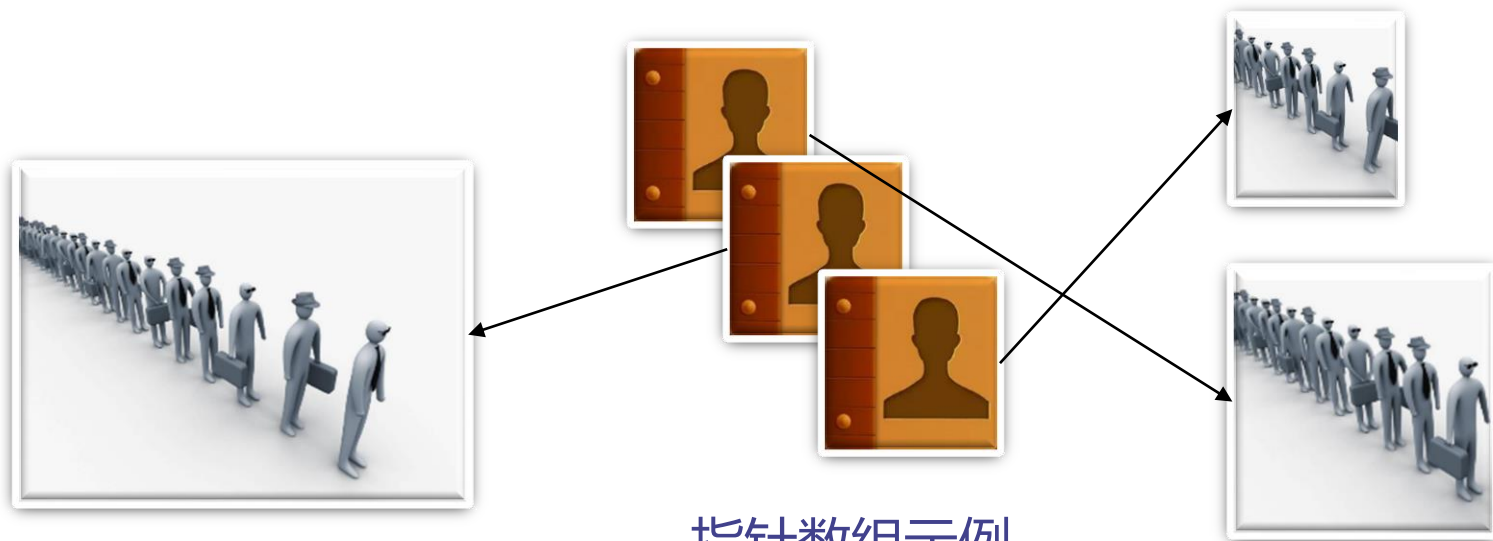


# C语言高级篇

## 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

指针数组, 数组指针, 傻傻分不清楚

指针函数, 函数指针, 傻傻分不清楚

数组指针, 二维数组, 傻傻分不清楚

数组形式参数, 指针形式参数, 傻傻分不清楚

心中有类型，指针如有神。初学者往往觉得指针很难，那只是人们心中的成见。

# 简单回顾

- 指针存储的是 数据实体的 地址

- ◆ 指针类型是一种数据类型，是从其它类型派生的类型：`xxx *` (`xxx`是某种数据类型)
- ◆ 指针类型的变量，也简称指针

- 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符（一元）
- ◆ `*`是解引用运算（一元）
- ◆ 函数参数的指针和返回指针的函数

- 指针与数组

- ◆ 数组名和指针的关系
- ◆ `char*`指针与字符串的关系
- ◆ `char*`指针 与 `char`型数组的关系

## 8.1 数组类型与数组指针

数组名和指针虽然在使用上很相似，尤其是数组名出现在表达式中几乎可以等同于指针来使用，但数组和指针却是**完全不同的数据类型**。

- C语言中的数组类型是相对与诸如int, double, float等单一类型而言的，数组类型是单一类型的聚合体。
- 数组类型由<元素类型> [<数组长度>]来描述。
- 例如int a[10], float b[20], char c[30]这三个数组的类型分别为int[10], float[20]和char[30]。
- 对于数组类型的变量（即我们常说的‘数组’，将数组看成数组类型的变量），对它用sizeof运算符返回的是整个数组所占内存空间的字节数。

## 8.1 数组类型与数组指针

数组名和指针虽然在使用上很相似，尤其是数组名出现在表达式中几乎可以等同于指针来使用，但数组和指针却是**完全不同的数据类型**。

```
1  #include<stdio.h>
2  int main()
3  {
4      double d_arr[] = { 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0};
5      double *pd = d_arr;
6      printf("sizeof d_arr is %d\n", sizeof(d_arr));
7      printf("sizeof pd is %d\n", sizeof(pd));
8      return 0;
9  }
```

sizeof d\_arr is 64  
sizeof pd is 8

## 8.1 数组类型与数组指针

数组既然是数组类型的数据实体，当然可以用取址符&取地址。

数组的地址，是它所占内存空间的起始地址，在数值上等于它首元素的地址。

数组的地址类型为指向数组的指针，简称数组指针。

数组指针变量的定义如下：

<类型> (\* <变量名>) [<元素个数>];

错误举例

正确示范

```
1  int a[10];
2  float b[20];
3  char c[30];
4  int(*pa)[10] = &a;
5  float(*pb)[20] = &b;
6  char(*pc)[30] = &c;
```

```
1  int a[10];
2  int (*pa1)[20] = &a; // &a的类型是 int(*)[10]，左边的类型是int(*)[20]，长度不匹配
3  float (*pa2)[10] = &a; //左边的类型是float(*)[10]，左右数组类型不匹配
4  int (*pa3)[10] = a; //左边的类型是int(*)[10]，右边数组名隐式类型转换为int*，左右类型不匹配
```

## 8.1 数组类型与数组指针

数组指针加1，实际地址变动是几？

```
1  #include<stdio.h>
2  int main()
3  {
4      int a[10];
5      int(*pa)[10] = &a;
6      printf("%p-%p = %d\n", pa + 1, pa, (void *)(pa + 1) - (void *)pa);
7      return 0;
8  }
```



0000000000061FE48-0000000000061FE20 = 40

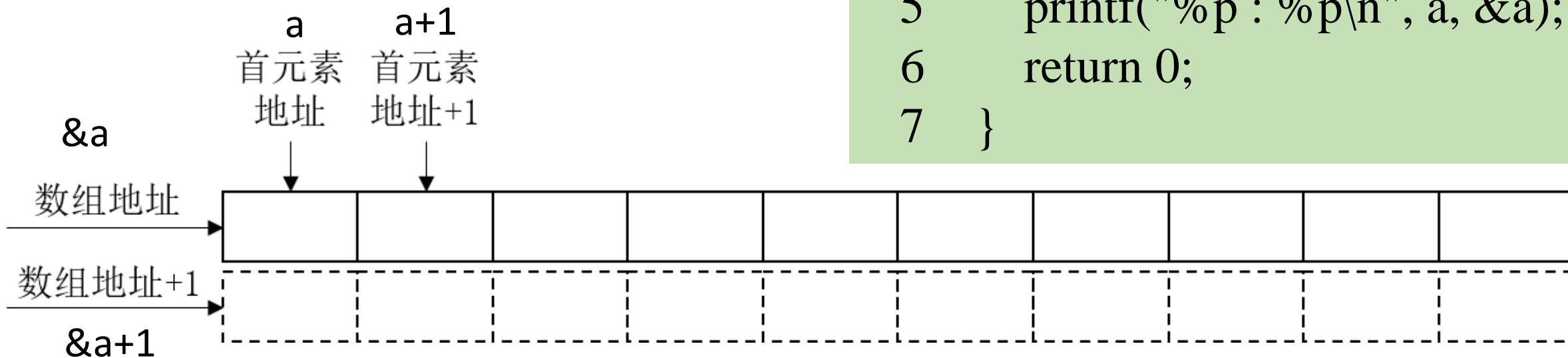
## 8.1 数组类型与数组指针

再看一段有意思的代码，便于我们深入理解数组的地址：

0000000000061FE20 : 0000000000061FE20

什么？a的值和a的地址竟然一样！

```
1  #include<stdio.h>
2  int main()
3  {
4      float a[10];
5      printf("%p : %p\n", a, &a);
6      return 0;
7  }
```





