北京航空航天大學
BEIHANG UNIVERSITY

程序设计基础

Fundamentals of Programming

北京航空航天大学 程序设计课程组

软件学院 谭火彬

2022年

北京航空航天大學
BEIHANG UNIVERSITY

第八讲 指针初步（2）

Pointer

- 数组指针
- 多重指针
- 指针数组
- 函数指针

简要回顾

- 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
  - ✓××类型的指针
- 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

提纲：指针（2）

- 8.1 数组类型与数组指针
- 8.2 多维（二维）数组与指针
- 8.3 多重指针
- 8.4 指针数组
- 8.5 函数指针

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，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
  - ✓int a[10]：a真正的数据类型是**数组类型 int [10]**（元素类型为int，数组长度为 10），拥有连续 10 个 int 型数据的内存空间
  - ✓数组类型的变量作为表达式，即数组名作为表达式，可以**隐式类型转换为指向数组首元素的指针**，从而参与指针运算或赋值给同类型指针变量
- ◆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**，属于聚合体类型

指针与数组的联系与区别

- 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4/8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，直到指针指向某存储空间
  - ✓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，其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：指针和数组

```
#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);
}
```

```
the length of str is: 6
Uppercase: 8
ABCDE
```

提示：数组作为参数时，实际传递的为指针（首地址）！
32位编译器指针大小为4字节，
64位编译器指针大小为8字节

数组指针

- 数组也是一种数据实体，也可以用取址符&进行取地址
  - ✓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，但和首元素地址的类型不一样
- 数组的地址类型为**指向数组的指针**，简称**数组指针**

<类型> (*<变量名>)[<元素个数>];

```
int a[10];
float b[20];
char c[30];
int (*pa)[10] = &a;
float (*pb)[20] = &b;
char (*pc)[30] = &c;
```

- 定义了三个数组指针，类型分别为 int (*)[10]、 float (*)[20]和 cha r(*)[30]，分别指向数组 a, b, c（而不是指向数组的第一个元素）

数组指针的运算

◆数组指针的本质还是指针，只不过指向的数据实体不是单一类型，而是数组类型，指针的所有运算都适用

✓数组指针每次加（减）1，意味着指针位置向后（前）移动了整个数组空间的大小，指针值的变动为整个数组内存空间的字节数

```
int a[10];
int (*pa)[10] = &a;
printf("%#X - %#X = %d\n", pa+1, pa, (void *)(pa+1)-(void *)pa);
printf("%p - %p = %d", pa+1, pa, (pa+1)-pa);
```

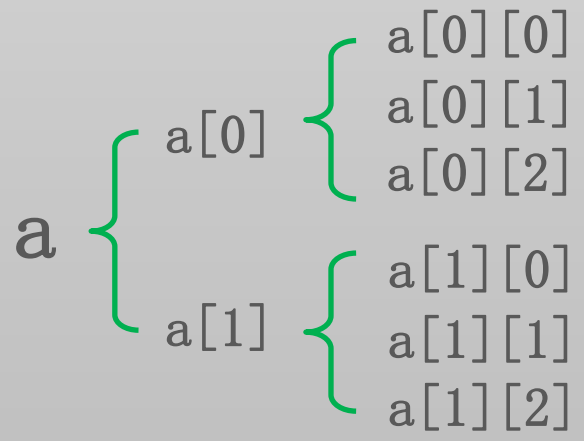
程序输出

```
0X61FDE8 - 0X61FDC0 = 40
000000000061FDE8 - 000000000061FDC0 = 1
```

8.2 多维（二维）数组与指针

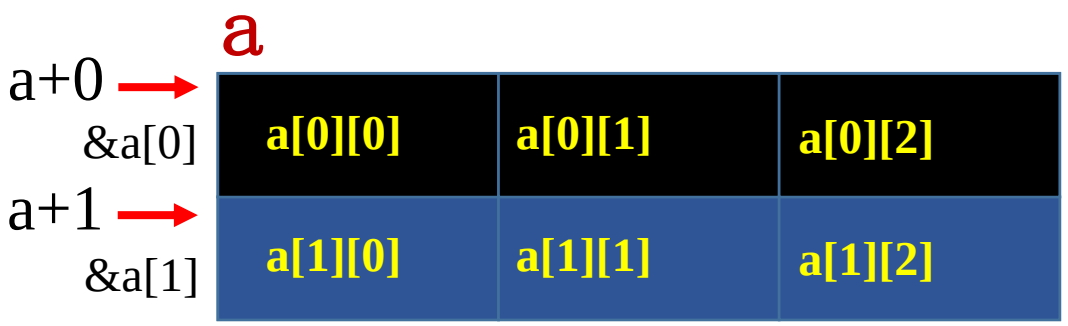
- 大于一维的数组称为多维数组，如：二维数组、三维数组
- 对于N维数组（N>1），可以看成是一个“一维数组”，该“一维数组”的每个元素是一个N-1维数组
- 对于N-1维数组的理解，可递归进行（数组的数组），直到数组元素变为单一类型

```
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）。

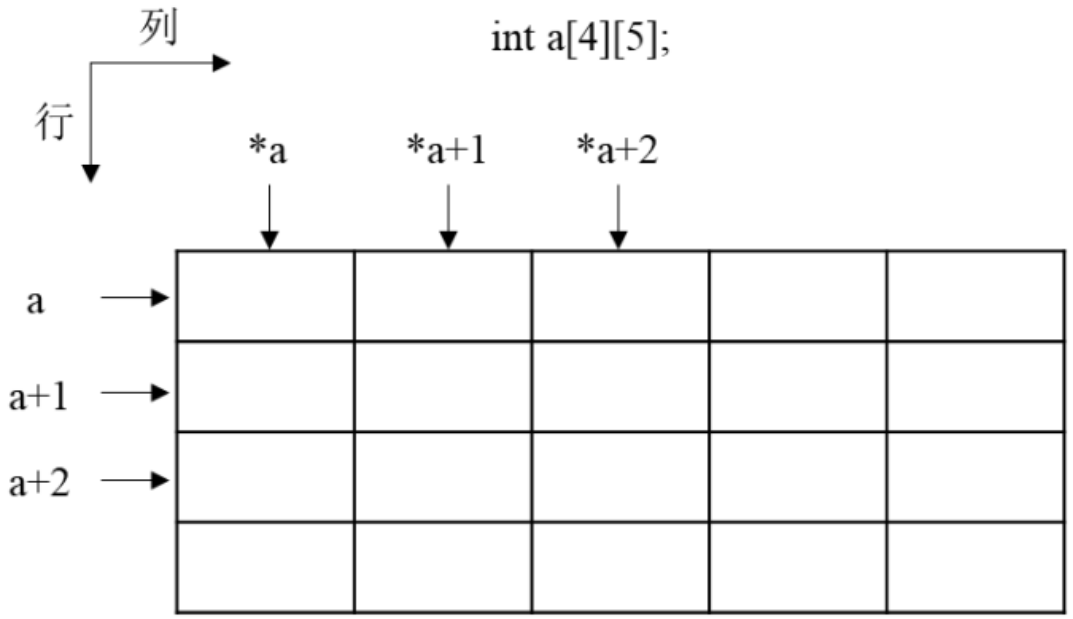
```
int a[2][3] = { {1,2,3}, {4,5,6} };
```



逻辑上，可看成2行3列，共6个元素

理解二维数组

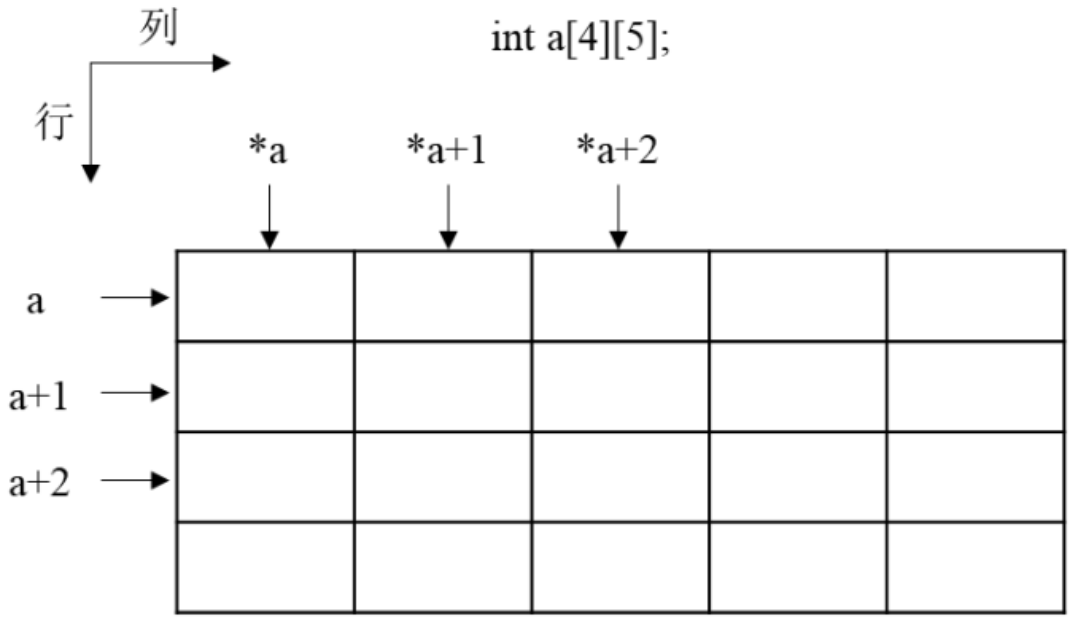
- 二维数组可以看作数组的数组，如：int a[4][5]
  - ✓可以看作一个长度为 4 的数组，如下图所示
  - ✓这个数组的每个元素又是一个长度为 5的 int 型数组（称为二维数组的行）
  - ✓a 的每一行以及每行的元素，在内存中连续存储
  - ✓在数学上， a 可看成一个 4× 5 的矩阵，共 4 行，每行 5 个元素



- 按照隐式类型转换规则，数组名 a 作为表达式可以看作指向首元素的指针，而二维数组的元素为一维数组，因此 a 作为表达式具有数组指针类型 int (*)[5]
- 它的值指向二维数组中的第 1 个数组，也就是第 1 行； a + 1 则指向二维数组中的第 2 个数组，即第 2 行，依次类推

理解二维数组

- 二维数组可以看作数组的数组，如：int a[4][5]
  - ✓可以看作一个长度为 4 的数组，如下图所示
  - ✓这个数组的每个元素又是一个长度为 5的 int 型数组（称为二维数组的行）
  - ✓a 的每一行以及每行的元素，在内存中连续存储
  - ✓在数学上， a 可看成一个 4× 5 的矩阵，共 4 行，每行 5 个元素



- 如果对 a 进行解引用： *a 代表第 1 行的数组，作为表达式*a 指向第 1 行数组的第 1 个元素，具有 int*类型；*a+1 则指向第 1 行数组的第 2 个元素，依次类推。
- 按照上面的指针运算规则，有*(a+i)+j 指向第 i+1 行数组的第 j+1 个元素，解引用后*(*(a+i)+j)代表二维数组中第 i+1 行 j+1 列的元素。
- *(*(a+i)+j) 等价于 a[i][j]

二维数组应用：C08-01-字符串排序（二维数组版）

◆C08-01：按字符串长度排序（二维数组版）

✓在标准输入上读入多行字符串（不超过 100 行，每行不超过 1000 个字符，每行的字符中可以包含空格），按照从短到长的顺序在标准输出上打印每行内容，如果长度一样则按字典序从小到大排列

◆问题分析

✓ 定义一个全局二维 char 型数组 char a[100][1002]保存输入的所有行

- a 每一个元素，是一个 char[1002]数组，用于保存输入的每一行字符串内容

✓输入结束后，对 a数组的每行按照字符串的长度进行冒泡排序

C08-01-字符串排序（二维数组版）

