = a;

    printf("%6s: %x\n", "a", a);
    printf("%6s: %x\n", "b", b);
    printf("%6s: %x\n", "aPtr", aPtr);
    printf("%6s: %x\n\n", "&aPtr", &aPtr);

    printf("%6s: %x\n", "abc", "abc");
    printf("%6s: %x\n", "abcde", "abcde");

    return 0;
}
```

- "hi", "abc" 等等，才是字符串。
- a, b是字符数组（a可以按字符串来处理，b不行），通常简称a是字符串，但并不表示a就是字符串。
- aPtr就是一个指针变量，当指向a后，也可以按字符串来处理。

输出

```
a: 61fefc
b: 61fefb
aPtr: 61fefc
&aPtr: 61fef4

abc: 408046
abcde: 40804a
```

# 字符数组实例

## 【例6-17】一行字符串倒置

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

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

    puts(a);
    str_rev(a);
    puts(a);

    return 0;
}
```

gets: 读入一行字符串 (回车结束, 支持空格读入)

scanf: 空格不能读入

**如果写成** scanf("%s", a); **结果怎么样?**

**puts vs printf?**

```
void str_rev(char s[])
{
    int hi=0, low=0;
    char temp;

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

## 6.4 常用的标准字符串函数

#include <string.h>

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

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

# 字符串输入输出函数： gets, puts

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

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

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

输入结束标志为空格、制表符、回车等（不能读入空格）。

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

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

# 字符串输入输出函数： gets, puts

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

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

本代码片段输出什么?  
哪些地方有错?



# 字符串输入输出函数： gets, puts

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

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

本代码片段输出什么?  
哪些地方有错?

输出

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

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

# 字符串输入输出函数：scanf, printf

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

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

输入相同

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

输出不同

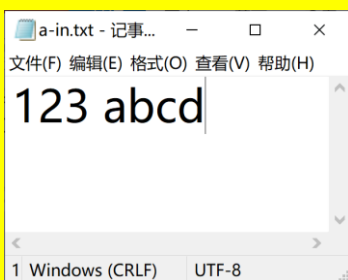
这两个1.23是  
否一样？

这里的1.23是  
字符串

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

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

输入:



```
char a[5], b[5];  
puts(gets(a));  
  
scanf("%s", b);  
printf("%s", b);
```

输出: ?

123 abcd

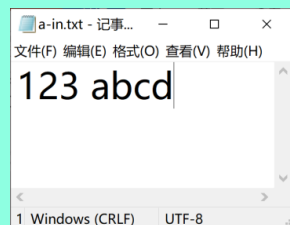
123  
abcd

针对第一组输入“123 abcd”:  
“123 abcd”是一行, 读入a, a越界访问, 此处运气好, 可行, 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。

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

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

输入:



a-in.txt - 记事...  
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)  
123 abcd  
1 Windows (CRLF) UTF-8



```
char a[5], b[5];  
scanf("%s", b);  
printf("%s", b);
```



输出: ?

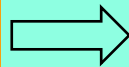
123 abcd



a-in.txt - 记事...  
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)  
123  
abcd  
1 Windows (CRLF) UTF-8



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

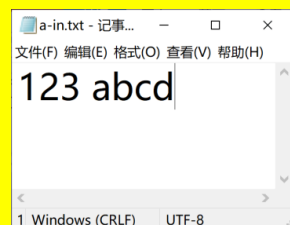


123

“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。

输入:



a-in.txt - 记事...  
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)  
123 abcd  
1 Windows (CRLF) UTF-8



```
char a[5], b[5];  
puts(gets(a));
```



输出:

123 abcd



a-in.txt - 记事...  
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)  
123  
abcd  
1 Windows (CRLF) UTF-8



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



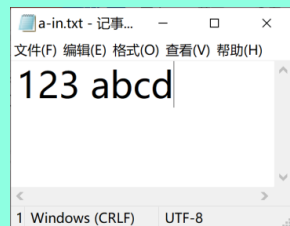
123  
abcd

“123 abcd”是一行，读入a，a越界访问，此处运气好，可行，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。

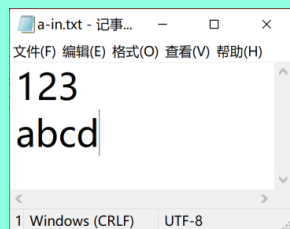
两行分别读入a和b，并进行相应输出。输出a时自动添加\n。

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

输入:



a-in.txt - 记事...  
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)  
123 abcd  
1 Windows (CRLF) UTF-8



a-in.txt - 记事...  
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)  
123  
abcd  
1 Windows (CRLF) UTF-8

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

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

`puts(gets(a));`

输出: ?

123 abcd

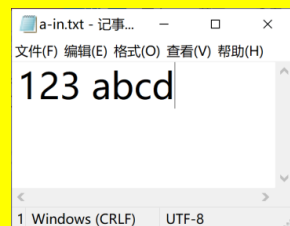
123

“123”和“abcd”分别读给b和a  
并进行相应输出，a后加 \n。

输出显示效果一样，但意义  
完全不一样!

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

输入:



a-in.txt - 记事...  
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)  
123 abcd  
1 Windows (CRLF) UTF-8



a-in.txt - 记事...  
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)  
123  
abcd  
1 Windows (CRLF) UTF-8

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

`puts(gets(a));`

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

输出:

123 abcd

123  
abcd

“123 abcd”是一行，读入a，a越界访问，  
此处运气好，可行，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。

## \*字符串输入函数：fgets

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

- 行输入函数 `char * fgets(char s[ ]);`

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

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

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

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

# \*输入函数gets的 数组越界非法写入 攻击示例

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

&key = 61fefafa, &str = 61fef0

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

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

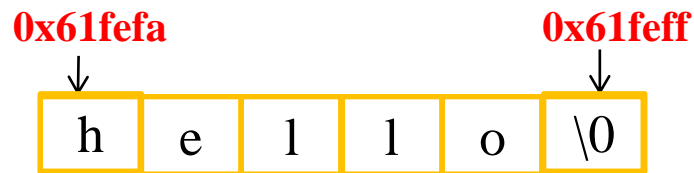
Password(hint: 123456):123456  
Correct password!

第一次尝试错误

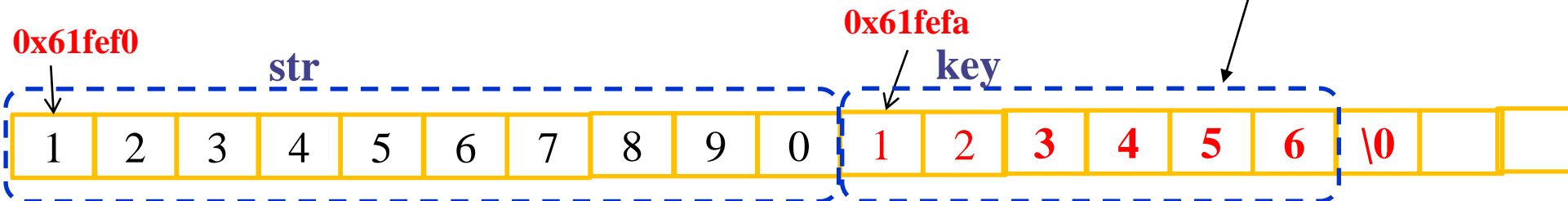
第二次修改了key

第三次成功破解

正确密码key:



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。



# \*字符串输出函数：fputs

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

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

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

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

 把 s 指向的字符串写入 fp 指向的文件

- 返回值

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



# \*字符串输入输出函数实例【例a6-5-2】

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

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

VS

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

左边输出的最后多一个空行（puts会在输出后加一个换行符'\n'）。

VS

上下两个程序的输出一样。

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

# \*字符串输出（构造）函数：sprintf

【例6-19】由参数确定输出的小数位数 从标准输入读入浮点数 $x$  ( $-10 < x < 10$ )和整数 $m$  ( $0 < m < 13$ ), 在标准输出上输出 $\sin(x)$ 的值, 保留到小数点后 $m$ 位数字。

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

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

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

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

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

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

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, 应掌握。sprintf 与 printf 类似。**

# \*字符串输入（构造）函数：sscanf

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

【例6-20】分析日期和时间 计算机显示的时间通常有特殊的格式，比如计算机给出的格式

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，并在屏幕上按行单独显示出来。如，红框的数据应显示为右边格式：

2018

Apr

4

17

28

28

+0800

```
int day, year, h, m, s;  
char mon[4], zone[6];  
char buf[] = "17/Apr/2018:10:28:28 +0800";  
sscanf(buf, "%d/%3c/%d:%d:%d:%d %s", &day, mon, &year, &h, &m, &s, zone);  
mon[3] = '\0'; // 什么作用?  
printf("%d\n%s\n%d\n%d\n%d\n%d\n%s", year, mon, day, h, m, s, zone);
```

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

## \*字符串复制函数：strcpy, strncpy

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

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

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

将字符串 `src` 中最多 `n` 个字符复制到字符数组 `dest` 中，返回 `dest` 的值。`n` 小于或等于 `src` 的长度时，只把 `src` 的前 `n` 个元素复制到 `dest`，这时可能需要手动在 `dest` 末尾添加 `'\0'`（即执行 `dest[n] = '\0';`）。如果 `dest` 中原来已经包括元素个数大于 `n` 的字符串，则不需要在 `dest` 中添加 `'\0'`。如果 `dest` 中不包括 `null` 终止符，可能造成严重的运行时错误。

## \*字符串复制函数: strcpy, strncpy

```
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); // does not copy null character  
z[11] = '\0';  
printf("Dest_2: %s\n", z);  
  
strncpy(z, "abcdefg hijklmn", 6);  
printf("Dest_3: %s\n", z);
```

输出:

Source: 1234 abcd ABC123

Dest\_1: 1234 abcd ABC123

Dest\_2: 1234 abcd A

Dest\_3: abcdefbcd A

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

```
strncpy(z, "abc", 6);  
printf("Dest_4: %s\n", z); // 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 弄 掉a ?a 捞

# \*strncpy的几个实例

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


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

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

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

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

## 连续运行三次的结果



```
命令提示符
C:\alac\example>a6-test

Z1_ini: x炭      棹a
Z2_ini:
Z1_cpy: 1234 abcd Ao鉞 @
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鉞 @
Z2_cpy: 1234 abcd A
Z3_cpy: 1234 abcd A

C:\alac\example>a6-test

Z1_ini: 躑?      棹a
Z2_ini:
Z1_cpy: 1234 abcd Ao鉞 @
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。

# \*字符串追加函数：strcat, strncat

(cat, concatenate, 连接)

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

将字符串 src 复制到字符串 dest 已有字符串后面（追加），src 的第一个字符重定义 dest 的\0终止符，返回 dest []。

- **char \*strncat(char dest[ ], const char src[ ], size\_t n);**

将字符串 src 中最多前n 个字符添加到字符串 dest 后面，src 的第一个字符重定义 dest 的\0终止符，返回 dest[]。

- 应用：今天的作业没有写完，明天接着写

# \*字符串追加函数: strcat, strncat

```
char x[] = "1234 abcd ABC123";  
char y[20] = "", z[20] = "";  
  
printf("Source: %s\n", x);  
printf("Dest_1: %s\n", strcat(y, x));  
  
strncat(z, x, 11);  
printf("Dest_2: %s\n", z);  
  
strncat(z, "abc", 6);  
printf("Dest_3: %s\n", z);  
  
strncat(z, "123456789abcdef0", 12);  
z[19] = '\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

- 把x的前11个字符追加到z, 无需手动在后面添加'\0', 因为strncat会自动添加。
- z已有14个元素, 追加12个元素后, 已超z的容量。z数组越界! 可能会导致越界部分内容覆盖掉别的有用数据! 这么用法是错误的!!!