## 8.1 数组类型与数组指针

数组指针解引用后，代表它所指向的数组变量，**可以看作数组名**，用作表达式时和数组名一样也可以隐式类型转换为指向数组首元素的指针，例如：

```
1  #include<stdio.h>
2  int main()
3  {
4      double d[10] = {0.0};
5      double(*pd)[10] = &d;
6      double *p = *pd;
7      *p = 1.2;
8      printf("%f \n", d[0]);
9      return 0;
10 }
```

第4行，定义了一个double[10]数组d。

第5行，将数组d的地址赋值给数组指针pd。

第6行，对pd进行解引用，\*pd等价与数组d，作为表达式时隐式类型转换为指向数组d首元素的指针，即具有double\*类型，并赋值给左边的同类型指针p。此时p指向数组d的第一个元素。

第7行，对p进行解引用，间接修改了数组d的第一个元素值。

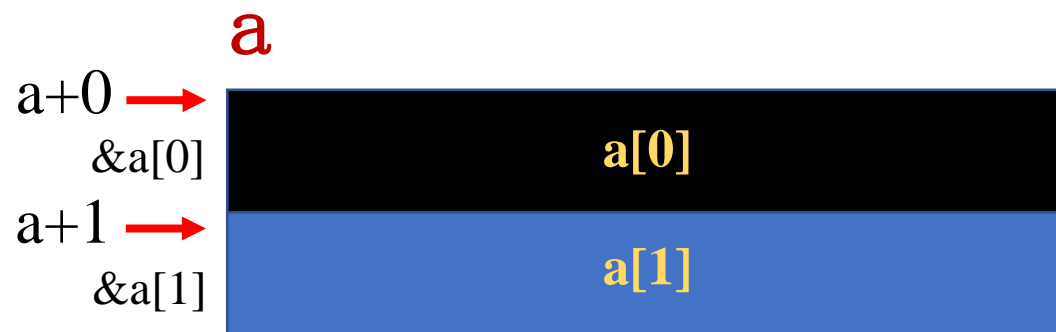
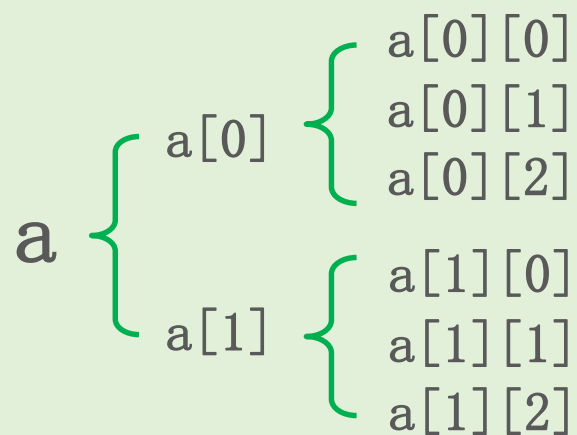
程序运行后屏幕输出：