```
#include <stdio.h>
#include <string.h>
char a[100][1002]; // 用 a[i]来存储第 i+1 行字符串
int main() {
    int i, j, k = 0;
    char tmp[1002];
    while (fgets(a[k++], 1000, stdin) != NULL);// 读入所有行
    for (i = 1; i < k; i++) // bubble sorting
        for (j = 0; j < k - i; j++)   {
            int dicflag =(strlen(a[j]) == strlen(a[j + 1]))&&(strcmp(a[j], a[j + 1])>0);
            if (strlen(a[j]) > strlen(a[j + 1]) || dicflag) {
                strcpy(tmp, a[j]);
                strcpy(a[j], a[j + 1]);
                strcpy(a[j + 1], tmp);
            }
        }
    for (i = 0; i < k; ++i)
        fputs(a[i], stdout);
    return 0;
}
```

二维数组作为函数参数

◆二维数组作为函数参数时，函数声明

```
void F(int a[4][5]);
```

✓函数 F 接受一个 int [4][5]的二维数组作为参数

◆函数声明中a的行数 4 可以不用指定，但列数 5 必须要指定，可以写成：

```
void F(int a[][5]);
```

◆根据数组指针的含义，上述函数声明其实等价于

```
void F(int (*a)[5]);
```

✓形式参数 a 的本质是一个指向 int [5]类型的数组指针

✓当函数调用时，只要实际二维数组的元素类型和列数符合函数中的声明，就可以将该二维数组名作为实际参数传给函数 F，在函数体内部通过a[i][j]就可以访问实际参数二维数组的各个元素

二维数组应用：C08-02 方阵相乘

◆C08-02 方阵相乘：写一个函数实现 4x4 矩阵相乘， 4x4 矩阵用二维 double 数组表示

```
#include <stdio.h>
void mat_multi(const double (*m1)[4], const double (*m2)[4], double (*m3)[4]);
void mat_print(const double (*c)[4]);
int main()
{
    double m1[4][4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16};
    double m2[4][4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16};
    double m3[4][4];
    mat_multi(m1, m2, m3);
    mat_print(m3);
    return 0;
}
```

程序输出

```
90.00    100.00    110.00    120.00
202.00   228.00    254.00    280.00
314.00   356.00    398.00    440.00
426.00   484.00    542.00    600.00
```

C08-02 方阵相乘（数组指针版）

```
void mat_multi(const double (*a)[4], const double (*b)[4], double (*c)[4])
{
    int i, j, k;
    for (i = 0; i < 4; ++i)
        for (j = 0; j < 4; ++j)
        {
            c[i][j] = 0;
            for (k = 0; k < 4; ++k)
                c[i][j] += a[i][k] * b[k][j];
        }
}
void mat_print(const double (*c)[4])
{
    int i, j;
    for (i = 0; i < 4; i++)
        for (j = 0; j < 4; j++)
            printf(j == 3 ? "%-10.2f\n" : "%-10.2f ", c[i][j]);
}
```

采用数组指针作为函数参数，在函数内部可以直接像访问二维数组那样用双下标访问元素

C08-02b方阵相乘（线性下标版）

```
void mat_multi(double *m1, double *m2, double *ret, int n)
{
    int i, j, k;
    for (i = 0; i < n; ++i)
        for (j = 0; j < n; ++j)
        {
            ret[i * n + j] = 0;
            for (k = 0; k < n; ++k)
                ret[i * n + j] += m1[i * n + k] * m2[k * n + j];
        }
}
void mat_print(const double *m, int n)
{
    int i, j;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            printf(j == n - 1 ? "%-10.2f\n" : "%-10.2f ", m[i * n + j]);
}
```

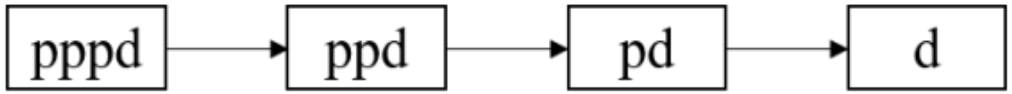
采用普通指针作为函数参数，将数组的第一个元素地址传进来，在函数内部将双下标映射为线性偏移量从而访问元素

8.3 多重指针

◆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

◆多重指针的定义： 类型 **标识符;

```
double d;
double *pd = &d;
double **ppd = &pd;
double ***pppd = &ppd;
```



对多重指针进行解引用，得到其数据实体，因为该数据实体也是指针，仍然可以继续解引用。每解一次引用，指针重数减 1，直到回溯到最终的数据实体（指针重数为 0，表示非指针变量）。

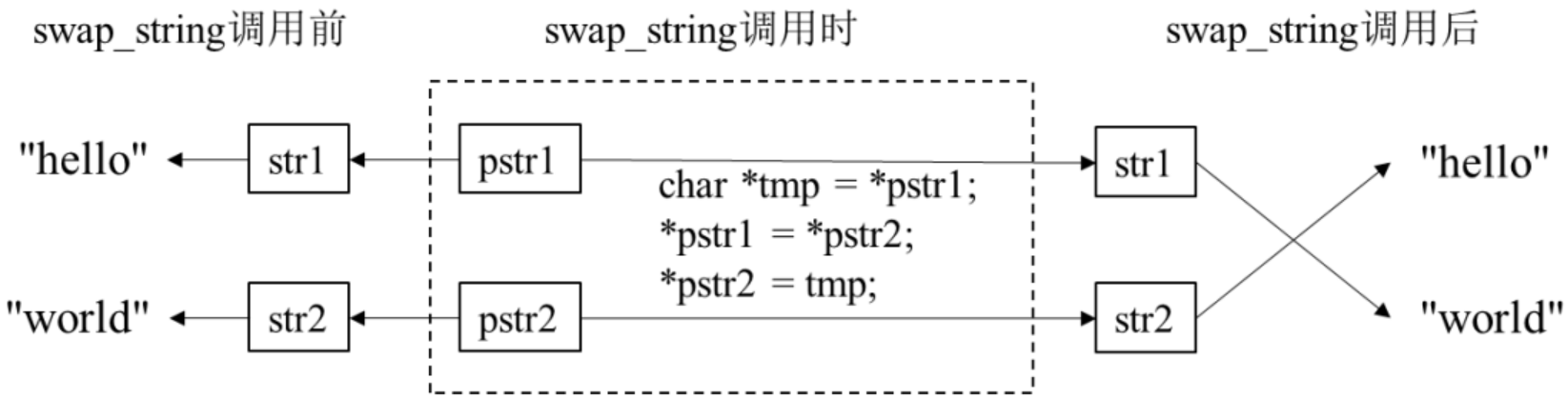
C08-03：交换字符串

◆C08-03：编写一个函数 swap_string，作用是交换两个字符串指针的指向

```c
#include <stdio.h>
void swap_string(char **pstr1, char **pstr2)
{
    char *tmp = *pstr1;
    *pstr1 = *pstr2;
    *pstr2 = tmp;
}
```

