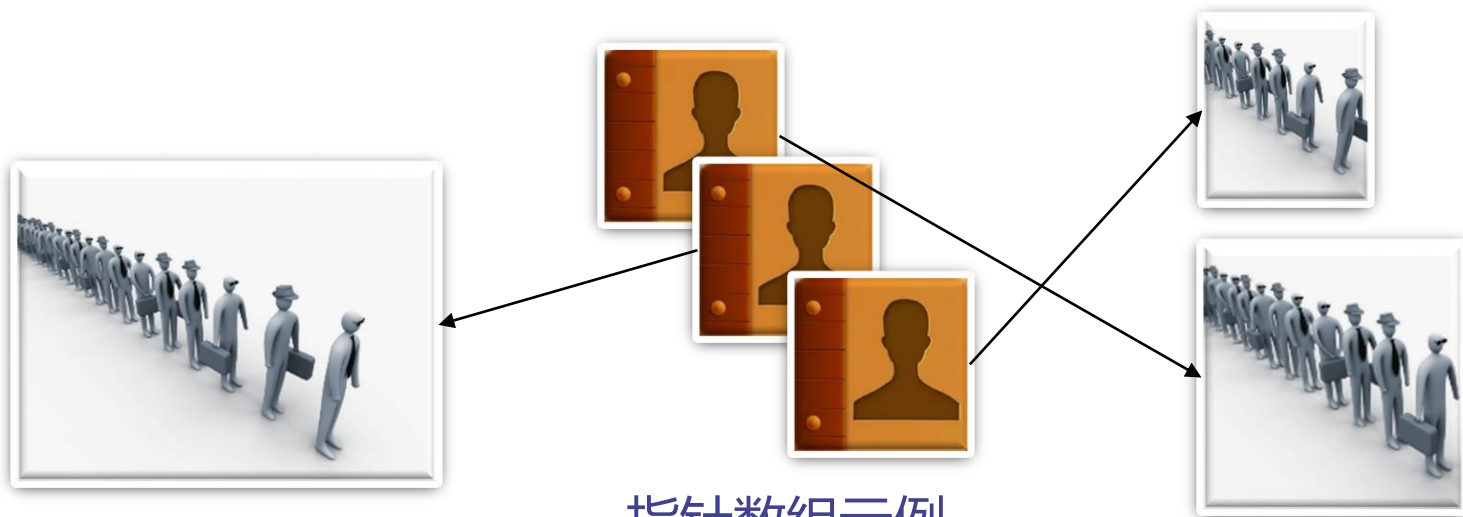


# C语言高级篇

## 第八讲

# 指针进阶

advanced pointer



指针数组示例

# 第八讲 指针进阶

---

## 本讲结构

1. 指针与数组
2. 数组类型与数组指针
3. 多维数组与数组指针
4. 多重指针
5. 指针数组
6. 命令行参数
7. 函数指针
8. 本章小结

## 8.1 指针与数组

指针与数组的关系非常密切，具体体现在以下两点

1. 数组名出现在表达式中，可以隐式类型转换为指向数组首元素的指针。

例如 int 型数组名可以看作指向数组首元素的 int\*型指针，因此数组名可以参与指针运算，可以赋值给同类型的指针变量。注意，数组名是右值，不能对它赋值或改变它的值，如：

```
float a[10];  
float *p = a; // p 指向数组 a 的第一个元素  
p++; // p 指向数组 a 的第二个元素  
a = p + 1; // 编译错误：数组名不能被赋值  
a++; // 编译错误：数组名是右值，不能用于++运算
```

## 8.1 指针与数组

指针与数组的关系非常密切，具体体现在以下两点

2. 可以通过“指针+偏移量”的方式访问数组的各个元素。

指针 p 一旦指向数组中的元素，可以像数组一样通过整型量下标访问对应的元素，下标表示相对于当前指针位置以元素为单位的偏移量。

```
float a[10];  
float *p = a + 3; // p 指向 a 的第 4 个元素 a[3]  
p[0] = 1.0f; // 等价于 a[3] = 1.0f  
p[-1] = 2.0f; // 等价于 a[2] = 2.0f  
p[1] = 3.0f; // 等价于 a[4] = 3.0f  
p[-4] = 0.0f; // 等价于 a[-1] = 0.0f。语法没有错误，但出现了数组越界访问
```

## 8.1 指针与数组

**例 8-1 指针遍历数组** 采用指针而非下标的形式遍历一个数组。遍历的含义是访问数组中的每个元素。

问题分析：基本做法是定义一个指针指向数组的首元素，然后在循环体内判断指针位置，如果没有超出数组范围则访问指针指向的内容，之后指针加 1 向后移动一个元素。

```
#include <stdio.h>
#define ArrayLen(x) (sizeof(x) / sizeof(x[0]))
int main()
{
    double d_arr[] = {1, 2, 3, 4, 5, 6, 7, 8};
    double *pd = NULL;
    for (pd = d_arr; pd < d_arr + ArrayLen(d_arr); pd++)
        printf("%.2f\n", *pd);
    return 0;
}
```

## 8.1 指针与数组

- 数组合法拥有它元素的所有空间，或者说元素空间已经确保。指针变量只拥有它自己的空间，它指向的空间不一定是合法的（例如野指针）

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

```
int *p;
```

- 第一种数组定义，分配5个连续的int型内存空间，能够容纳数组array的所有元素；
  - 第二种指针定义，只分配sizeof(int\*)，通常是4或8个字节给指针变量p，只存储一个地址。
- 数组拥有存储数据的空间；而指针变量，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);
```

## 8.2 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**。

例如 `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。

**数组名和指针在使用上很相近，所以概念很容易混淆！**

## 8.2.1 数组类型

- 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，属于聚合体类型。
- 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。

```
double d_arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};  
double *pd = d_arr;  
printf("sizeof d_arr is %d\n", sizeof(d_arr));  
printf("sizeof pd is %d\n", sizeof(pd));
```

程序输出

```
sizeof d_arr is 80  
sizeof pd is 8
```



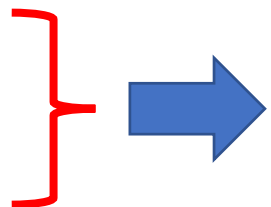
## 8.2.2 数组指针

- 数组既然是数组类型的数据实体，当然也可以用取址符&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，数组指针变量的定义如下：

<类型> (\*<变量名>)[<元素个数>];

### 例 8-2 数组指针定义示例