1.200000

## 8.2 二维数组与数组指针

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。

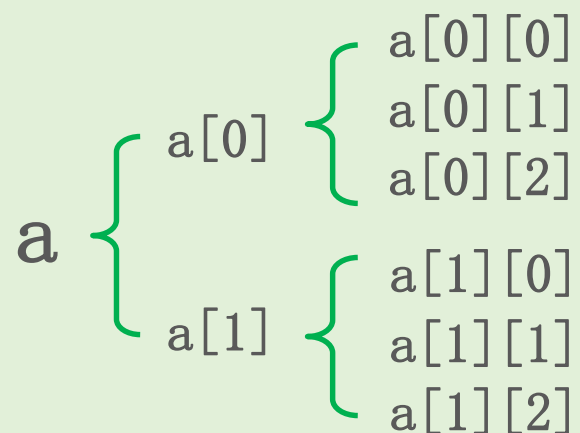
```
int a[2][3];
```



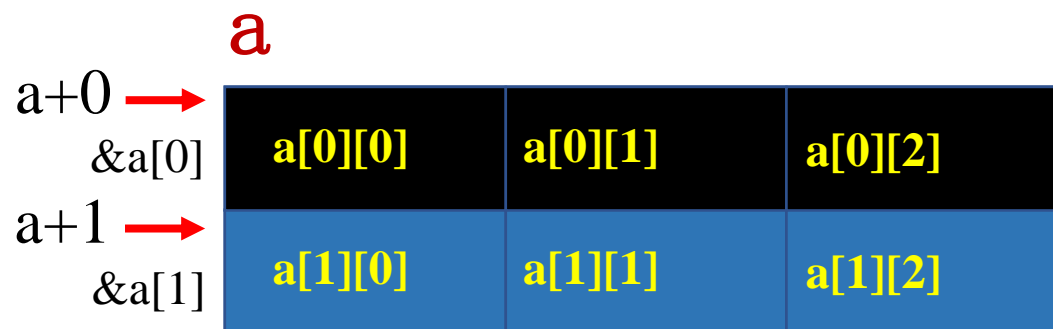
## 8.2 二维数组与数组指针

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）。

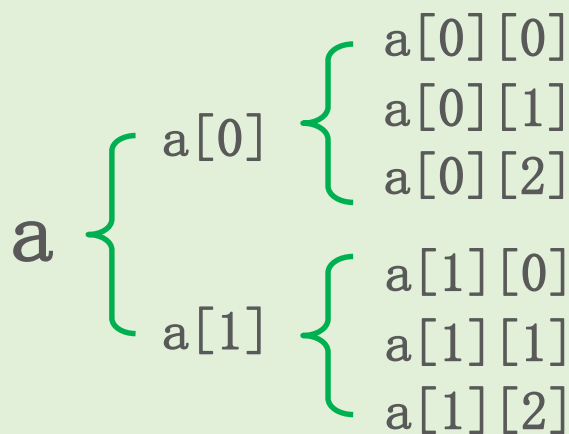


逻辑上，可看成2行3列，共6个元素。

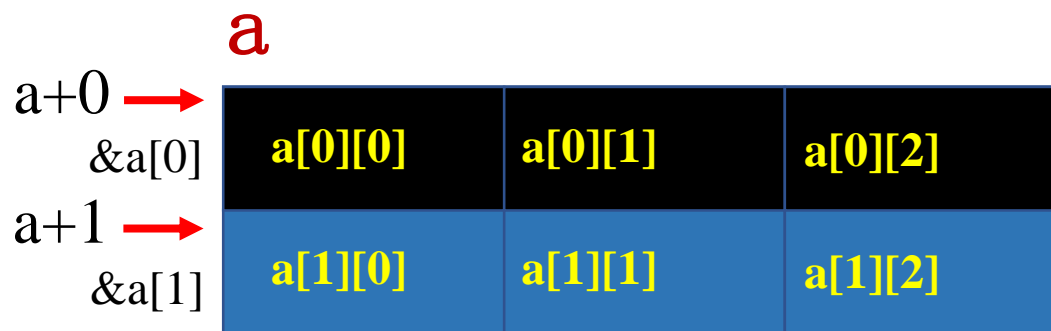
## 8.2 二维数组与数组指针

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。

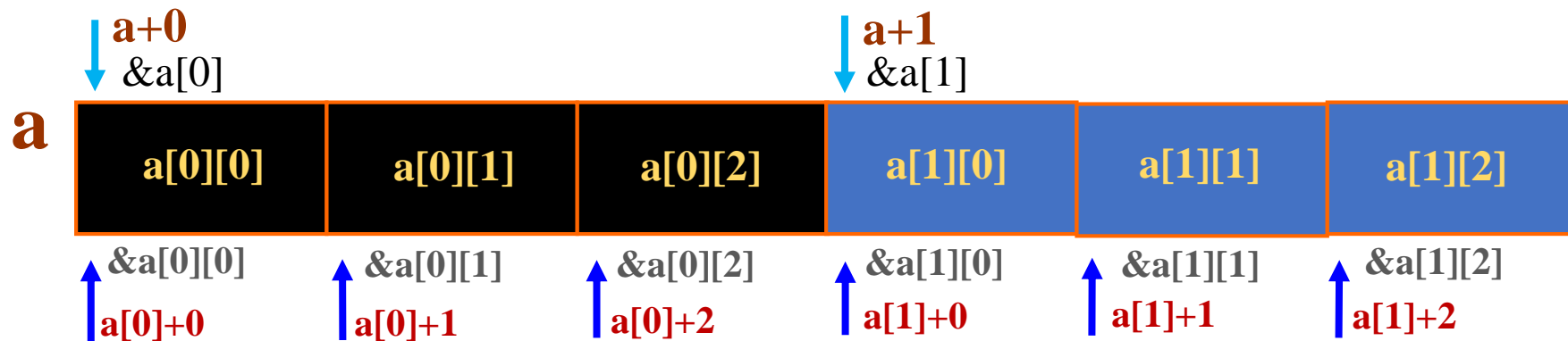
```
int a[2][3];
```



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）。

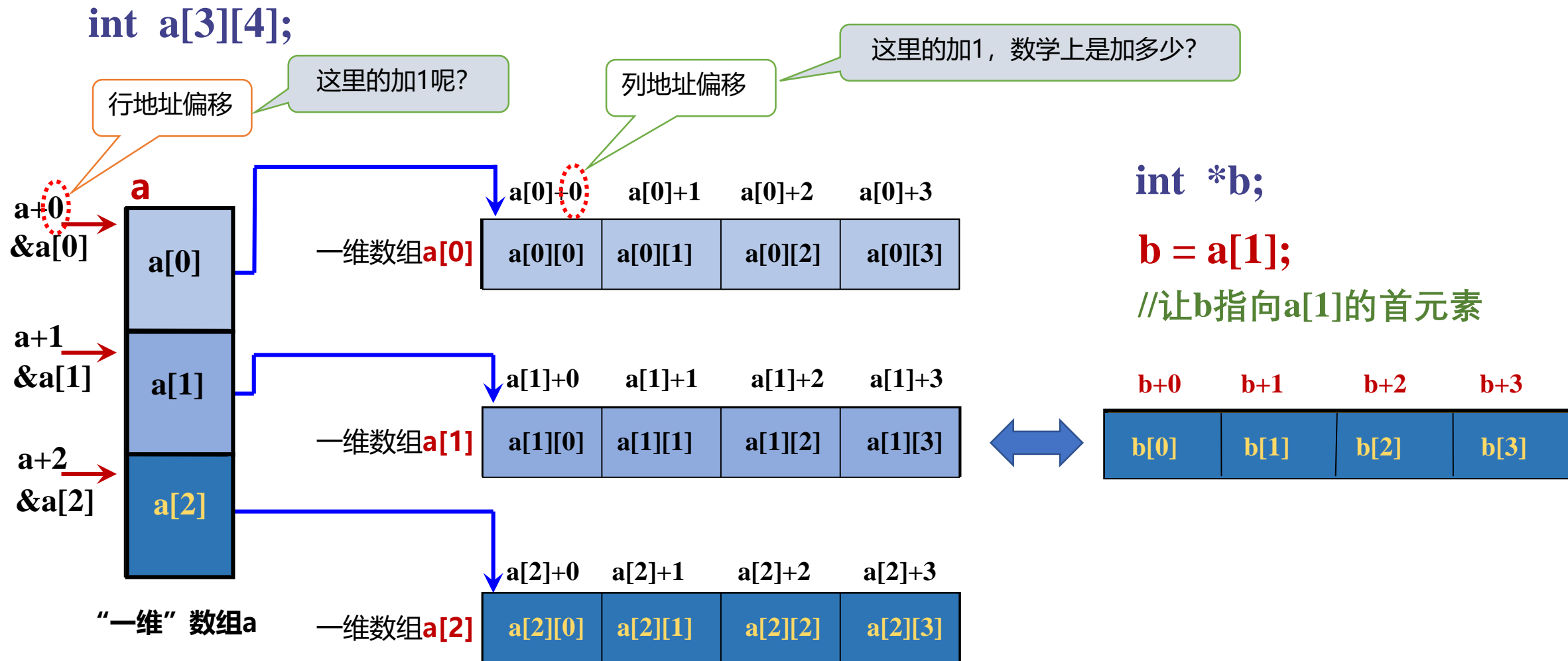


逻辑上，可看成2行3列，共6个元素。



物理上，其实就是在内存中连续存放的6个元素。

## 8.2 二维数组与数组指针



## 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
...									
...									
...									

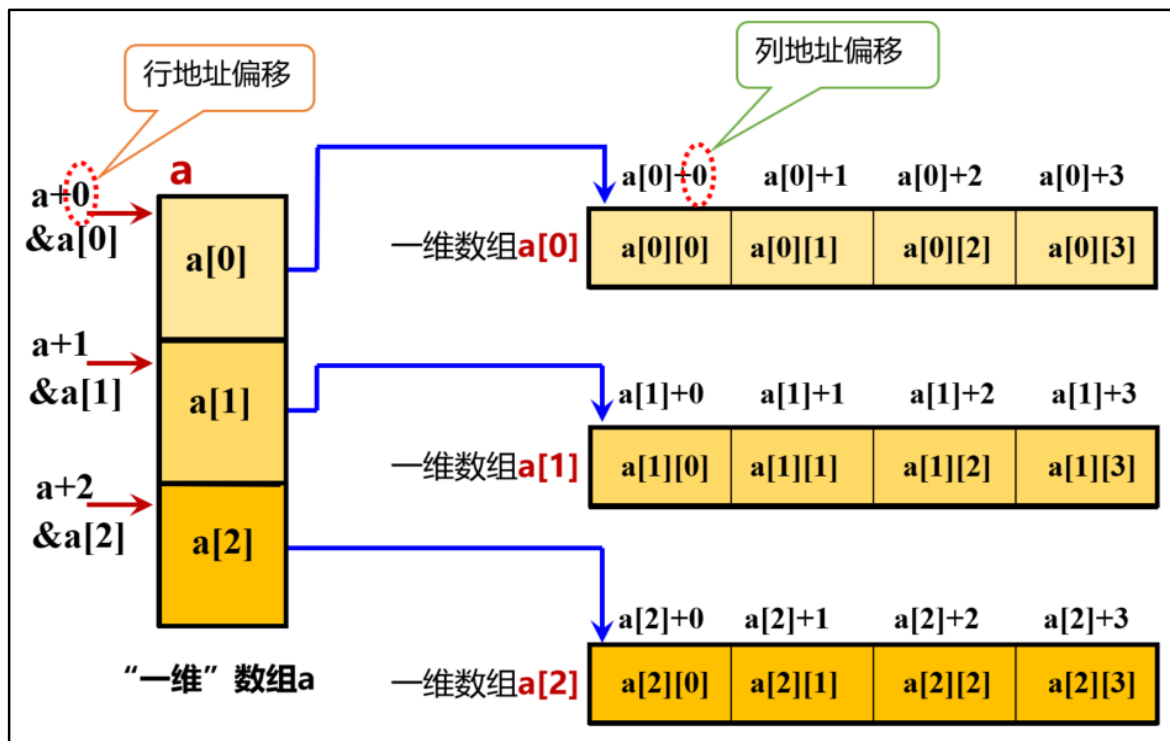
```
printf("Please enter a string of week:");
scanf("%s", x);

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**

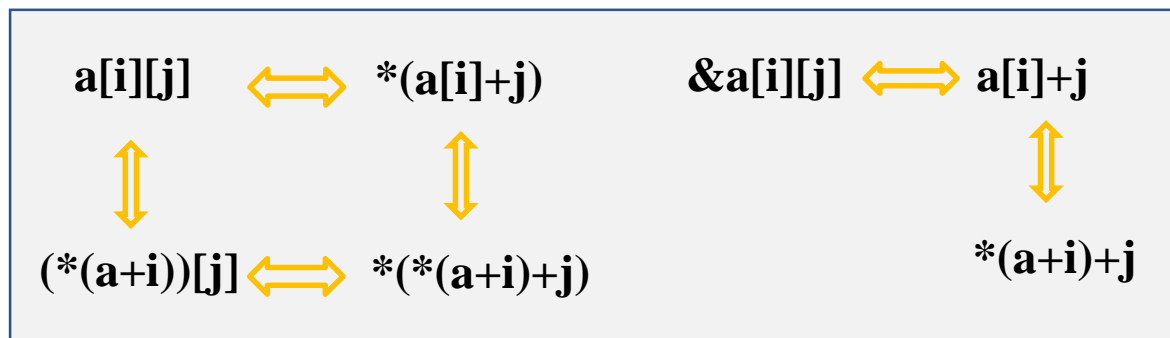
代表第*i*行第*j*列的地址

列地址

**\*(\*(a+i)+j)** 即 **a[i][j]**

代表第*i*行第*j*列的元素

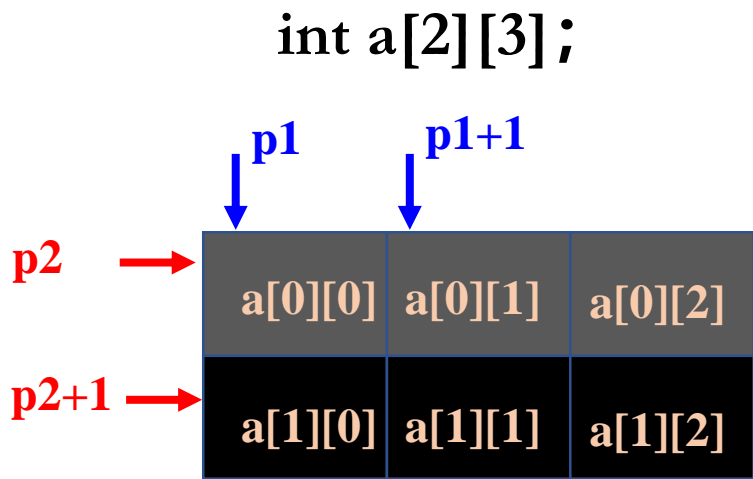
元素



在指向行的指针前面加一个\*，就转换为指向列的指针。

在指向列的指针前面加一个&，就转换为指向行的指针。

# \*指针与二维数组——列指针与行指针（形象描述）



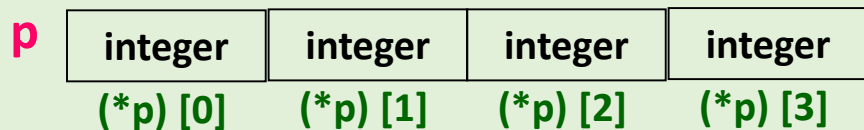
用列指针初始化，相对于数组起始地址的偏移量  $i*m+j$  ( $m$ 为2维数组列数)

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

**回顾行指针定义：**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



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
p →	5	6	7	8
	9	10	11	12

	1	2	3	4
p ↘	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=1; 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);
}
```

用指针作函数参数，可以返回多个值。**MonthDay**函数计算了**month**和**day**两个值。

# 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个月第几日

```
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}
};
```

两段代码功能互逆，请认真对比分析，  
仔细阅读

```
int dayofYear(int year, int *pMonth, int *pDay)
{
    int i, leap, day=0;
    leap = isLeap(year);
    for (i=1; i<*pMonth; i++)
        day += (*(dayTab+leap)+i);
    day += *pDay;
    return day;
}
```

```
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;
}
```

## 8.2 二维数组与数组指针

二维数组作为函数的形参：

```
#include <stdio.h>