```c
int main()
{
    char *str1 = "hello";
    char *str2 = "world";
    swap_string(&str1, &str2);
    printf("%s, %s\n", str1, str2);
    return 0;
}
```

world, hello



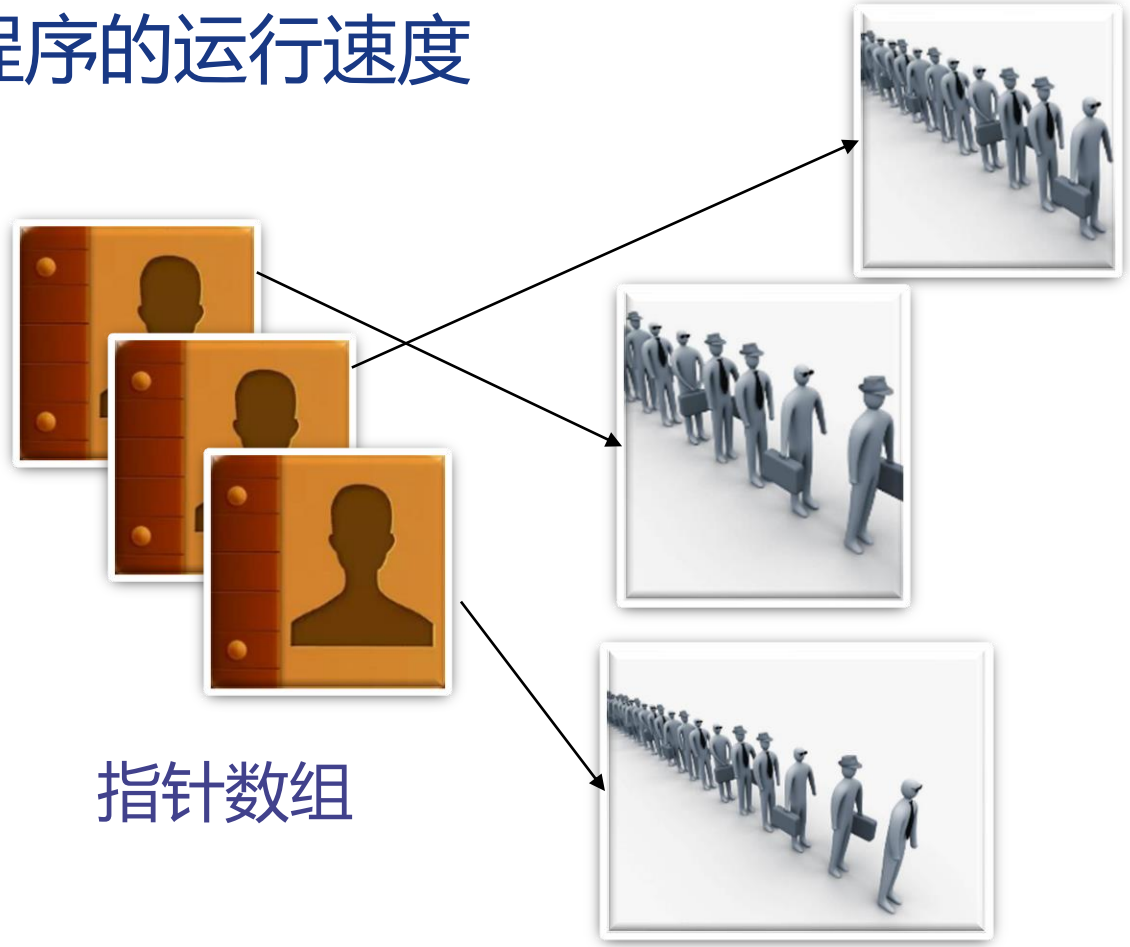
8.4 指针数组

◆指针数组：是一个数组，但数组元素是指针类型

✓利用数组管理数据的地址，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



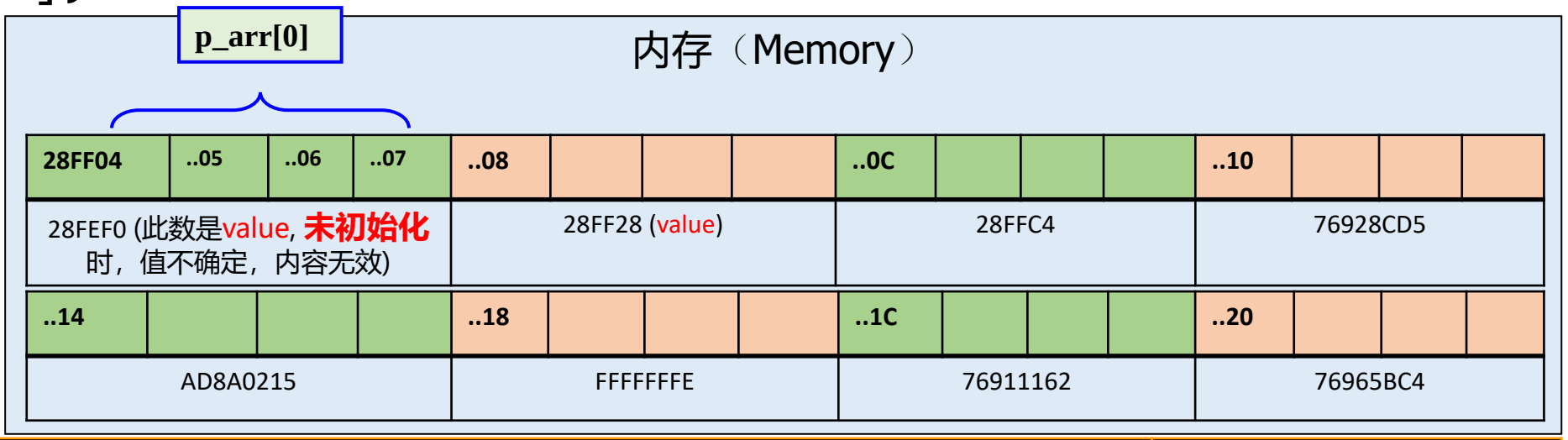
指针数组

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] 地址即为数组的首地址(&p_arr[0])是 0x28FF04

p_arr[0] 值未初始化时，其值是不确定的，无意义(表中 0x28FEF0 是确定值)

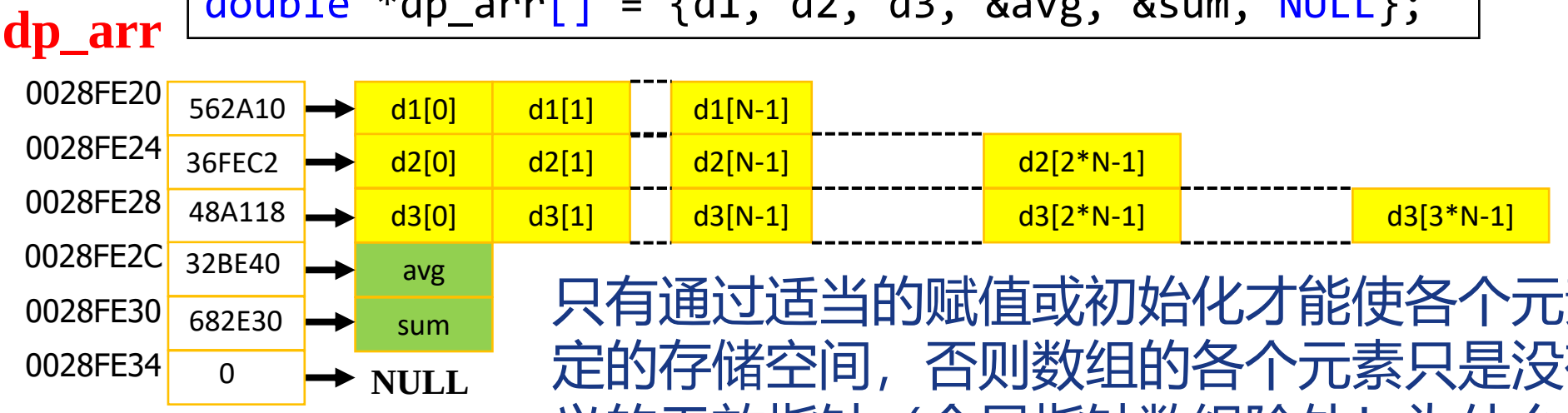
p_arr[1], p_arr[2], .. 同理

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};
```



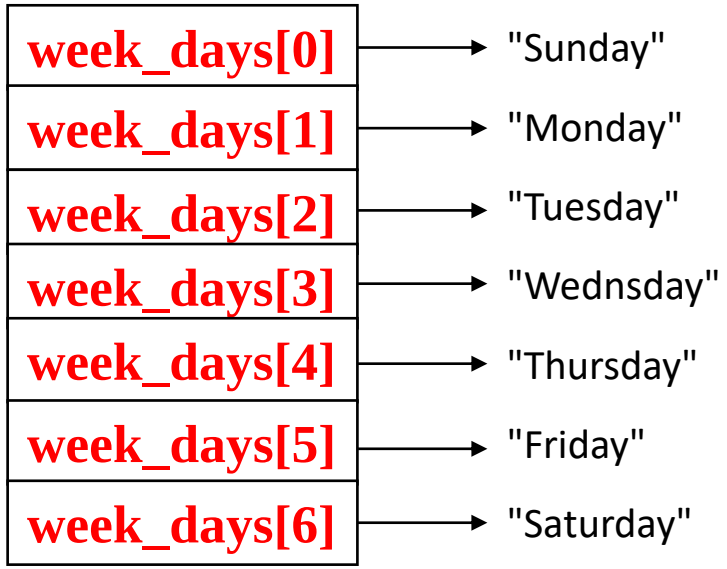
只有通过适当的赋值或初始化才能使各个元素指向确定的存储空间，否则数组的各个元素只是没有任何含义的无效指针（全局指针数组除外！为什么？）

