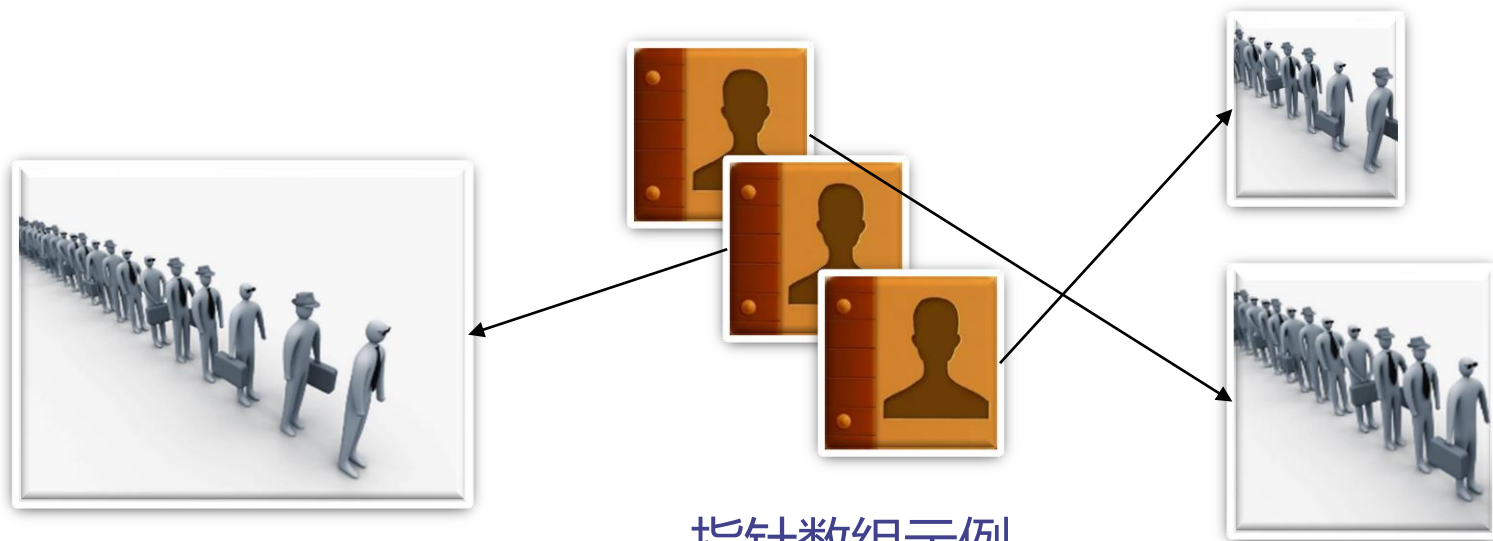


# C语言高级篇

## 第八讲

# 指针初步(2)

introduction to pointer (2)

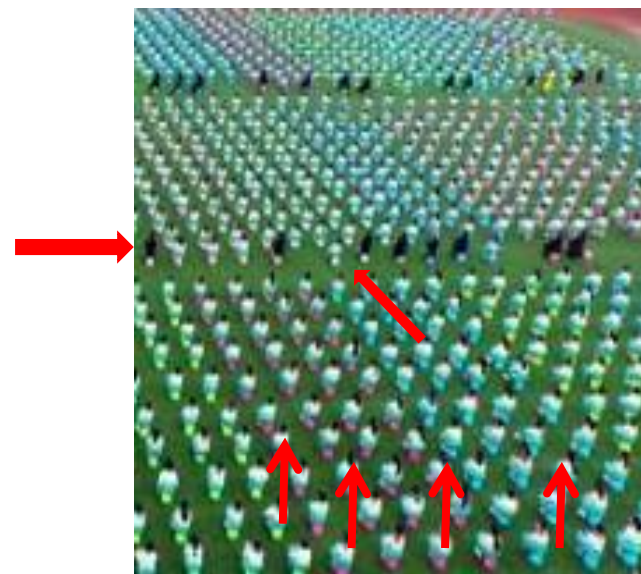


指针数组示例

## 第八讲 指针初步(2)

### 学习要点

1. 数组指针
2. 多重指针
3. 指针数组
4. 函数指针



# 简单回顾

- 指针是数据实体的地址

- ◆ 指针是一种数据类型，是从其它类型派生的类型
- ◆  $\times \times$ 类型的指针

- 指针的运算

- ◆ 指针的加减整数，指针比较，指针相减（指向同一个数组才有意义）
- ◆ 强制类型转换和通用类型void \*

- 指针变量是保存指针的变量

- ◆ &是取数据实体地址的运算
- ◆ \* 是进行间接寻址的运算
- ◆ 函数参数的指针和返回指针的函数

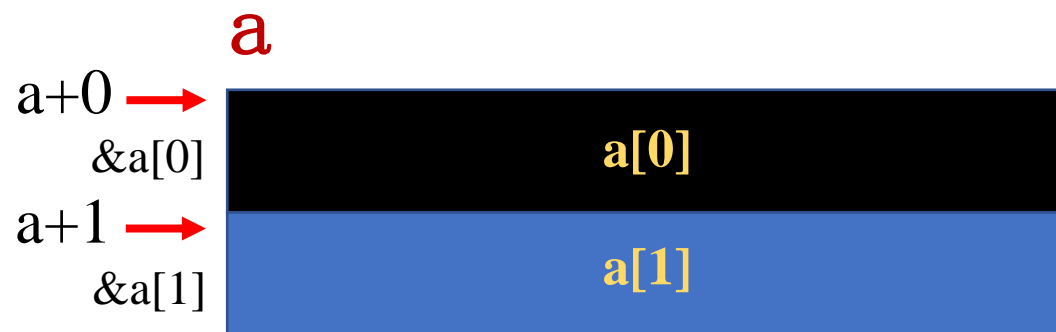
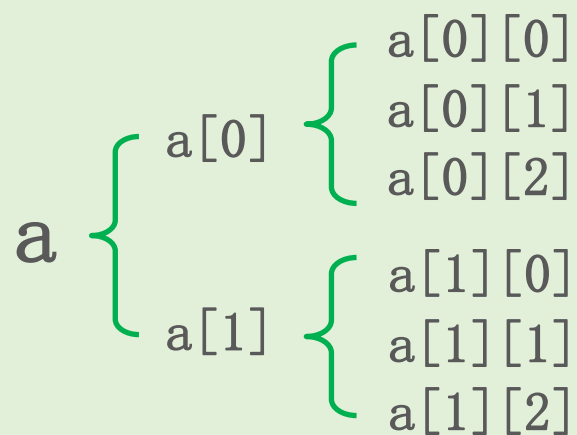
- 指针与数组