# \*字符串比较函数： strcmp, strncmp

按姓名拼音排序

Song Xiao

Song You

Zhou Jielun

- **int strcmp(char s1[ ], char s2[ ]);**

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

- **int strncmp(char s1[ ], char s2[ ], size\_t n);**

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

## \*字符串比较函数: strcmp, strncmp

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

输出

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

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

# 字符串检查计算函数：strlen

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

返回字符串s的字符个数，长度中不包括终止符 '\0' 。

如下几个函数需要对指针有较深入的理解，以后再介绍。

- `char * strchr(char s[ ], int c);`
- `char * strrchr(char s[ ], int c);`  
返回字符 c 在字符串 s 中第一次和最后一次出现的**位置的指针**。如果 s 中没有 c，两个函数都返回NULL。
- `char * strstr(char *s, char *sub_str);`  
返回子字符串 sub\_str 在字符串 s 中第一次出现的**位置的指针**。如果 s 中没有 sub\_str，返回NULL。

**思考题：**如果想要返回子字符串 sub\_str 在字符串 s 中最后一次出现的位置的指针，如何实现函数 `char * strrstr(char *s, char *sub_str);` ?

# 字符串检查计算函数：strlen

**int strlen(char s[ ]);** // 返回s的字符个数（不包括终止符'\0'）

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

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

输出

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

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

## 6.5 使用数组的常用数据结构\*

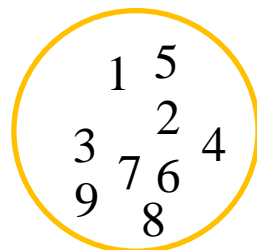
- 数据结构

- ◆ **定义**：计算机存储、组织数据的方式
- ◆ **作用**：针对特定问题选择合适的数据结构，可提高运行或存储效率
- ◆ **特点**：数据结构只明确**数据组织方式**（静态）  
但没有提供**数据操作方法**（动态）

计算机没实现，  
就要人来实现

- 使用数组的常用数据结构

- ◆ **队**：先进先出
- ◆ **栈**：先进后出
- ◆ **散列表**：按“键值”索骥



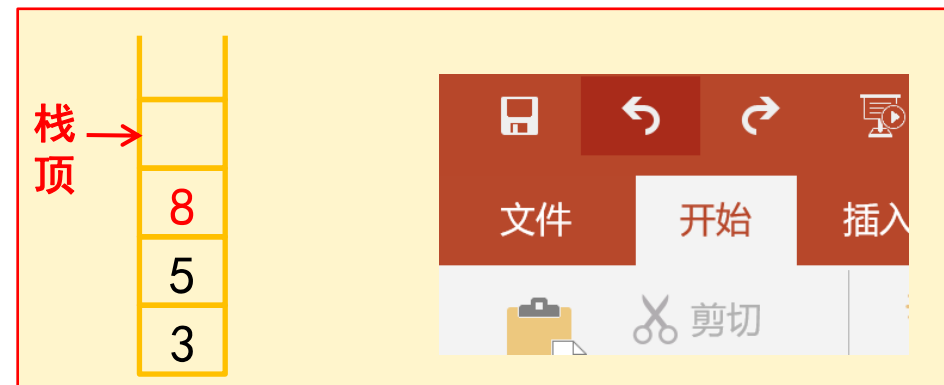
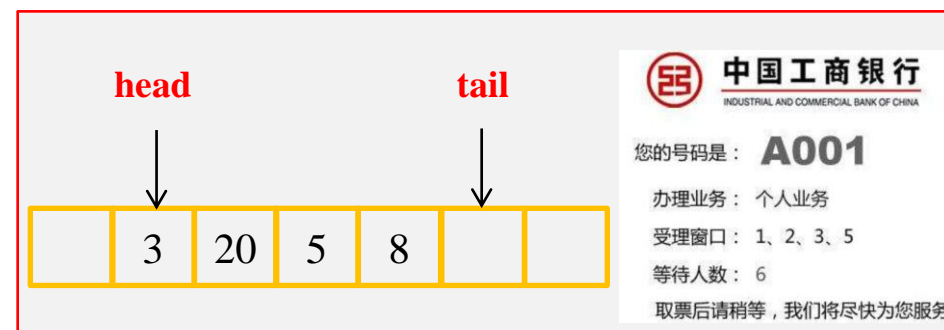
哪个查找目标  
值更快速？



静态为根基，动态为灵魂

静若处子，动若脱兔

——《孙子·九地》



# 队 (Queue)

- 队，也称“队列”，是一种先进先出的数据结构
  - ◆ 特点：先进先出 (First In First Out, FIFO)
  - ◆ 对象：顺序存储的数据集
  - ◆ 存取：数据从一端写入（队尾入队），从另一端读出（队头出队）
  - ◆ 应用：打印机的任务队列；输入/输出数据的缓存存储
- 队的操作
  - ◆ 队的初始化
    - 为队分配存储空间
    - 说明队头 (head) 和队尾 (tail) 的位置
  - ◆ 数据的入队（从队尾写入数据）
  - ◆ 数据的出队（从队头取出数据）
  - ◆ 检查队是否为空
  - ◆ 检查队存储空间是否已满



井然有序



混乱无序

# 队的存储结构

- 队的存储空间

- ◆ 定义数组，指明队中各元素的数据类型

- 队头和队尾的位置标记

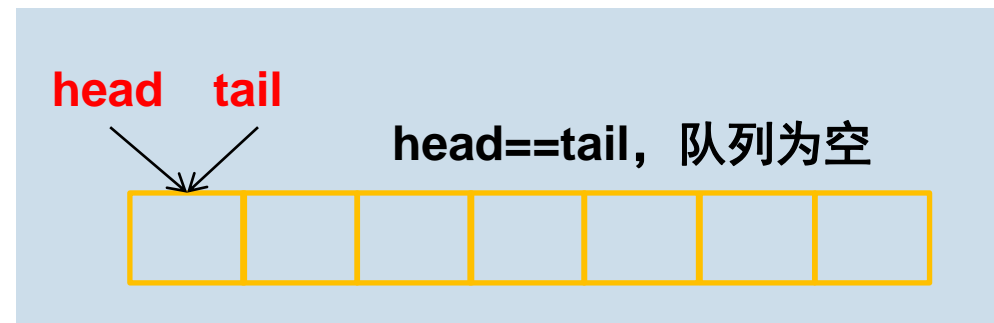
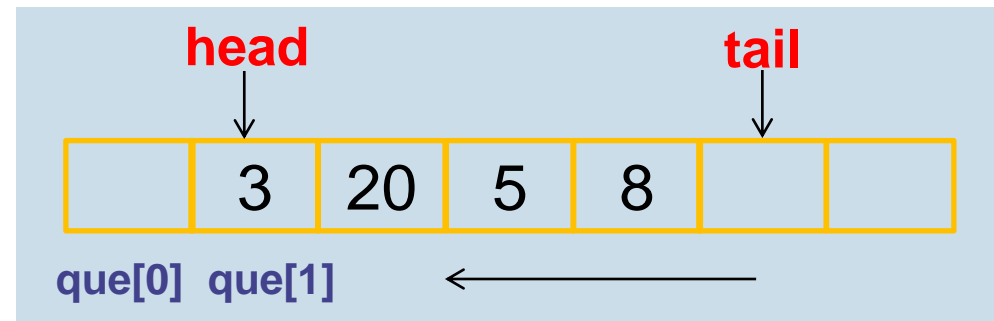
- ◆ 定义位置标记变量（head, tail），保存队头和队尾在数组中的下标
- ◆ 用于指示队头和队尾的位置

- 队中元素个数

- ◆ 定义个数变量（count），保存当前队列中元素实际个数
- ◆ 定义长度变量（MAXSIZE），表示队列存储的最大元素个数

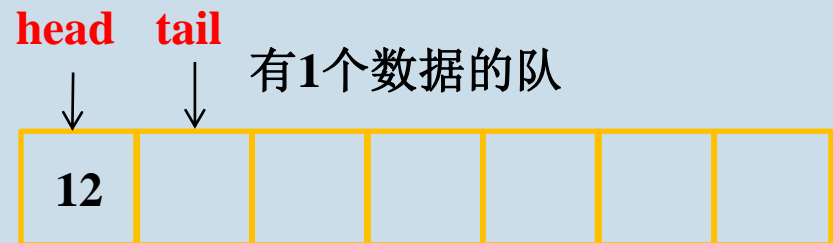
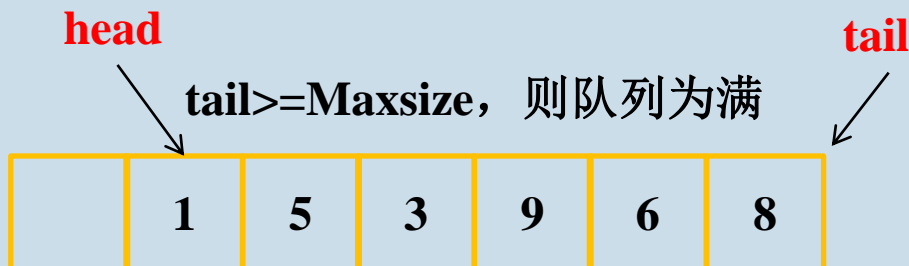
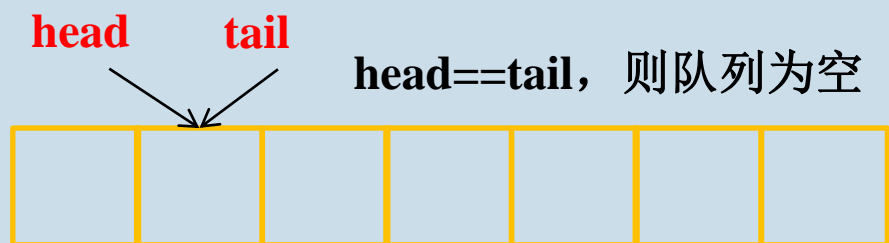
- 队的初始化

- ◆ 初始状态队中没有数据，即空队
- ◆ 将队头和队尾的位置标记变量设置0



# 队的操作

- ◆ 判断队是否为**空**：head和tail是否相等
- ◆ 判断队是否已**满**：tail是否等于  $n-1$ （ $n$ 表示该数组长度）
- ◆ **入队**操作：1) 先将数据插入队尾；2) 再将tail加1
- ◆ **出队**操作：1) 先从队头读取并删除数据；2) 再将head加1





# 队的操作-出队

出队操作:

- 1) 先从队头读取并删除数据;
- 2) 再将head加1

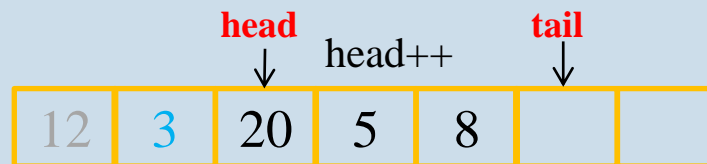
当前队



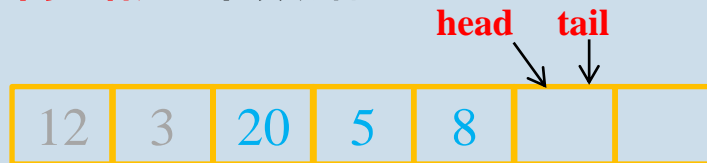
出队1个数据 head++



再出队1个数据



再出队3个数据 head==tail, 队列为空



# 一个int型队的实现

```
int queue[MAXSIZE], head = 0, tail = 0;
```

```
void queuePush(int v)
```

```
{
```

```
    queue[tail++] = v;
```

```
}
```

queue[tail]=v;  
tail++;

```
int queuePop()
```

```
{
```

```
    return queue[head++];
```

```
}
```

temp=queue[head];  
head++;  
return temp;

```
int queueEmpty()
```

```
{
```

```
    return head >= tail;
```

```
}
```

```
int queueFull()
```

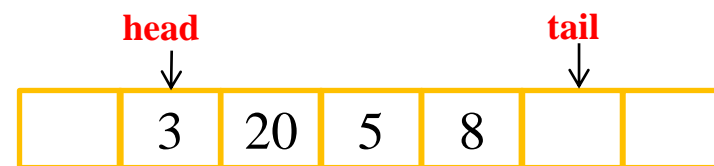
```
{
```

```
    return tail >= MAXSIZE;
```

```
}
```

注意:

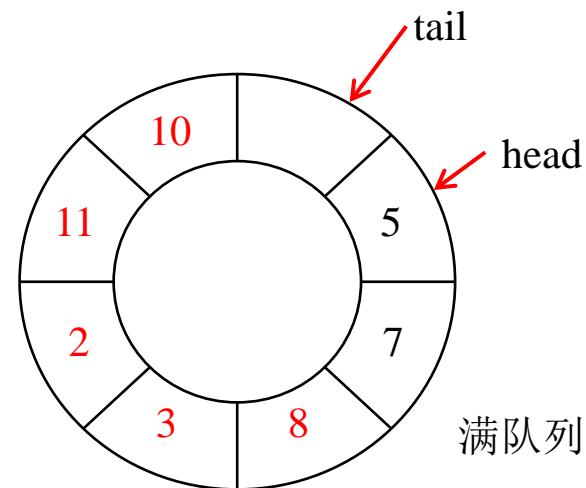
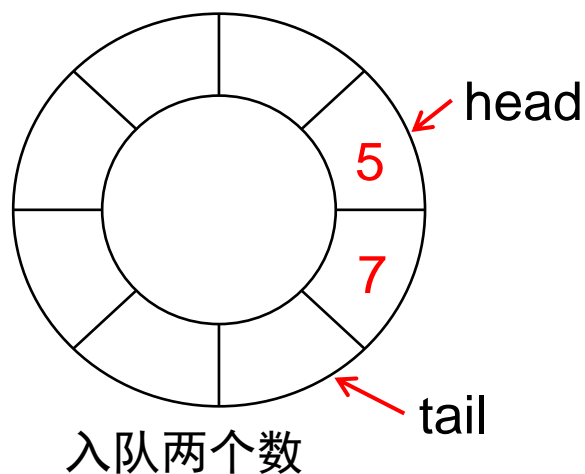
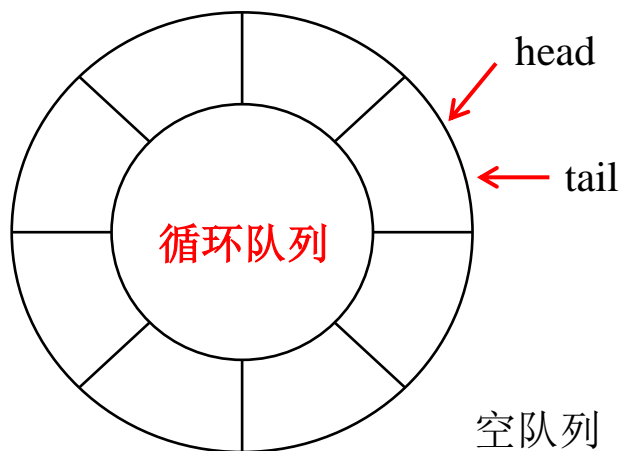
- 入队前，要判断队列是否已满
- 出队前，要判断队列是否已空



queue[0] queue[1]