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

示例：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

```
//指针数组实现
char *week_days[] = {
        "Sunday",
        "Monday",
        "Tuesday",
        "Wednesday",
        "Thursday",
        "Friday",
        "Saturday"
};
```

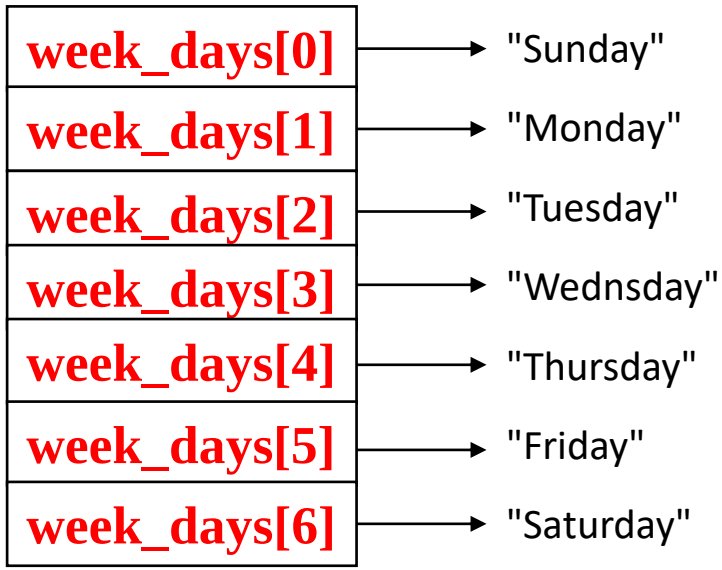


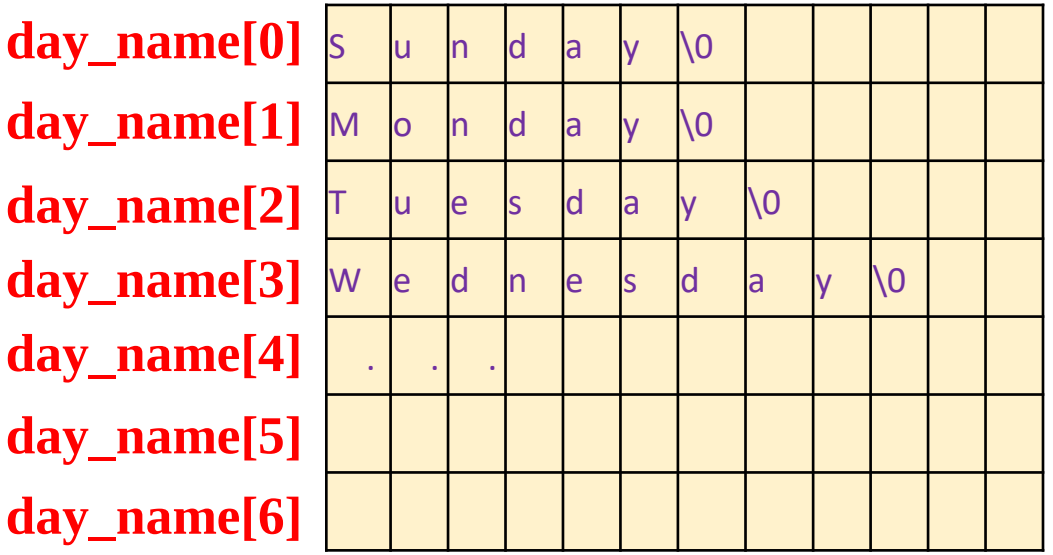

```
void week_day(int x, int y, int n, int k)
{
        int m;
        m = (n - x + y + k) % 7;
        printf("%s\n", week_days[m]);
}
```

指针数组 VS. 二维数组

```
//指针数组实现
char *week_days[] = {
        "Sunday", "Monday", "Tuesday",
        "Wednesday","Thursday",
        "Friday", "Saturday"
};
```

```
//二维数组
char day_name[][12] = {
    "Sunday", "Monday", "Tuesday",
    "Wednesday", "Thursday",
    "Friday",  "Saturday"
};
```





一维指针数组与二维数组

◆指针数组与二维数组的区别

- ✓（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

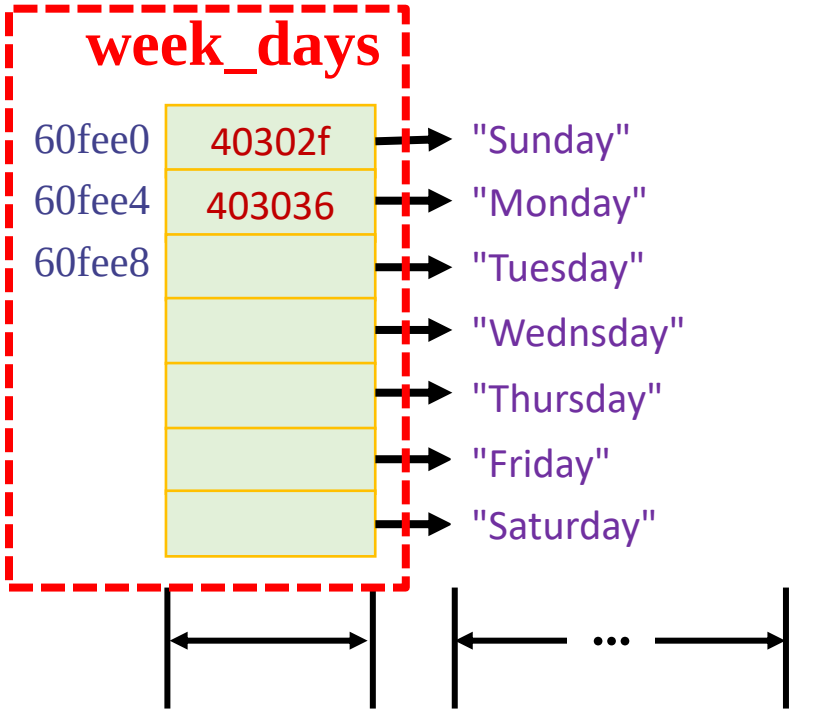
一维指针数组与二维数组

- （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

```
//指针数组
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：

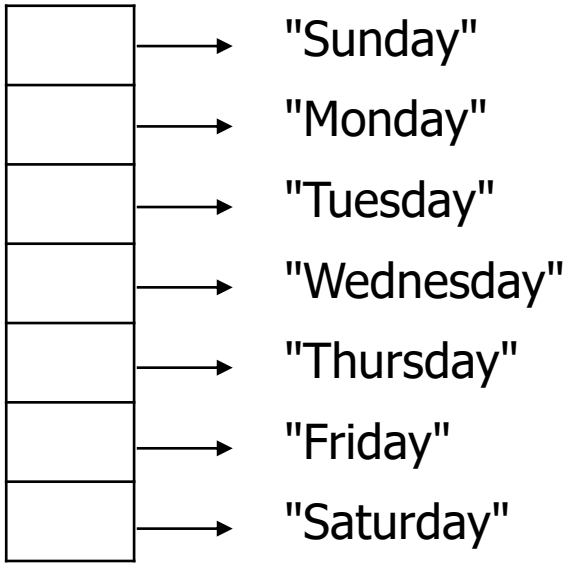
```
C:\Users\song\Desktop\sytestC\testc\temptemp-dyna-array.exe
60fee0 -> 40302f
60fee4 -> 403036
60fee8 -> 40303d
60feec -> 403045
60fef0 -> 40304f
60fef4 -> 403058
60fef8 -> 40305f
```



一维指针数组与二维数组

◆(2) 二维数组每一行中元素个数是在数组定义时明确规定的，并且是完全相同的；而指针数组各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

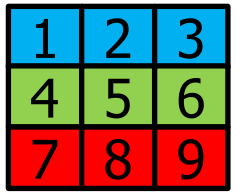