- ◆ 指向一维数组的指针
- ◆ 指向字符数组和字符串的指针

## 8.1 指向二维数组的指针

C语言将二维数组看作一维数组的嵌套，每个一维数组的元素又是一个一维数组。

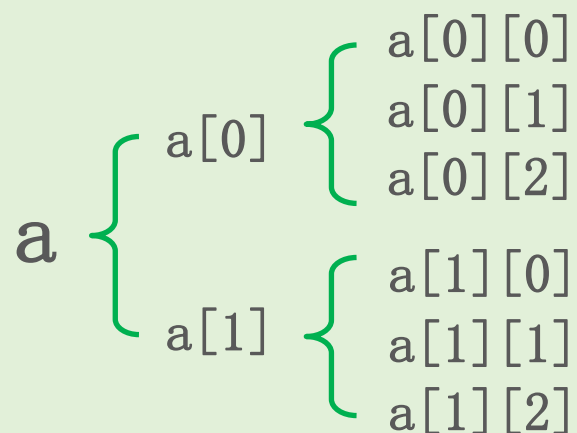
```
int a[2][3];
```



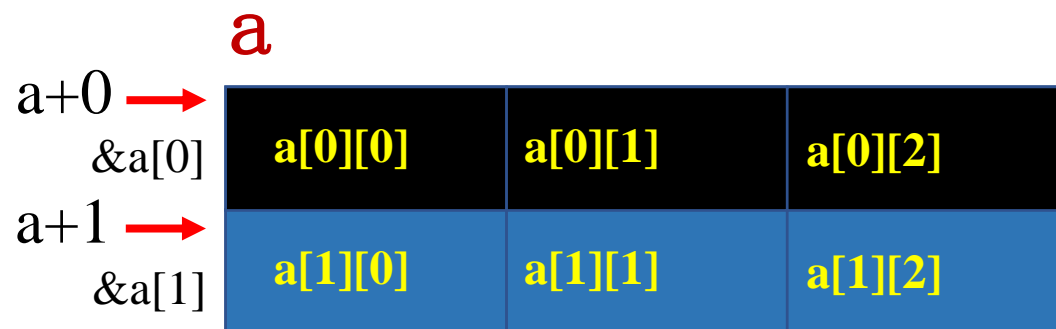
## 8.1 指向二维数组的指针

C语言将二维数组看作一维数组的嵌套，每个一维数组的元素又是一个一维数组。

```
int a[2][3];
```



数学上，可以把`a`看成一个集合：有2个元素，每个元素是一个子集合，每个子集合有3个数值（原子元素）；有6个元素，每个元素是一个数值（原子元素）。

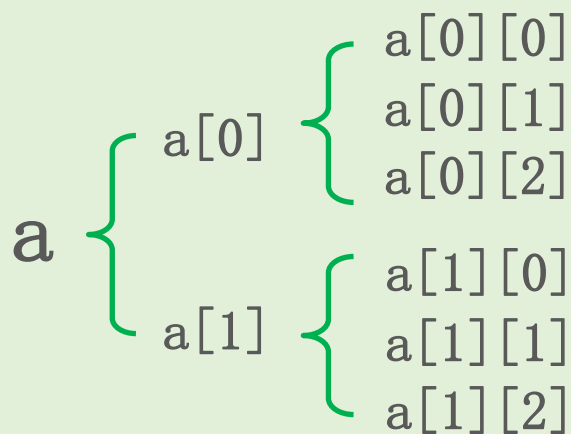


逻辑上，可看成2行3列，共6个元素。

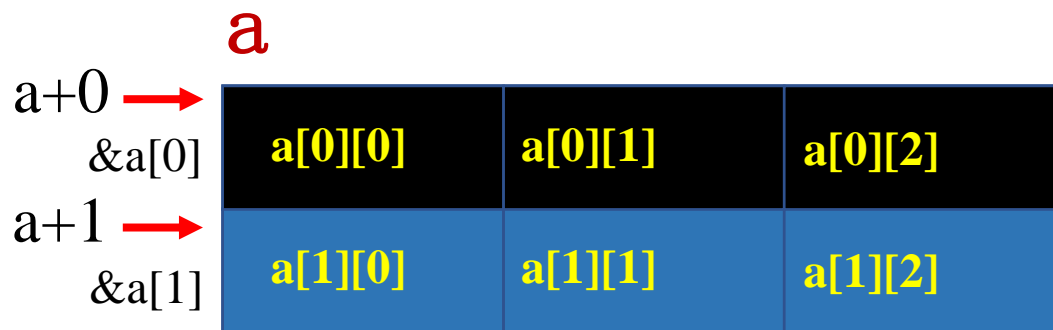
## 8.1 指向二维数组的指针

C语言将二维数组看作一维数组的嵌套，每个一维数组的元素又是一个一维数组。

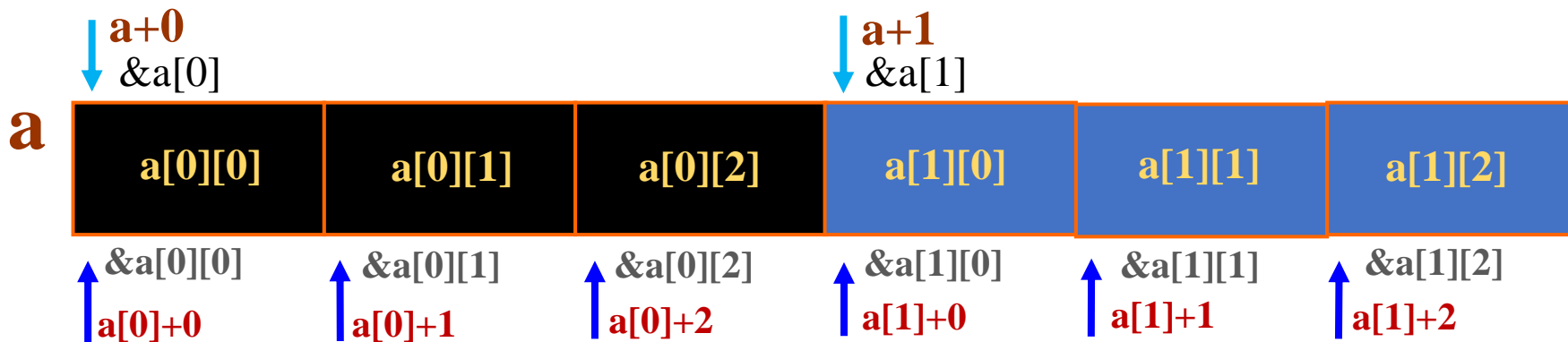
```
int a[2][3];
```



数学上，可以把`a`看成一个集合：有2个元素，每个元素是一个子集合，每个子集合有3个数值（原子元素）；有6个元素，每个元素是一个数值（原子元素）。



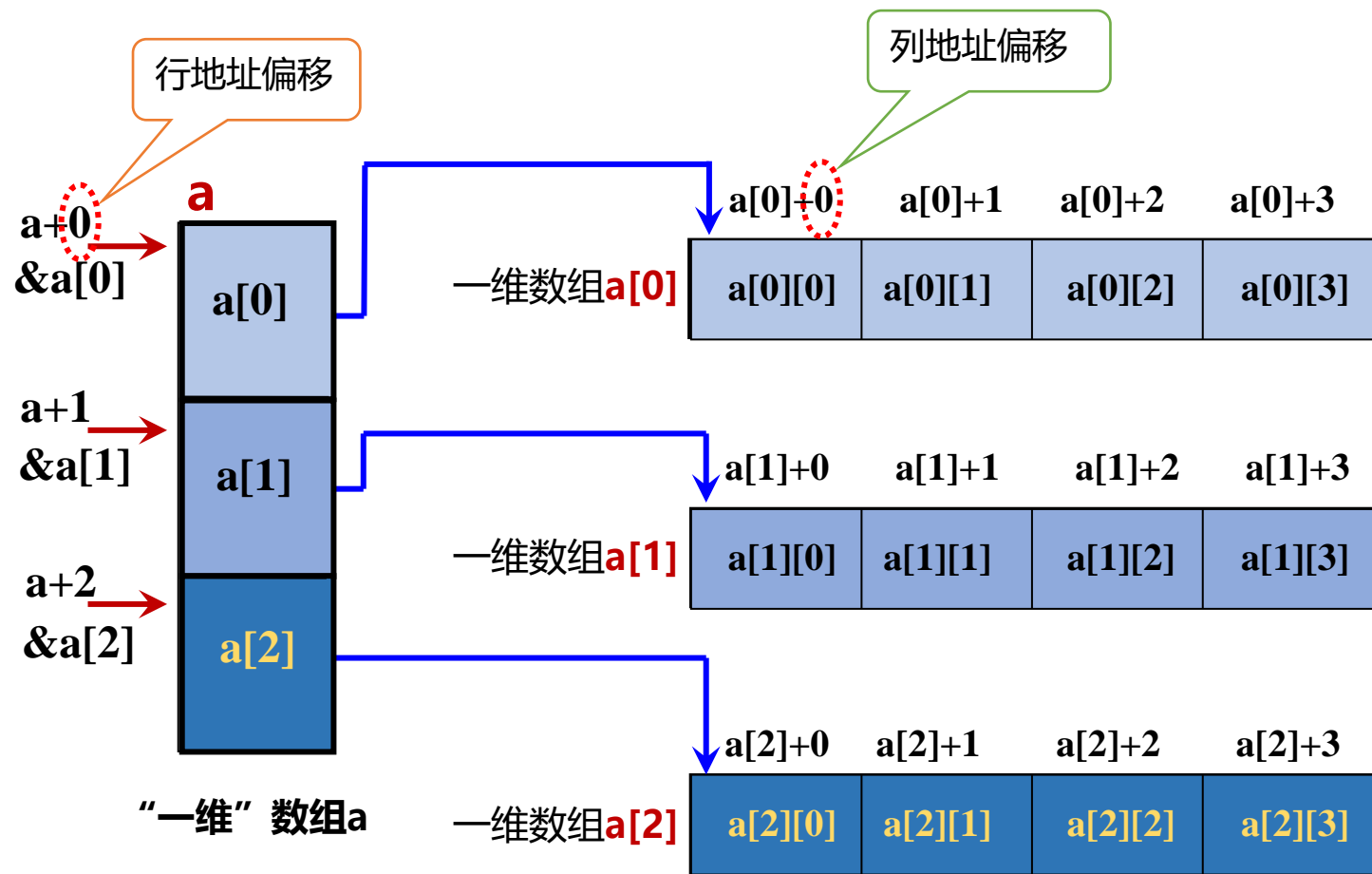
逻辑上，可看成2行3列，共6个元素。



物理上，其实就是在内存中连续存放的6个元素。

### 8.1.1 指针与二维数组

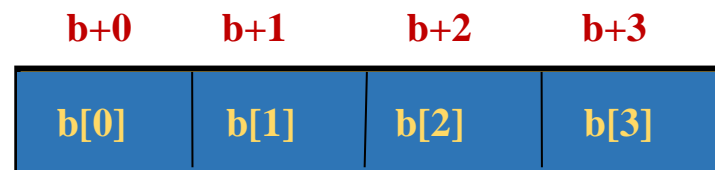
```
int a[3][4];
```



```
int *b;
```

```
b= a[1];
```

```
//让b指向a[1]的首元素
```



## [例8-1] 英文星期几对应的数字

任意输入英文的星期几，在查找星期表后输出其对应的数字。

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

```
int main()
```

```
{
```

```
    char x[10],i;
```

```
    char weekDay[][10] =
```

```
    {
```

```
        "Sunday",
```

```
        "Monday",
```

```
        "Tuesday",
```

```
        "Wednesday",
```

```
        "Thursday",
```

```
        "Friday",
```

```
        "Saturday"
```

```
    };
```

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

```
printf("Please enter a string of week:");
scanf("%s", x);
```

```
for (i = 0; i<7;i++ )
```

```
{
```

```
    if (strcmp(x, weekDay[i]) == 0)
```

```
    {
```

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

```
        return 0;
```

```
    }
```

```
}
```

```
printf("Not found!\n");
```

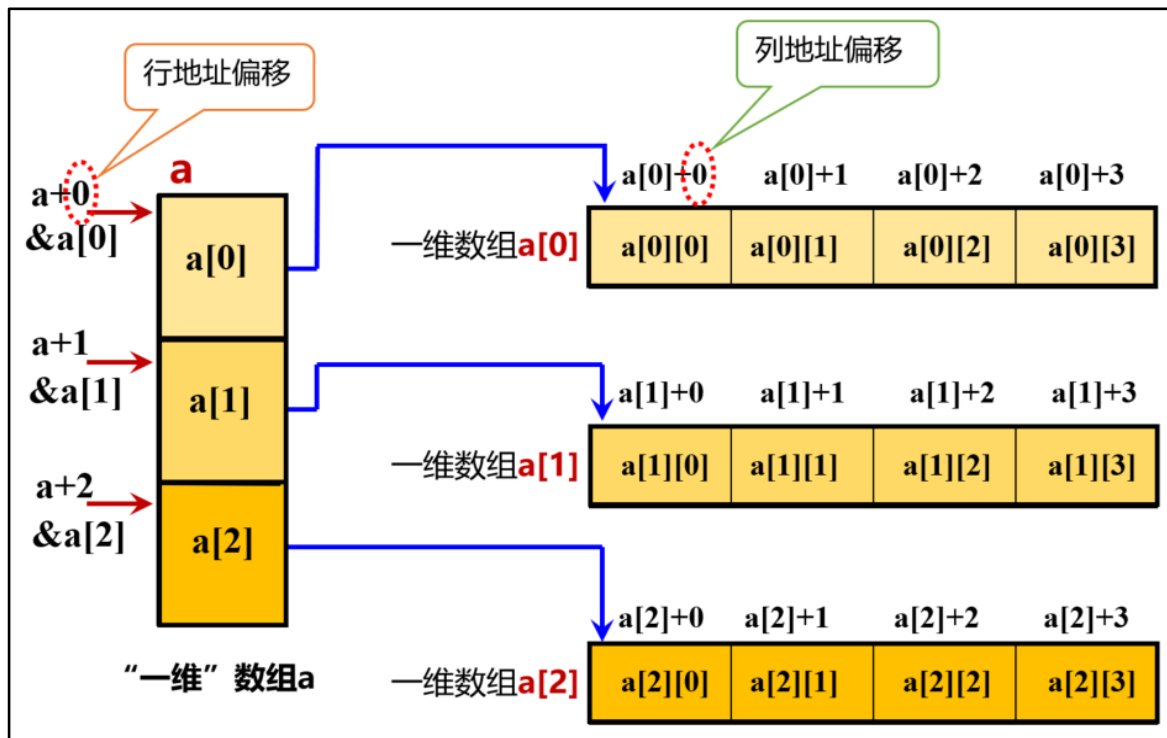
```
return 0;
```

```
}
```



# \*指针与二维数组的行地址与列地址

`int a[3][4];`



**a** 代表二维数组的首地址，第0行的地址。

**a+i** 即 **&a[i]** 代表第*i*行的地址

行地址

**\*(a+i)** 即 **a[i]**

代表第*i*行第0列的地址

列地址

**\*(a+i)+j** 即 **a[i]+j**

代表第*i*行第*j*列的地址

列地址

**\*(\*(a+i)+j)** 即 **a[i][j]**

代表第*i*行第*j*列的元素

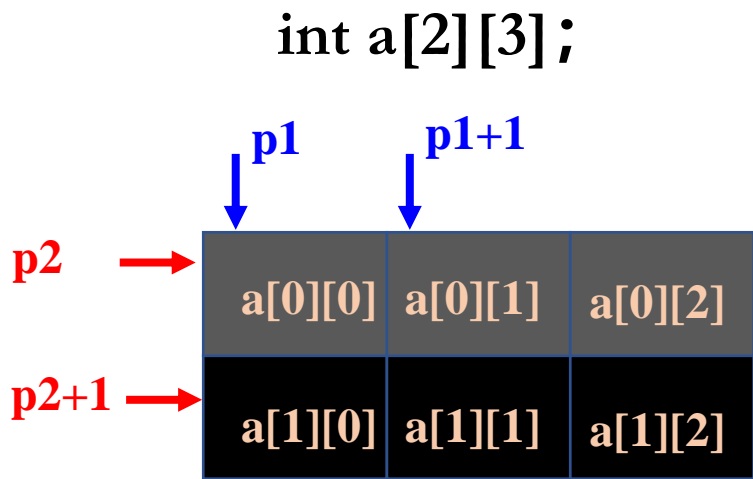
元素

$a[i][j] \iff *(a[i]+j)$        $\&a[i][j] \iff a[i]+j$

$(*(a+i))[j] \iff (*(a+i)+j)$        $*(a+i)+j$

在指向行的指针前面加一个\*，  
就转换为指向列的指针。

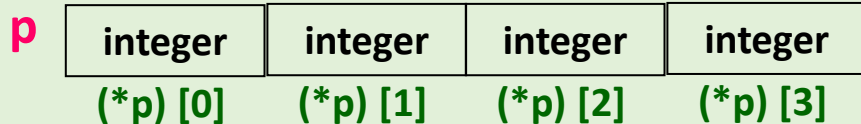
# \*指针与二维数组——列指针与行指针（形象描述）



**行指针定义：**类型 (\* 标识符) [常量表达式]

例如: `int (*p) [4];`

p为指向由4个整数组成的一维数组的指针变量



## 列指针

```
int* p1;
p1 = *a;
for (i = 0; i<2; i++ )
    for (j = 0; j<3; j++ )
        printf("%d", *(p1 + i*3 + j ));
```

## 行指针（数组指针）

```
int (*p2)[3];
p2 = a;
for (i = 0; i<2; i++ )
    for (j = 0; j<3; j++ )
        printf("%d", (*(p2+i)+j ));
```

## 行指针同样不做越界检查

```
int a[3][4]= {1,2,3,4,5,6,7,8,9,10,11,12};
int (*p)[4],i,j;

printf("\n The 2nd line::");
p = a+1;
for(j=0; j<4; j++)
    printf("%d ", (*p)[j]);
printf("\n");


printf("\n The half line::");
p = &a[0][2]; // 指向虽正确, 但类型不匹配, warning
for(j=0; j<4; j++)
    printf("%d ", (*p)[j]);
printf("\n");
```

输出


The 2nd line :: 5 6 7 8

The half line :: 3 4 5 6

a[3][4]



1	2	3	4
5	6	7	8
9	10	11	12



1	2	3	4
5	6	7	8
9	10	11	12

## \*[例8-2]日期转换问题(1)

任意给定某年某月某日，打印出它是这一年的第几天。 例：2019.4.1是2019年的第91<sup>st</sup> 天

输入样例: 2019 4 1

输出样例: 91

输入样例: 2020 4 1

输出样例: 92

```
#include <stdio.h>

int dayofYear( int, int *, int *);
int isLeap(int);
int dayTab[2][13] = {
    {0,31,28,31,30,31,30,31,31,30,31,30,31},
    {0,31,29,31,30,31,30,31,31,30,31,30,31}
};

int main() {
    int yearday, year, month, day;
    printf("input year month day: ");
    scanf("%d%d%d", &year, &month, &day);

    yearday = dayofYear(year, &month, &day);
    printf("%d", yearday);
    return 0;
}
```

```
int dayofYear(int year, int *pMonth,
              int *pDay)
{
    int i, leap, day=0;
    leap = isLeap(year);

    for (i=0; i<*pMonth; i++)
        day += *(dayTab+leap+i);

    day += *pDay;
    return day;
}
```

```
int isLeap(int year)
{
    return (
        ((year%4 == 0) && (year%100 != 0))
        || (year % 400 == 0) );
}
```