```
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 char (\*)[30]，分别指向数组 a, b, c（而不是指向数组的第一个元素）。

## 8.2.2 数组指针

数组指针的本质还是指针，只不过它指向的数据实体不是单一类型，而是数组类型。关于指针的所有运算都适用于它。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。

### 例 8-3 数组指针的内存占用示例

```
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

## 8.2.2 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：

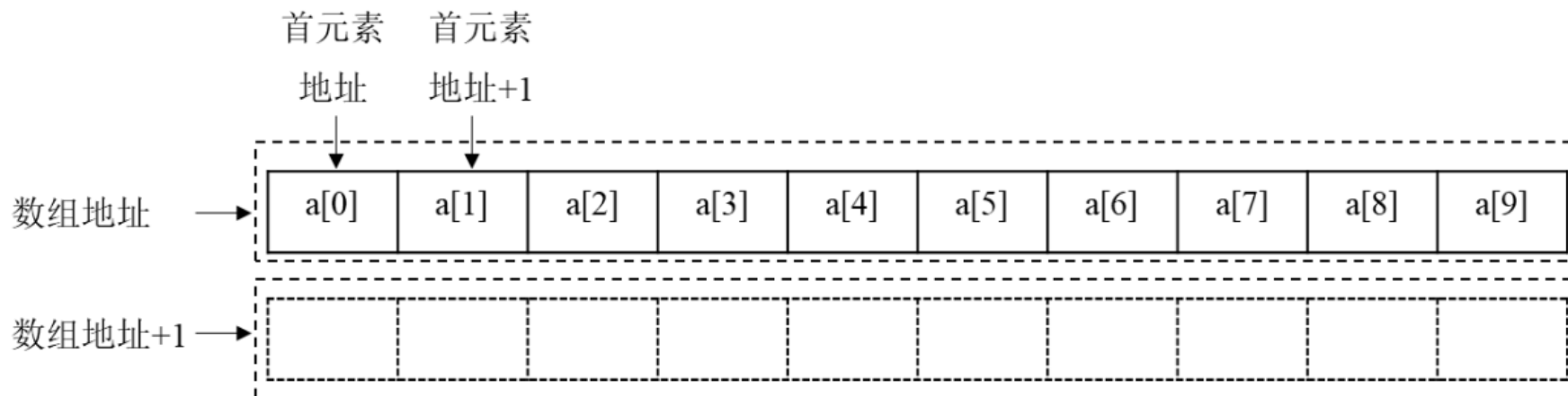
```
float a[10];  
printf("%p : %p\n", a, &a);  
return 0;
```

程序输出



什么？a的值和a的地址竟然一样！

000000000061FE20 : 000000000061FE20



如果 `a` 是一个数组 `a` 和 `&a` 做表达式时具有相同的值，但却具有不同的类型。因为类型不同，加 1 时指针指向的位置也不同：`a+1` 指向数组 `a` 的第二个元素；`&a+1` 指向数组 `a` 最后一个元素之后的位置。

## 8.2.2 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。

```
double d[10] = {0};  
double (*pd)[10] = &d;  
double *p;  
p = *pd;  
*(p+1) = 1.2;  
printf("%f \n", d[1]);  
//输出1.200000
```

中文习惯将主语放最后，修饰它的定语放前面。数组指针，主语是指针，因此它的本质是指针，是指向一个数组（而不是指向数组元素）。指针数组（详见第 8.5 节），主语是数组，因此它的本质是数组。按这样的方式区分，便于记忆，也便于理解。

## 8.3 多维数组与数组指针

- 大于一维的数组称为多维数组，常用的多维数组包括二维数组、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，直到数组元素变为单一类型。

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

$a$  {  $a[0]$  {  $a[0][0]$   
 $a[0][1]$   
 $a[0][2]$   
 $a[1]$  {  $a[1][0]$   
 $a[1][1]$   
 $a[1][2]$

数学上，可以把a看成是一个集合：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} };
```

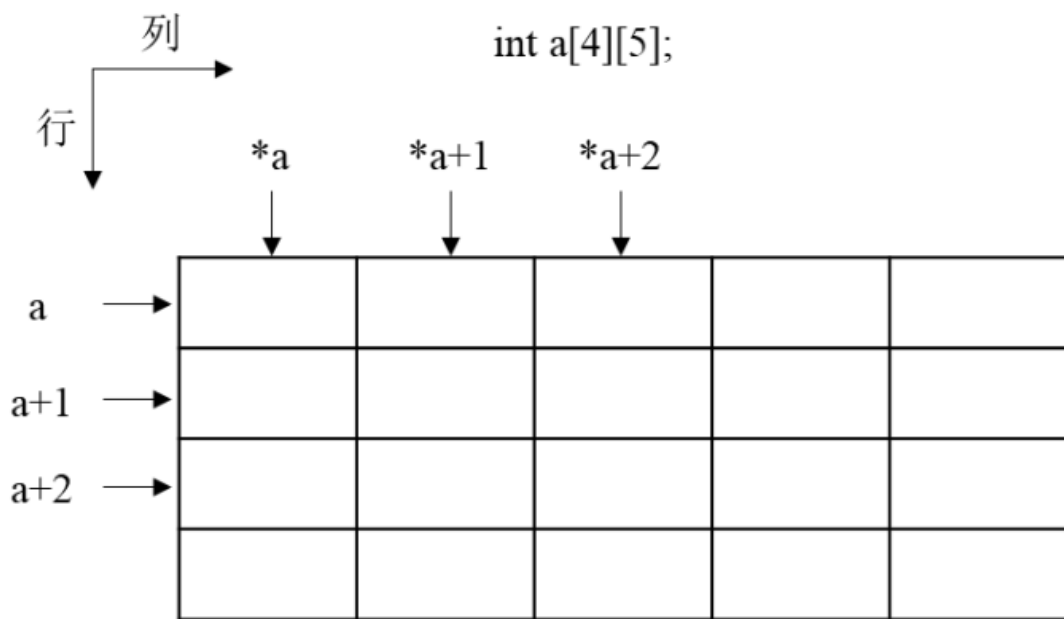
	$a$		
$a+0 \rightarrow$ $\&a[0]$	$a[0][0]$	$a[0][1]$	$a[0][2]$
$a+1 \rightarrow$ $\&a[1]$	$a[1][0]$	$a[1][1]$	$a[1][2]$