```
char day_name[][12];
//存储空间为7*12=84个字节
```

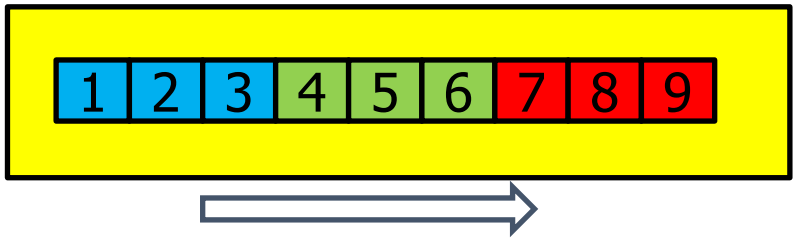
```
char *week_days[];
//存储空间为7个指针+各字符串本身的长度
```

一维指针数组与二维数组

- （3）二维数组中全部元素的存储空间是连续排列的；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

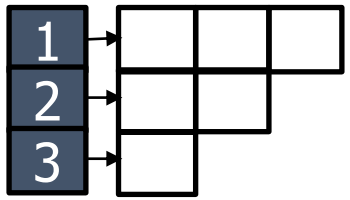


逻辑结构

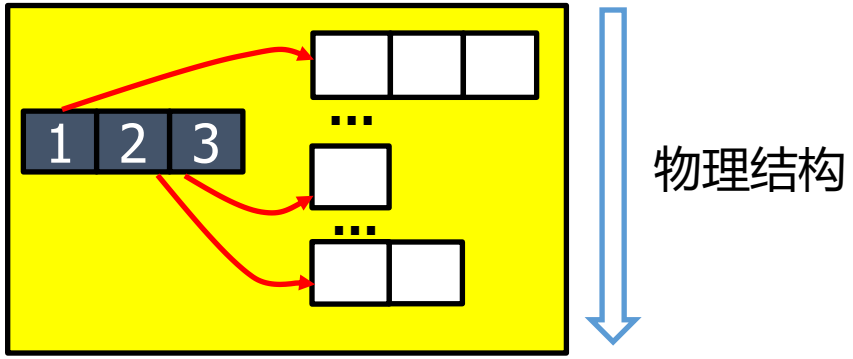


物理结构（存储空间）

二维数组的存储空间连续排列



逻辑结构



指针数组的存储空间连续排列
指向的数据元素之间通常不连续

一维指针数组与二维数组

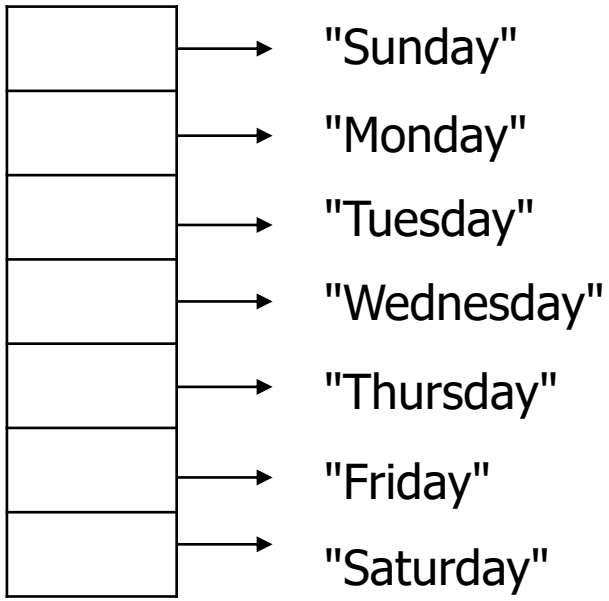
```
char day_name[][12] = { "Sunday",…}
```

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

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

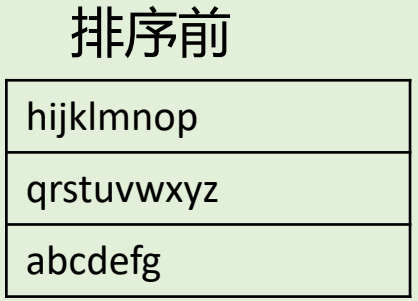