## \*[例8-3]日期转换问题(2)

已知某一年的第几天，计算它是该年的第几月第几日。

```
void MonthDay( int, int, int *, int *);
int isLeap(int);
int dayTab[2][13] ={
    {0,31,28,31,30,31,30,31,31,30,31,30,31},
    {0,31,29,31,30,31,30,31,31,30,31,30,31}
};
int main() {
    int yearday, year, month=0, day=0;
    printf("input year and yeardays:");
    scanf("%d%d", &year, &yearday);
    MonthDay(year, yearday, &month, &day);
    printf("Mon: %d, Day: %d", month, day);
    return 0;
}
```

```
void MonthDay(int year, int yearday,int *pMonth,int *pDay)
{
    int i, leap;
    leap = isLeap(year);
    for (i=1; yearday>dayTab[leap][i]; i++)
        yearday -= (*(dayTab+leap)+i);
    *pMonth = i;           // 月
    *pDay = yearday;       // 日
}
```

```
int isLeap(int year)
{
    return(((year%4 == 0)&&(year%100 != 0))
        ||(year%400 == 0) );
}
```

## 日期转换问题对比

- 任意给定某年某月某日，打印出它是这一年的第几天，例如：2019.4.1是2019年的第91<sup>st</sup> 天
- 已知某一年的第几天，计算它是该年的第几月第几日

```
int dayTab[2][13] ={
    {0,31,28,31,30,31,30,31,31,30,31,30,31},
    {0,31,29,31,30,31,30,31,31,30,31,30,31}
};
```

```
int dayOfYear(int year, int *pMonth, int *pDay)
{
    int i, leap, day=0;
    leap = isLeap(year);
    for (i=0; i<*pMonth; i++)
        day += *(*(dayTab+leap)+i);
    day += *pDay;
    return day;
}
```

两段代码功能互逆，请认真对比分析，  
仔细阅读

```
void MonthDay(int year, int yearday, int *pMonth, int *pDay)
{
    int i, leap;
    leap = isLeap(year);
    for (i=1; yearday>dayTab[leap][i]; i++)
        yearday -= *(*(dayTab+leap)+i);
    *pMonth = i;
    *pDay = yearday;
}
```

## 8.1.2 二维数组和指针作为函数的参数

二维数组作为函数的形参:

```
#include <stdio.h>
void set_char(char *c[])
{
    c[0][0]='b';
}
int main()
{
    char c[10][10];
    c[0][0] = 'a';
    set_char(c);
    printf("%c\n", c[0][0]);
    return 0;
}
```

c[10][10]与  
\*c[]不匹配

```
#include <stdio.h>
void set_char(char *c[10])
{
    c[0][0]='b';
}
int main()
{
    char c[10][10];
    c[0][0] = 'a';
    set_char(c);
    printf("%c\n", c[0][0]);
    return 0;
}
```

\*c[10]是指  
针数组, 不  
是行指针!

Message

In function 'main':

[Warning] passing argument 1 of 'set\_char' from incompatible pointer type

[Note] expected 'char \*\*' but argument is of type 'char (\*)[10]'

## 8.1.2 二维数组和指针作为函数的参数

二维数组作为函数的形参：

```
#include <stdio.h>
void set_char(char c[][10])
{
    c[0][0]='b';
}
int main(){
    char c[10][10];
    c[0][0] = 'a';
    set_char(c);
    printf("%c\n", c[0][0]);
    return 0;
}
```

```
#include <stdio.h>
void set_char(char (*c)[10])
{
    c[0][0]='b';
}
int main(){
    char c[10][10];
    c[0][0] = 'a';
    set_char(c);
    printf("%c\n", c[0][0]);
    return 0;
}
```

(\*c)[10]才是指向  
含有10个元素的  
行指针，即，  
**数组指针！**

b

-----  
Process exited after 0.1637 seconds with return value 0  
请按任意键继续. . .






## 8.1.2 二维数组和指针作为函数的参数

- 二维数组是特殊的一维数组，其元素也是一维数组，并按行存储。
- 二维数组作为函数的参数，参数说明中应指明数组的列数，而行数可省略。
- 在函数参数声明中，数组和指针等价，函数参数一般多用指针。

如前一个例子中：

`void set_char(char c[][10])`  `void set_char(char (*c)[10])`

可写为： `void set_char(char c[10][10])`

`void set_char(char c[][])`  `void set_char(char *c[])`  `void set_char(char *c[10])` 

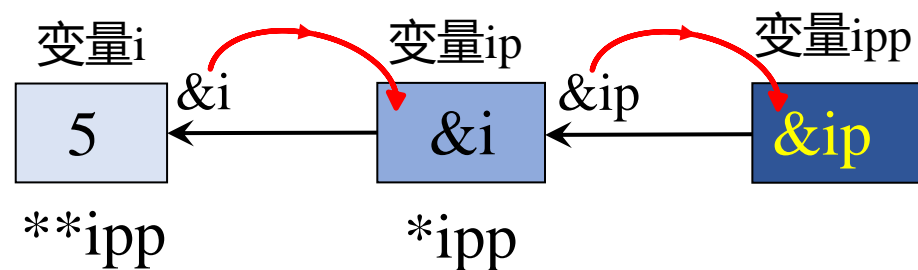
既然有数组，为什么要用指针？

指针能快速方便地指向需要去的地方，很灵活。上天入地，无所不能。

## \* 8.2 多重指针

- 如果指针变量中保存的是另一指针变量的地址，该指针变量就称为**指向指针的指针**。
- 多级指针：即多级间接寻址 (Multiple Indirection)
- 多重指针的定义： 类型 \*\*标识符;

```
int main()
{
    int i = 5;
    int *ip = &i;
    int **ipp = &ip;
    printf("i = %d, **ipp = %d\n", i, **ipp);
    **ipp = 10;
    printf("i = %d, **ipp = %d\n", i, **ipp);
    return 0;
}
```



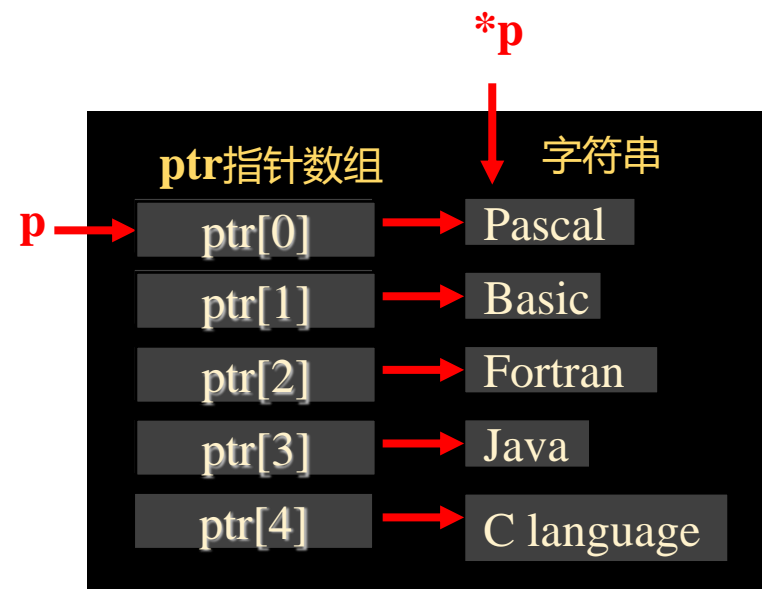
输出:

i = 5, \*\*ipp = 5

i = 10, \*\*ipp = 10

## [例8-4] 多重指针与指针数组

```
#include <stdio.h>
int main()
{
    int i;
    char *ptr[] = {"Pascal", "Basic", "Fortran",
                  "Java", "C language"};
    char **p; //声明指向指针的指针p
    p = ptr;  //用p指向指针数组首地址
    for (i = 0; i < 5; i++)
    {
        printf("%s\n", *p);
        p++;
    }
    printf("%c\n", *(*(--p) + 3));
    return 0;
}
```



输出

Pascal  
Basic  
Fortran  
Java  
C language

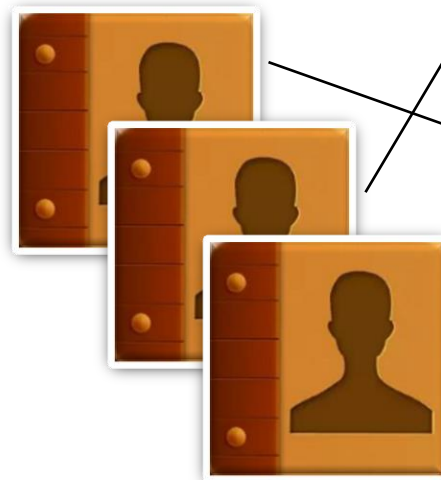
输出?

## 8.3 指针数组

- 元素类型为指针的数组称为指针数组。
- 常用于管理各类数据的索引。
- 组织数据、简化程序、提高程序的运行速度。



元素数组



指针数组



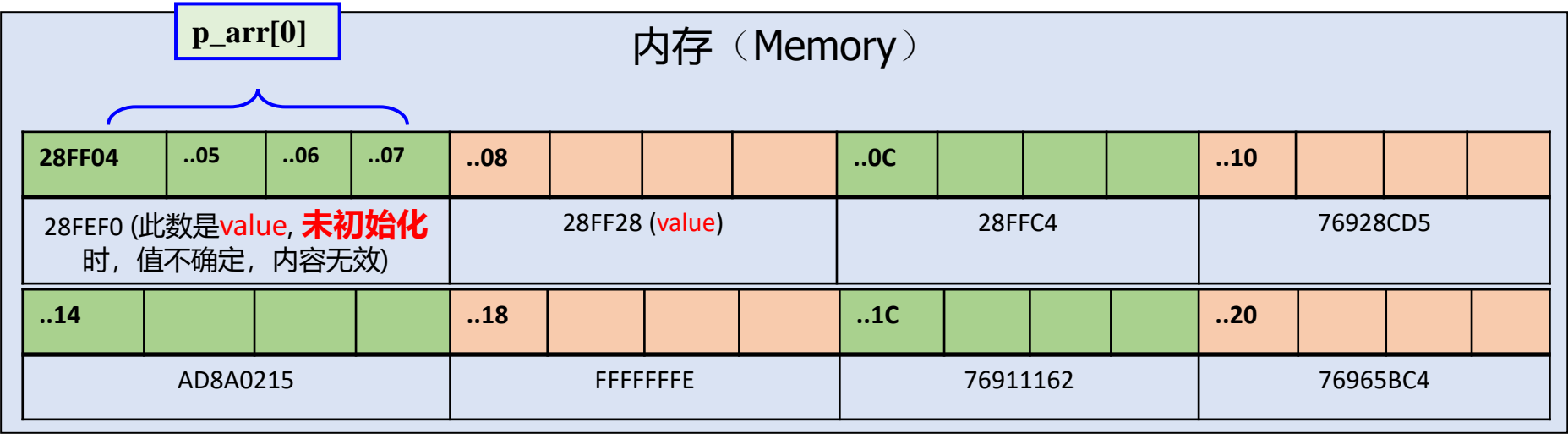
# 8.3.1 一维指针数组

指针数组类似于普通数组，为说明元素是指针，需在类型与数组名之间加上表示指针的\*。

指针数组定义：数据类型 \*标识符[常量表达式];

```
int *p_arr[N];           // 一维指针数组的声明
```

未初始化的指针内容(value)是无效的!



p\_arr[0] 是一个指针变量, 其指向int

p\_arr[0] 其首地址(&p\_arr[0])是 28FF04

p\_arr[0] 未初始化时, 其值是不确定的, 无意义(表中 0028FEF0 是确定值)

p\_arr[1],

p\_arr[2], ..

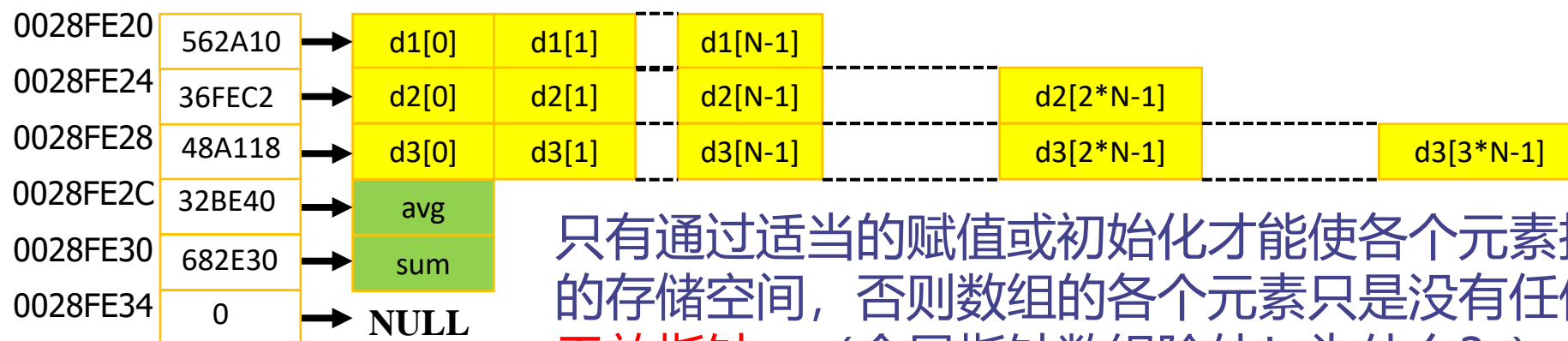
同理

## 8.3.1 一维指针数组

一维指针数组的初始化：指针数组可以在定义时初始化，但指针数组的初始化表中只能包含变量的地址、数组名，以及表示无效指针的常量NULL。

```
double d1[N], d2[2 * N], d3[3 * N], avg, sum;  
double *dp_arr[] = {d1, d2, d3, &avg, &sum, NULL};
```