逻辑上，可看成2行3列，共6个元素。

多维数组可以看成数组的数组

## 8.3.1 二维数组的深入理解

二维数组可以看作数组的数组。例如 `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（按习惯，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  $4 \times 5$  的矩阵，共 4 行，每行 5 个元素。

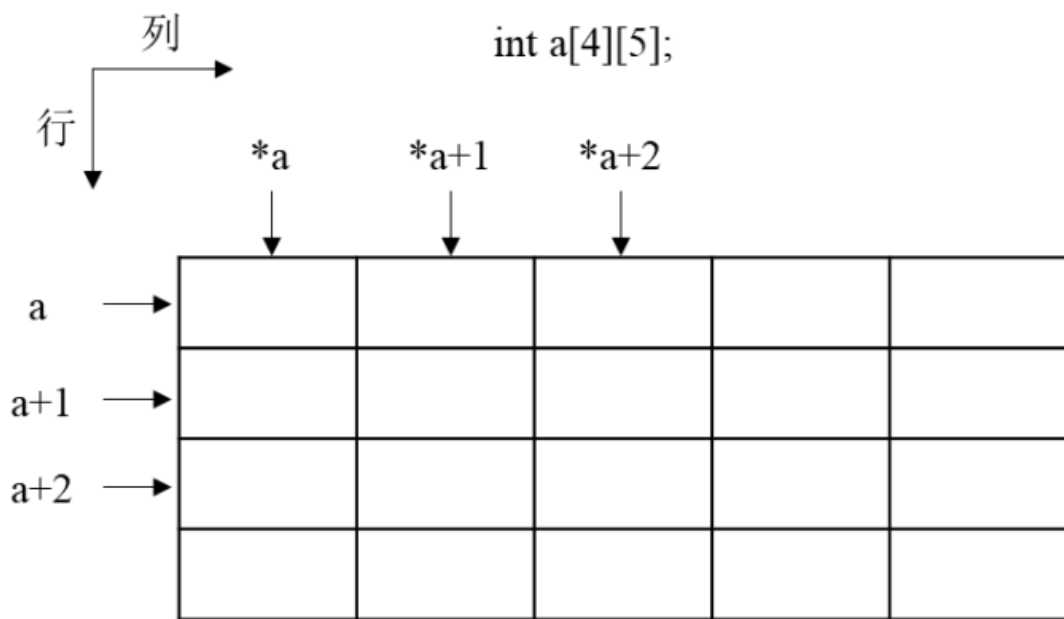


- 按照隐式类型转换规则，数组名 `a` 作为表达式可以看作指向首元素的指针，而二维数组的元素为一维数组，因此 `a` 作为表达式具有数组指针类型 `int (*)[5]`。
- 它的值指向二维数组中的第 1 个数组，也就是第 1 行；`a + 1` 则指向二维数组中的第 2 个数组，即第 2 行，依次类推。

**二维数组名 `a` 可看作行指针（数组指针）**

## 8.3.1 二维数组的深入理解

二维数组可以看作数组的数组。例如 `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（按习惯，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  $4 \times 5$  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]`

**对行指针进行解引用，得到列指针（元素指针）**

## 8.3.1 二维数组的深入理解

**例 8-5 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 的每一个元素，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。排序结束后，a 中的字符串数据将变得有序。

```
#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
```

见下页



## 8.3.1 二维数组的深入理解

**例 8-5 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。

[接上页](#)

```
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;
}
```

本例中，冒泡排序的元素交换环节需要进行三次字符串的整体数据复制，效率较低。第 8.5 节给出了一个更高效的算法。

## 8.3.2 二维数组作为函数参数的本质

二维数组作为函数参数时，函数声明

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

表示函数 F 接受一个 int [4][5]的二维数组作为参数。注意上述函数声明中 a 的行数 4 可以不用指定（编译器会忽略），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，这就是为什么列数不能省略的原因，因为数组指针需要数组长度这个信息。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。