# 循环队\*

- 特点：队头和队尾相连，循环利用执行完出队操作后空出来的存储空间
- 操作：
  - 逻辑上，插入、删除、判空和顺序队列相同
  - 队头和队尾下标变量进行模MAX\_N操作
  - 判满和顺序队列不同**



```
int queueFull(){  
    return head == (tail+1)%MAX_N;  
}
```



循环队列判满

- 当循环队列同时存入N个数据时， $head == tail$ ，但 $head == tail$ 表示空队
- 牺牲一个空间，用来判断队满
- 长度为N的存储空间，最多只能保存N-1个有效数据

# 队的应用实例

**【例6.2-4】N位超级质数** 从标准输入读入一个整数N ( $N < 9$ )，生成所有满足下列条件的N位超级质数：左侧前任意连续位均是质数。例如，23、233、2339、23399就分别是这样的2位、3位、4位、5位超级质数

## 方法1：直观方法

- 遍历所有N位正整数 
- 并逐一检查每一个数的前一位、两位、……、直至N位是否构成质数 
- 效率很低，当N较大时计算速度慢

```
int i, j, n, i_test, flag, num=0;
scanf("%d", &n);
//10^(N-1) <= i <= 10^N-1
for(i=pow(10,n-1); i < pow(10,n); i++) {
    flag = 1;
    i_test = i;
    for(j=1; j<=n; j++) {
        if( 1 == i_test || !isPrime(i_test) ) {
            flag = 0;
            break;
        }
        i_test /= 10; // 依次检查 i 的前n-1, n-2, ...位
    }
    if(1==flag) printf("%d: %d\n", ++num, i);
}
```

## 队的应用实例

**【例6.2】N位超级质数** 从标准输入读入一个整数N ( $N < 9$ )，生成所有满足下列条件的N位超级质数：左侧前任意连续位均是质数。例如，23、233、2339、23399就分别是这样的2位、3位、4位、5位超级质数

- **方法2：改进方法**
  - 以所有的一位质数作为种子集合 {2, 3, 5, 7}
  - 每个种子后增加一个数字，检查新生成的两位数是否质数，如果是，则将其放进种子集合中  
{2, 3, 5, 7, 23, 29, 31, ... }
  - 重复这一过程，生成出所有长度的 N 位质数
  - 若以进入集合的顺序检查种子，即可使用队作为种子集合的存储结构
  - 具体参考书上的程序

# 队的应用实例

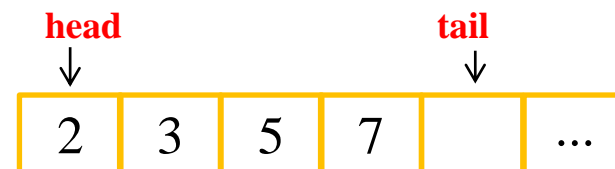
**【例6.2】N位超级质数** 从标准输入读入一个整数N ( $N < 9$ )，生成所有满足下列条件的N位超级质数：左侧前任意连续位均是质数。例如，23、233、2339、23399就分别是这样的2位、3位、4位、5位超级质数

- **方法3：更优雅的改进方法**
  - ◆ 以所有的一位质数作为种子集合 {2, 3, 5, 7}
  - ◆ 每个种子后增加一个数字（种子出队），检查新生成的两位数是否质数，如果是，则将其放进种子集合中（入队）  
{2, 3, 5, 7, 23, 29, 31, ... }
  - ◆ 重复这一过程，生成出所有长度的 N 位质数

# 队的应用实例—超级质数

## 算法伪代码

- Step1: 初始化: 将个位数的质数入队列queue
- Step2: 如果队列不为空
  - 出队, 将队首元素给变量a
  - 判断a的位数是否为n
    - ◆ 如果是, 输出a
    - ◆ 如果不是, 将 $10*a+1$ ,  $10*a+3$ ,  $10*a+7$ ,  $10*a+9$ 四个数中的质数依次放到队尾 (入队)
  - 返回Step2



## 队的应用实例—超级质数

## 队列可视化（n=2为例）

初始状态:

2	3	5	7													
---	---	---	---	--	--	--	--	--	--	--	--	--	--	--	--	--

Diagram illustrating the initial state of a queue implemented using an array. The array has 17 slots. The first four slots contain the values 2, 3, 5, and 7. The remaining 13 slots are empty. The 'head' pointer is positioned at the first slot (index 0), and the 'tail' pointer is positioned at the fifth slot (index 4).

2出队:

2	3	5	7														
---	---	---	---	--	--	--	--	--	--	--	--	--	--	--	--	--	--

Diagram illustrating a queue structure. The queue is represented by a horizontal array of 18 slots. The first four slots contain the numbers 2, 3, 5, and 7. The remaining 14 slots are empty. Above the second slot (containing 3) is the word "head" with a downward arrow. Above the fifth slot (empty) is the word "tail" with a downward arrow.

2开头的扩展质数入队:

The diagram shows a queue structure. The first six cells contain the numbers 2, 3, 5, 7, 23, and 29. The next ten cells are empty. A red 'head' label with a downward arrow points to the cell containing 3. A red 'tail' label with a downward arrow points to the first empty cell after 29.

3出队:

2	3	5	7	23	29												
---	---	---	---	----	----	--	--	--	--	--	--	--	--	--	--	--	--

Diagram illustrating the queue state after 3 is dequeued. The queue is represented as an array. The element 5 is now at the front of the queue. The 'head' pointer is at index 2 (value 5) and the 'tail' pointer is at index 6 (empty). The elements currently in the queue are 2, 3, 5, 7, 23, and 29.

3开头的扩展  
质数入队:

2	3	5	7	23	29	31	37									
---	---	---	---	----	----	----	----	--	--	--	--	--	--	--	--	--

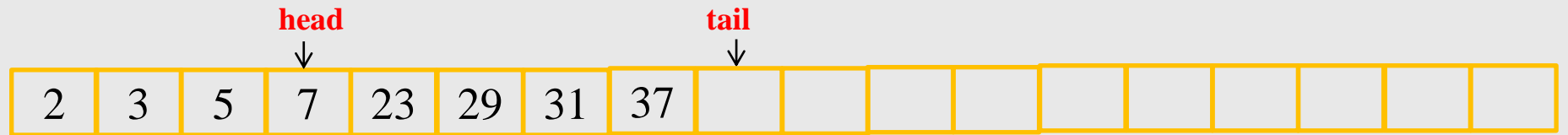
Diagram illustrating the queue state after the first step. The queue contains the numbers 2, 3, 5, 7, 23, 29, 31, and 37. The number 5 is marked as the **head** and the number 37 is marked as the **tail**. The queue is represented as a horizontal array of cells.



# 队的应用实例—超级质数

## 队列可视化（ $n=2$ 为例）

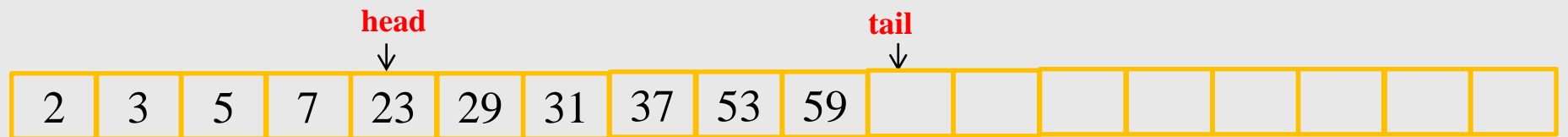
5出队:



5开头的扩展  
质数入队:



7出队:

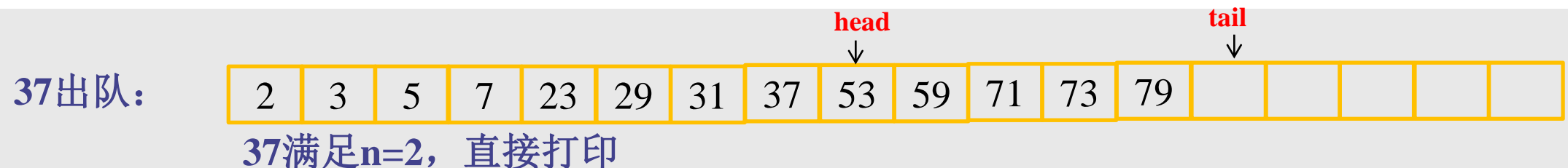
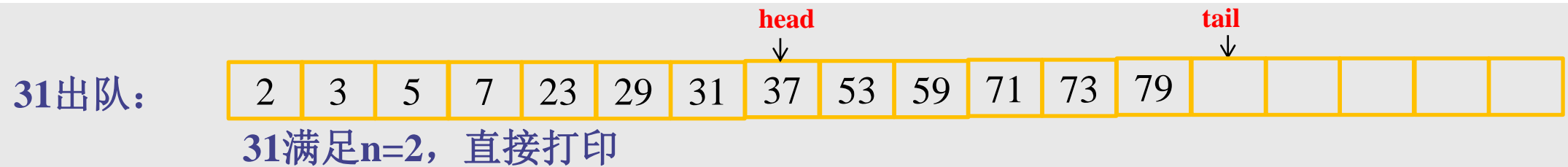
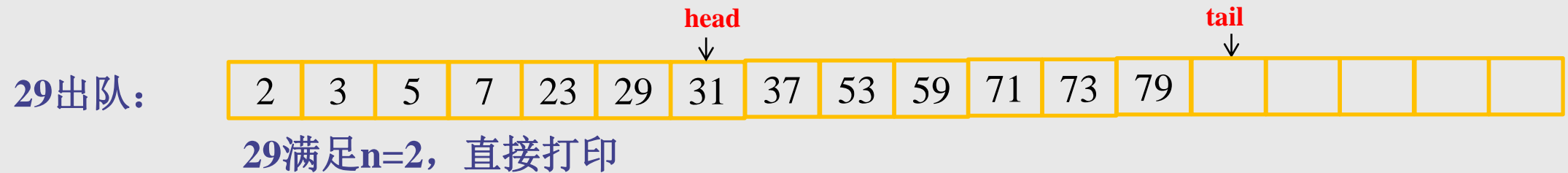
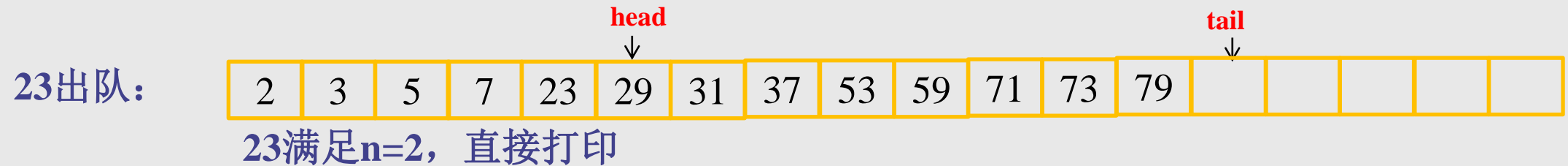


7开头的扩展  
质数入队:



# 队的应用实例—超级质数

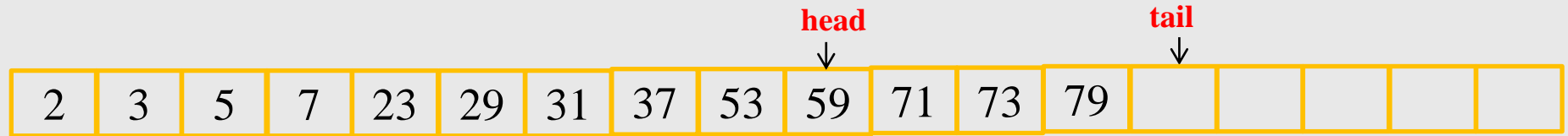
## 队列可视化（ $n=2$ 为例）



# 队的应用实例—超级质数

## 队列可视化（ $n=2$ 为例）

53出队:



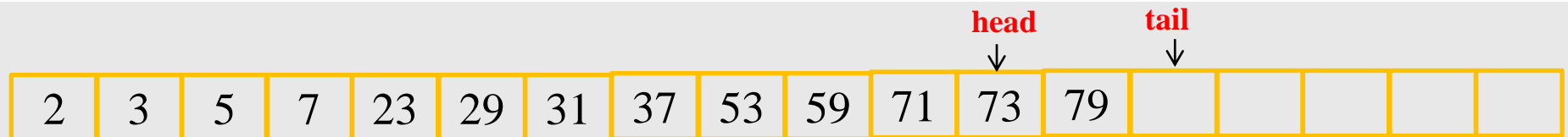
53满足 $n=2$ ，直接打印

59出队:



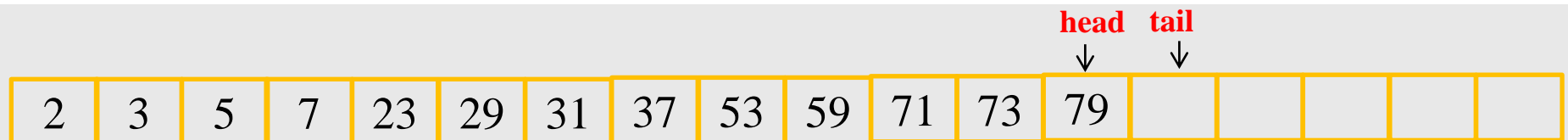
59满足 $n=2$ ，直接打印

71出队:



71满足 $n=2$ ，直接打印

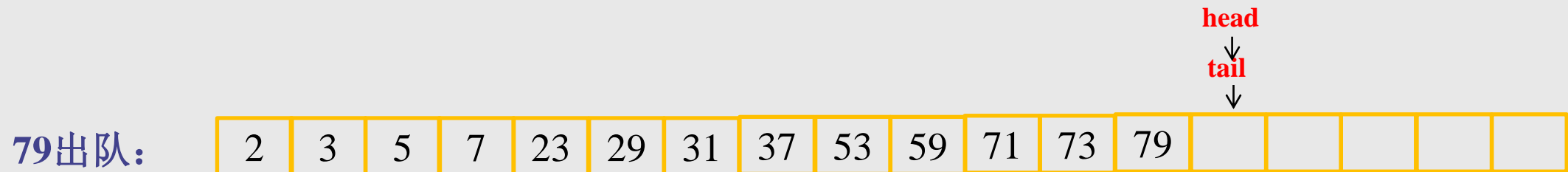
73出队:



73满足 $n=2$ ，直接打印

# 队的应用实例—超级质数

## 队列可视化（ $n=2$ 为例）



79满足 $n=2$ ，直接打印

队列为空（ $\text{head} == \text{tail}$ ），循环结束，程序结束！

其实：

- 还可以再快些，第一个 $n$ 位超级质数出队后，后面的所有元素（一直到队尾）都是超级质数，依次输出即可。
- 还可以再再快些，第一个 $n-1$ 位超级质数出队后，... ?