void set_char(char *x[])
{
    x[0][0]='b';
}
int main()
{
    char c[5][10];
    c[0][0] = 'a';
    set_char(c);
    printf("%c\n", c[0][0]);
    return 0;
}
```

等价

```
#include <stdio.h>
void set_char(char *x[10])
{
    x[0][0]='b';
}
int main()
{
    char c[5][10];
    c[0][0] = 'a';
    set_char(c);
    printf("%c\n", c[0][0]);
    return 0;
}
```

c[5][10]与\*x[]  
类型不匹配,

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]'

## 8.2 二维数组与数组指针

二维数组作为函数的形参：

```
#include <stdio.h>
void set_char(char x[][10])
{
    x[0][0]='b';
}
int main()
{
    char c[5][10];
    c[0][0] = 'a';
    set_char(c);
    printf("%c\n", c[0][0]);
    return 0;
}
```

```
#include <stdio.h>
void set_char(char (*x)[10])
{
    x[0][0]='b';
}
int main()
{
    char c[5][10];
    c[0][0] = 'a';
    set_char(c);
    printf("%c\n", c[0][0]);
    return 0;
}
```

(\*x)[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. . .
```


## 8.2 二维数组与数组指针

- 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[5][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。

# 指针与数组的联系与区别

- 在C语言中，数组和指针之间最大的不同在于它们最初定义时的标识方法不同，下面两个声明之间最根本的区别就是内存分配。

```
int array[5];
```

```
int *p;
```

- 第一种声明中内存分配给 array 5个连续的int型字节内存，能够容纳该数组的所有元素；
- 第二种声明只分配sizeof(int\*)，通常4或8个字节，只存储一个地址。
- 声明的数组拥有存储数据的空间；而声明的指针变量，不与任何存储空间相关联，直到该指针变量指向某存储空间。
- 如果 p=array；指针变量p和数组array指向相同的地址，二者均可访问该数组。
- 使用指针访问数组元素的方式，其真正便利之处在于允许指针变量指向动态分配的内存空间，从而达到程序运行时根据所需大小创建存储数据空间的目的。

```
char* cp;
```

```
cp=(char*)malloc(10);
```



# 课堂测试：指针面试题

下面程序的输出是什么

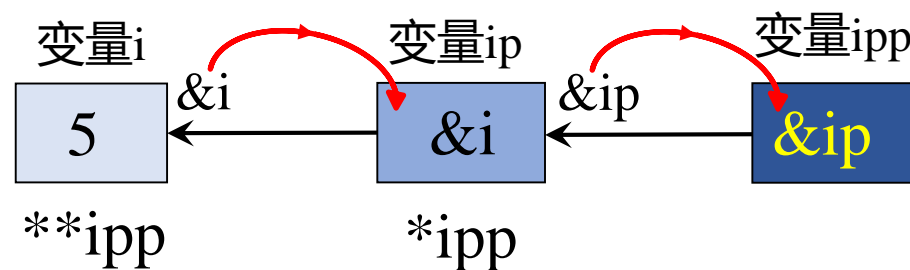
```
#include <stdio.h>
void UpperCase( char str[] ) // 将 str 中的小写字母转换成大写字母
{
    int i;
    printf("Uppercase: %lu\n", sizeof(str));
    for( i = 0; str[i] != '\0'; ++i )
        if(str[i] >= 'a' && str[i] <= 'z')
            str[i] -= ('a' - 'A' );
}
int main()
{
    char str[] = "aBcDe";
    printf("the length of str is: %lu\n", sizeof(str));
    UpperCase( str );
    printf("%s\n", str);
    return 0;
}
```

the length of str is: 6  
Uppercase: 4  
ABCDE

## \* 8.3 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 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);
```