## 8.3.2 二维数组作为函数参数的本质

**例 8-6 方阵相乘（数组指针版）** 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

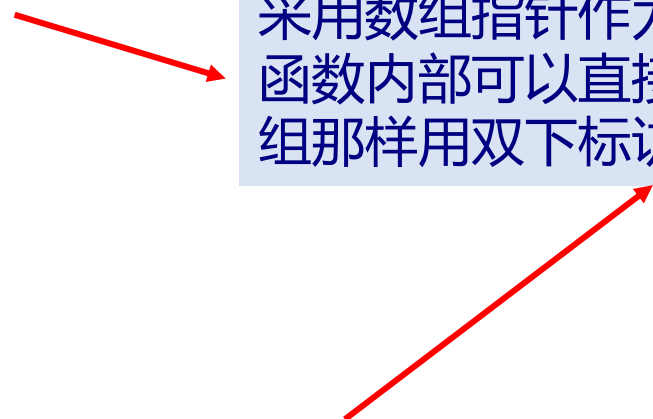
## 8.3.2 二维数组作为函数参数的本质

**例 8-6 方阵相乘（数组指针版）** 写一个函数实现 4x4 矩阵相乘，4x4 矩阵用二维 double 数组表示。

```
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];
        }
}

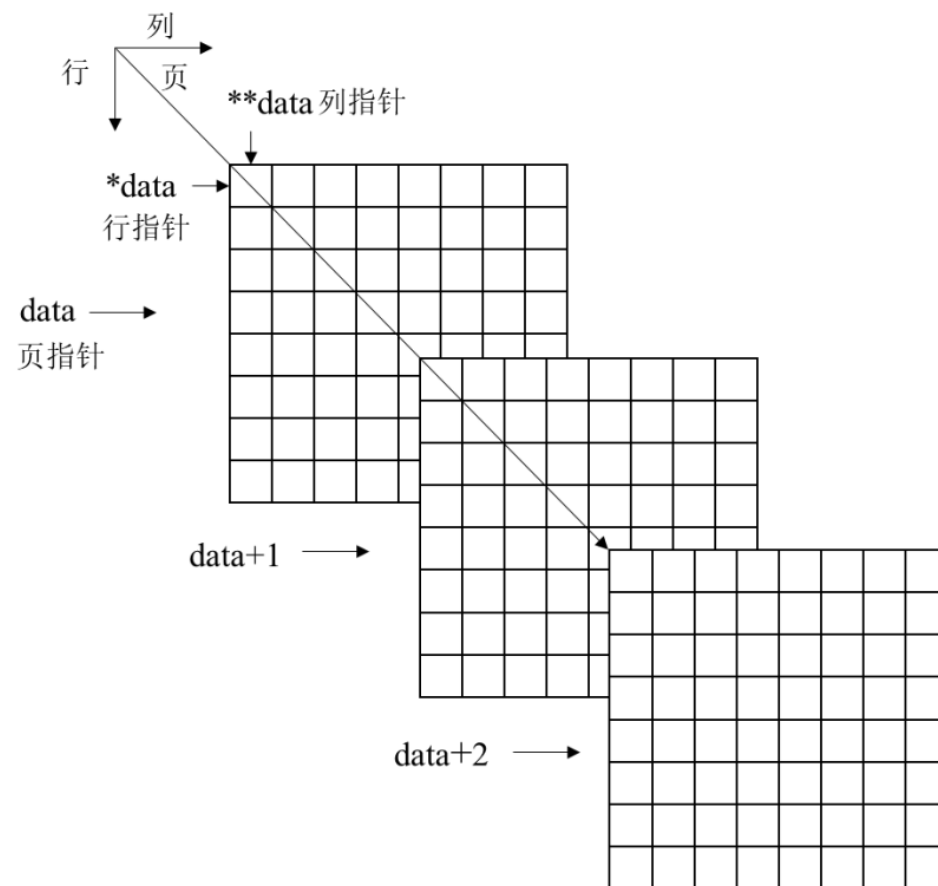
void mat_print(const double (*c)[4])
{
    int i, j;
    for (i = 0; i < 4; i++)
        for (j = 0; j < 4; j++)
            printf(j == 3 ? "%-10.2f\n" : "%-10.2f ", c[i][j]);
}
```

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



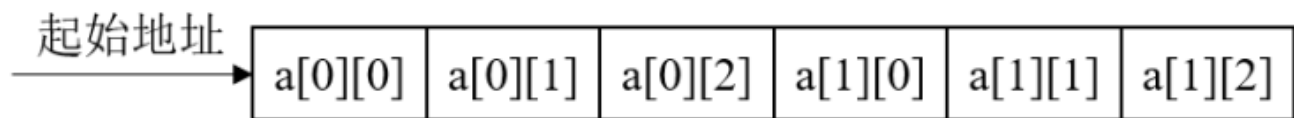
## 8.3.3 高维数组的深入理解

- 一个三维数组 `int data[D][M][N]`，可以将它看作一个长度为 `D` 的数组，数组的每个元素是 `int [M][N]` 的二维数组。
- `data` 作为表达式可以看作指向二维数组 `int [M][N]` 的指针，具有数组指针类型 `int (*)[M][N]`。
- 如果将三维数组中每个 `int [M][N]` 的二维数组视为一页，则 `data` 可称为页指针。
  - 对页指针解引用，`*(data+k)` 成为行指针，指向第 `k+1` 页的第 1 行；
  - 再解引用，`*(*(data+k)+i)` 成为列指针，指向第 `k+1` 页的第 `i+1` 行的第 1 个元素；
  - 最后解引用，`*(***(data+k)+i+j)` 对应三维数组中第 `k+1` 页中第 `i+1` 行的第 `j+1` 个元素，即： `data[k][i][j]`。



## 8.3.4 多维数组下标与线性地址偏移

- 不管多少维的数组，它们的本质都是内存中一段连续的内存空间，元素从数组的起始地址开始连续存放。多维数组元素的下标与该元素的存储位置（以数组元素为单位相对数组起始地址的偏移量）之间存在一个关系映射。
- 对于一维数组很简单，下标即为偏移量。对于多维数组的下标，C语言采用“行优先存储”（row-major）的规则，即优先存储最右侧下标对应的元素。



- 在“行优先存储”规则下，一个  $M \times N$  的二维数组元素  $a[i][j]$  的下标与元素位置偏移量  $loc$  之间的关系为： $loc = i \times N + j$

即  $a[i][j]$  元素的存储地址为  $\&a[0][0] + loc$ 。通过上面的公式，可以将二维数组的二维下标转换成相对于数组首地址的线性偏移量（以数组元素为单位），从而直接访问该元素。

## 8.3.4 多维数组下标与线性地址偏移

“行优先存储”的本质其实是“最右边的下标变化最快 (rightmost subscript varies fastest)”，即在内存中从起始地址依次遍历数组的每个元素时，相邻元素中越靠右侧的下标变化的速度越快。与之相对的是“列优先存储” (column-major) 或“最左边的下标变化最快 (leftmost subscript varies fastest)”。Fortran, Matlab 等语言采用“列优先存储”。

**例 8-7 方阵相乘 (线性下标版)** 写一个函数实现  $n$  阶方阵相乘。在数组访问中使用线性下标。

问题分析： $n$  阶方阵既可用二维数组表示，用两个下标访问，也可用以“行优先存储”的一维数组表示。不管矩阵是二维数组的形式还是一维数组的形式，它的元素只要是“行优先存储”，就可以通过下标映射公式计算它相对于第一个元素地址的线性偏移量，从而对它进行访问。

## 8.3.4 多维数组下标与线性地址偏移

**例 8-7 方阵相乘（线性下标版）** 写一个函数实现  $n$  阶方阵相乘。在数组访问中使用线性下标。

```
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



## 8.4 多重指针

- **多重指针就是指向指针的指针。** 指针变量也是变量，它也要占用内存空间，指针变量的地址就是一个多重（二重）指针。同理，二重指针的地址是一个三重指针，以此类推。多重指针的定义和普通指针一样，都是 `<类型> * <变量名>`，但是其中 `<类型>` 是指针类型。