# 队的应用实例—超级质数

## //example6-16-better

```
#include <stdio.h>
#include <math.h>
#define MAX_Q 1020
#define NumOf(a) (sizeof(a)/sizeof(a[0]))
int hd = 0, tail=4, queue[MAX_Q] = {2, 3, 5, 7}; //queue初始只包含一位质数
int digits[]={1, 3, 7, 9}; //可用于后n-1位的数字表
int isPrime( int num );
int main(){
    int i, m, n, a, upper;
    scanf("%d", &n);
    upper = pow(10, n);
    while(hd != tail) { //队列不为空
        a = queue[hd++]; //出队
        m = 10 * a;
        if (m < upper) {
            for(i=0; i<NumOf(digits); i++)
                if(isPrime(m+digits[i]))
                    queue[tail++] = m+digits[i]; //入队
        }
        else
            printf("%d\n", a);
    }
}
```

- MAX\_Q的估计：设每个n位的种子生成两个n+1位的质数，则8位以下的全部超级质数的数量小于1020
- 质数扩展时，只使用了digits[]中的4个数字
- 质数判断函数的声明（在某个地方一定要有它的定义）
- 判断是否为N位
- **可能的改进：使用循环队列**

```
int isPrime( int num ){
    int tmp = sqrt( num);
    for(int i= 2; i <=tmp; i++)
        if(num %i == 0)
            return 0 ;
    return 1 ;
}
```

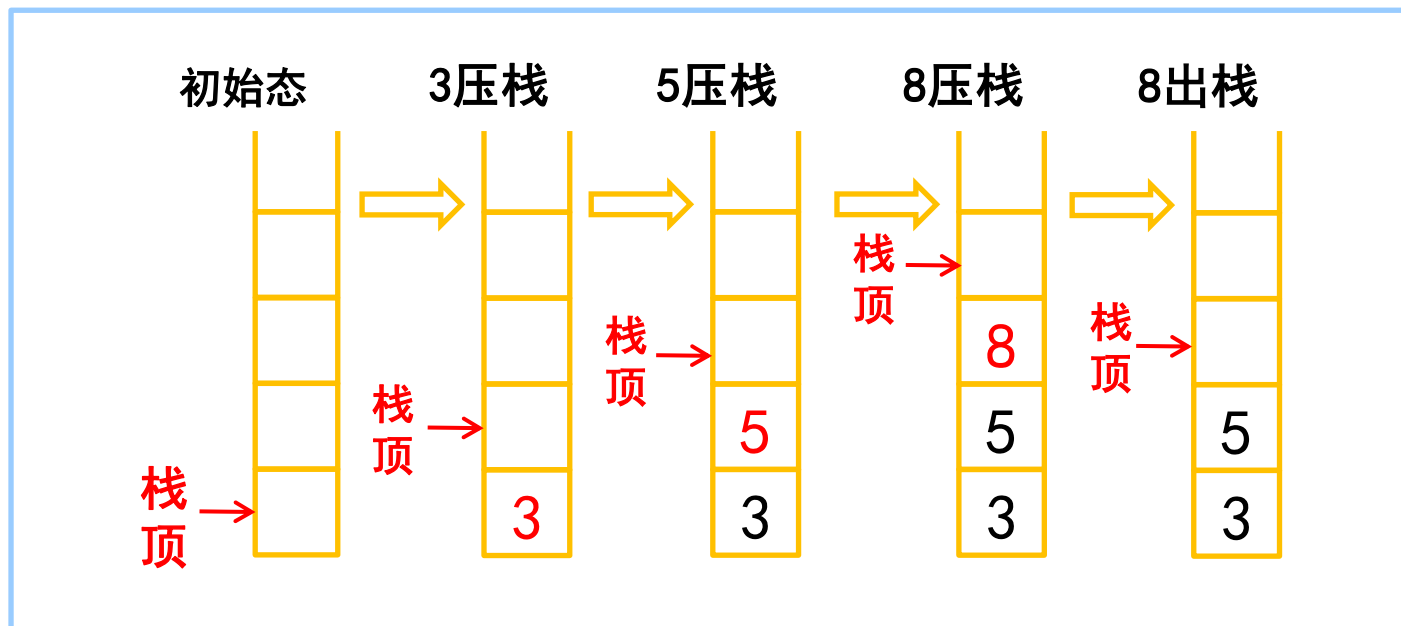
# 栈 (Stack)

- 栈：一种后进先出（Last In First Out, LIFO）的数据结构
  - ◆ 对象：顺序存储的数据集
  - ◆ 存取：数据从一端写入（栈顶入栈），从**同一端**读出（栈顶出栈）
  - ◆ 应用：网页中的回退、函数的调用和返回序列、后缀表达式的计算



- 对栈的操作

- ◆ 栈的初始化
  - 为栈分配存储空间
  - 定义栈顶位置
- ◆ 数据的进栈（压栈）
- ◆ 数据的出栈（弹栈）
- ◆ 判断栈是否为空



# 栈的操作

- 入栈：1) 先将数据插入栈顶；2) 再将栈顶标记top加1
- 出栈：1) 先将top减1；2) 再从栈顶读出数据
- 判断栈是否为空：判断top是否等于0

## 一个int型栈的实现

```
int stack[MAXSIZE], top = 0; // 栈初始化
```

```
void stackPush(int v)
```

```
{  
    stack[top++] = v;  
}
```

```
int stackPop()
```

```
{  
    return stack[--top];  
}
```

top--;  
return stack[top];

```
int stackEmpty()
```

```
{  
    return top <= 0;  
}
```

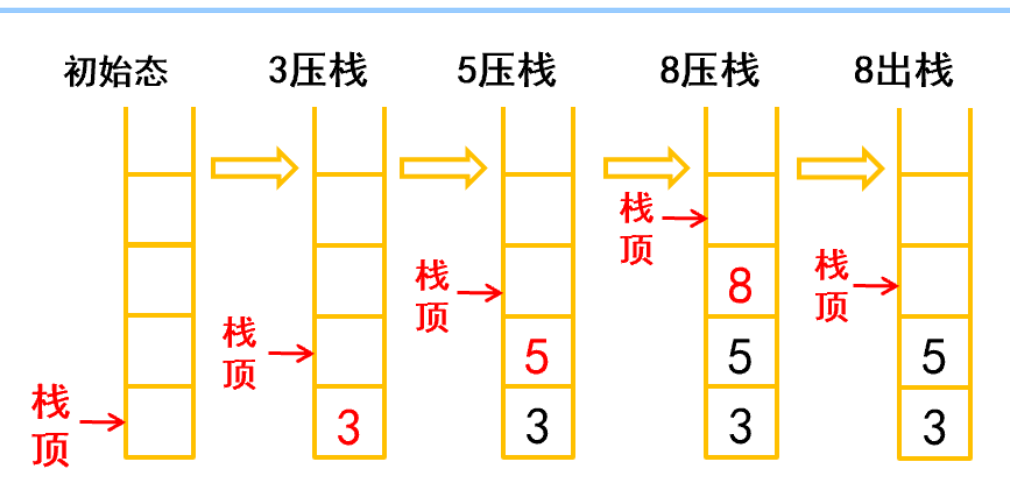
```
int stackFull()
```

```
{  
    return top >= MAXSIZE;  
}
```

stack[top]=v;  
top++;

注意：

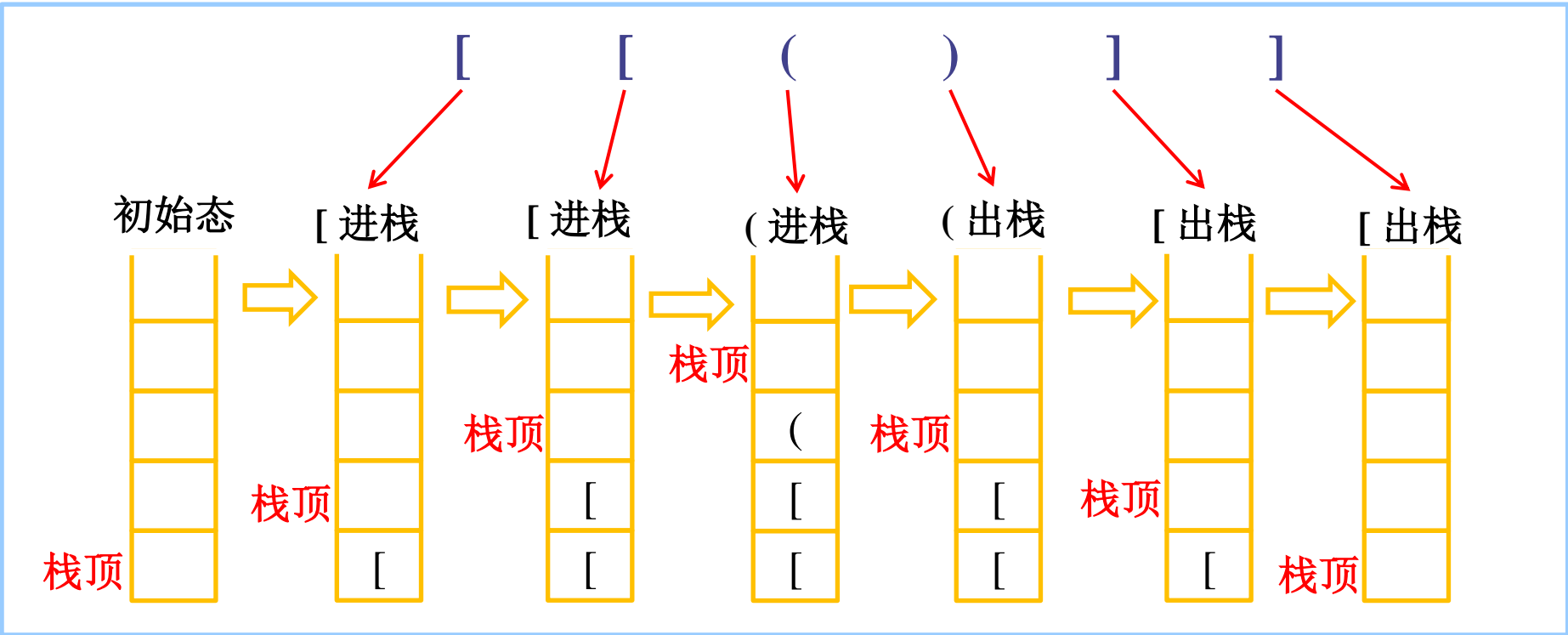
- 入栈前，要判断栈是否已满
- 出栈前，要判断栈是否已空



# 栈的应用实例—括号匹配

**【例6-14】 括号匹配** 写一个程序，从键盘读入一个有方括号和圆括号组成的字符串，检查该字符串中的括号是否匹配正确。当匹配正确是返回符号常量OK，否则返回ERR。例如，正确的匹配：[]()[]和[(]()()[]()；不正确的匹配：[()[]和[(]()()[]()]