**dp\_arr**



只有通过适当的赋值或初始化才能使各个元素指向确定的存储空间，否则数组的各个元素只是没有任何含义的无效指针。（全局指针数组除外！为什么？）

```
double* dp_arr[N]; // 若在函数内定义，未初始化，无效指针
```

[例8-5] 星期几 已知某月x日是星期y，该月有n天，设计一个函数，在标准输出上以文字方式输出下一个月的k日是星期几。

```
char *week_days[] =  
{  
    "Sunday",  
    "Monday",  
    "Tuesday",  
    "Wednesday",  
    "Thursday",  
    "Friday",  
    "Saturday"  
};
```

指针数组

week_days[0]	→	Sunday
week_days[1]		Monday
week_days[2]		Tuesday
week_days[3]	...	Wednesday
week_days[4]		Thursday
week_days[5]		Friday
week_days[6]	→	Saturday

```
void week_day(int x, int y, int n, int k)  
{  
    int m;  
    m = (n - x + y + k) % 7;  
    printf("%s\n", week_days[m]);  
}
```

## [例6-9] vs [例8-5] (二维数组的星期几)

```
// 例6-9
```

```
char day_name[][12] =  
{  
    "Sunday",  
    "Monday",  
    "Tuesday",  
    "Wednesday",  
    "Thursday",  
    "Friday",  
    "Saturday"  
};
```

二维数组

day\_name[0]

S	u	n	d	a	y	\0					
---	---	---	---	---	---	----	--	--	--	--	--

day\_name[1]

M	o	n	d	a	y	\0					
---	---	---	---	---	---	----	--	--	--	--	--

day\_name[2]

T	u	e	s	d	a	y	\0				
---	---	---	---	---	---	---	----	--	--	--	--

day\_name[3]

W	e	d	n	e	s	d	a	y	\0		
---	---	---	---	---	---	---	---	---	----	--	--

day\_name[4]

.	.	.									
---	---	---	--	--	--	--	--	--	--	--	--

day\_name[5]

--	--	--	--	--	--	--	--	--	--	--	--

day\_name[6]

--	--	--	--	--	--	--	--	--	--	--	--

```
void week_day(int x, int y, int n, int k)  
{  
    int m;  
    m = (n - x + y + k) % 7;  
    printf("%s\n", day_name[m]);  
}
```

与指针数组有什么不同?



## 8.3.1 一维指针数组

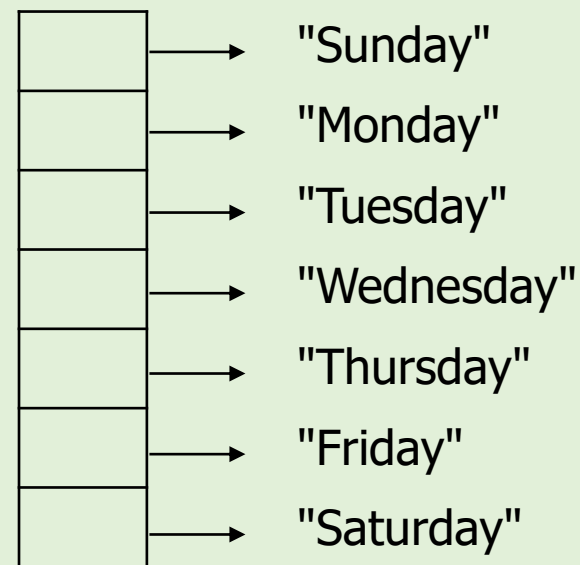
### 二维字符数组与字符指针数组的不同结构

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
W	e	d	n	e	s	d	a	y	\0	\0	\0
T	h	u	r	s	d	a	y	\0	\0	\0	\0
F	r	i	d	a	y	\0	\0	\0	\0	\0	\0
S	a	t	u	r	d	a	y	\0	\0	\0	\0

例6-9中的二维字符数组

```
char day_name[][12];
```

space is  $7*12$



例8-5中的字符指针数组

```
char *week_days[];
```

space is  $7*4+x1+x2+...+x7$

## 8.3.1 一维指针数组

指针数组与二维数组的区别有以下三点：

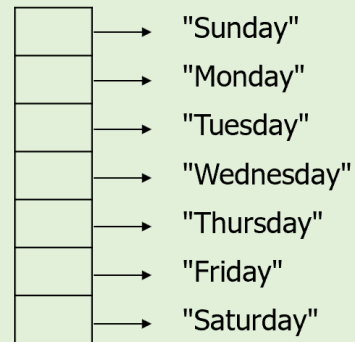
(1) 指针数组**只为指针分配了存储空间**，其所指向的数据元素所需要的存储空间是通过其他方式另行分配的。

(2) 二维数组每一行中元素的个数是在数组定义时明确规定的，并且是完全相同的；而指针数组中各个指针所**指向的存储空间**的长度不一定相同。

(3) 二维数组中全部元素的存储空间是连续排列的；而在指针数组中，只有**各个指针的存储空间是连续排列的**，其所指的数据元素的存储顺序取决于存储空间的分配方法，并且常常是不连续的。

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
W	e	d	n	e	s	d	a	y	\0	\0	\0
T	h	u	r	s	d	a	y	\0	\0	\0	\0
F	r	i	d	a	y	\0	\0	\0	\0	\0	\0
S	a	t	u	r	d	a	y	\0	\0	\0	\0

例6-9中的二维字符数组  
`char day_name[][12];`  
space is 7\*12



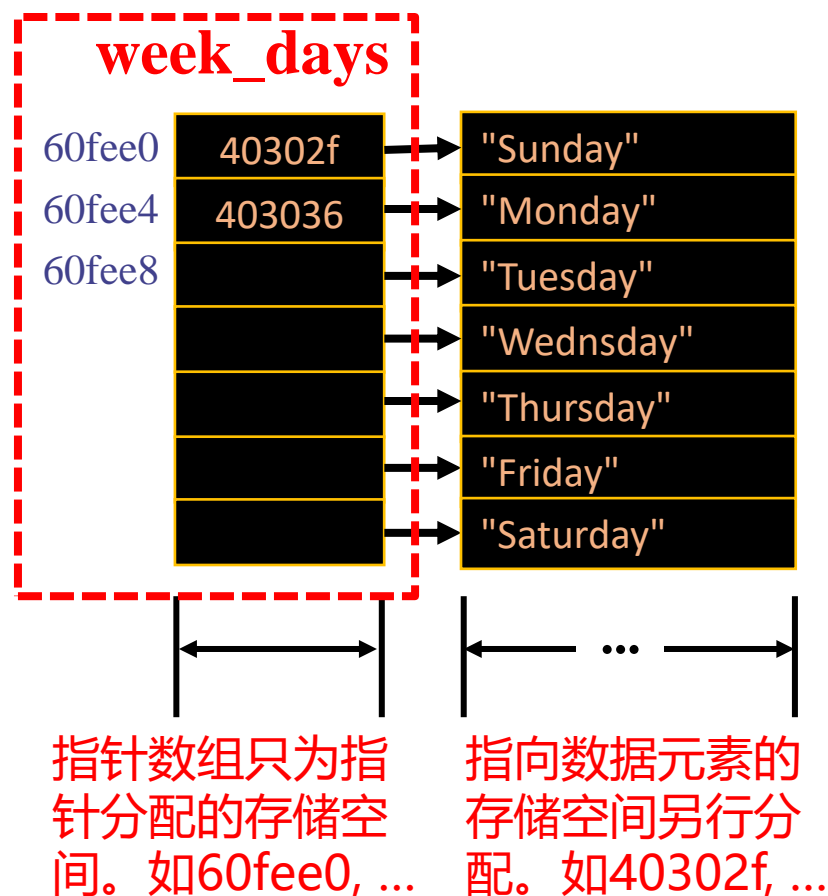
例8-5中的字符指针数组  
`char *week_days[];`  
space is 7\*4+x1+x2+...+x7

(1) 指针数组**只为指针分配了存储空间**，其所指向的数据元素所需要的存储空间是通过其他方式另行分配的。

```
char *week_days[] = {  
    "Sunday",  
    "Monday",  
    "Tuesday",  
    "Wednesday",  
    "Thursday",  
    "Friday",  
    "Saturday"  
};  
for(i=0; i<7; i++)  
{  
    printf("%x -> ", &(week_days[i]));  
    printf("%x\n", week_days[i]);  
}
```

程序输出为:

```
C:\Users\song\Desktop\sytestC\testc\temptemp-dyna-array.exe  
60fee0 -> 40302f  
60fee4 -> 403036  
60fee8 -> 40303d  
60feec -> 403045  
60fef0 -> 40304f  
60fef4 -> 403058  
60fef8 -> 40305f
```



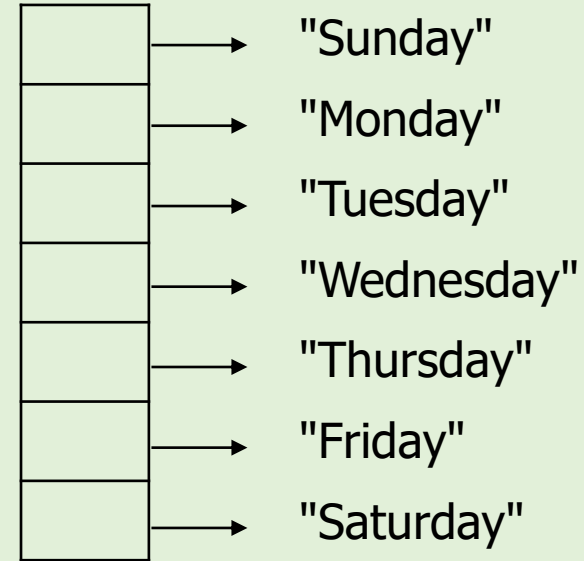
(2) 二维数组每一行中元素的个数是在数组定义时明确规定的，并且是完全相同的；而指针数组中各个指针所指向的存储空间长度不一定相同。

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
W	e	d	n	e	s	d	a	y	\0	\0	\0
T	h	u	r	s	d	a	y	\0	\0	\0	\0
F	r	i	d	a	y	\0	\0	\0	\0	\0	\0
S	a	t	u	r	d	a	y	\0	\0	\0	\0

例6-9中的二维字符数组

```
char day_name[][12];
```

space is  $7*12$

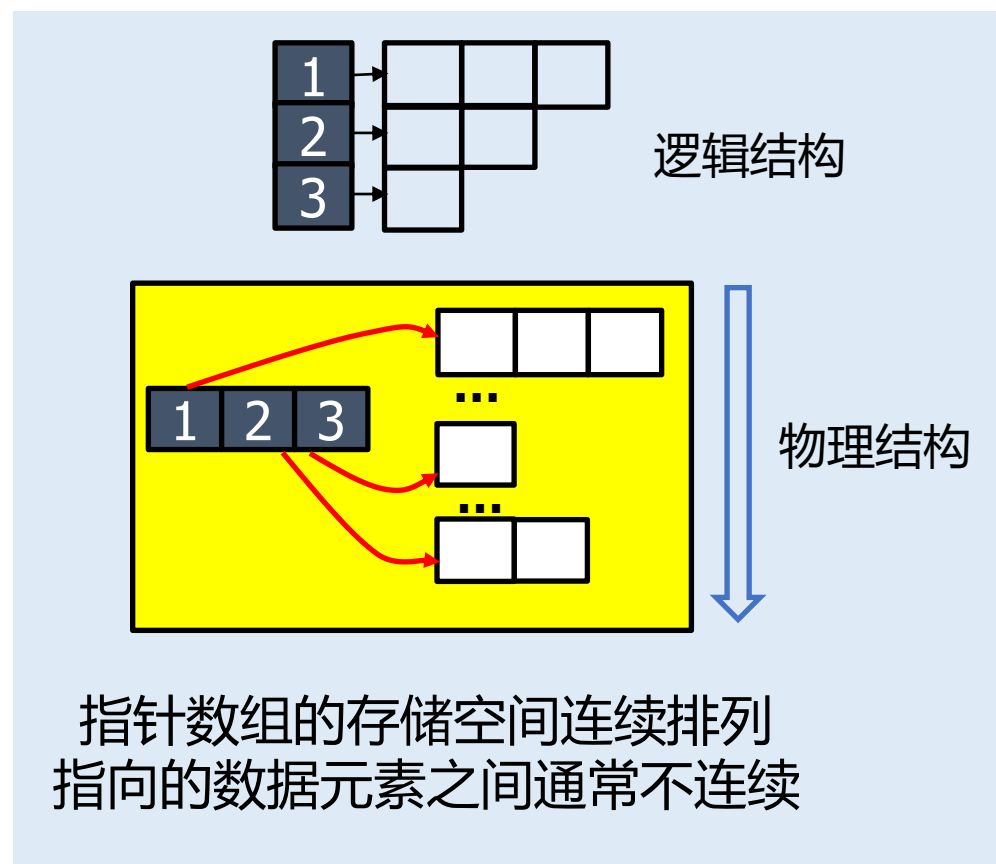
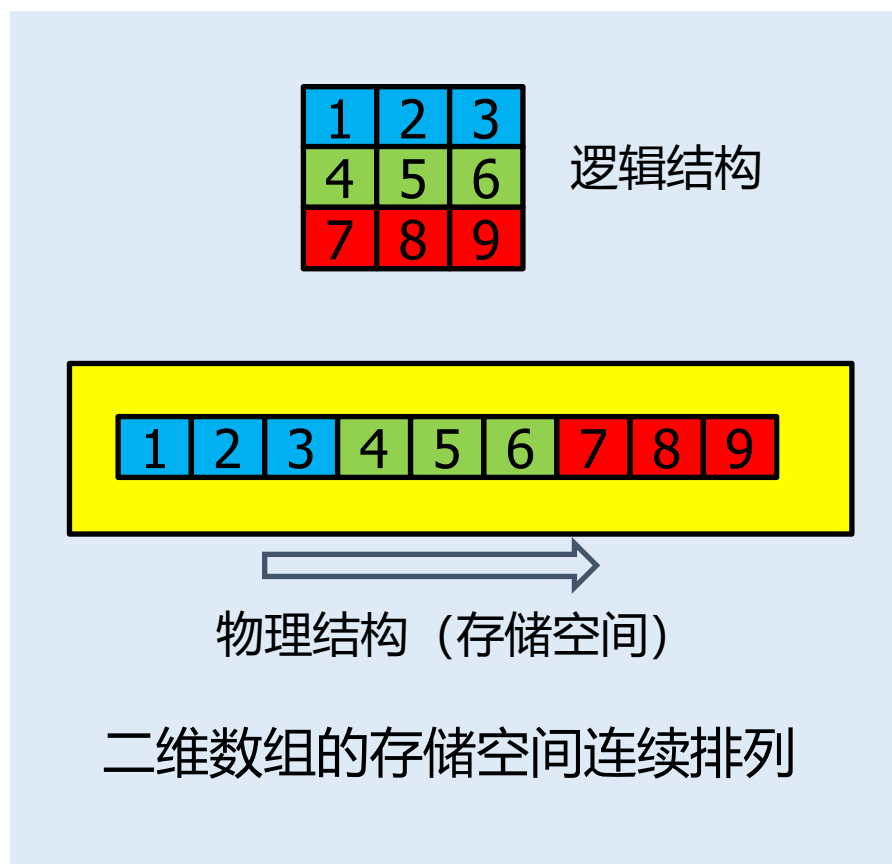