```
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）。

## 8.4 多重指针

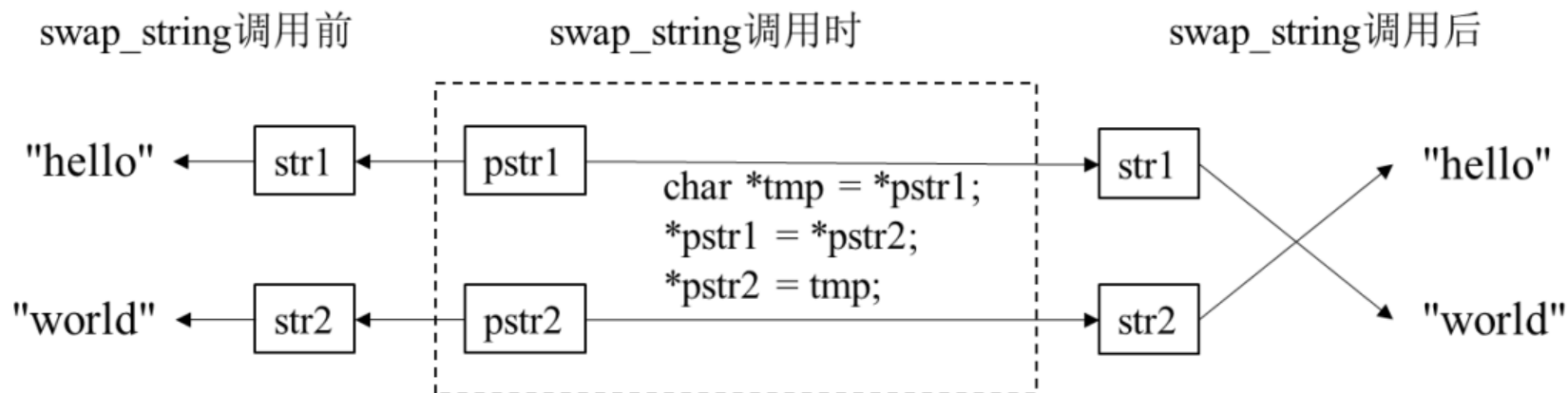
**例 8-8 交换字符串** 编写一个函数 `swap_string`，作用是交换两个字符串指针的指向。

问题分析： 函数的目的要交换两个指针的值，因此可使用二重指针作为形式参数去接收外部指针变量的地址，代码如下：

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

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

world, hello

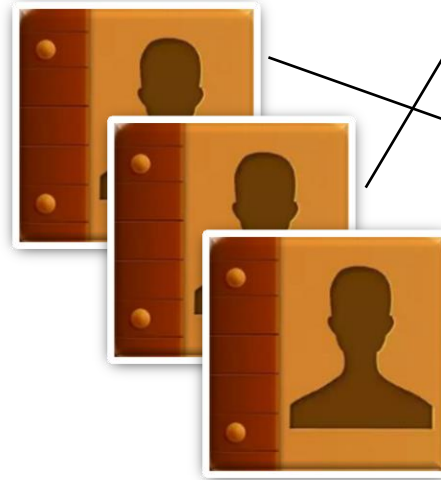


## 8.5 指针数组

- 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



## 8.5.1 指针数组

- 指针数组就是元素类型为指针的数组。例如：

```
int *piArr[10];  
char *pcArr[20];  
float *pfArr[30];  
double *pdArr[40];
```

分别定义了四个数组，数组元素类型分别是 int\*， char\*， float\*以及 double\*，数组内容均未进行初始化，未初始化的指针（野指针）不能访问其指向的内容。

- 指针数组可以在定义时初始化，如：

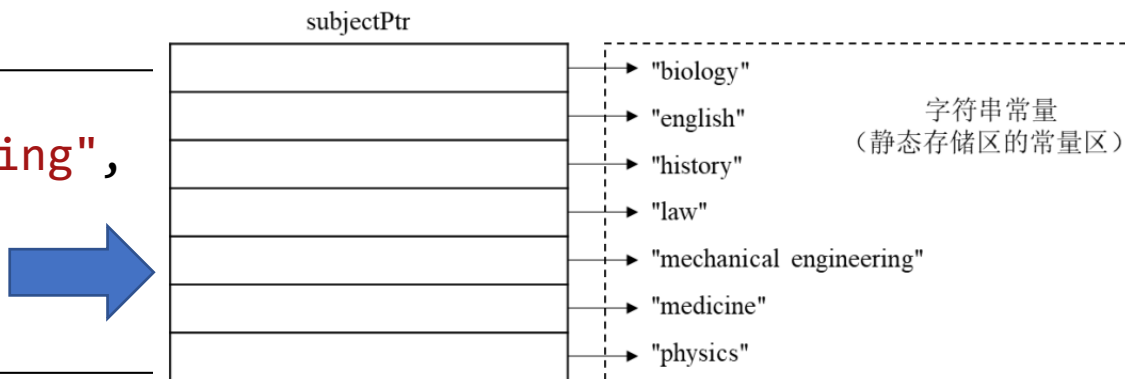
```
char *pcArr[] = {"I", "Love", "C", "Language"};
```

定义了一个长度为 4 的指针数组，其中 pcArr[0]指向字符串常量"I"； pcArr[1]指向字符串常量"Love"； pcArr[2]指向字符串常量"C"； pcArr[3]指向字符串常量"Language"。注意数组名 pcArr 作为表达式时可看作指向首元素的指针，即它具有二重指针类型 char \*\*。

### 8.5.1 指针数组

### 例 8-9 指针数组定义示例

```
#include <stdio.h>
#define LEN 23
int main()
{
    int i;
    char *subjectPtr[] =
    {
        "biology",
        "english",
        "history",
        "law",
        "mechanical",
        "medicine",
        "physics"
    };
}
```

[illegible][illegible]