**题解分析：**以输入[ [ ( ) ] ]为例进行分析

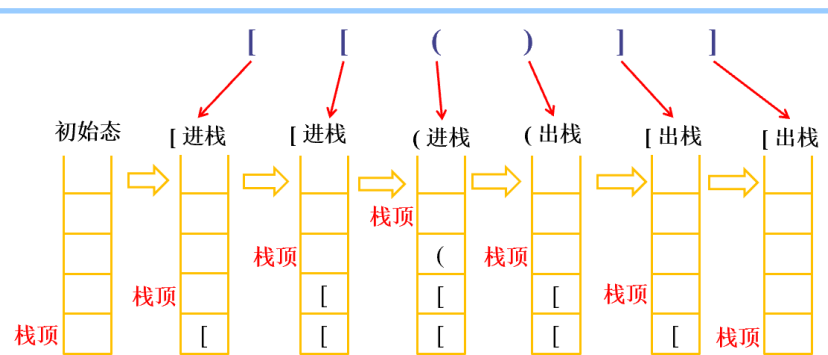




# 栈的应用实例—括号匹配

## 检验括号匹配的算法

1. 逐个读入输入的字符，直至输入结束
2. 当遇到左括号时将其入栈
3. 遇到右括号时，弹出栈顶元素并检查
  - (1) 若弹出的元素与其匹配，则转向 1
  - (2) 若弹出的元素与其不匹配，或者栈为空，则说明输入序列错误
4. 当输入结束时，检查栈是否为空
  - (1) 若栈为空，则说明输入序列正确
  - (2) 若栈不为空，则说明输入序列错误



int isMatch() //检验括号匹配函数

```
{
    int c, d;
    while ((c = getchar()) != '\n')
    {
        if (c == '[' || c == '(') // '[' == c || '(' == c
            stackPush(c);
        else {
            if (stackEmpty())
                return 0; //左括号少于右括号
            d = stackPop();
            if ((']' == c && '(' == d) || (')' == c && '[' == d))
                continue;
            return 0; // 左右括号类型不同
        }
    }
    return stackEmpty(); // 栈为空时正确
}
```

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

```
#define MAXSIZE 100
```

```
int stack[MAXSIZE], top = 0;
```

```
void stackPush(int v); //进栈
```

```
int stackPop(); //出栈
```

```
int stackEmpty(); //栈空判断
```

```
int stackFull(); //栈满判断
```

```
int isMatch(); //括号匹配检查
```

```
int main()
```

```
{
```

```
    int match_Ok;
```

```
    match_Ok = isMatch();
```

```
    if (match_Ok == 0)
```

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

```
    else
```

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

```
    return 0;
```

```
}
```

# 散列表 (Hash table)

散列表，也叫杂凑表、**哈希表** (Hash table)

- 根据数据的某种属性直接确定数据在存储结构中的位置 (**直接访问**)
  - ◆ 数据的某种属性——键值 (key)，取决于具体问题和数据的特征
  - ◆ 映射函数——从键值到存储位置的映射
- **当以数组为存储结构时，使用映射函数将键值映射为数组下标，以便数据定位**

	Hash(key)	key	学号	姓名	性别	其他
0	←	170601	张三	男	.....	
1	←	170602	李四	男	.....	
2	←	170603	王五	男	.....	
3	←	170604	赵六	男	.....	
4	←	170605	汪武	男	.....	
5	←	.....	.....	.....	.....	
...		.....	.....	.....	.....	

以“学号”作为key，  
通过映射函数  
Hash(key)把每条记录映射到对于的存储位置0, 1, 2 ...上  
(也可以选其他列的数据作为key)

# 散列表 (Hash table)

- 哈希冲突

- 王五和汪武都是WangWu，映射到同一个位置，称为哈希冲突（本书不讨论消除冲突等较复杂问题，感兴趣的同学可阅读“数据结构”或“算法”相关书籍）
- 本课程主要介绍没有冲突的哈希应用

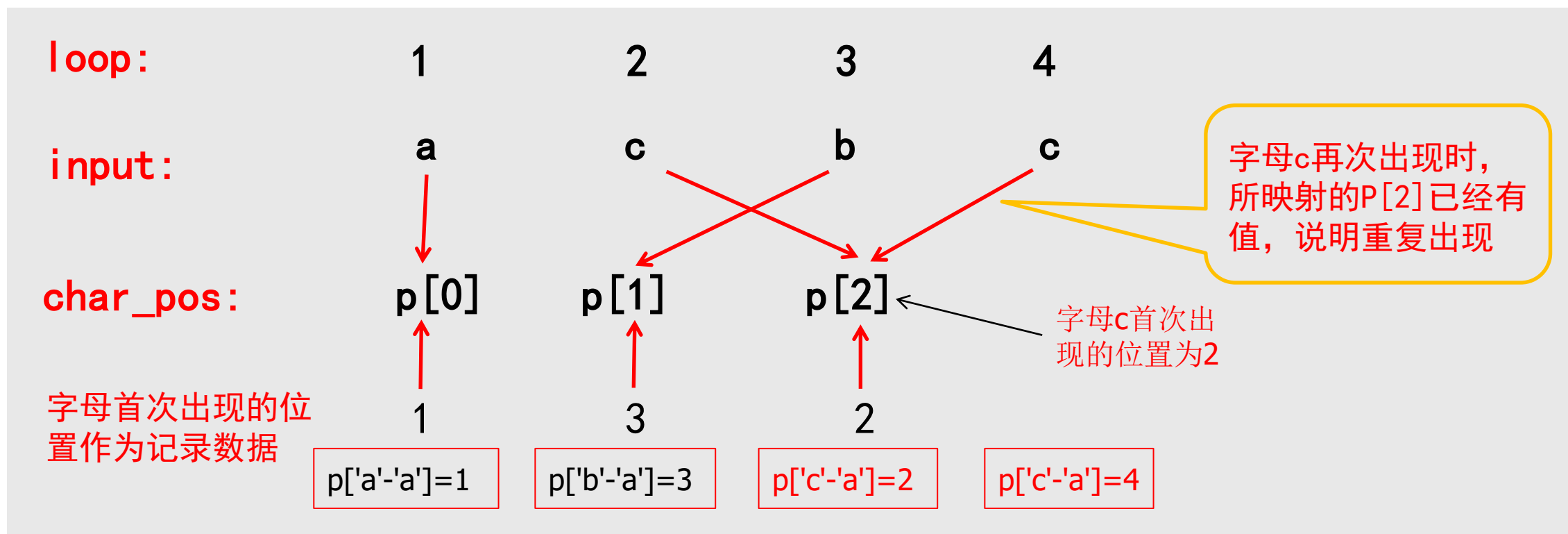
Hash(key)	学号	姓名	性别	其他
.	170601	张三	男	.....
.	170602	李四	男	.....
i	170603	王五	男	.....
.	170604	赵六	男	.....
j	170605	汪武	男	.....
.	.....	.....	.....	.....
...	.....	.....	.....	.....

以“姓名拼音”作为key，通过映射函数Hash(key)把每条记录映射到对于的存储位置上

# 散列表的应用实例—查找重复字符

**【例6-12】 首个重复字母的位置。**从标准输入上读入一个字符串，查找字符串中首个重复出现的小写字母，在标准输出上输出该字母及其在字符串中第一次和第二次出现的位置。字符的位置从1开始，两次出现的位置及其与字母间以冒号(:)分隔。如果字符串中没有重复的小写字母，输出0。

**题解分析：**以输入acbcd为例分析    **输出：** c:2:4



# 散列表的应用实例—查找重复字符

```
//example6-12.c
#include <stdio.h>
#include <ctype.h>
#define MAX_N 30
int char_pos[MAX_N];
int main(){
    int i, x;
    for(i = 1; (c = getchar()) != EOF; i++){
        if (islower(x))
            if (char_pos[x-'a'] > 0){
                printf("%c:%d:%d\n", x, char_pos[x-'a'], i);
                return 0;
            }
        else // 首次出现, 记录出现的位置
            char_pos[x-'a'] = i;
    }
    printf("0\n");
    return 0;
}
```

数组char\_pos[]是散列表的存储空间, 定义为全局变量, 默认初始化为0

键值: 字母在字母表中的顺序作为散列表的键值

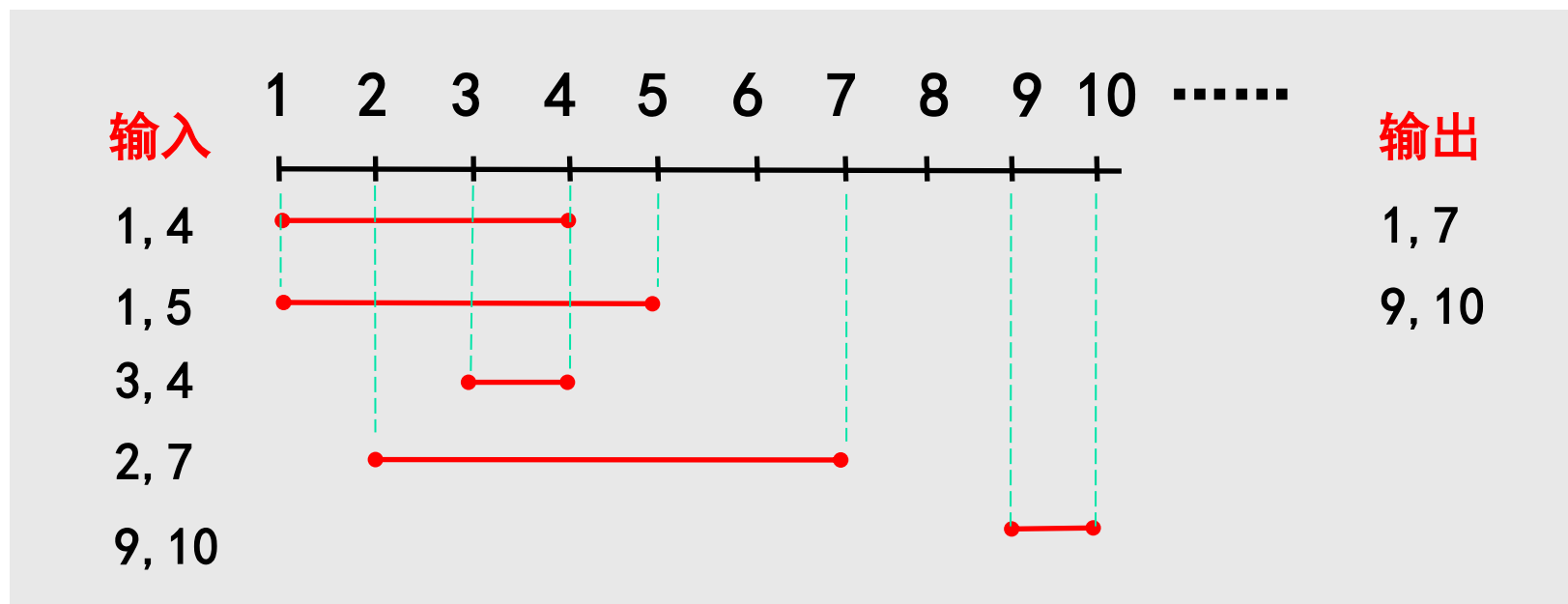
Hash 函数:  $H(x) = x - 'a'$

使用映射函数将键值映射为数组下标, 以便数据定位

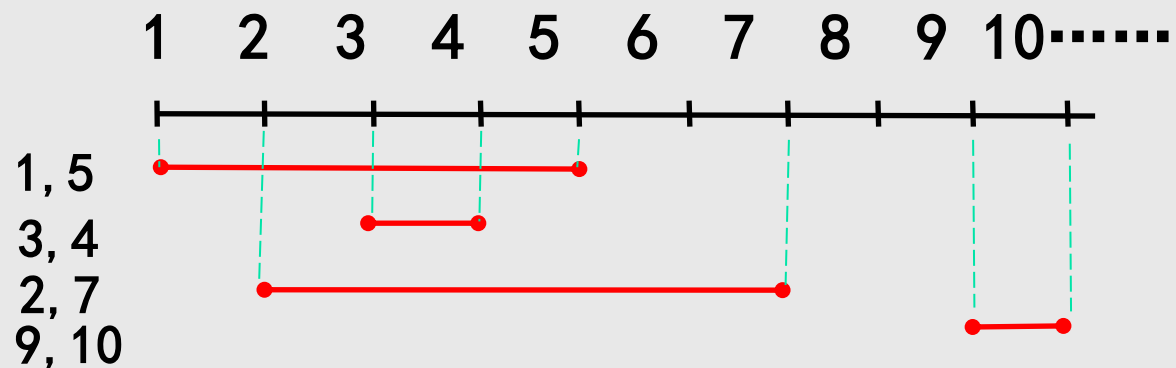
# 散列表的应用实例—区间合并

**【例6-13】**从标准输入上读入 $n$  ( $n \leq 50000$ ) 个闭区间  $[a_i, b_i]$  ( $0 \leq a_i < b_i \leq 10^6$  且均为整数)，将这些区间合并为不相交的闭区间。例如，区间  $[1,4]$ 、 $[1,5]$ 、 $[3,4]$ 、 $[2,7]$ 、 $[9,10]$  可以合并为区间  $[1,7]$  和  $[9,10]$ 。写一个程序，从标准输入中读入  $n$  行，每行包含两个有空格分隔的整数  $a_i$  和  $b_i$ ，表示区间  $[a_i, b_i]$ 。将计算结果写在标准输出上，每行包含两个用空格分开的整数  $x_i$  和  $y_i$  表示区间  $[x_i, y_i]$ 。各区间按升序排列输出。