输出:

```
i = 5, **ipp = 5  
i = 10, **ipp = 10
```

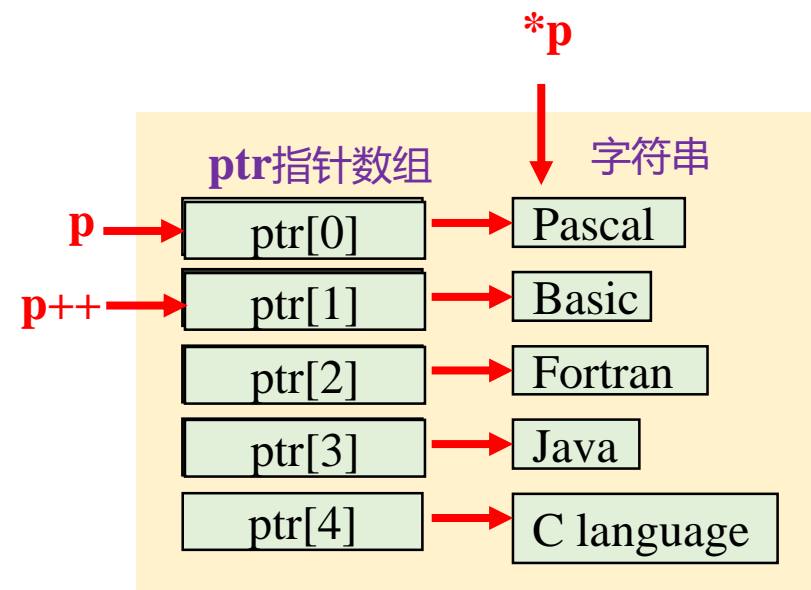
## 例8-4 多重指针与指针数组

```
#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?

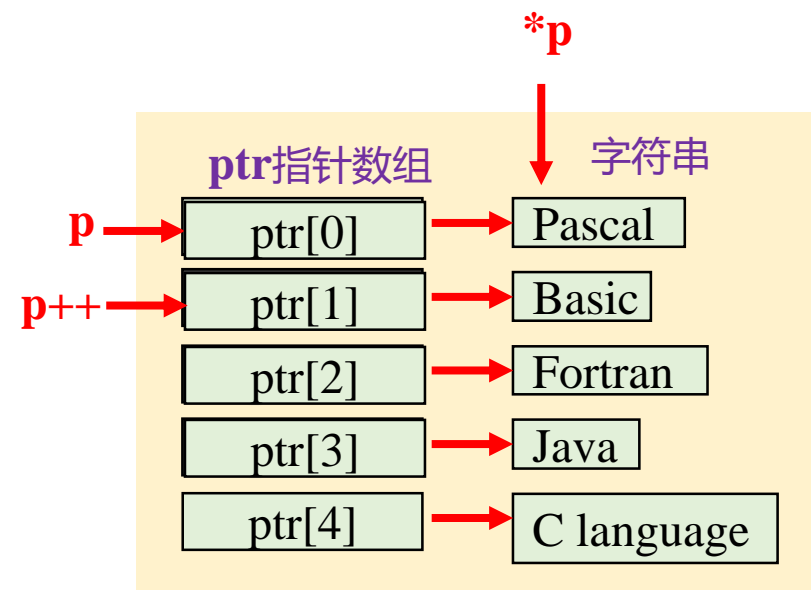
## 例8-4 多重指针与指针数组

```
#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

a

# 二重指针的应用示例

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
void getmemory(char *p,int num)
{
    p=(char *) malloc(num);
    strcpy(p,"hello world");
}
int main()
{
    char*str=NULL;
    getmemory(str, 100);
    printf("%s",str);
    printf("\n");
    free(str);
    return 0;
}
```

getmemory调用后，因为值传递，对主函数中的变量没有影响，str依然是NULL，输出空串

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
void getmemory(char **p, int num)
{
    *p=(char *) malloc(num);
    strcpy(*p,"hello world");
}
int main()
{
    char*str=NULL;
    getmemory(&str,100);
    printf("%s",str);
    printf("\n");
    free(str);
    return 0;
}
```

二级指针（指向指针的指针）

hello world

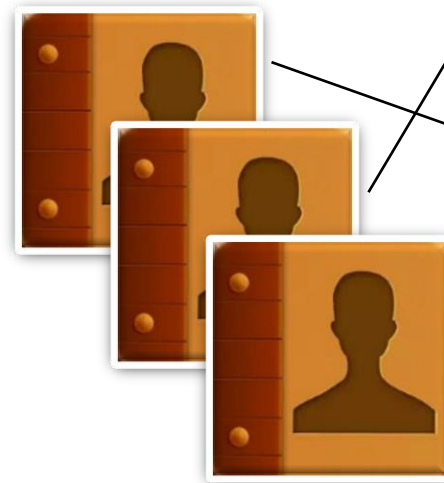
因为使用了二重指针，str在getmemory调用后指向了用malloc申请到的内存空间

## 8.4 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



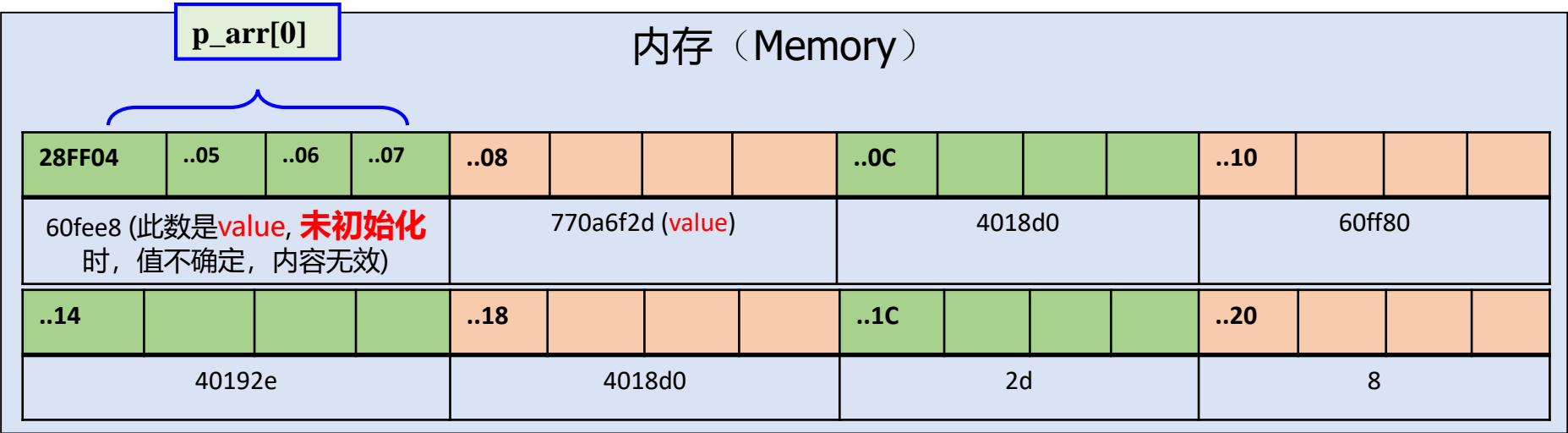
# 8.4.1 一维指针数组

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], ..