## 8.5.1 指针数组

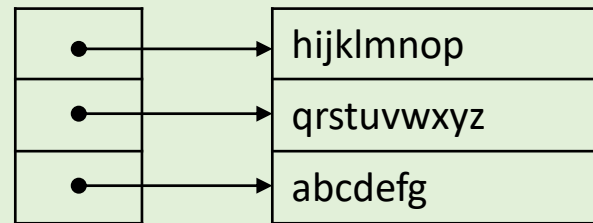
- 指针数组常被用作**数据索引**，以加快数据**定位、查找、交换和排序**等操作的速度。
- 在一些文字处理程序中，数据一般以“行”为单位保存在二维数组中，在数据处理的过程中，对各行位置的交换，以及整行内容的删除和新行的添加是频繁进行的操作（**计算代价很大**）。为提高程序的运行速度，往往使用指针数组作为实际数据的索引。

排序前

hijklmnop
qrstuvwxyz
abcdefg

直接对二维字符数组排序

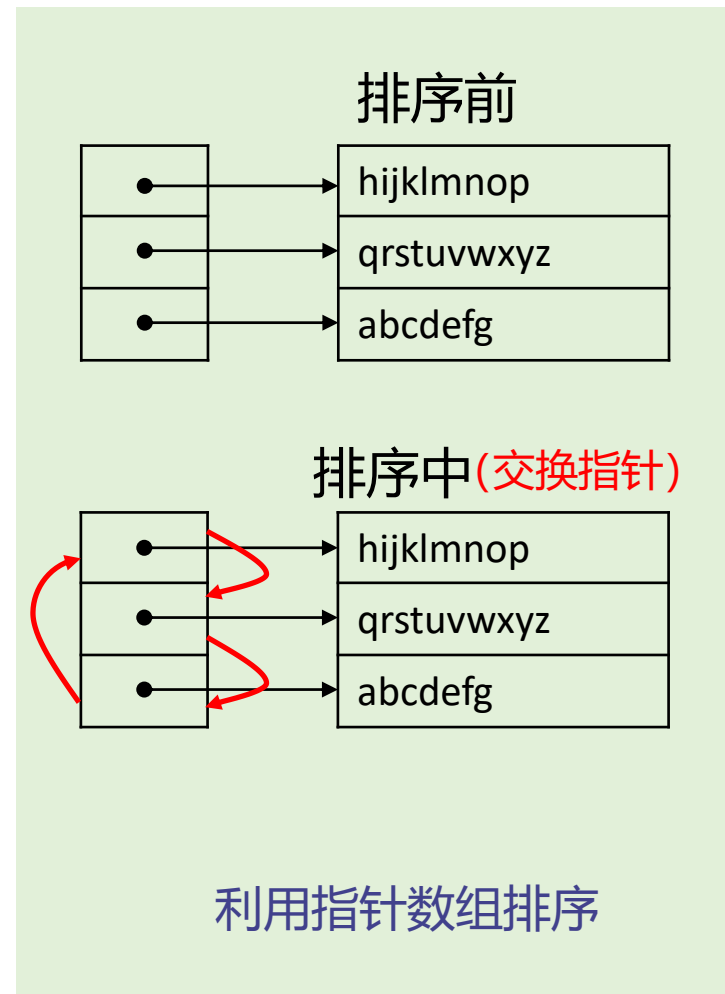
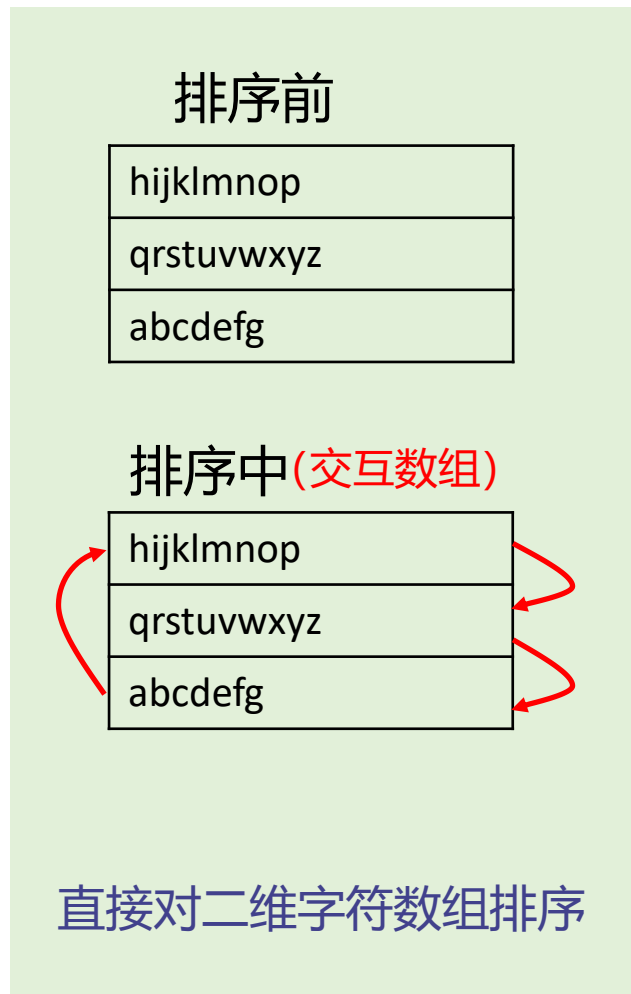
排序前



利用指针数组排序

## 8.5.1 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。



## 8.5.1 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

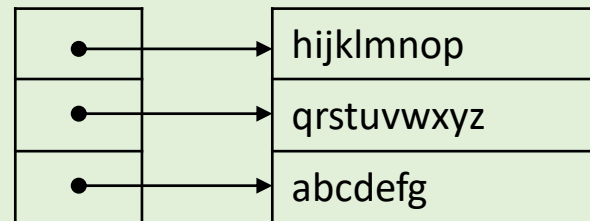
hijklmnop
qrstuvwxyz
abcdefg

排序后

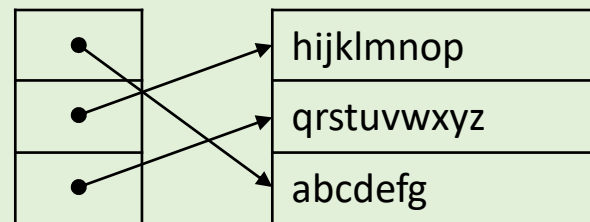
abcdefg
hijklmnop
qrstuvwxyz

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



## 8.5.1 指针数组

### 例 8-10 按字符串长度排序（指针数组版） 题意见例 8-5

问题分析： 定义一个全局二维数组 `char a[100][1002]` 保存输入的所有行。定义一个整型数组 `int lens[100]` 和一个指针数组 `char *lines[100]` 分别保存每行的长度和指向该行字符串的指针。输入结束后，对 `lens` 数组进行冒泡排序并同时相应交换 `lines` 中对应的指针。排序结束后数组 `lines` 中按字符串从短到长依序存放各个字符串的指针，但数组 `a` 里存的数据，即原始数据没有发生任何改变（没有对数组 `a` 进行写操作）。