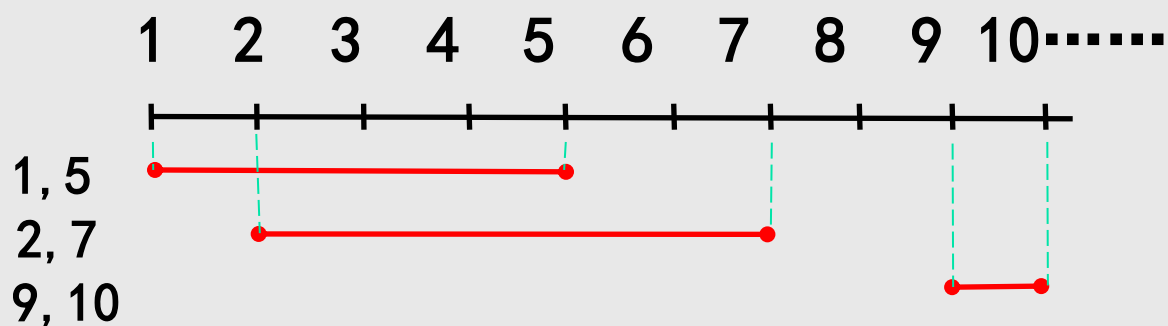
## 题目分析



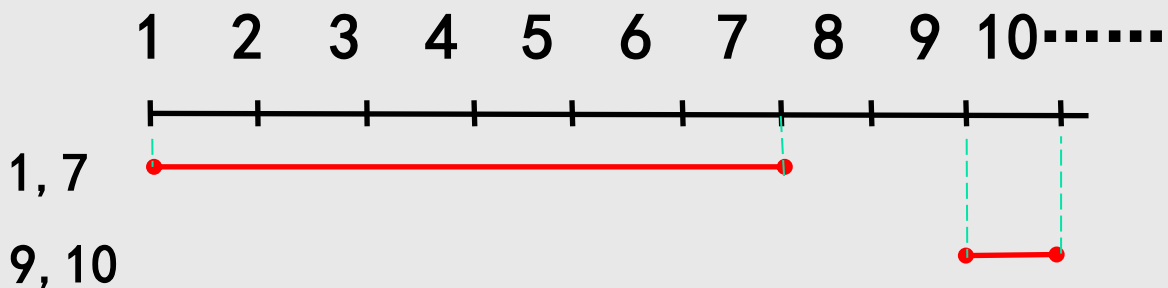
# 散列表的应用实例—区间合并



[1, 4]和[1, 5]有交叉  
扩展上界, 为[1, 5]



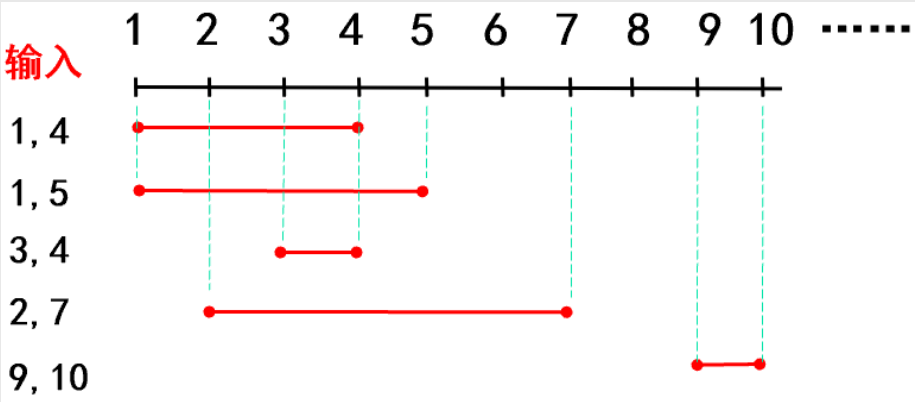
[3, 4]包含在[1, 5]中, 合并后仍  
为[1, 5]



[2, 7]与[1, 5]有交叉 (2在[1, 5]  
里), 扩展上界后为[1, 7]  
[9, 10]和[1, 7]没有交叉, 不合并

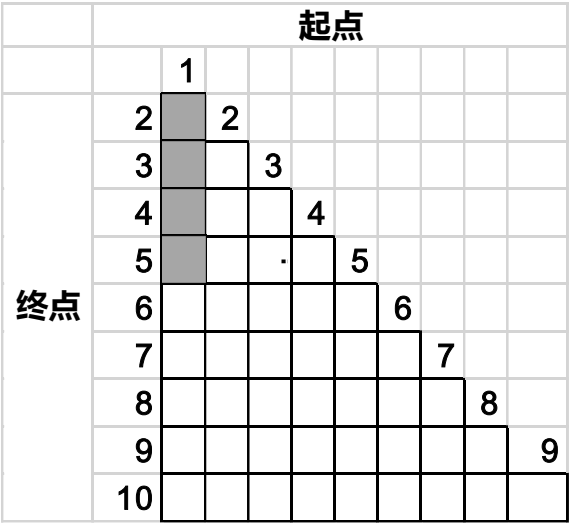
# 散列表的应用实例—区间合并

每根红线都是一名乘客的上下车记录，求车上连续有乘客的区间有哪些？

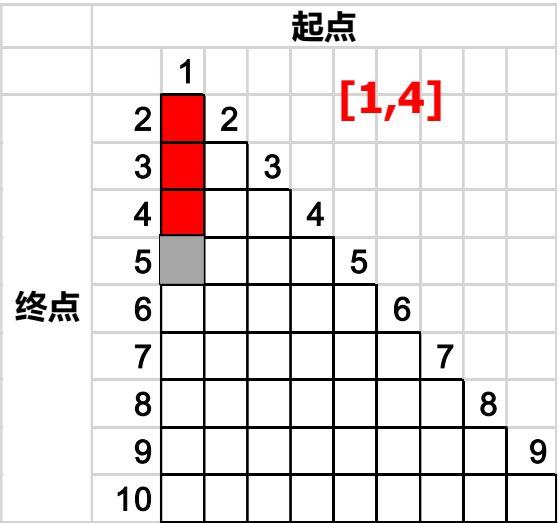


每个数字就是一站

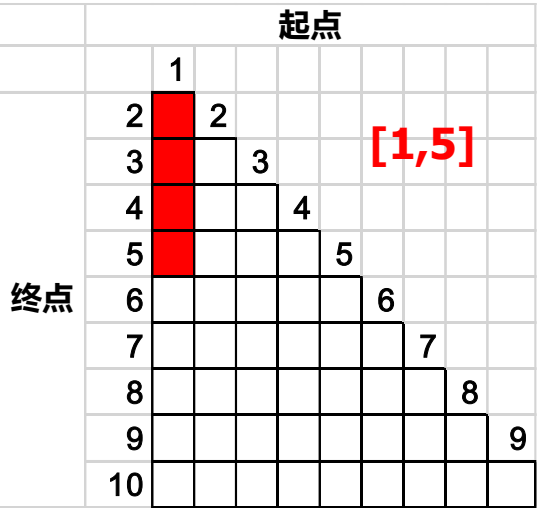
如果你是乘务员，你该怎么算？



1. 先准备一张这样的表



2. 逐条红线，从起点到终点涂格子



3. 如果有更长的就覆盖 (多个人同时上车，以远的为准)



# 散列表的应用实例—区间合并

每根红线都是一名乘客的上下车记录，求车上连续有乘客的区间有哪些？

[2,7]



44内

北官厅

----

北官厅

(BEIGUANTING)

(BEIGUANTING)

前门西(QIANMENXI)

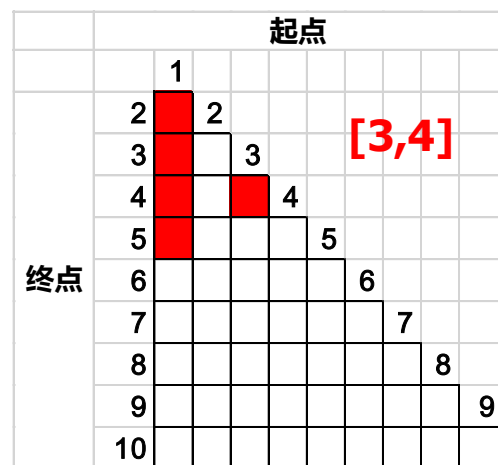
营业时间: 北官厅5:30—24:00

1	2	4	5	6	7	9	10	11	12	13	14	15	16	17	19	20	21	22	24	25
北	东	朝	雅	建	东	崇	台	前	前	和	宣	复	京	阜	西	玉	新	德	鼓	安
直	十	阳	国	宝	文	厂	基	平	武	椿	兴	成	直	桃	街	胜	楼	定	雍	北
官	条	门	宝	便	路	口	口	口	口	路	口	口	口	口	口	口	口	口	口	口
厅	北	南	南	路	南	西	西	东	东	西	南	院	北	南	口	西	西	西	东	厅

10公里以内(含)票价2元 每增加5公里以内(含)加价1元

每个数字就是一站

如果你是乘务员，你该怎么算？



继续涂格子



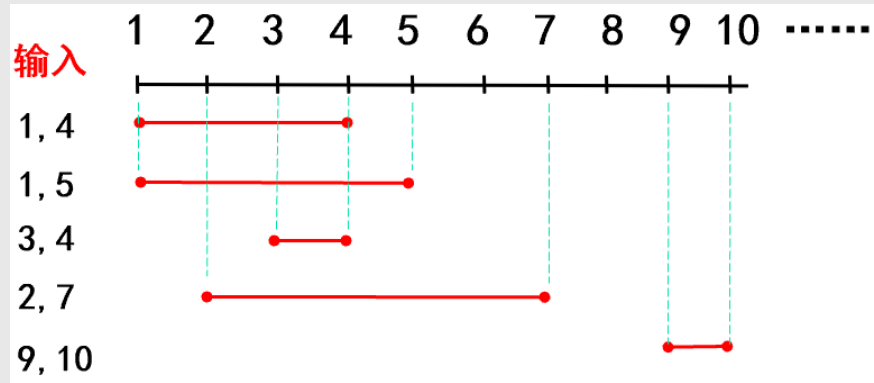
继续涂格子



涂完了

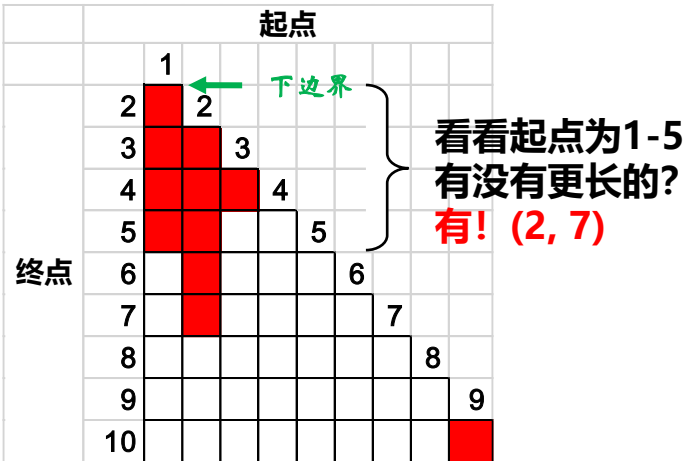
# 散列表的应用实例—区间合并

每根红线都是一名乘客的上下车记录，求车上连续有乘客的区间有哪些？



每个数字就是一站

如果你是乘务员，你该怎么算？



4. 数区间，先确定下边界



5. 数区间，找上边界

乘务员的这张表很重要！  
这张表就叫散列/哈希表

# 散列表的应用实例—区间合并

//example 6-13.c

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

```
#define MAX_N (1024*1024)
```

```
int range[MAX_N];
```

```
void print_zone()
```

```
{
```

```
    int i, n;
```

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

```
        if(range[i]==0) // 没有区间
            continue;
```

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

```
        for(n=range[i]; i<=n; i++){
```

```
            if(range[i] > n) // 新区间更远
```

```
                n = range[i]; // 扩展右区间
```

```
            i--; // 回到当前区间①的上界n
```

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

```
        }
```

```
}
```

```
int main(){
```

```
    int a, b;
```

```
    while(scanf("%d%d",&a, &b) == 2){
```

```
        if(b>range[a])
```

```
            range[a]=b;
```

```
    }
```

```
    print_zone();
```

```
    return 0;
```

```
}
```

初始化（相同下界时，扩展上界）

[1,4]

[1,5]

[1,5]

i为当前区间①的下界（左界）

区间有交叉时（ $i \leq n$ ），上界（右界）扩展（if语句成立时上界扩展  $n = \text{range}[i]$ ；if不成立，新区间不存在或包括在区间①中）。

该部分代码完成，第一个区间的上界不断扩大，直到[1,7]。（注意：n是变化的）

打印扩展后的上界

即便输入顺序为：

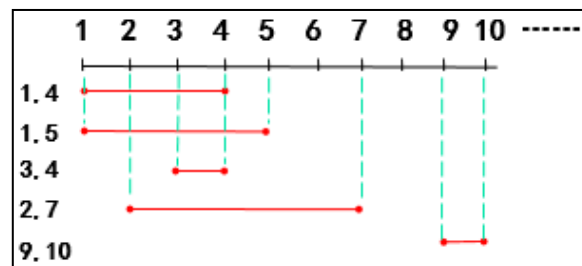
9 10

2 7

3 4

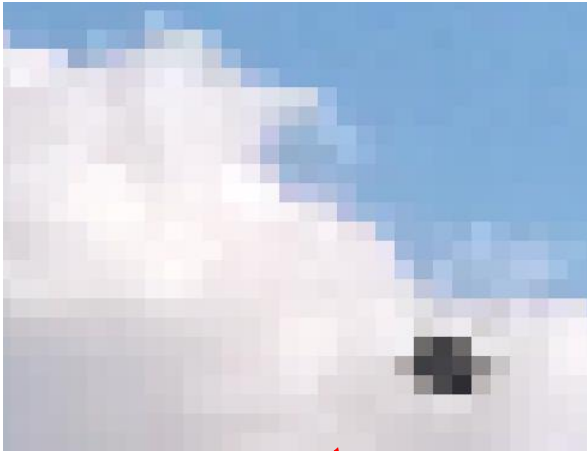
...