例8-5中的字符指针数组

```
char *week_days[];
```

space is  $7*4+x1+x2+...+x7$

(3) 二维数组中全部元素的存储空间是连续排列的；在指针数组中，只有各个指针的存储空间连续排列，其所指的数据元素的存储顺序取决于存储空间的分配方法，并且元素之间常常是不连续的。



## 8.3.1 一维指针数组

### 例6-9中的二维字符数组

```
char day_name[][LEN] = { "Sunday", ... }
```

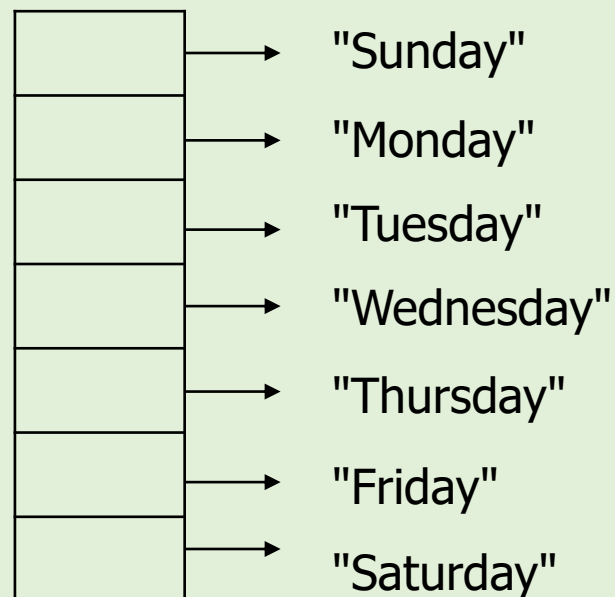
S	u	n	d	a	y	\0	\0	\0	\0	\0	\0
M	o	n	d	a	y	\0	\0	\0	\0	\0	\0
T	u	e	s	d	a	y	\0	\0	\0	\0	\0
W	e	d	n	e	s	d	a	y	\0	\0	\0
T	h	u	r	s	d	a	y	\0	\0	\0	\0
F	r	i	d	a	y	\0	\0	\0	\0	\0	\0
S	a	t	u	r	d	a	y	\0	\0	\0	\0

```
day_name[0][0] = 's'; // 'S' => 's',  
                      // 改变元素值, OK
```

二维字符数组可以读写。

### 例8-5中的字符指针数组

```
char *week_day[] = { "Sunday", ... }
```



```
*(week_day[0]) = 's'; // 运行错误!  
                  // 常量数据不能改
```

在本例，指针数组所指向的字符串是常量，指针数组元素是变量，可以指向不同位置。

## 8.3.1 一维指针数组

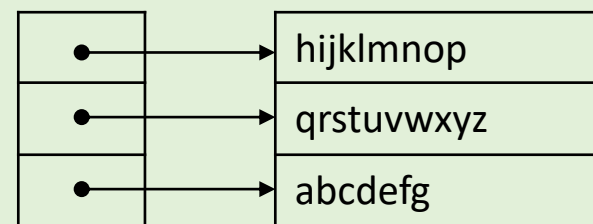
- 指针数组常被用作**数据索引**，以加快数据**定位、查找、交换和排序**等操作的速度。
- 在一些文字处理程序中，数据一般以“行”为单位保存在二维数组中，在数据处理的过程中，对各行位置的交换，以及整行内容的删除和新行的添加是频繁进行的操作（**计算代价很大**）。为**提高程序的运行速度**，往往使用**指针数组作为实际数据的索引**。

排序前

hijklmnop
qrstuvwxyz
abcdefg

直接对二维字符数组排序

排序前



利用指针数组排序

## 8.3.1 一维指针数组

- 指针数组常被用作**数据索引**，以加快数据**定位、查找、交换和排序**等操作的速度。
- 在一些文字处理程序中，数据一般以“行”为单位保存在二维数组中，在数据处理的过程中，对各行位置的交换，以及整行内容的删除和新行的添加是频繁进行的操作（**计算代价很大**）。为提高程序的运行速度，往往使用指针数组作为实际数据的索引。

排序前

hijklmnop
qrstuvwxyz
abcdefg

排序中(交互数组)

hijklmnop
qrstuvwxyz
abcdefg

直接对二维字符数组排序

排序前

•	hijklmnop
•	qrstuvwxyz
•	abcdefg

排序中(交换指针)

•	hijklmnop
•	qrstuvwxyz
•	abcdefg

利用指针数组排序



## 8.3.1 一维指针数组

- 指针数组常被用作**数据索引**，以加快数据**定位、查找、交换和排序**等操作的速度。
- 在一些文字处理程序中，数据一般以“行”为单位保存在二维数组中，在数据处理的过程中，对各行位置的交换，以及整行内容的删除和新行的添加是频繁进行的操作（**计算代价很大**）。为**提高程序的运行速度**，往往使用**指针数组**作为**实际数据的索引**。