```
*(week_day[0]) = 's'; // 运行错误!
                      // 常量数据不能改
```

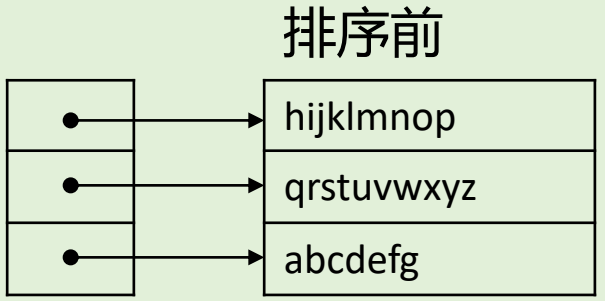
在本例，指针数组所指向的字符串是常量，
指针数组元素是变量，可以指向不同位置

使用一维指针数组

- 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



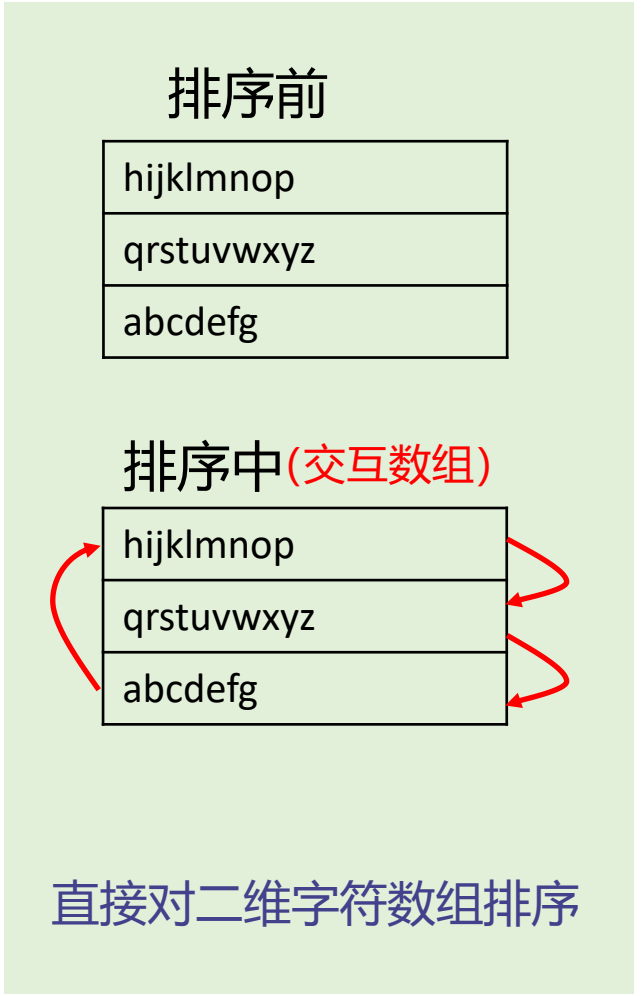
直接对二维字符数组排序



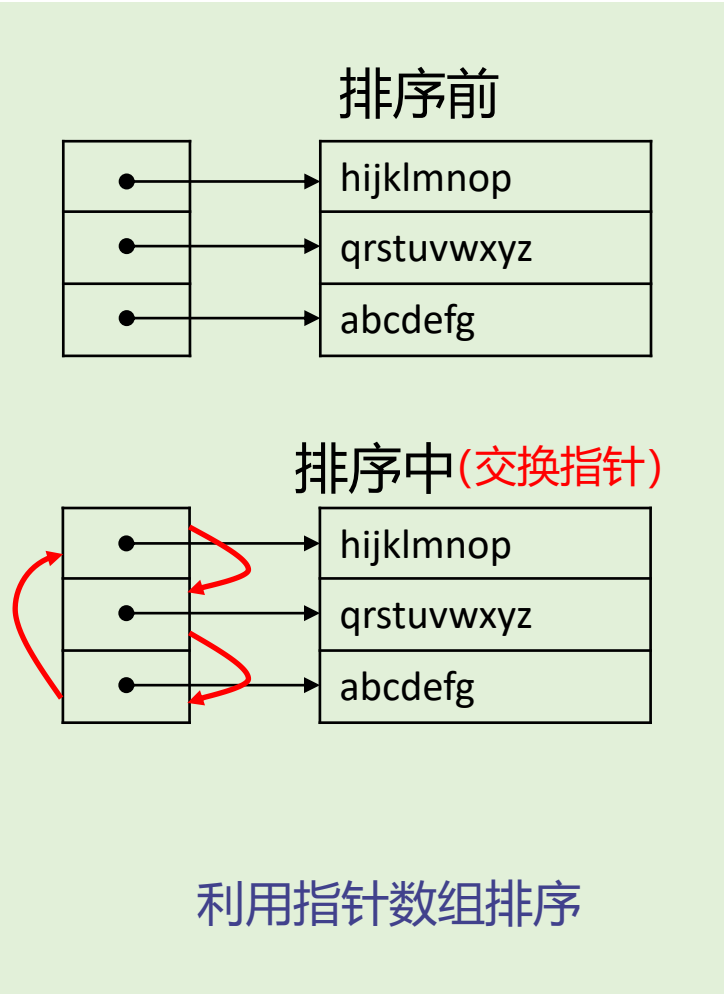
利用指针数组排序

使用一维指针数组

- 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



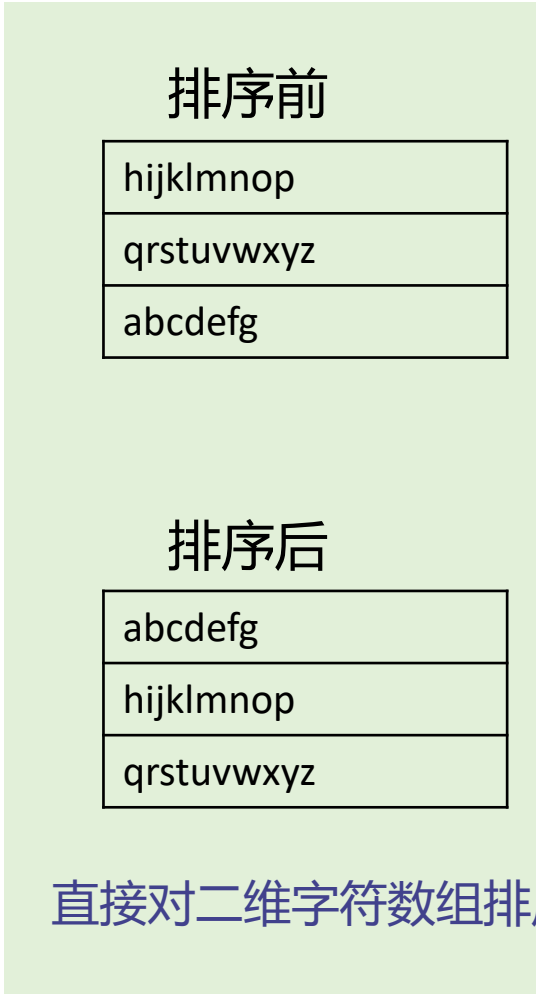
直接对二维字符数组排序

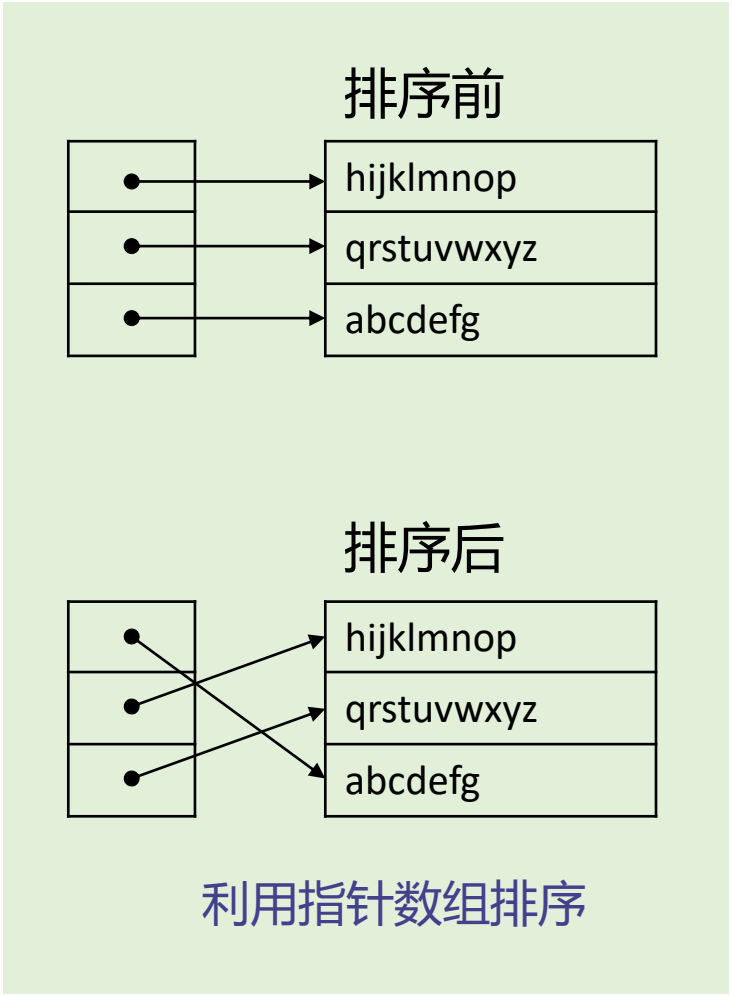


利用指针数组排序

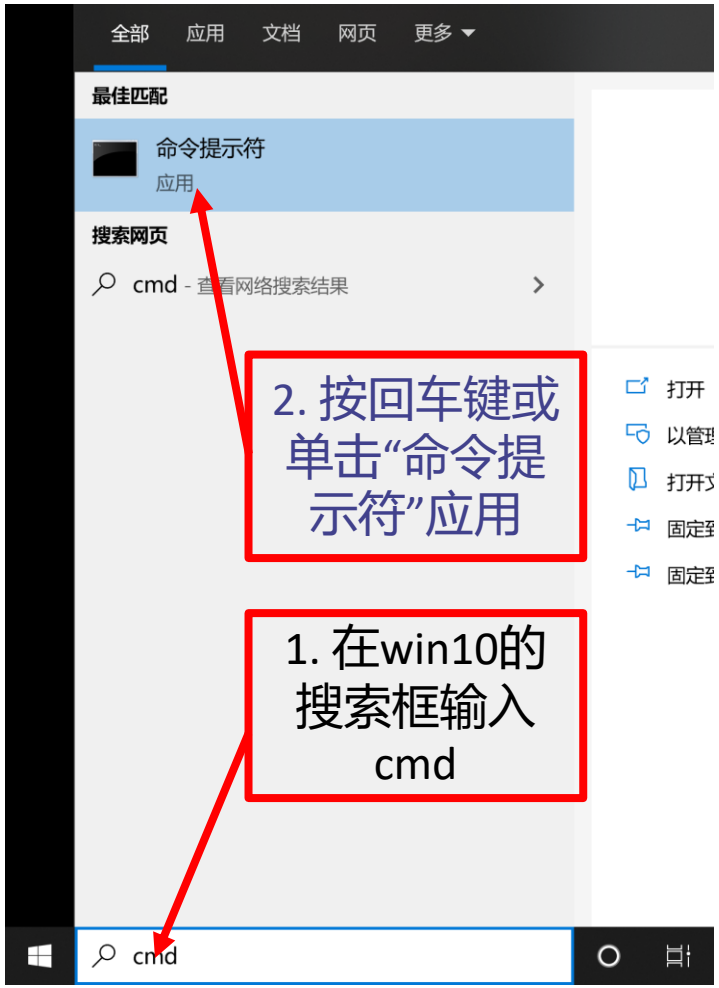
使用一维指针数组

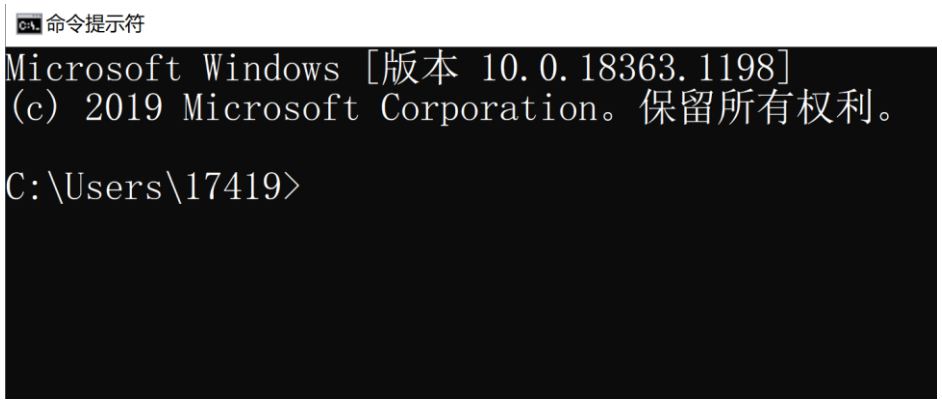
- 指针数组常被用作数据索引，以加快数据定位、查找、交换和排序等操作的速度
- 在一些文字处理程序中，数据一般以“行”为单位保存在二维数组中，在数据处理的过程中，对各行位置的交换，以及整行内容的删除和新行的添加是频繁进行的操作（计算代价很大）。为提高程序的运行速度，往往使用指针数组作为实际数据的索引

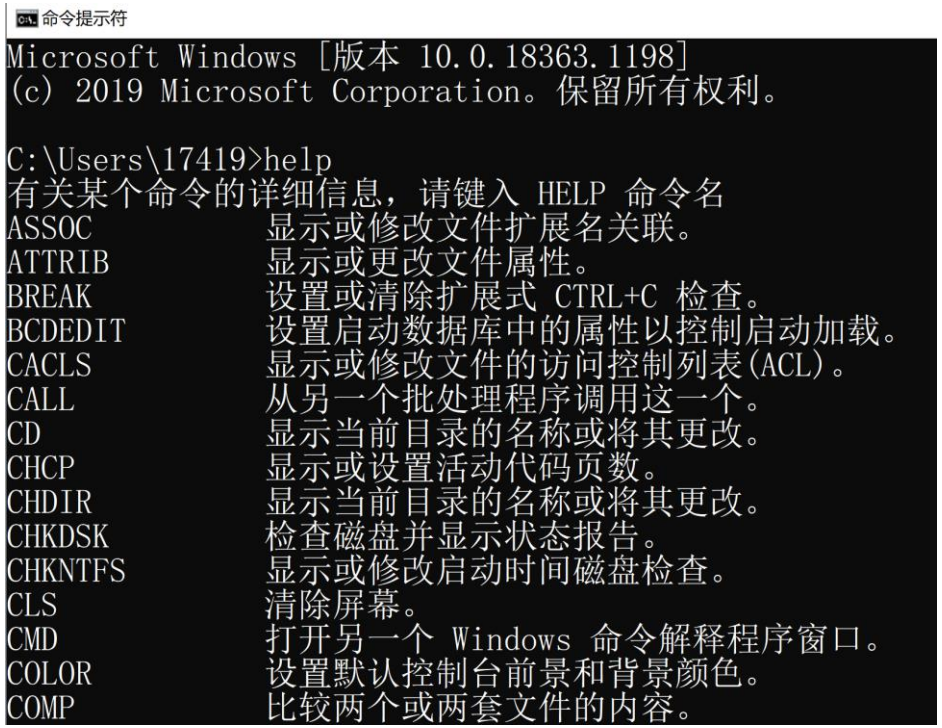




一维指针数组的应用：命令行参数







使用命令行参数

◆通过命令行参数，用户可以告知程序各类运行参数/选项，从而影响根据需要来决定程序干什么、怎么干

命令提示符