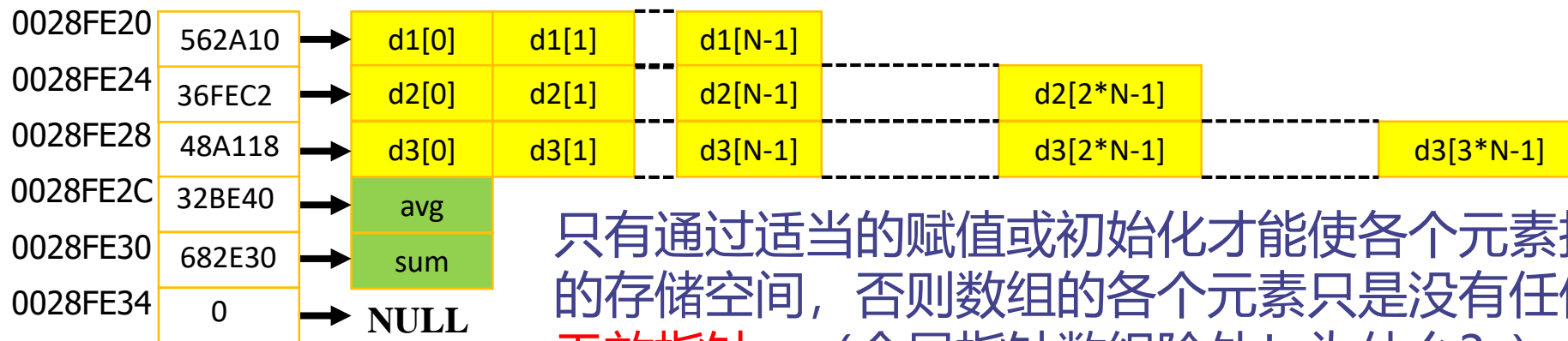
同理

## 8.4.1 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

与指针数组有什么不同?

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]);
}
```

## 8.4.1 一维指针数组

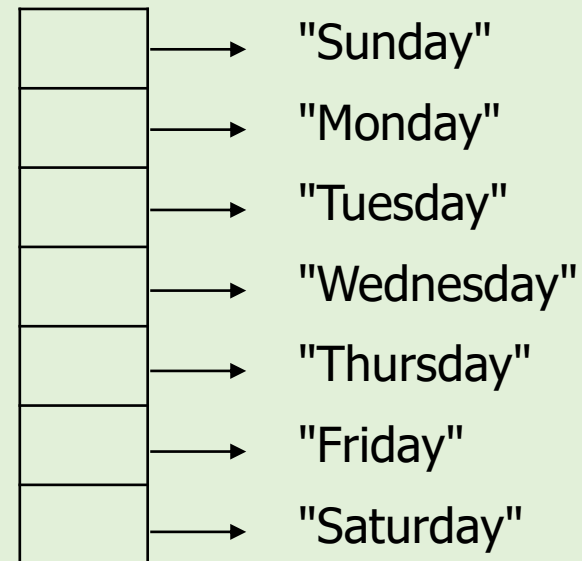
### 二维字符数组与字符指针数组的不同结构

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

## 8.4.1 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。

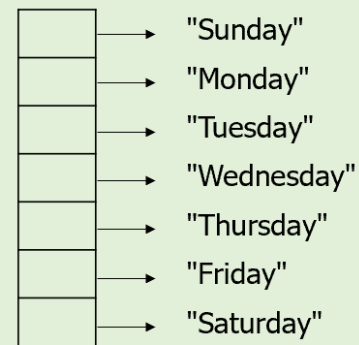
(3) 二维数组中全部元素的存储空间是连续排列的；而在指针数组中，只有**各个指针的存储空间是连续排列的**，其所指的数据元素的存储顺序取决于存储空间的分配方法，并且常常是不连续的。

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[];
```