区间[9 10]被hash到最后的位置，即9，是最后一个区间，无需排序。即，每个区间是一个元素，被hash后，hash的位置是该区间的左界。



# 6.6 二维数组与多维数组

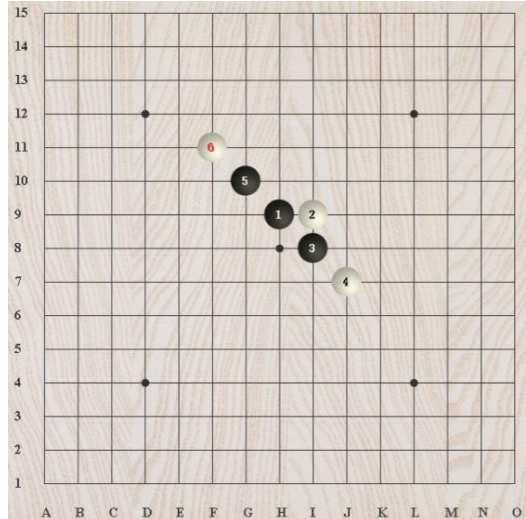
- 一维数组（单下标数组）：字符串，语音信号， $\sin(t_k)$ ，...
- 二维数组（双下标）应用更广泛：平面图像，excel表格，...



	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	魏	####	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	



# 二维数组定义及初始化

- 二维数组定义：数据类型 数组名[行数][列数];其中行数和列数是**常量表达式**
  - 例如：int a[3][2];//3行2列，3x2=6个数组元素
- 二维数组初始化：数据类型 数组名[行数][列数]={初始化数据}

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

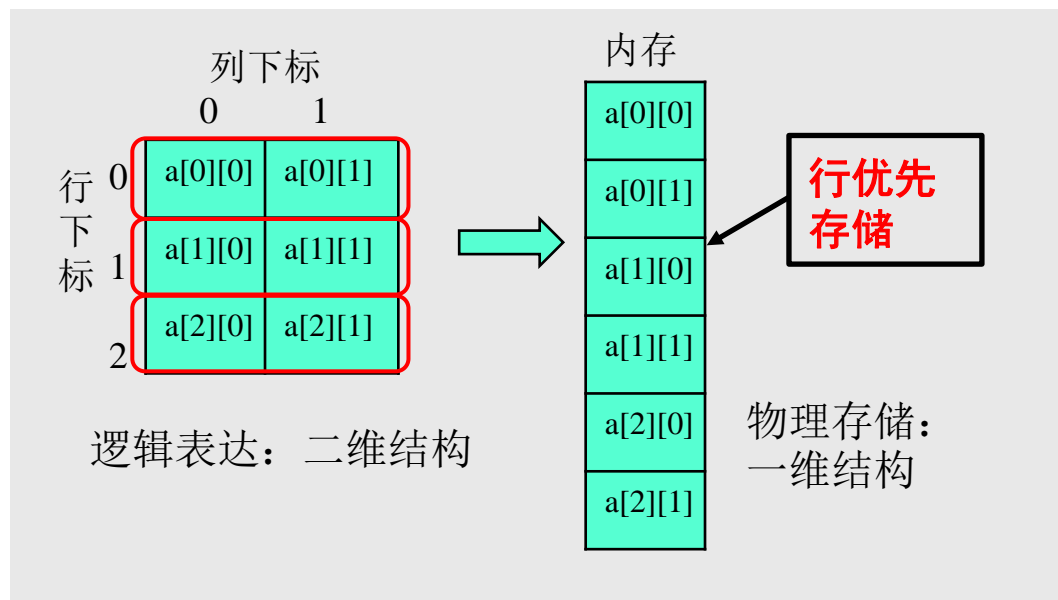
// 定义若干局部数组 int a[2][3] = {{1, 2, 3}, {4, 5, 6}}; int b[2][3] = {1, 2, 3, 4, 5, 6}; int c[2][3] = {{1, 2}, {4}};  int d[ ][3] = {{1, 2}, {4}};  int e[2][3] = {1, 2, 3, 4}; int f[1][3];	数组 a, c, d 的定义方式是好的习惯 <ul style="list-style-type: none"><li>• 按序对每一个数组元素初始化赋值</li><li>• 按序对每一个数组元素初始化赋值</li><li>• 每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</li><li>• 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</li><li>• 按行优先规则顺序初始化</li><li>• 无初始化，默认“随机值”</li></ul>
--	--

二维数组初始化的方法：
• 可以定义时初始化
• 定义时无初始化：全局二维数组默认为0，局部二维数组赋值为随机值
• 不完全初始化：初始化值的个数不超过二维数组范围时，其余部分默认为0
• 越界初始化：初始化值的个数超过数组范围时，可能造成运行时错误

初 始 化 后 每 个 数 组 的 值	1 2 3 4 5 6	a
	1 2 3 4 5 6	b
	1 2 0 4 0 0	c
	1 2 0 4 0 0	d
	1 2 3 4 0 0	e
	2008358566 2008295927 -1	f

# 二维数组的存储排列方式

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



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

# 二维数组的存储排列方式

- 二维数组的存储是行优先的。设定义数组

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

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

内存 (Memory)									
	.....	60FEF8 a: a[0][0]	60FEFC a[0][1]	60FF00 a[0][2]	60FF04 a[0][3]	60FF08 a[1][0]	60FF0C a[1][1]	60FF10 a[1][2]	60FF14 a[1][3]
60FF18 a[2][0]	60FF1C a[2][2]	...	...	...					

注：为便于理解，这里假设每个单元格占四个字节。

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



# 二维数组的存储排列方式

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

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

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

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

语法糖：

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





# 二维数组的存储排列方式

语法糖：

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



# 二维字符数组

- 二维字符数组初始化

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

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

二维数组当作一维数组使用，这个一维数组中的每一个元素是个一维数组

- 二维字符数组的引用

a[0][0]

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

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

运行结果:

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

# 二维数组使用实例

【例6-23】 星期几    已知本月有n天，第x天是星期y，求下月k日是星期几

```
#include <stdio.h>
char day_name[][12] =
{
    "Sunday",
    "Monday",
    "Tuesday",
    "Wednesday",
    "Thursday",
    "Friday",
    "Saturday"
};
```

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

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

```
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", day_name[m]);
    return 0;
}

int weekday(int x, int y, int n, int k)
{
    return (n - x + k + y) % 7;
}
```

## 二维数组使用实例

### 【例6-24】数据中的最长行 输入若干行字符串，输出最长行的长度和字符串

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

程序算法分析（示例）：

目前最长：arr[longest = 0] = "abcd"

输入前：arr[in = 1] = "ab"

第2行更短，新的输入存入第2行，即下一条：

输入后：arr[in = 1] = "abcdefghi"

若 len > max\_len（新输入的字符串更长），

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

输入前：arr[in = 0] = "abcd"

目前最长：arr[longest = 1] = "abcdefghi"

保持长的第2行，下一次新输入存入第1行，即：

待输入：arr[in = 0] = "??"

若输入字符串比目前最长的字符串短，不做处理，

接着检查下一个输入字符串，存入当前行。

输入文件结束时，输出结果。

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

# 二维数组作为函数参数

- 数组参数形式：int a[][4]

可省略行数，但不能省略列数。多维数组中可省略第一个下标，但不能省略其他下标。

- 访问二维数组元素：

在二维数组中，每行是一个一维数组，要找到特定行中的元素，函数要知道每一行有多少元素，以便在访问数组时跳过适当数量的内存地址。如定义 int a[5][4]，当访问元素a[3][2]时，函数知道跳过内存中前 3 行的12个元素以访问第 4 行（行下标为3），然后访问这一行的第 3 个元素（列下标为2），即数组中该元素前面有  $3*4+2$ 个元素

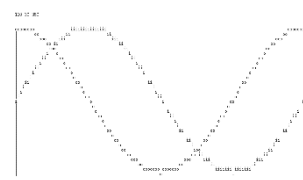
- 实参是否有足够的行数，需要由编程人员来保证
- 为使函数处理n行的矩阵，则需要将n作为一个参数传递给函数

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

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

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

## 二维数组作为函数参数



**\*\*【例6-26】绘制函数曲线** 在标准输出上水平宽度为 $w$ 、垂直高度为 $h$ （均以字符为单位）的窗口中用字符'\*'（也可以用其他字符）画出三角函数 $\sin$ 和 $\cos$ 在以字符个数为单位的角度区间 $[0, \text{ang})$ 的图像，在图像中画出 $x$ 轴和 $y$ 轴，其中坐标的 $x$ 轴从左到右平行于屏幕的横轴， $y$ 轴自下而上平行于屏幕的纵轴。

### 分析：

计算机屏幕以左上角为原点。这里，我们把一个字符看成屏幕上的一个点（实际计算机绘图中，计算机的像素点要比字符小很多）。

step1: 屏幕坐标（绘图坐标） $\rightarrow$  数学坐标：

$x = u * \text{ang} / w$  【这里角度为度，转换为弧度（数学函数的参数），则为

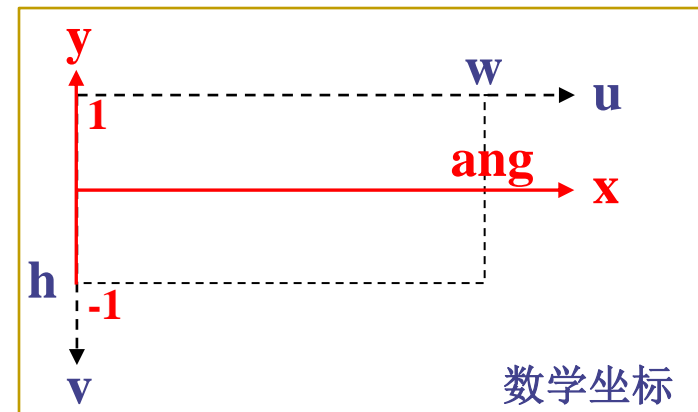
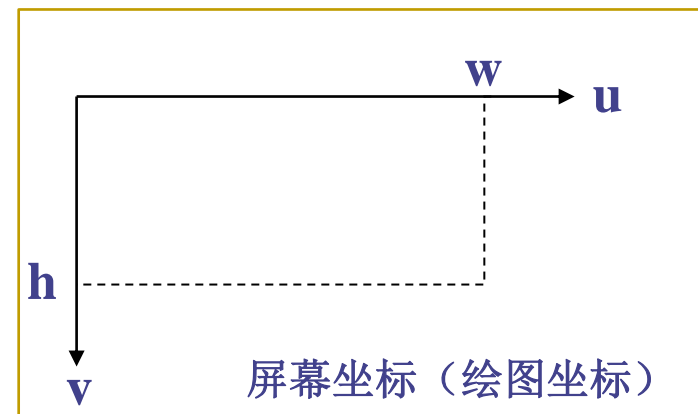
$x = u * (\text{ang} / w) * (\pi / 180)$  】

step2: 计算  $y = \sin(x)$  和  $\cos(x)$  ( $-1 \leq y \leq 1$ )

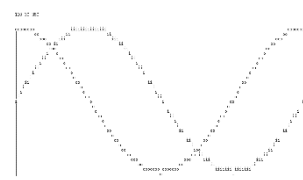
step3: 数学屏幕  $\rightarrow$  绘图坐标:  $v = -y * h / 2 + h / 2$

step4: 在坐标  $(v, u)$  处输出字符 '\*'

$u \rightarrow x \rightarrow y = \sin(x) \rightarrow v$ , then draw( $u, v$ )