排序前

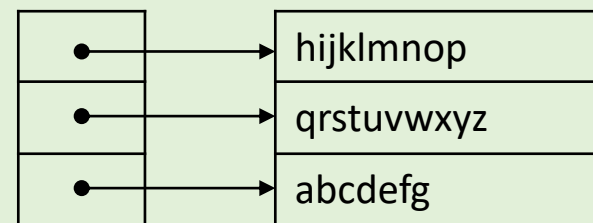
hijklmnop
qrstuvwxyz
abcdefg

排序后

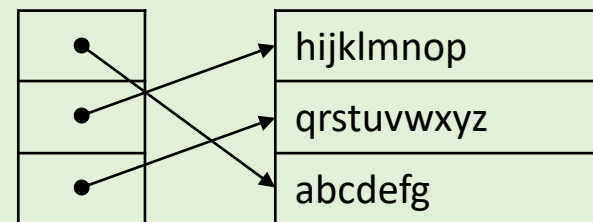
abcdefg
hijklmnop
qrstuvwxyz

直接对二维字符数组排序

排序前



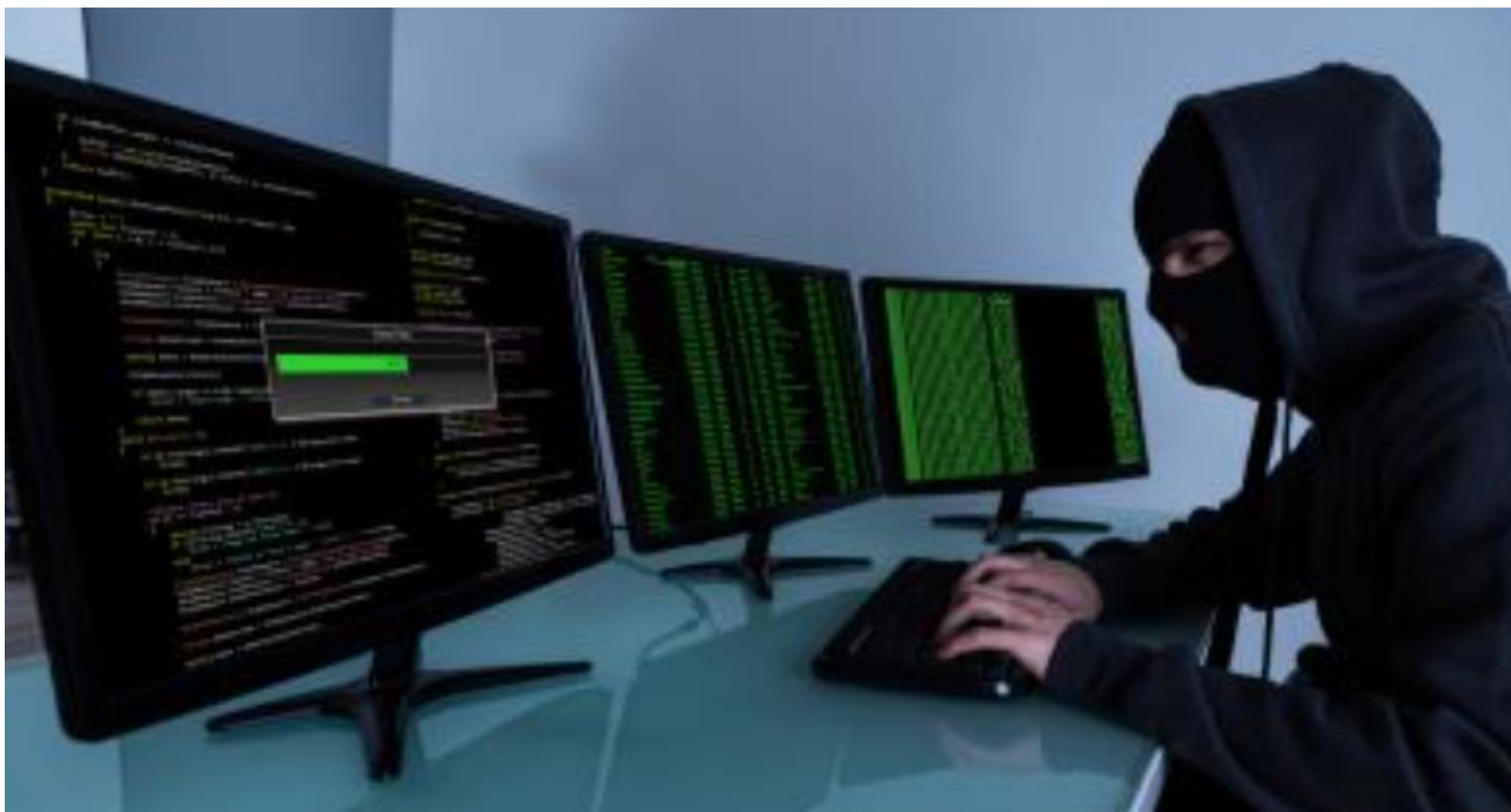
排序后



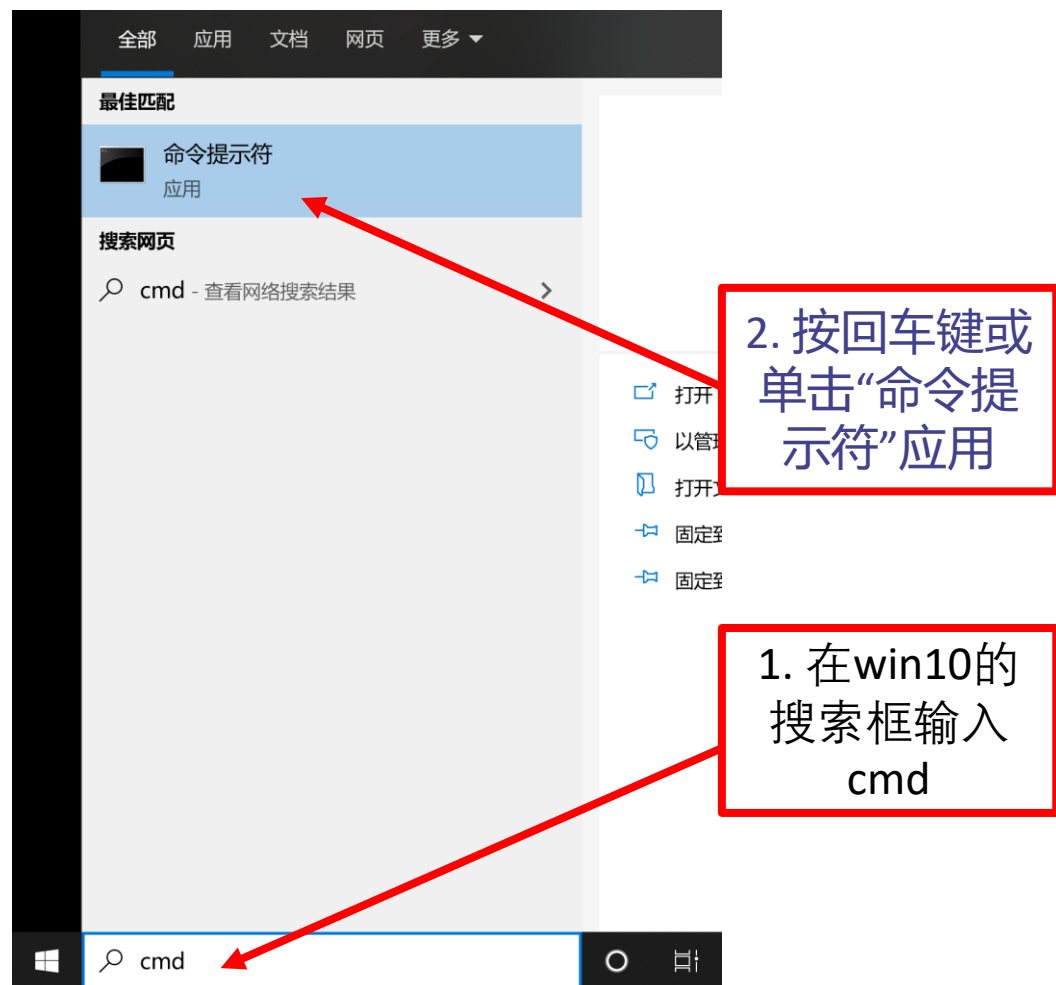
利用指针数组排序

## \*\* 8.3.2 命令行参数

高手都喜欢用命令行来操作电脑



## \*\*\* 8.3.2 命令行参数



命令提示符

```
Microsoft Windows [版本 10.0.18363.1198]  
(c) 2019 Microsoft Corporation. 保留所有权利。
```

```
C:\Users\17419>
```

命令提示符

```
Microsoft Windows [版本 10.0.18363.1198]  
(c) 2019 Microsoft Corporation. 保留所有权利。
```

C:\Users\17419>help

有关某个命令的详细信息，请键入 HELP 命令名

ASSOC	显示或修改文件扩展名关联。
ATTRIB	显示或更改文件属性。
BREAK	设置或清除扩展式 CTRL+C 检查。
BCDEDIT	设置启动数据库中的属性以控制启动加载。
CACLS	显示或修改文件的访问控制列表(ACL)。
CALL	从另一个批处理程序调用这一个。
CD	显示当前目录的名称或将其更改。
CHCP	显示或设置活动代码页数。
CHDIR	显示当前目录的名称或将其更改。
CHKDSK	检查磁盘并显示状态报告。
CHKNTFS	显示或修改启动时间磁盘检查。
CLS	清除屏幕。
CMD	打开另一个 Windows 命令解释程序窗口。
COLOR	设置默认控制台前景和背景颜色。
COMP	比较两个或两套文件的内容。

## **\*\* 8.3.2 命令行参数**

命令提示符

```
Microsoft Windows [版本 10.0.18363.1198]  
(c) 2019 Microsoft Corporation。保留所有权利。
```

```
C:\Users\17419>
```

命令提示符

```
C:\Users\17419>cd C:\alac\code
```

```
C:\alac\code>copy test.c sy.c  
已复制          1 个文件。
```

```
C:\alac\code>_
```

## \*\* 8.3.2 命令行参数

### Windows命令行示例

PING某IP主机 : ping 192.168.0.1

删除文件 : del D:\my.txt

使用反斜杠\, 不要使用正斜杠/

查看网卡配置 : ipconfig /all

关闭计算机 : shutdown /s /t 10

10s延时

### UNIX/Linux命令行示例

文件拷贝 : cp src\_file dest\_file

列出当前目录 : ls -l

-l: 列出当前目录下所有文件的详细信息

切换目录 : cd ~

查找文件 : find . -name "\*.c"

-name "\*.c":将目前目录及其子目录下所有延伸档名是 c 的文件列出来

## \*\* 8.3.2 命令行参数

在C语言中，当要编写具有命令行参数的程序时，程序中的main()函数需要使用如下的函数原型：

```
int main(int argc, char *argv[]); // => char **argv
```

假设运行程序program时在终端键盘上输入下列命令：

```
% program f1 6
```

在程序program中，argc的值等于3，argv[0]，argv[1]和argv[2]的内容分别是"program"，"f1"和"6"。

## \*\* 8.3.2 命令行参数

【例8-6】 计算命令行参数的代数和 对命令行中输入的若干个整数求和，在标准输出上输出结果。

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    int i, sum = 0, val;
    for (i = 1; i < argc; i++)
    {
        sscanf(argv[i], "%d", &val);
        sum += val;
    }
    printf("%d\n", sum);
    return 0;
}
```

对argv的下标遍历  
是从1开始的?

## \* 8.4 函数指针

- 指针函数: `char *strstr(char *s, char *s1);`

主语是函数，该函数返回一个指针

- 函数指针: `int (*f_name) (...);`

主语是指针，`f_name`是一个变量，指向一个返回`int`类型的函数

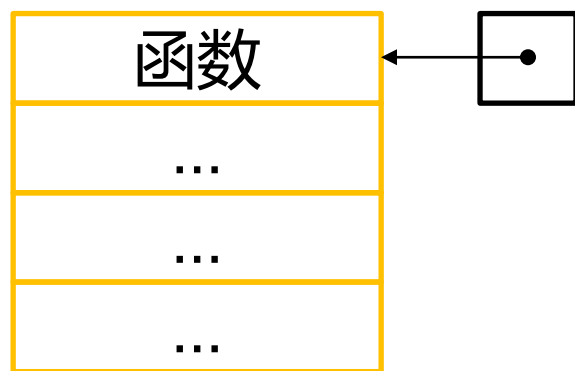
- 同样，“指针数组”【如，`char *a[N]`】与  
“数组指针”【如，`char (*a)[N]`】是同一个道理。



## \* 8.4 函数指针

函数指针: `int (*f_name) (...);`

函数名表示的是一个函数的可执行代码的入口地址，也就是指向该函数可执行代码的指针。函数指针类型为提高程序描述的能力提供了有力的手段，是实际编程中一种不可或缺的工具。



函数存储在内存里；  
内存有地址；  
函数有指针。

## 8.4.1 函数指针变量的定义

函数指针: `int (*f_name) (...);`

- 函数指针类型是一种泛称，其具体类型由函数原型确定。  
(参数个数、参数类型、返回值类型)
- 定义一个函数指针类型的变量需要按顺序说明下面这几件事：
  - 1) 说明指针变量的变量名；
  - 2) 说明这个变量是指针；
  - 3) 说明这个指针指向一个函数；
  - 4) 说明这个变量所指向函数的原型，包括参数表和函数的返回值类型。

## 8.4.1 函数指针变量的定义

**keywords:** 变量名、指针、指向函数、函数类型

```
double (*func) (double x, double y);  
// 等价于  
double (*func) (double, double);
```

定义一个函数指针变量:

<返回类型> (\*<标识符>) (<参数表>); // <标识符>应为一个合法的变量名

例如:

```
int (* funPtr) (int, int);  
void (* funPtr) (int, int, int);  
int (* funPtr) (double, char *);  
int * (* funPtr) ();
```

## 8.4.1 函数指针变量的定义

```
double (*func) (double x, double y); //定义一个函数指针
// vs
double sum(double x, double y);      //函数声明
```

```
double sum(double x, double y) {
    return x + y;
}                                     // sum函数定义
...
func = sum;      // 把函数sum赋值给func, func指向sum, 操作func即操作sum
s1 = (*func)(u, v); // 调用, 与sum(u, v)所调用的是同一个函数
s2 = func(u, v);    // 等价于(*func)(u, v)
```

为了方便起见, 在C语言中也允许将函数指针变量直接按函数调用的方式使用: `func(u, v)` 与 `(*func)(u, v)` 完全等价!

## 8.4.2 具有函数指针参数的库函数

一般函数的参数采用普通数值或指针，函数内部执行与参数类型相关的固定计算方法，对参数进行计算。

```
int add(int a, int b); // 普通数值  
int toupper(char* src, char* dst); // 指向变量或数组的指针
```

如何设计“动态”绑定的计算函数，实现动态计算方法？

将“动态”绑定的函数以参数形式传递给计算框架函数。

“静态”绑定：函数声明（编译）时就已经确定了；

“动态”绑定：函数运行时才能确定，且运行时可以变！

## 【例8-7】：使用函数指针作为参数的选择排序

```
void seSort(int [], int, int (*)(int, int) );
void swap( int *, int * );
int  ascending( int, int );
int  descending( int, int );

int main()
{
    int order; // 1 = ascending, 2 = descending
    int counter; // array index
    int a[N] = {2, 6, 4, 8, 10, 12, 89, 68, 45, 37};
    printf("Enter 1, sort in ascending order,\n");
    printf("Enter 2, sort in descending order:\n");
    scanf("%d", &order);
    printf("Data items in original order\n");
    for ( counter = 0; counter < N; counter++ )
        printf("%4d", a[ counter ]);
```

函数原型中变量名（包括函数指针名）可以省略。

函数的声明和作为参数使用

```
if ( 1 == order )
{
    seSort( a, N, ascending );
    printf("\nData in ascending order\n");
}
else
{
    seSort( a, N, descending );
    printf("\nData in descending order\n");
}

// output sorted array
for (counter = 0; counter < N; counter++)
    printf("%4d", a[ counter ]);
printf("\n");
return 0;
}
```

## 【例8-7】：使用函数指针作为参数的选择排序

函数框架：动态绑定（运行时绑定，确定compare的逻辑）

形参为带两个int参数，返回int的函数指针

静态绑定的函数，编译就确定

```
void seSort(int a[], int size, int (*compare)(int, int) )
{
    int smallOrLarge;
    int i, index;
    for ( i = 0; i < size - 1; i++ )
    {
        // first index of remaining vector
        smallOrLarge = i;

        for ( index = i + 1; index < size; index++ )
            if ( !(*compare)(a[smallOrLarge], a[index]) )
                smallOrLarge = index;
        swap(&a[smallOrLarge], &a[i]);
    }
}
```

```
void swap( int * element1Ptr,
           int * element2Ptr )
{
    int hold = *element1Ptr;
    *element1Ptr = *element2Ptr;
    *element2Ptr = hold;
}

int ascending( int a, int b )
{
    return a < b;
}

int descending( int a, int b )
{
    return a > b;
}
```

函数指针的调用，即调用传进来的函数体

选择排序：升序时，把最大的选出来，放最后；次大的，放倒数第二；以此类推。

## 【例8-7】：使用函数指针作为参数的选择排序

```
int main()
{
...
    if ( order == 1 )
    {
        seSort( a, N, ascending ); //运行时
    }
...
}
```

“运行时：程序执行到这里的时候”

```
void seSort(int a[], int size, int (*compare)(int, int))
{
...
    if ( !(*compare)(a[smallOrLarge], a[index]) )
...
}
```



相当于

```
void seSort(int a[], int size, int (*compare)(int, int))
{
...
    if ( !ascending(a[smallOrLarge], a[index]) )
...
}
```

- “静态”绑定：函数声明（编译）时就已经确定了；
- “动态”绑定：函数运行时才能确定，且运行时可以变！



## 8.4.2 具有函数指针参数的库函数

举例： qsort() 快速排序函数（标准库函数）

```
void qsort( void *base, size_t num, size_t wid, int (*comp)(const void *e1, const void *e2) );
```

base: 是指向所要排序的数组的指针（void\*指向任意类型的数组）；

num: 是数组中元素的个数；

wid: 是每个元素所占用的字节数；

comp: 是一个指向数组元素比较函数的指针，该比较函数的两个参数是类型位置的指针，const表示指针指向的内容是只读的，在comp所指向的函数中不可被修改。

qsort: 负责框架调用和给(\*comp)传递所需参数，根据(\*comp)的返回值决定如何移动数组；

(\*comp): 负责比较两个元素，返回负数、正数和0，分别表示第一个参数先于、后于和等于第二个参数

## 8.4.2 具有函数指针参数的库函数

**【例8-8】使用qsort()对一维double数组排序** 给定一个所有元素均已被赋值的double型数组，使用qsort()对数组元素按升序和降序排序。

qsort 怎么实现的？用户看不到（不透明），是用快速排序实现。

前面选择排序seSort的框架跟这个原理相似，但选择排序“透明”。

```
int rise_double(const void *p1, const void *p2)
{
    if ( *(double *)p1 < *(double *)p2 ) return -1;
    if ( *(double *)p1 > *(double *)p2 ) return 1;
    if ( *(double *)p1 == *(double *)p2 ) return 0;
}

int fall_double(const double *p1, const double *p2)
{
    if ( *p1 > *p2 ) return -1;
    if ( *p1 < *p2 ) return 1;
    if ( *p1 == *p2 ) return 0;
}

double a[N_ITEMS];
...
// 按升序排序
qsort(a, N_ITEMS, sizeof(double), rise_double);