```
C:\Users\17419>cd C:\a1ac\code

C:\a1ac\code>copy test.c sy.c
已复制         1 个文件。

C:\a1ac\code>_
```

高手都喜欢用命令行来操作电脑

使用命令行参数

◆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

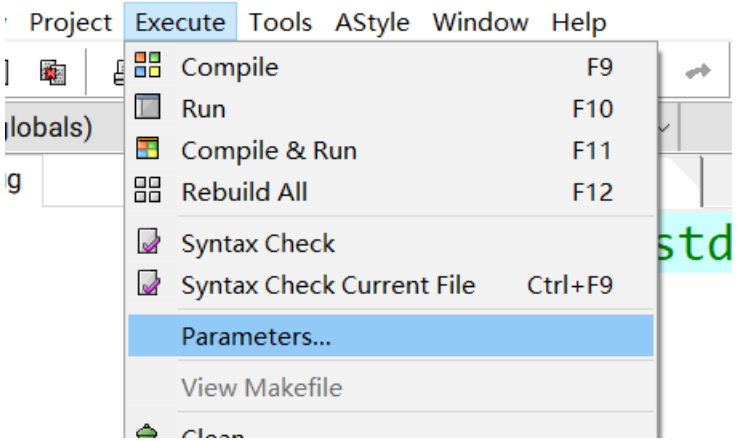
文件拷贝	： cp src_file dest_file	
列出当前目录	： ls –l	-l: 列出当前目录下所有文件详细信息
切换目录	： cd ~	
查找文件	： find . -name "*.c"	-name "*.c":将目前目录及其子目录下所有延伸档名是 c 的文件列出来

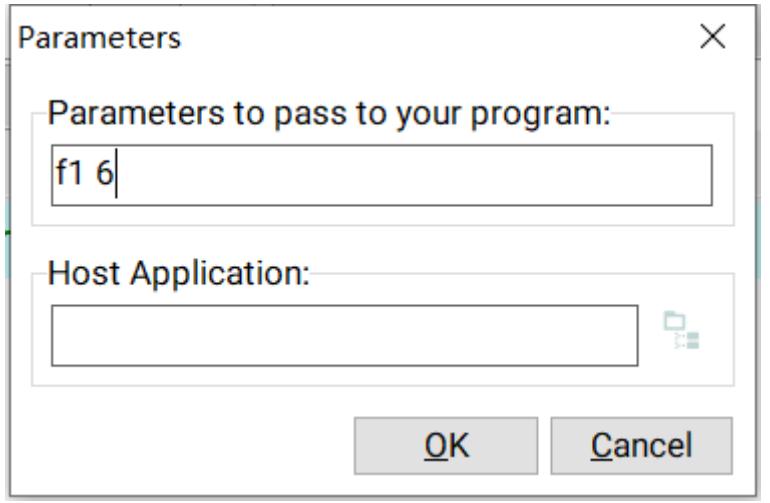
编写命令行程序

- 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"。





C08-04：使用命令行参数

◆C08-04：计算命令行参数的代数和：通过命令行参数，输入若干个整数，对这些整数求和，在标

- ✓argc 表示包含程序名在内的参数个数
- ✓argv[0]指向命令行上调用程序时输入的程序名（字符串）
- ✓从 argv[1]到 argv[argc-1]分别顺次指向命令行中第 1 个到第 argc-1 个参数（字符串形式）

```
#include <stdio.h>
int main(int argc, char *argv[]){
    int i;
    printf("%s: ", argv[0]);
    for (i = 1; i < argc; i++)
        printf("%s ", argv[i]);
    return 0;
}
```

8.6 函数指针

◆函数指针：是一个指针变量，该指针变量可以指向某个函数

```
//f_name是一个函数指针变量，可以指向一个函数
//这个函数接受一个int型参数，返回一个int型值
int (*f_name)(int);
```

◆注意与指针函数的区别

✓指针函数是一个函数，函数返回类型的值

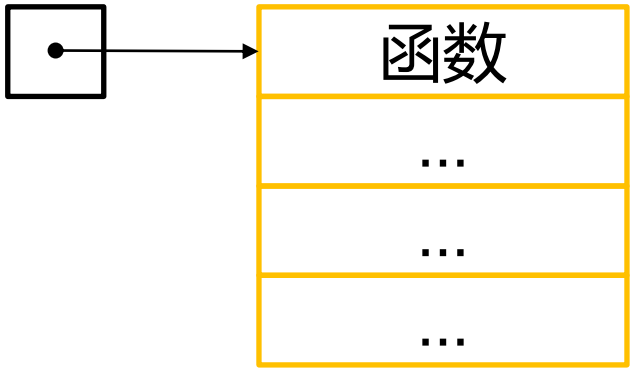
```
//strstr是个函数，返回类型为char*
char  *strstr(char *s, char *s1);
```

◆类似于指针数组和数组指针的区别

```
char *a[10];    //（一维）指针数组
char (*a)[10]; //数组指针，可以指向char[10]的数组
```

函数指针

- 函数指针的类型由函数原型决定，即：函数的参数个数、参数类型和返回类型
- 函数指针可以指向同类型的函数
  - ✓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



定义函数指针

◆定义一个函数指针变量

<返回类型> (*<标识符>) (<参数表>);

```
//例如:
int (* funPtr1) (int, int);
void (* funPtr2) (int, int, int);
int (* funPtr3) (double, char *);
int * (* funPtr4) ();
```

```
//声明一个函数指针func,
//可以指向含有两个double参数，返回double类型的函数
double (*func) (double x, double y);
// 等价于，参数的变量名一般省略
double (*func) (double, double);
```

使用函数指针

```
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) { //sum函数定义
     return x + y;
}
…
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)
```

◆为了方便使用，C语言中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！

使用函数指针

◆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
```

◆利用函数指针，可以设计“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

C08-05-函数指针应用：通用冒泡排序算法

◆写一个通用的冒泡排序函数，可以对任何数据类型的数组进行任何形式（升序或降序）的稳定排序

◆问题分析：

✓考虑数组类型是不确定的，采用void*作为函数接口

```
void g_bub_sort(void *array, int len, int elemSize, int (*cmp)(const void *, const void *));
```

✓其中： array 是数组起始地址， elemSize 是数组单个元素所占内存空间的字节数， len 是数组长度（数组元素个数）， cmp 是函数指针，指向具有如下函数原型的比较函数：

```
int cmp(const void *e1, const void *e2);
```

- cmp 比较两个元素 e1 和 e2 的顺序关系，如果 e1 指向的元素排在 e2 指向元素的前面，cmp 返回 1，否则返回 0。因为 cmp 不会改变被比较的元素，因此参数是 const 指针
- 在冒泡排序的算法框架中需要交换顺序不对的两个元素的值，但数组元素类型在函数设计时是未知的，不能简单地通过赋值运算完成元素交换。需要设计一个通用函数来完成元素逐个字节的内容交换（原始内存空间复制）

C08-05-函数指针应用：通用冒泡排序算法

◆（1）内容交换的通用函数实现

```
void swap_elem(void *e1, void *e2, int elemSize)
{
    int i;
    char tmp;
    for (i = 0; i < elemSize; ++i)
    {
        tmp = *(char *)(e1 + i);
        *(char *)(e1 + i) = *(char *)(e2 + i);
        *(char *)(e2 + i) = tmp;
    }
}
```

e1 和 e2 分别指向要交换数据内容的两个内存空间的地址， elemSize 是内存空间的字节数

通过强制类型转换，将每个字节重新解释成 char 型并交换内容，直到所有字节的数据交换完成

C08-05-函数指针应用：通用冒泡排序算法

◆(2) 冒泡排序的通用函数实现

```
void g_bub_sort(void *array, int len, int elemSize, int (*cmp)(const void *, const void *))
{
    int i, j;
    for (i = 1; i < len; i++) // bubble sorting
        for (j = 0; j < len - i; j++)
            if (cmp(array + (j + 1) * elemSize, array + j * elemSize))
                swap_elem(array + j * elemSize, array + (j + 1) * elemSize, elemSize);
}
```

if 语句是通过函数指针调用比较函数来判断元素 j 和元素 j+1 是否有序，如果不是有序则交换它们的内容（如果 j+1 元素应该排在 j 元素的前面， cmp 会返回真，表示 j和 j+1 元素不是有序的）。函数指针指向的实体将在函数调用时确定

C08-05-函数指针应用：通用冒泡排序算法

◆编写比较函数

(3) 比较函数示例（整数型升序）

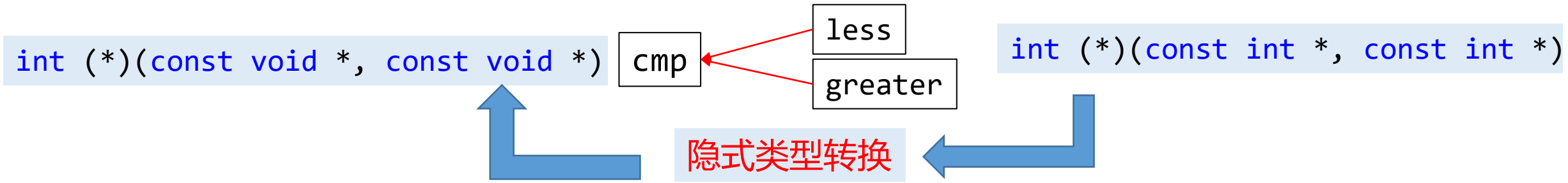
```
int less(const int *e1, const int *e2)
{
    return *e1 < *e2;
}
```

(4) 比较函数示例（整数型降序）

```
int greater(const int *e1, const int *e2)
{
    return *e1 > *e2;
}
```