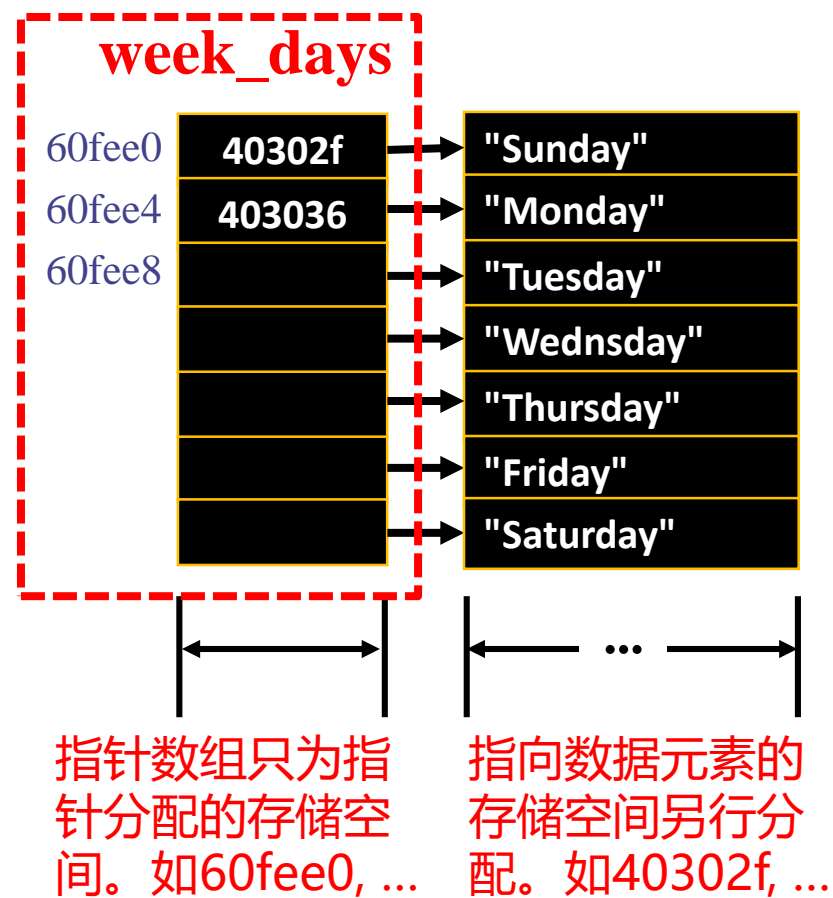
**space is 7\*4+x1+x2+...+x7**

(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]);
}
```

输出:

60fee0	->	403024
60fee4	->	40302b
60fee8	->	403032
60feec	->	40303a
60fef0	->	403044
60fef4	->	40304d
60fef8	->	403054



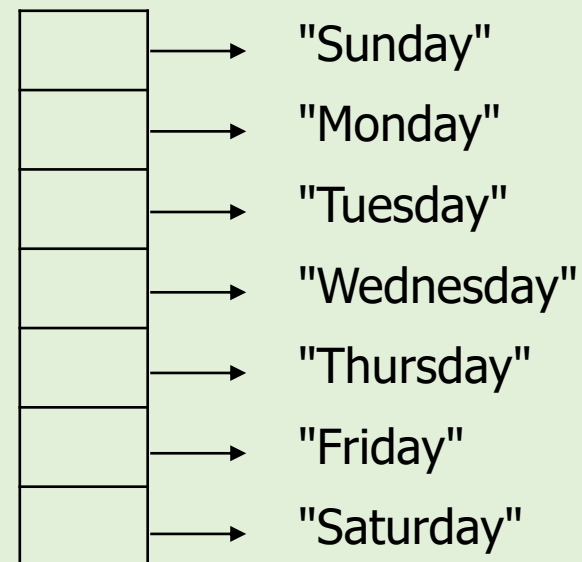
(2) 二维数组每一行中元素的个数是在数组定义时明确规定的，并且是完全相同的；而指针数组中各个指针所指向的存储空间长度不一定相同。

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  $7*12$

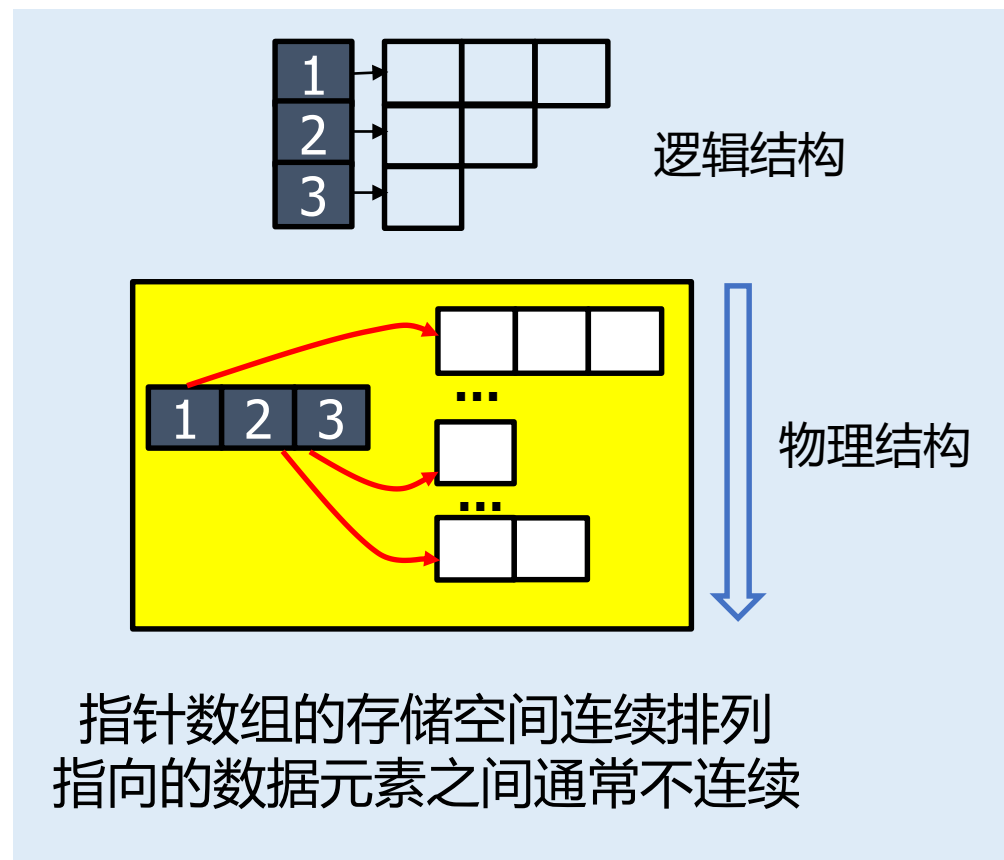
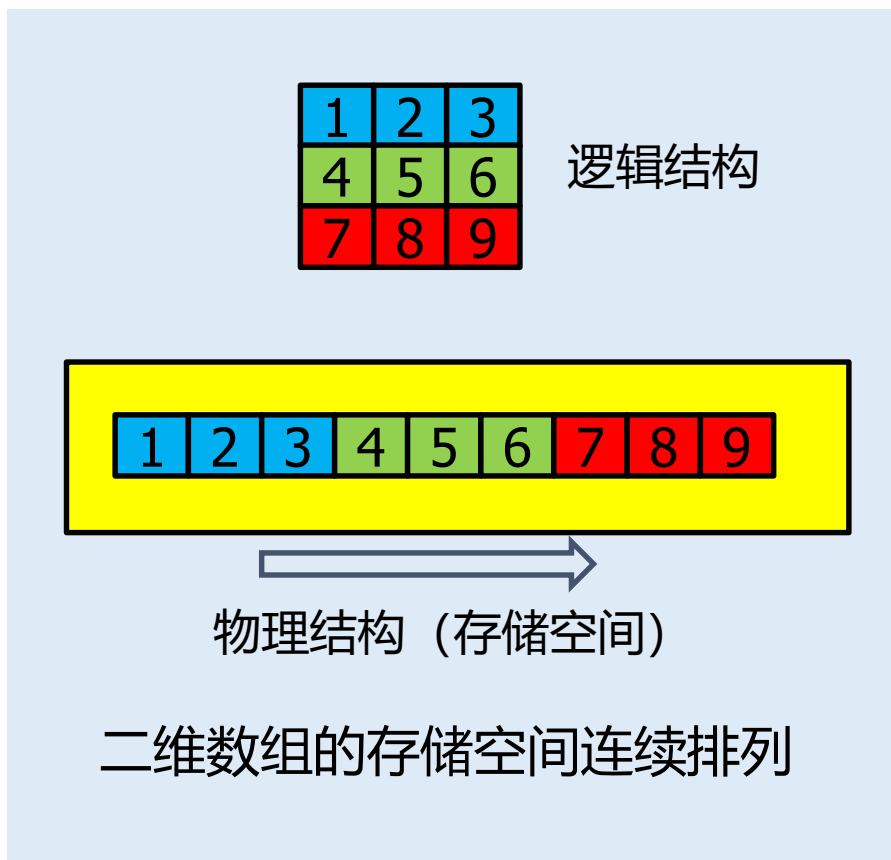


例8-5中的字符指针数组