## 8.5.1 指针数组

### 例 8-10 按字符串长度排序（指针数组版） 题意见例 8-5

```
#include <stdio.h>
#include <string.h>
char a[100][1002]; // save lines
int main()
{
    int i, j, k = 0, tmp;
    char *lines[100], *temptr;           // pointer to each line
    int lens[100];                       // keep line's length
    while (fgets(a[k], 1000, stdin) != NULL) // read all lines
    {
        lines[k] = a[k];
        lens[k] = strlen(lines[k]);
        k++;
    }
}
```

这一句话编译器做了什么类型的隐式转换？

## 8.5.1 指针数组

### 例 8-10 按字符串长度排序（指针数组版） 题意见例 8-5

```
for (i = 1; i < k; i++) // bubble sorting
    for (j = 0; j < k - i; j++)
    {
        int dicflag = (lens[j] == lens[j + 1]) && (strcmp(lines[j], lines[j + 1]) > 0);
        if (lens[j] > lens[j + 1] || dicflag)
        {
            tmp = lens[j];
            lens[j] = lens[j + 1];
            lens[j + 1] = tmp;
            tmptr = lines[j];
            lines[j] = lines[j + 1];
            lines[j + 1] = tmptr;
        }
    }
for (i = 0; i < k; ++i)
    fputs(lines[i], stdout);
return 0;
}
```

请对比例8-5，说出本例的优点

## 8.5.2 指针数组与二维数组

- 指针数组和二维数组，语法上都可以用双下标访问元素。例如有如下定义：

```
int *a[12];  
int b[12][2];
```

则 `a[0][0]`和 `b[0][0]`在语法上都正确，具有 `int` 类型。

- `b[0][0]`一定是合法的 `int` 型变量，因为它属于二维数组 `b` 的元素；`a[0][0]`则不一定是合法的 `int` 型变量，因为 `a[0]`可能是野指针，或指向非法的内存空间。
- 这种可能的逻辑错误在编译阶段不会暴露出来，但是在运行时，一旦指针数组元素没有指向合法的空间，进行类似的下标访问时，轻则程序崩溃，重则留下了巨大的逻辑错误隐患。只有当 `a[i]`指向一个一维数组时，`a[i][j]`才是合法的变量。

## 8.5.2 指针数组与二维数组

### 例 8-11 指针数组错用示例

```
#include <stdio.h>
void print_mat4x4(int *a[4])
{
    int i, j;
    for (i = 0; i < 4; i++)
        for (j = 0; j < 4; j++)
            printf(j == 3 ? "%d\n" : "%d ", a[i][j]);
}
int main()
{
    int array[4][4] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};
    print_mat4x4(array);
    return 0;
}
```