排序时调用 g_bub_sort(a, len, sizeof(int), less)即可实现对整型数组 a 从小到大排序。

排序时调用 g_bub_sort(a, len, sizeof(int), greater)即可实现对整型数组 a 从小到大排序。



C08-05-函数指针应用：通用冒泡排序算法

◆（5）应用通用冒泡排序函数

```
int main()
{
    int i;
    char a[] = {"adfbdeadfgdeadgcdeaghc"};
    double b[] = {2, 6, 5, 1, 7, 10, 3, 9, 7, 8};
    g_bub_sort(a, strlen(a), sizeof(char), less);
    for (i = 0; i < strlen(a); i++)
        printf("%c", a[i]);
    printf("\n");
    g_bub_sort(b, sizeof(b) / sizeof(b[0]), sizeof(double), greater);
    for (i = 0; i < 10; i++)
        printf("%0.2f ", b[i]);
    printf("\n");
    return 0;
}
```

排序算法主框架 g_bub_sort 不用做任何修改，用户只需根据要排序的数据类型、排序目标（升序或降序）重写比较函数即可。

使用带有函数指针的库函数

◆快速排序库函数qsort（stdlib.h头文件）

```
//<stdlib.h>中提供的快速排序函数
//qsort：负责框架调用和给(*comp)传递所需参数
//根据(*comp)的返回值决定如何交换数组的成员；
//base：需要排序的数组首地址（void*指向任意类型的数组）；
//num：参与排序的数组元素的个数；
//wid：数组中每个元素所占用的字节数；
//comp：指向数组元素比较函数的指针，接收两个元素的地址
//(*comp)：函数负责比较两个元素，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
void qsort( void *base, size_t num, size_t wid,
      int (*comp)(const void *e1, const void *e2) );
```

C08-06：使用qsort排序

◆给定一个具有n个元素的double型数组，使用qsort()对数组元素按升序和降序排序

```
double a[100];//需要排序的数组
int n;//数组中实际元素的个数
…
//按升序排序
qsort(a, n, sizeof(double), rise_double);
// 按降序排序
qsort(a, n, sizeof(double), fall_double);
```

```
//升序排序，最基础的写法
int rise_double(const void *p1, const void *p2)
{
    double *d1=(double *)p1;//转换为目标类型
    double *d2=(double *)p2;//转换为目标类型
    if(*d1 < *d2) return -1;//根据大小返回值
    else if(*d1 > *d2) return 1;
    else return 0;
}
```

```
//降序排序，直接类型转换
int fall_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;
}
```

C08-06：使用qsort排序

◆下列写法可能存在一些错误

```
//不一定正确的写法，参数直接写目标类型（部分编译器不支持）
//建议参数采用const void *，在代码中转换为目标类型，更通用!
int fall_double(const double *p1, const double *p2)
{
  if ( *p1 >  *p2 ) return -1;
  if ( *p1 <  *p2 ) return 1;
  if ( *p1 == *p2 ) return 0;
}
```

```
//肯定错误的写法，思考为什么？
int rise_double(const void *p1, const void *p2)
{
    return (int)(*(double *)p1 - *(double *)p2);
}
int fall_double(const double *p1, const double *p2)
{
    return (int)(*p1 - *p2);
}
```

思考：如果数组元素是int类型，可否使用 return *(int*)p1- *(int*)p2?

C08-07：点集排序

◆C08-07：点集排序，已知一个N×2的二维 double 数组存储二维平面上 N 个点的坐标，每一行表示一个点，第一列的数表示x的坐标，第2列表示y坐标。使用 qsort 函数对这 N 个点先根据 x 坐标值从小到大排序，如果x坐标相同，则按照y坐标值从大到小排序

◆问题分析

✓用二维数组double pt[N][2]保存这 N 个点的坐标，每行代表一个点的 x 和 y坐标

✓使用qsort 函数的关键是定义满足需求的比较函数，考虑数组内存空间中数据构成和含义：

- pt中每一行两个 double元素代表一个坐标，将这2个double看作一个单元，那么数组 pt 中有 N 个连续单元
- 明确比较函数中元素指针指向的内容：要把2个double元素看作一个排序单元，因此元素指针应该指向 2 个连续的 double 元素空间

double pt[N][2]

1.1	2.2
1.3	1.3
2.6	4.6
0.2	1.7
…	…

一行两个元素看成一个排序单元

C08-07：点集排序

◆使用快速排序

```
#include <stdio.h>
#include <stdlib.h>
#define LINE 6
#define ROW 2
int cmp(const void *d1, const void *d2);
int main()
{
    double pt[LINE][ROW] = {{1.0, 2.0}, {3.0, 4.0}, {2.5, 9.1},
                            {3.5, 3.0}, {3.0, 7.0}, {3.5, 2.0}};
    qsort(pt, LINE, 2 * sizeof(double), cmp);
    for (int i = 0; i < LINE; i++)
        for (int j = 0; j < ROW; j++)
            printf(j == 1?"%.1f\n" : "%.1f ", pt[i][j]);
    return 0;
}
```

C08-07：点集排序

◆编写比较函数

```
int cmp(const void *d1, const void *d2)
{
    double *pt1, *pt2;
    pt1 = (double*)d1;  //转换成目标类型
    pt2 = (double*)d2;  //转换成目标类型
    if (pt1[0] < pt2[0]) return -1;//0号位置为x坐标，升序
    else if (pt1[0] > pt2[0]) return 1;
    else{                       //x坐标相等，则比较y坐标，降序
        if (pt1[1] < pt2[1]) return 1;
        else if (pt1[1] > pt2[1]) return -1;
        else return 0;
    }
}
```

使用带有函数指针的库函数

◆折半查找函数bsearch（stdlib.h）

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

C08-08：查询质数表

◆查质数表：给定一个按升序排列的包含N个质数的质数表，通过查表判断一个正整数是否是质数

```c
#include <stdio.h>
#include <stdlib.h>
#define N 1000
int primes[N];
int main(){
    int n;
    //生成包含N个元素的质数表
    init_primes(primes, N);
    scanf("%d", &n);
    //在质数表中查询n是否是质数
    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);
}
```

```c
//比较函数，用于查找质数表中的元素
int comp_int(const void *p1, const void *p2)
{
    int *e1=(int *)p1;
    int *e2=(int *)p2;
    if(*e1>*e2) return 1;
    else if(*e1<*e2) return -1;
    else return 0;
}
```

待查元素指针　查找数组　数组大小　元素大小　比较函数

如何判断某个数是否为质数

◆最基本的判断质数的函数

```
//判断n是否为质数
int isPrime (int n)  {
  int i;
  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

算法改进：优化质数判断

◆假设已经有初始质数表，在此基础上判断某个数是否为质数

```
//利用已有的质数表primes，判断n是否为质数
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;
}
```

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≤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≤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。

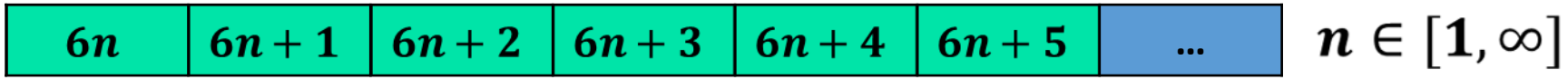
② 故，假设 n = a×p，p为质数，且p必须大于sqrt(n) 。
那么 a < sqrt(n),并且 a 不能是质数，否则就跟①矛盾。
a是合数可分解：令a= b×q，这里q是质数，且q<sqrt(n) 。

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!

高效的质数表生成算法

◆改进的质数判断函数和高效质数表初始化方法

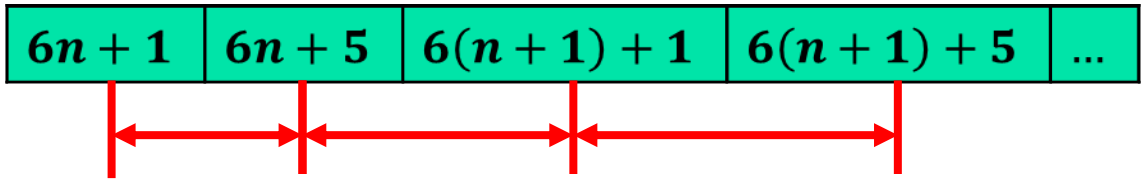
$$\{6,7,8,9,\dots,\infty\} \Leftrightarrow \bigcap_{n=1}^{+\infty}\{6n,6n+1,6n+2,6n+3,6n+4,6n+5\}$$



$n \in [1,\infty]$

× ✓ × × × ✓

2倍数 2倍数 3倍数 2倍数



$n \in [1,\infty]$

4 2 4 ...

$n = 1 \Rightarrow 6n + 1 = 7(\text{start})$

$STEPS = \{4,2,4,2,\dots\}$

采用4/2步长，不需要在isPrime里模2, 3!

快多少?

高效的质数表生成算法

```
//构造≤Q的质数表（Q>=5）
void init_primes(int primes[], int Q)
{
    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
    }
}
```

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 …**

关于使用指针的原则总结

- 地址的有效性：永远要清楚每个指针指向了哪里
- 空间的有效性：永远要清楚指针指向的是什么，确保通过指针间接访问变量的正确性

总结

- ◆数组指针
  - ✓数组与数组指针
  - ✓二维数组与指向一维数组的指针
- ◆多重指针
  - ✓指针的指针
- ◆指针数组
  - ✓数组的成员为指针、命令行参数
- ◆函数指针
  - ✓使用快速排序、折半查找库函数
  - ✓快速素数生成算法