# 二维数组作为函数参数



## \*\* 【例6-26】绘制函数曲线 sin和cos

```
void init(char arr[][MAX_W],
          int w, int h)
{
    int i; char s[MAX_W];
    for(i=1; i<w; i++) s[i] = ' ';
    s[0] = '|'; s[w] = '\0';
    for(i=0; i<=h; i++) strcpy(arr[i], s);
    for(i=0; i<w; i++) arr[h/2][i] = '-';
}
```

```
void draw_curve(char arr[][MAX_W],
                int w, int h, int ang)
{
    int u, v;
    double x;
    for(u=0; u<w; u++)
    {
        x = (double) u*ang/w*M_PI/180.0;
        v = (int) (h/2-sin(x)*h/2);
        arr[v][u] = 'i'; // 用 i 画sin
        v = (int) (h/2-cos(x)*h/2);
        arr[v][u] = 'o'; // 用 o 画cos
    }
}
```

```
#define MAX_H 100
#define MAX_W 400
void init(char arr[][MAX_W], int, int);
void draw_curve(char arr[][MAX_W], int, int, int);
char arr[MAX_H][MAX_W];
int main()
{
    int w, h, ang, i;
    printf("input w(<320), h(<60), ang(<=720):\n");
    scanf("%d%d%d", &w, &h, &ang);
    printf("\n\n");
}
```

init(arr, w, h); // 初始化屏幕，并画xy坐标轴

draw\_curve(arr, w, h, ang); // 计算画图函数

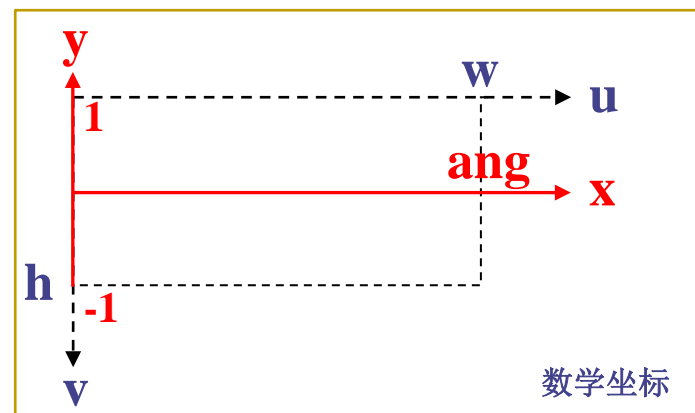
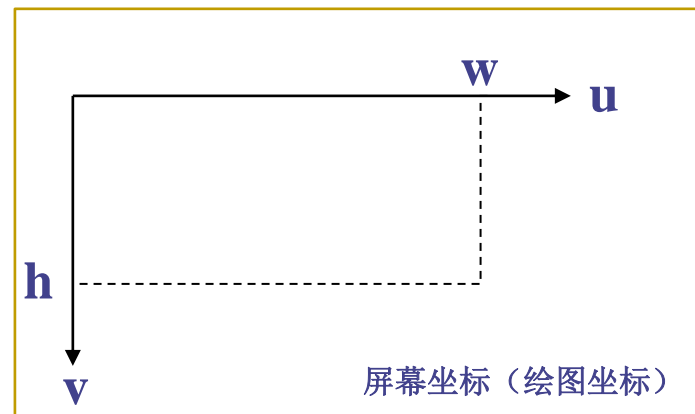
```
for(i=0; i<=h; i++)
    puts(arr[i]); // 这才是在真正的画，draw是计算
printf("\n\n"); return 0;
}
```

$u \rightarrow x \rightarrow y = \sin(x) \rightarrow v$ ,  
then draw(u, v)

$x = u * \text{ang} / w * \text{PI} / 180$   
 $v = -y * h / 2 + h / 2$

输入 w, h, ang, 如 120, 30, 360

表示uv坐标的u轴字符数区间[0 120]，对应xy坐标的x坐标角度[0 360]（单位：度，代码里会换算为弧度）；  
v轴字符区间[0 30]，对应xy坐标的y坐标值[1 -1]



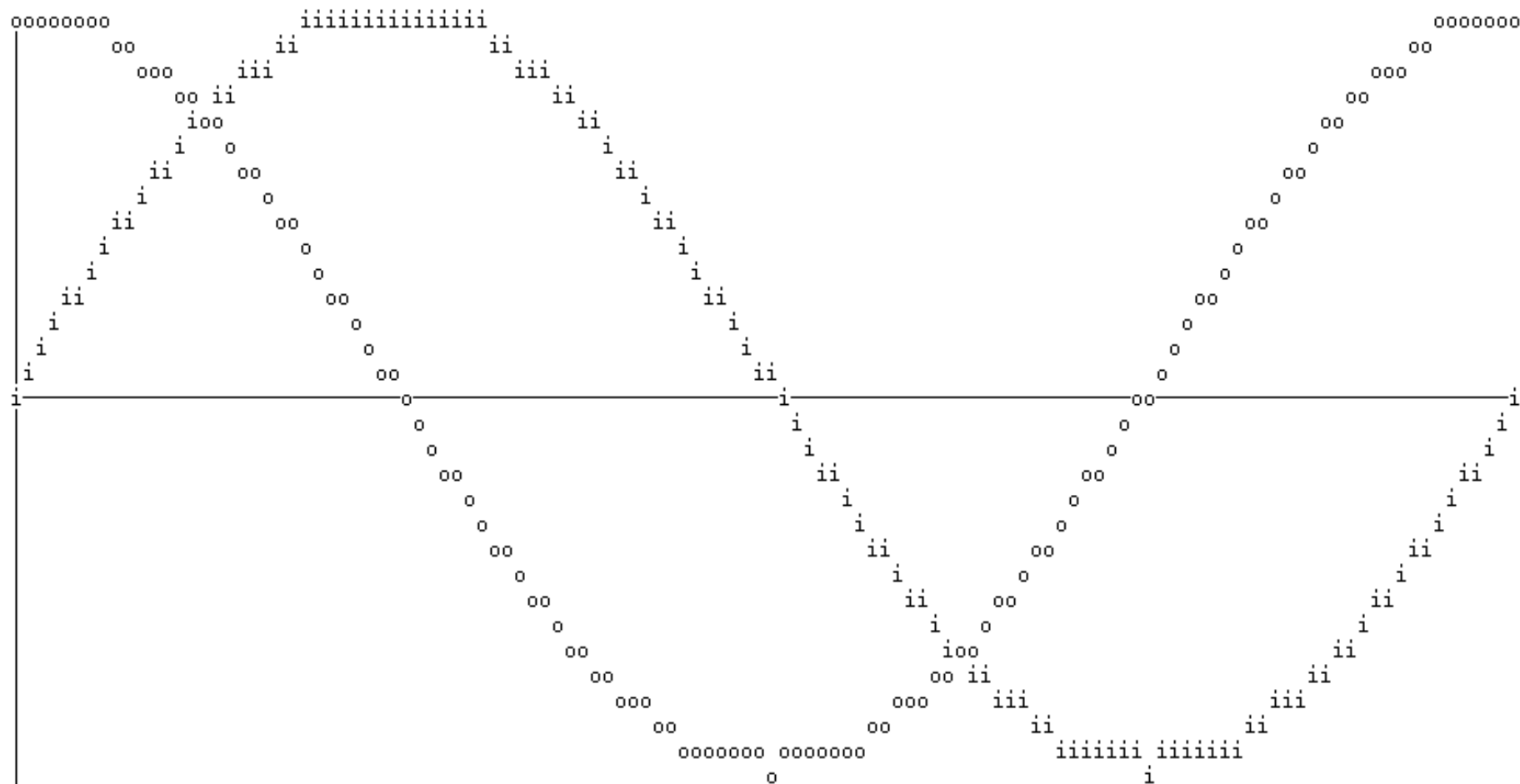


# 二维数组作为函数参数

## \*\* 【例6-26】绘制函数曲线 sin和cos

120 30 360

运行程序，输入 120 30 360  
后的运行结果  
(窗口字号为 14号)





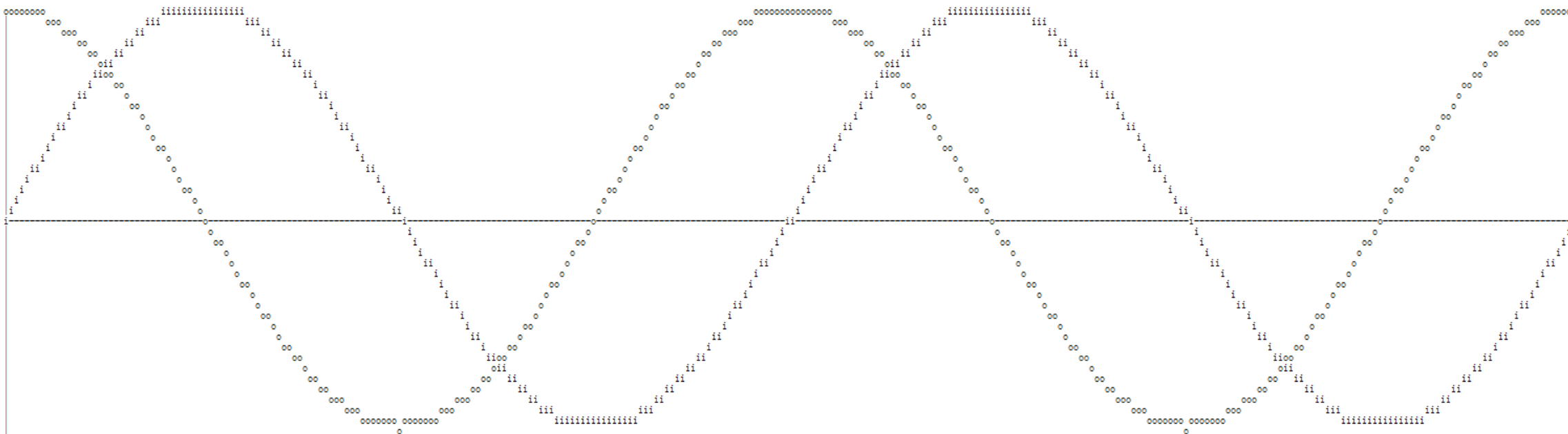
# 二维数组作为函数参数

## \*\* 【例6-26】绘制函数曲线 sin和cos

运行程序（用输入重定向命令【如下】）后的结果

C:\a-1C\example>c6-26-draw < c6-26\_in.txt

（\*in.txt 文件中的数据为：300 40 720）（窗口字号为10号，图经过按比例缩小）



# 二维数组的应用：成绩处理

【例a6-6】有一个m行n列的表格存放成绩，表示m个同学，n门课，求：

- (1) 每门课的最高（低）成绩以及获得该成绩的同学的学号（这里用数组的行号表示）
- (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）？
- .....

# 二维数组的应用：成绩处理

【例a6-6】 $m \times n$ 的成绩表格( $m$ 个同学,  $n$ 门课), 求: 每门课的最高(低)成绩; 每门课的平均成绩; 每个同学的平均成绩。(代码未完成, 请读者自行完成其余代码)

```
#define stu 6
#define course 3
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");
}
```

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

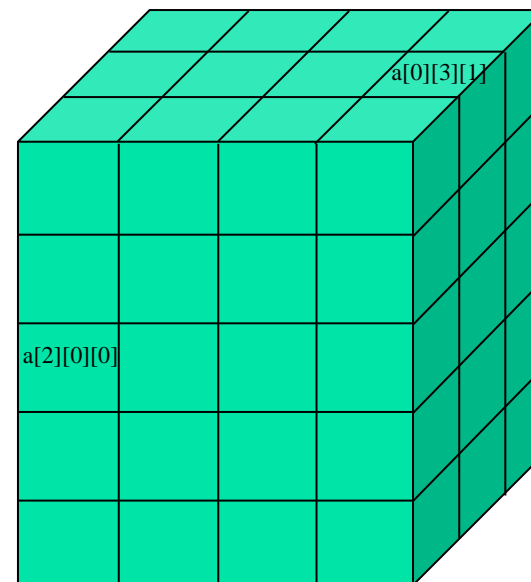
# 多维数组

## 多下标数组可以有多于两个的下标

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

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

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



# 多维数组的本质：一维数组



语法糖：  
多维数组是糖衣，  
一维数组才是药！

Wiki: In computer science, **syntactic sugar** is syntax within a programming language that is designed to **make things easier to read or to express**. It makes the language "**sweeter**" for **human use**: things can be expressed more clearly, more concisely, or in an alternative style that some may prefer.

1. 一方面永远不要忘记一维数组，另一方面将二维数组的各种定义视作工具，用哪个顺手就用哪个。
2. 实在没有顺手的，果断舍弃二维数组，直接采用一维数组，当断不断，反受其乱！（药虽苦，狠心吞下，疗效快，吃药的痛苦跟生病的痛苦相比？当然要吃药！如果药吃起来很甜，还能治病，当然最好！）

# 本讲总结

---

- 数组的应用：排序与查找（重点掌握）
- 字符串和字符数组（重点掌握）
- 了解基本的标准库字符串处理函数（掌握常用库函数）
- 使用数组的常用数据结构（理解）
  - ◆ 队列、栈、散列表
- 二维数组定义、结构和访问（掌握）
  - ◆ 一方面永远不要忘记一维数组，另一方面将二维数组的各种定义视作工具，用哪个顺手就用哪个
  - ◆ 实在没有顺手的，果断舍弃二维数组，直接采用一维数组也可以解决问题
- 多维数组定义（了解）

## 课堂作业

- 宏定义 `#define ArrayNum(x) (sizeof(x)/sizeof(x[0]))` 的作用是什么？
- 数组作为函数参数时，传递的是什么？
- 对n个元素进行选择排序时，共执行了多少次比较？冒泡排序呢？
- 在n个元素中查找某一个值，用线性查找方法，需要比较多少比较？二分查找呢（什么条件下可以使用二分查找）？
- 字符数组与字符串是否相同？
- 队、栈、哈希表，各有什么特点？分别举一个应用进行说明。
- C语言的二维数组 `a[m][n]` 的存储是行优先还是列优先？

[illegible]

# 课后作业

---

- 根据课件和教材内容复习
- 教材章节后相关习题
- 通过OJ平台做练习赛
- 上机实践题
  - ◆ 把本课件和书上讲到的所有例程输入计算机，运行并观察、分析与体会输出结果
  - ◆ 编程练习课后习题内容