// 按降序排序
qsort(a, N_ITEMS, sizeof(double), fall_double);
```

## 8.4.2 具有函数指针参数的库函数

【例8-8】使用qsort()对一维double数组排序 给定一个所有元素均已被赋值的double型数组，使用qsort()对数组元素按升序和降序排序。

```
int rise_double(const void *p1, const void *p2)
{
    if ( *(double *)p1 < *(double *)p2 ) return -1;
    if ( *(double *)p1 > *(double *)p2 ) return 1;
    if ( *(double *)p1 == *(double *)p2 ) return 0;
}

int fall_double(const double *p1, const double *p2)
{
    if ( *p1 > *p2 ) return -1;
    if ( *p1 < *p2 ) return 1;
    if ( *p1 == *p2 ) return 0;
}
```

```
int rise_double(const void *p1, const void *p2)
{
    return (int)(*(double *)p1 - *(double *)p2);
}
```

如上：书上的代码，若 \*p1 - \*p2 的结果为0.5或-0.5时，都返回0，跟希望的结果不同。会出问题。

如左：fall\_double函数的参数，这种用法有的编译器可能会warning，最好都按rise\_double函数那样，用const void \*

rise\_double() 的参数为通用类型指针const void\*，在函数内部需要进行强制类型转换。=> 可匹配任意类型指针  
fall\_double() 的参数直接定义为const double\*，在函数内部的避免参数类型转换。=> 描述上更加简洁（更“严格”的编译器会warning）  
在C语言中，两种方法都可以。

# 8.4.2 具有函数指针参数的库函数

应用：qsort()对二维数组按行排序，即，把二维数组看成按行组成的一维数组

【例8-9】输出数据的编号（顺序统计量）。从标准输入上读入  $n$  ( $1 < n < 200000$ ) 个整数，将其按数值从小到大连续编号（**第i小**），相同的数值具有相同的编号。在标准输出上按输入顺序以<编号>: <数值>的格式输出这些数据，各数据之间以空格符分隔，以换行符结束。

输入样例 5 3 4 7 3 5 6      输出样例 3:5 1:3 2:4 5:7 1:3 3:5 4:6

5
3
4
7
3
5
6

输入

5	3
3	1
4	2
7	5
3	1
5	3
6	4

编号

编号如何求?

- 算法:
- 1. 读入数据并记录读入顺序;
  - 2. 对数据按大小排序后编号;
  - 3. 再对数据按输入顺序排序;
  - 4. 按顺序输出编号及其数据。

N行3列数组

5	1	3
3	2	1
4	3	2
7	4	5
3	5	1
5	6	3
6	7	4

输入      输入顺序      数据编号  
                输出顺序      (即第几小)

# 8.4.2 具有函数指针参数的库函数

【例8-9】输出数据的编号（顺序统计量）。从标准输入上读入  $n$  ( $1 < n < 200000$ ) 个整数，将其按数值从小到大连续编号（**第  $i$  小**），相同的数值具有相同的编号。在标准输出上按输入顺序以<编号>: <数值>的格式输出这些数据，各数据之间以空格符分隔，以换行符结束。

输入样例 5 3 4 7 3 5 6      输出样例 3:5 1:3 2:4 5:7 1:3 3:5 4:6

(1) 读入数据并记录顺序

5	1	
3	2	
4	3	
7	4	
3	5	
5	6	
6	7	

输入    输入    数据  
数值    顺序    编号

(2) 按数值大小排序后编号

3	2	1
3	5	1
4	3	2
5	1	3
5	6	3
6	7	4
7	4	5

输入    输入    数据  
数值    顺序    编号

(3) 按输入顺序再排序

5	1	3
3	2	1
4	3	2
7	4	5
3	5	1
5	6	3
6	7	4

输入    输入    数据  
数值    顺序    编号

➡ (4) 按输入顺序输出编号及其数值

```
int data[MAX_N][3];
int main()
{
    int i, n;
    for(n=0; scanf("%d", &data[n][0])!=EOF; n++)
        data[n][1] = n; // 存数据的输入顺序
    qsort(data, n, sizeof(data[0]), s_rank);
    gen_rank(data, n);
    qsort(data, n, sizeof(data[0]), s_order);
    for (i = 0; i < n; i++)
    {
        if (i != 0) // 首个输出数值前没有空格
            putchar(' ');
        printf("%d:%d", data[i][2], data[i][0]);
    }
    return 0;
}
```

输入数据

```
int s_rank(const int *p1, const int *p2)
{
    return p1[0] - p2[0]; // 第一列元素比较
}
```

这是书上的代码，直接用这两个函数可能会报错！为什么？

```
int s_order(const int *p1, const int *p2)
{
    return p1[1] - p2[1]; // 第二列元素比较
}
```

第一个qsort()对data中的数据按值大小排序

第二个qsort()对data中的数据按输入顺序排序

qsort执行过程中，s\_rank()中的两个参数p1和p2分别指向data的两组相邻元素（每组数据是一行，3个数），根据两组数据中对应位置（这里是第一列）的两个元素的大小关系决定如何排序。

```

int data[MAX_N][3];
int main()
{
    int i, n;
    for(n=0; scanf("%d", &data[n][0])!=EOF; n++)
        data[n][1] = n; // 存数据的输入顺序
    qsort(data, n, sizeof(data[0]), s_rank);
    gen_rank(data, n);
    qsort(data, n, sizeof(data[0]), s_order);
    for (i = 0; i < n; i++)
    {
        if (i != 0) // 首个输出数值前没有空格
            putchar(' ');
        printf("%d:%d", data[i][2], data[i][0]);
    }
    return 0;
}

```

输入数据

```

int s_rank(const int *p1, const int *p2)
{
    return p1[0] - p2[0]; // 第一列元素比较
}

```

这是书上的代码，直接用这两个函数可能会报错！为什么？

qsort要求参数为void（参考例8-8的用法）；相减还可能造成数据越界。

```

int s_order(const int *p1, const int *p2)
{
    return p1[1] - p2[1]; // 第二列元素比较
}

```

第一个qsort()对data中的数据按值大小排序

第二个qsort()对data中的数据按输入顺序排序

qsort执行过程中，s\_rank()中的两个参数p1和p2分别指向data的两组相邻元素（每组数据是一行，3个数），根据两组数据中对应位置（这里是第一列）的两个元素的大小关系决定如何排序。

## 8.4.2 具有函数指针参数的库函数

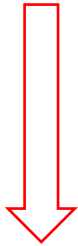
gen\_rank()的功能是给data中的数据按大小编号（填第三列）

```
void gen_rank(int data[][3], int n)
{
    int i;
    data[0][2] = 1;
    for (i = 1; i < n; i++)
        if (data[i][0] == data[i - 1][0])
            data[i][2] = data[i - 1][2];
        else
            data[i][2] = data[i - 1][2] + 1;
}
```

int data[][3]

3	2	1	直接给1
3	5	1	若相等，连续排
4	3	2	若不等，则加一
5	1	3	
5	6	3	
6	7	4	
7	4	5	

数值      输入      数据  
             顺序      编号





## 8.4.2 具有函数指针参数的库函数

### \*\*\* 举例： bsearch()二分查找函数（标准库函数）

```
void *bsearch (const void *key, const void *base, size_t num,  
              size_t wid, int (*comp) (const void *e1, const void *e2));
```

key: 指向待查数据的指针;

base: 指向所要查找的数组的指针;

num: 数组中元素的个数;

wid: 每一个元素所占用的字节数;

comp: 一个指向比较函数的指针;

e1: 指向key;

e2: 指向当前正在检查的数组元素。

当 base 所指向的数组中有与 key 所指向的数据的属性一致的元素时，bsearch() 返回该元素的地址，否则返回NULL。

## 实例分析：判断质数

**\*\* 【[例8-10] ] 查质数表。** 给定一个按升序排列的包含N个质数的指数表，通过查表判断一个正整数是否是质数。

```
int n;  
int primes[N]; //质数表  
init_primes(primes, N); //质数表初始化, 自行定义  
scanf("%d", &n);  
if (bsearch(&n, primes, N, sizeof(int), comp_init) != NULL)  
    printf("%d is a prime\n", n);  
else  
    printf("%d is not a prime\n", n);
```

```
int comp_int(const int *p1, const int *p2)  
{  
    return *p1 - *p2;  
}
```

注意溢出问题

待查元  
素指针

查找  
数组

数组  
大小

元素  
大小

比较  
函数

# 实例分析：判断质数

最容易想到的求质数算法

```
int isPrime (int n)  // n为正整数
{
    if (n == 1)
        return 0;
    for(int i = 2; i <= sqrt(n); i++)
    {
        if(n % i == 0)
            return 0;
    }
    return 1;
}
```

- 从2到 $\sqrt{n}$ 遍历，step 为 1，查所有数。
- 可以从3开始，step 为 2时，不查偶数，则会快一倍！
- 还可以再快些？

存在的问题：

1. 大量的遍历
2.  $\sqrt{n}$ 函数计算慢且不精确

# 实例分析：判断质数

## 改进的质数判断函数和高效质数表初始化方法

```
int isPrime(int primes[], int n)
{
    int i;
    for(i=0; primes[i]*primes[i] <= n; i++)
    {
        if (n % primes[i] == 0)
            return 0;
    }
    return 1;
}
```

判断  $n$  是否为质数

用  $\text{int} * \text{int}$  对比  $\text{sqrt}$ ，快且准！

利用已生成的质数表，减少大量遍历

质数表 `primes[]` 如何生成？

定理：数  $n$  若不能被  $\leq \text{sqrt}(n)$  的所有质数整除，则  $n$  必为质数。

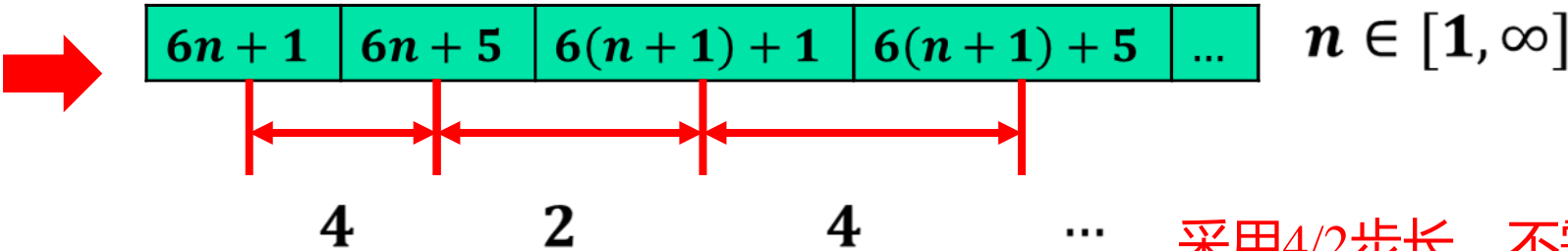
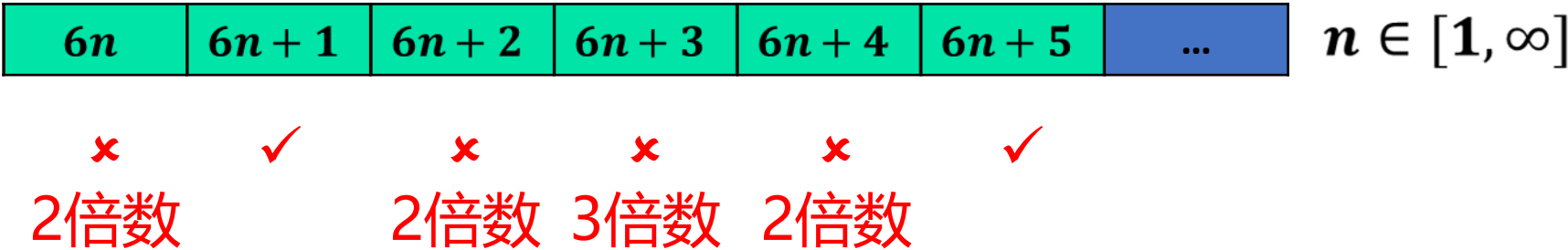
证明：用反证法

- ① 先假设  $n$  不能被  $\leq \text{sqrt}(n)$  的质数整除，且为合数，它必能分解为一个质数与另一个数相乘。
- ② 故，假设  $n = a \times p$ ， $p$  为质数，且  $p$  必须大于  $\text{sqrt}(n)$ 。那么  $a < \text{sqrt}(n)$ ，并且  $a$  不能是质数，否则就跟①矛盾。  
 $a$  是合数可分解：令  $a = b \times q$ ，这里  $q$  是质数，且  $q < \text{sqrt}(n)$ 。
- ③ 所以： $n$  能被小于  $\text{sqrt}(n)$  的质数  $q$  整除！与①假设矛盾！  
所以，若  $n$  不能被  $\text{sqrt}(n)$  的质数整除的话， $n$  必为质数！ 证毕！

# 实例分析：判断质数

改进的质数  
判断函数和  
高效质数表  
初始化方法

$$\{6, 7, 8, 9, \dots, \infty\} \Leftrightarrow \bigcap_{n=1}^{+\infty} \{6n, 6n+1, 6n+2, 6n+3, 6n+4, 6n+5\}$$



➡  $n = 1 \Rightarrow 6n + 1 = 7(\text{start})$   
 $STEPS = \{4, 2, 4, 2, \dots\}$

采用4/2步长，不需要在isPrime里模2, 3!  
快多少?

# 实例分析：判断质数

## 改进的质数判断函数和高效质数表初始化方法