```
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。



## 8.4.1 一维指针数组

### 例6-9中的二维字符数组

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

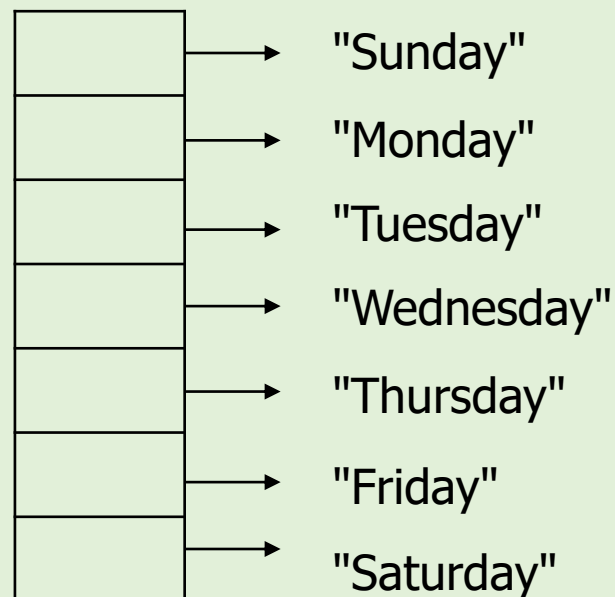
S	u	n	d	a	y	\0	\0	\0	\0	\0	\0
M	o	n	d	a	y	\0	\0	\0	\0	\0	\0
T	u	e	s	d	a	y	\0	\0	\0	\0	\0
W	e	d	n	e	s	d	a	y	\0	\0	\0
T	h	u	r	s	d	a	y	\0	\0	\0	\0
F	r	i	d	a	y	\0	\0	\0	\0	\0	\0
S	a	t	u	r	d	a	y	\0	\0	\0	\0

```
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。



## 8.4.1 一维指针数组

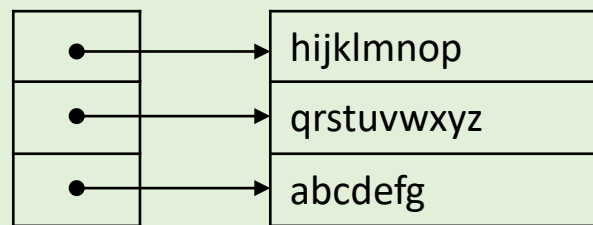
- 指针数组常被用作数据索引，以加快数据定位、查找、交换和排序等操作的速度。
- 在一些文字处理程序中，数据一般以“行”为单位保存在二维数组中，在数据处理的过程中，对各行位置的交换，以及整行内容的删除和新行的添加是频繁进行的操作（计算代价很大）。为提高程序的运行速度，往往使用指针数组作为实际数据的索引。

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

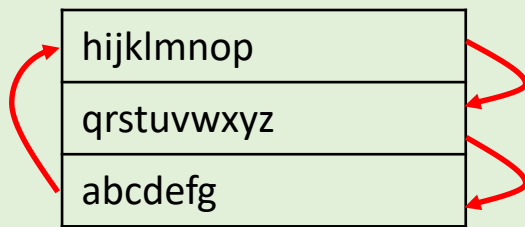
## 8.4.1 一维指针数组

- 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

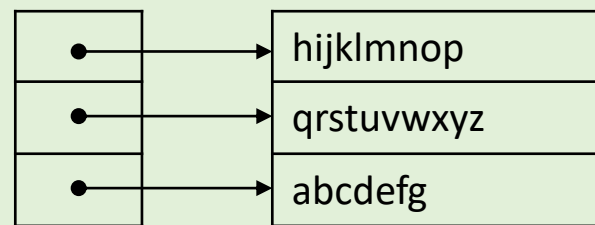
hijklmnop
qrstuvwxyz
abcdefg

排序中(交互数组)

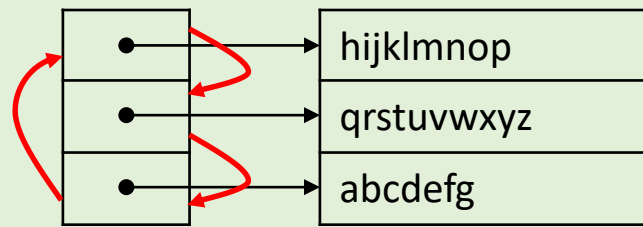


直接对二维字符数组排序

排序前



排序中(交换指针)



利用指针数组排序

## 8.4.1 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

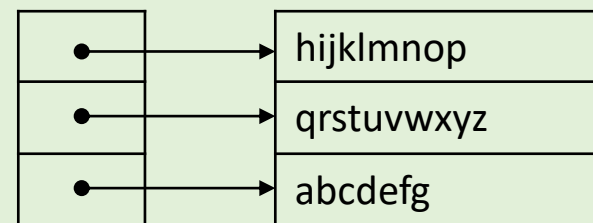
hijklmnop
qrstuvwxyz
abcdefg

排序后

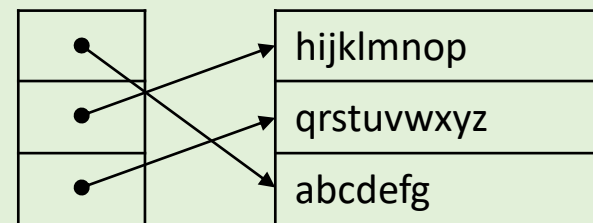
abcdefg
hijklmnop
qrstuvwxyz

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

## 8.5 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) (...);`

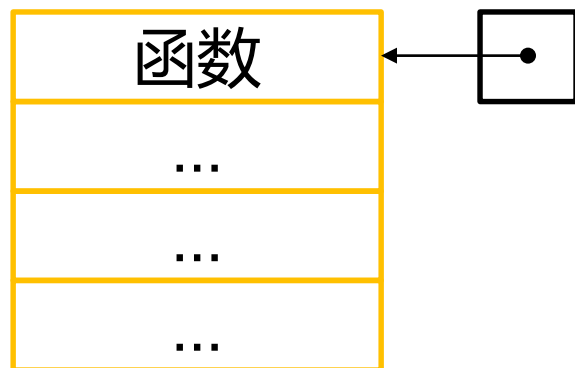
主语是指针，`f_name`是一个变量，指向一个返回`int`类型的函数

- 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。

## 8.5 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。

## 8.5.1 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。

## 8.5.1 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) ();
```

## 8.5.1 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!



## 8.5.2 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）

```
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]);
    }
}
```

函数指针的调用，即调用传进来的函数体

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 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;
}
```

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！

## 8.5.2 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。

## 8.5.2 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;
    return 0;
}

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

## 8.5.2 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;
    return 0;
}

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

## 8.5.2 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

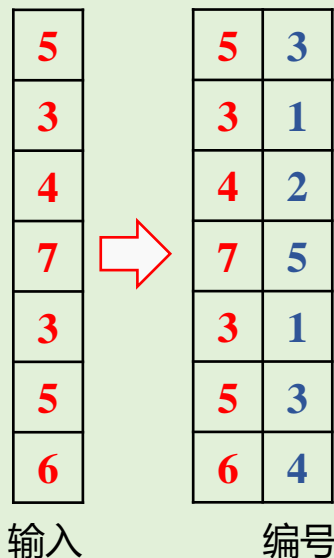
例8-9 输出数据的编号（顺序统计量） 从标准输入上读入  $n$  ( $1 < n < 200000$ ) 个整数，将其按数值从小到大连续编号（**第i小**），相同的数值具有相同的编号。在标准输出上按输入顺序以 **<编号>: <数值>** 的格式输出这些数据，各数据之间以空格符分隔，以换行符结束。

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



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)



# 8.5.2 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	输出样例
5	3:5
3	1:3
4	2:4
7	5:7
3	1:3
5	3:5
6	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？如何修改？

```
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
}

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

```

这是书上的代码，直接用这两个函数可能会报错！为什么？如何修改？

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

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。

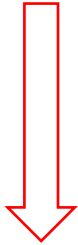
gen\_rank()的功能是给data中的数据按大小编号（填第三列）

```
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	数值等, 编号同
4	3	2	数值异, 编号增
5	1	3	
5	6	3	
6	7	4	
7	4	5	

数值      输入      数据  
            顺序      编号



## 8.5.2 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_int) != 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。  
实际应用时请改  
为例8-8的样式。

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 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int\*int 对比 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 $\leq \sqrt{n}$ 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 $\leq \sqrt{n}$ 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 $\sqrt{n}$ 。那么  $a < \sqrt{n}$ ，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  $q < \sqrt{n}$ 。
- ③ 所以：n 能被小于 $\sqrt{n}$ 的质数 q 整除！与① 假设矛盾！所以，若n不能被 $\sqrt{n}$ 的质数整除的话，n必为质数！ 证毕！



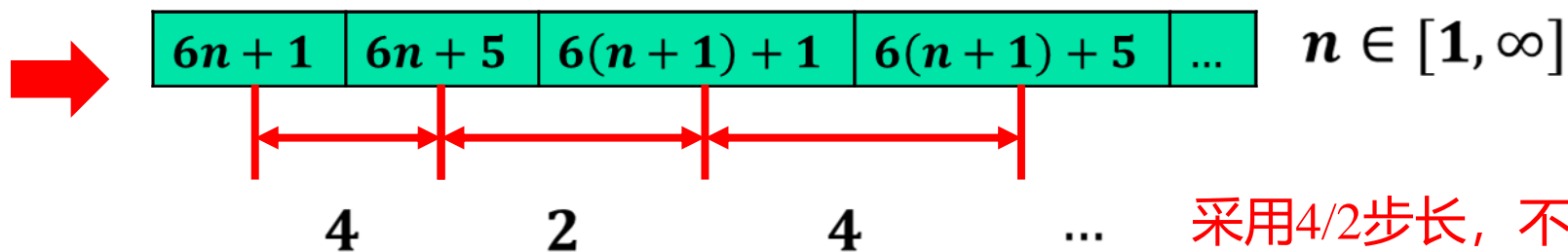
# 实例分析：判断质数

改进的质数  
判断函数和  
高效质数表  
初始化方法

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

$6n$	$6n+1$	$6n+2$	$6n+3$	$6n+4$	$6n+5$	...	$n \in [1, \infty]$
------	--------	--------	--------	--------	--------	-----	---------------------

$\times$   $\checkmark$   $\times$   $\times$   $\times$   $\checkmark$   
2倍数      2倍数 3倍数 2倍数



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

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Q个质数的质数表 (Q>=3)
{
    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 53 55 59 61 ...

## 关于使用指针的原则总结

---

- 永远要清楚每个指针指向了哪里-地址的有效性
- 永远要清楚指针指向的是什么-间接访问变量的正确性

## 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