```
void init_primes(int primes[], int Q) //构造 $\leq Q$ 的质数表 ( $Q \geq 5$ )
{
    int count=3, num, step;
    primes[0] = 2; primes[1] = 3; primes[2] = 5; //头3个质数
    num = 7; step = 2; //初始为2
    while(count < Q)
    {
        step = 6 - step; // 构造 4-2-4-2-...序列, 减少遍历
        if (isPrime(primes, num))
            primes[count++] = num;
        num += step; // 下一个可能的质数
    }
}
```

头3个质数直接给

只需要检查 $6n+1$ 与 $6n+5$ ;  
num=7, 11, 13, 17, 19  
...  
即4-2-4-2...序列

质数表: 2 3 5 7 11 13 17 19 23 25 29 31 35 37 41 43 47 49 51 55 57 61 ...

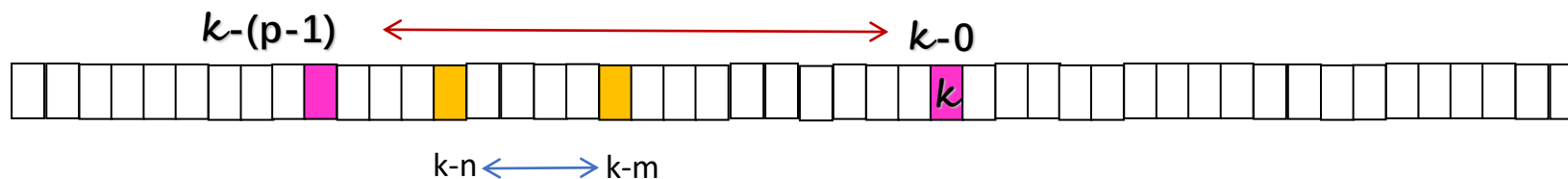
## \*\* [最后一个有趣的例子]母牛的数量 (方法1~8)

一头 $x$ 年出生的母牛从 $x+m$ 年到 $x+n$ 年间每年生出一头母牛，并在 $x+p$ 年被淘汰。写一个程序，从标准输入上按顺序读入整数 $m, n, p, k$  ( $3 < m < n < p < 60, 0 < k < 60$ )，设第0年有一头刚出生的母牛，计算第 $k$ 年时共存有多少头未被淘汰的母牛。

题解分析：

- 第 $k$ 年母牛的**总数量** $T(k)$ 为第 $k-(p-1)$ 年到第 $k$ 年新出生母牛数量之和（不超过 $p$ 岁）（超过 $p$ 岁的，即 $k-p$ 年及以前出生的在第 $k$ 年时都死了）。

$$T(k) = N(k-(p-1)) + N(k-(p-2)) + \dots + N(k-0) \text{ ----- (1)}$$



- 第 $k$ 年**新生母牛** $N(k)$ 等于 $k-m$ 年到 $k-n$ 年出生母牛数量之和（即这期间出生的母牛在第 $k$ 年有生产能力）（ $k-n$ 年前出生的牛太老了，不能再生产， $k-m$ 年后出生的牛太小，还不能生产）

$$N(k) = N(k-n) + N(k-(n-1)) + \dots + N(k-m) \text{ ----- (2)}$$

基于(1)和(2)，可写递归程序。

基本情况： $T(x)=1$  ( $x < m$ );  $N(m) = 1, N(x)=0$ , 当  $0 \leq x < m$ .

## \*\*母牛的数量

```
#include<stdio.h>

int m, n, p, k;

int Tcows(int k);
int Ncows(int k);
int Dcows(int k); // 第k年死亡的牛

void main()
{
    scanf("%d%d%d%d", &m, &n, &p, &k);
    printf("%d", Tcows(k));
}

int Dcows(int k)
{
    if( k < p ) return 0;
    else return Ncows(k - p);
}
```

```
int Tcows(int k) {
    if (k < 0)
        return 0;
    else if (k < m)
        return 1;
    else
        return Tcows(k-1) + Ncows(k) - Dcows(k);
}

int Ncows(int k) {
    int i, cows;
    if( k == 0 ) return 1;
    else if( k < m ) return 0;
    else {
        cows = 0;
        for(i = m; i <= n; i++) cows += Ncows(k-i);
        return cows;
    }
}
```



## \*\*母牛的数量

```
// 母牛数量计算
// 书上的实现
#include <stdio.h>
#define N 64
int sum_prev(int *cows, int m, int n);
int cows(int m, int n, int p, int k);

int main()
{
    int m, n, p, k;
    scanf("%d%d%d%d",&m,&n,&p,&k);
    printf("%d ",cows(m, n, p, k));
    return 0;
}
```

```
int sum_prev(int *cows, int m, int n)
{
    int i, s = 0;
    for(i = m; i <= n; i++)
        s += cows[-i];
    return s;
}

int cows(int m, int n, int p, int k)
{
    int cow_buf[N * 2] = {0}, i, *new_cow;

    new_cow = &cow_buf[N];
    new_cow[0] = 1;
    for(i = 1; i <= k; i++) {
        new_cow[i] = sum_prev(&new_cow[i], m, n);
        printf("%d:%d\n",i,new_cow[i]);
    }

    return sum_prev(&new_cow[k], 0, p - 1);
}
```

## 小结

---

- 掌握二维数组在内存中的存放方式
- 理解二维数组的行指针和列指针
- 理解数组作为函数参数其实就是指针做参数
- 多重指针的概念与应用
- 掌握指针数组的概念和用法
- 理解一维指针数组与二维数组的区别
- 理解函数指针的定义与使用方法
- 掌握qsort()和bsearch()函数的使用方法

# 补充：再论指针与数组

## 课堂练习

```
#include <stdio.h>
int main()
{
    int a[16]= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16};
    int *p, *q, *r;
    p = a;
    q = &a[4];
    r = &a[7];
    printf("%d, %d, %d", p[4], q[0], r[2]);
    return 0;
}
```

当指针指向一个数组时，他们在很多行为和操作上表现得都是相同和一致的。经常会被混用！

程序输出结果？

# 补充：指针变量与数组的区别！

- 数组名不是变量！
  - 数组分配完成后，数组名就是固定的，地址也是固定的
  - 绝对不能把数组名当作变量来进行处理（对数组名赋值等）
- 数组是开辟一块连续的内存空间，数组名代表整个数组
- 指针变量（无论指向任何类型）通常是一个占4B的整数（or 8B，实际大小取决于计算环境，如操作系统、编译器的仿真环境等）

```
char *p;  
char ch[10];  
p = ch; //不要写成p = &ch这样的操作?  
printf("%d\n", sizeof(p)); //输出?  
printf("%d\n", sizeof(*p)); //输出?
```

更多思考： p+1 is ?

4

1

前者（可能为8）是它本身占多少字节，后者是指向的单元占多少字节

&ch 表示是否合法？

```
char (*q) [10];  
char ch[10];  
q = &ch;
```



q+1 is ?

# 补充：关于数组名在应用时的说明

```
#include <stdio.h>
int main()
{
    int a[10] = {1};
    int x, y, xp, yp;


    x = (int)(a);
    y = (int)(a + 1);
    xp = (int>(&a);
    yp = (int)((&a) + 1);

    printf("a = %d, &a = %d\n", a, &a);
    printf("%d + 1 = %d\n", x, y);
    printf("%d + 1 = %d\n", xp, yp);

    return 0;
}
```

**a** 是数组名是个指针常量，是数组第一个元素的地址

**&a** 产生的是一个指向数组的指针，而不是一个指向某个指针常量的指针！



```
a = 6487520, &a = 6487520
6487520 + 1 = 6487524
6487520 + 1 = 6487560
```

请解释输出的含义

## 补充：再论指针数组与数组指针

//指针数组

```
char *aPtr[5]={ "123" , "1234" , "12345" };
```

// aPtr是数组名，是数组 {aPtr[0], aPtr[1], aPtr[2]... }的首地址。

//数组指针

```
char ch[10];
```

```
char (*cPtr)[10];
```

// cPtr是指针变量，指向一个包含10个字符元素的数组。

```
cPtr = &ch;
```

```
printf("%d, %d, %d, %d\n", aPtr[0], aPtr[1], aPtr[2], aPtr[3]);
```

```
printf("%d, %d, %d", cPtr[0], cPtr[1], cPtr[2]);
```

程序输出结果（请解释输出的含义）：

4227108, 4227112, 4227117, 0

6422238, 6422248, 6422258

## 补充：回文字符串的一个例子

```
#include <stdio.h>
#include <string.h>
int isPalindrome(char *, int);
char aLine[2<<29];
int main() {
    int n;
    char *p=aLine;

    fgets(p, 2<<29, stdin);
    n = strlen(p);          //字符串长度, 包括'\n'
    if('\n'==*(p+n-1)) n--;

    if(isPalindrome(p, n))
        printf("A palindrome!\n");
    else
        printf("It's not a palindrome.!\n");

    return 0;
}
```

回文串就是一个正读和反读都一样的字符串,比如 “level” 或者 “noon” 等等就是回文串。

```
int isPalindrome(char *p, int n)
{
    int i, nhalf;
    nhalf = n>>1;

    for(i=0; i<nhalf; i++)
        if(*(p+i) != *(p+n-1-i))
            return 0;

    return 1;
}
```

## 补充：字符与字符串

```
#include <stdio.h>
int main()
{
    printf("%d\n", "a");
    printf("%d\n", 'a');
    return 0;
}
```

"a"返回的是分配的地址，  
'a'返回的是ASCII码值。

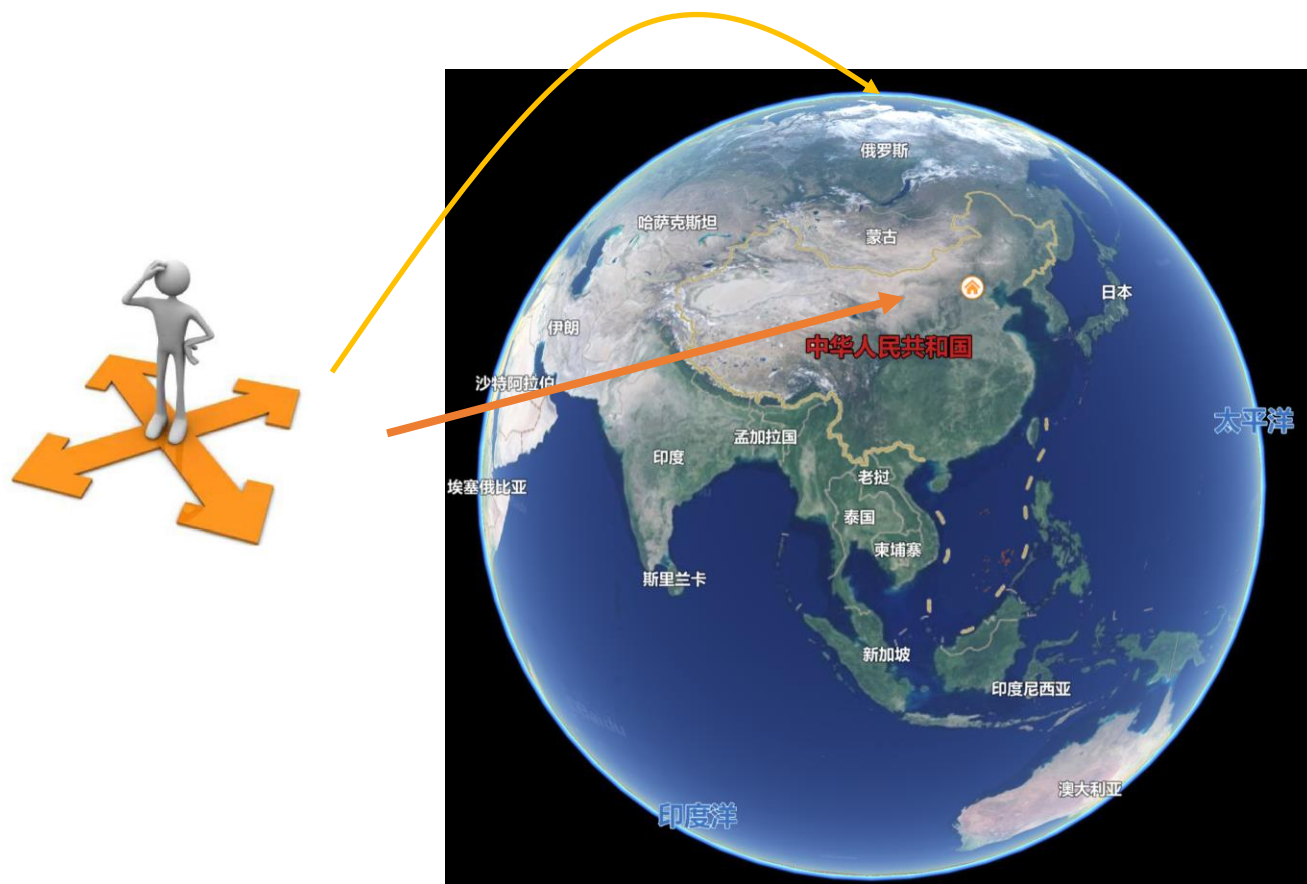
字符是小整数，字符串是  
大整数（地址）

4206628  
97



更多补充读物：指针与数组的进一步认识

## 指针本质探寻



本课完