程序的本意，是写一个 print\_mat4x4 函数，打印一个 4x4 的 int 型二维数组。

如果运行这个程序，会发生运行时错误，程序崩溃了。

思考题：如何修改？

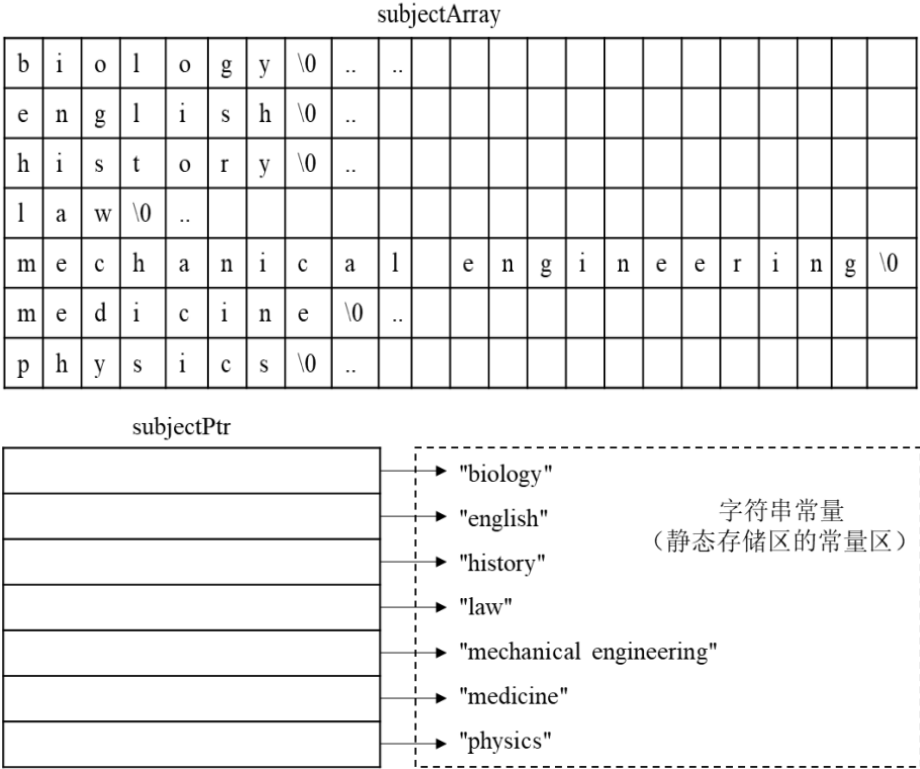
# 8.5.2 指针数组与二维数组

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**。

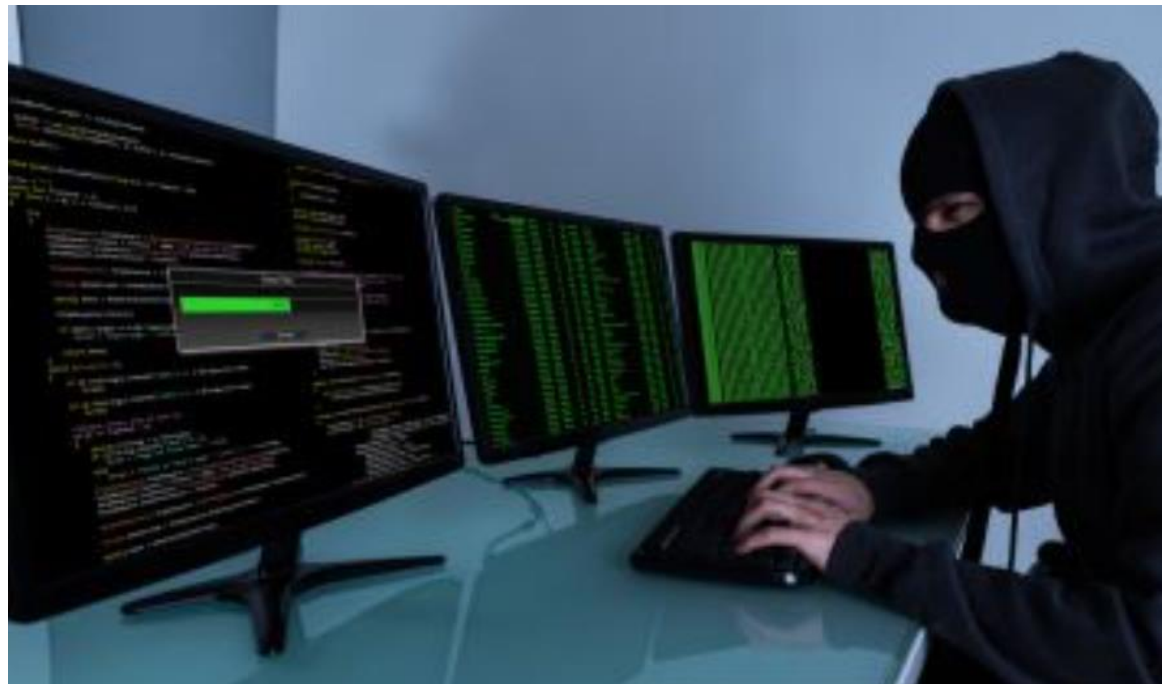
(3) 二维数组中全部元素的存储空间是连续排列的；而在指针数组中，只有**各个指针的存储空间是连续排列的**，其所指的数据元素的存储顺序取决于存储空间的分配方法，并且常常是不连续的。



例 8-9 中的指针数组和二维数组

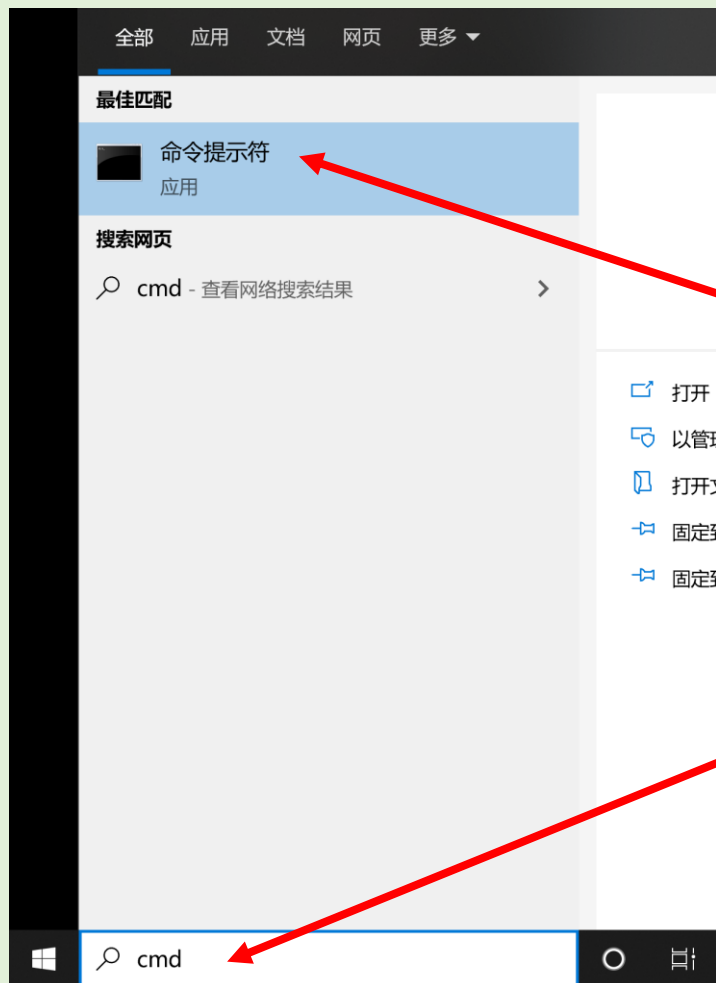
## 8.6 命令行参数\*

- 当程序执行时，程序中的 main 函数由操作系统调用，操作系统可能会传进来一些参数。
- 例如用过 Linux 命令 ls 的读者都知道它的作用是列出当前目录下的子目录或文件。ls 命令是可以带参数的，例如 ls -a、ls -l、ls -la 等，这些 ls 后的参数称为命令行参数。
- 命令行参数将以字符串的形式传进程序的 main 函数中。如果程序要接收命令行参数，需要在 main 函数定义中指定参数。



高手都喜欢用命令行来操作电脑

## 8.6 命令行参数\*



2. 按回车键或  
单击“命令提  
示符”应用

1. 在windows  
的搜索框输入  
cmd

```
C:\> 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>
```

```
C:\> 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>_
```

通过命令行参数，可以将参数传递给主函数



## 8.6 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

## 8.6 命令行参数\*

### 例 8-12 命令行参数程序示例

```
#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;
}
```

- 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)。

- 假设上述程序编译后生成的可执行程序名为 c8\_12, 在命令行 (windows 下的 cmd 窗口或 linux 系统的 shell) 上输入 (不包括\$, 它是 shell 提示符): `$ c8_12 tell me you love c`
- 程序执行时, main 函数的参数值如下: argc 等于 6, 因为输入的命令有 6 个字符串; argv[0] 指向字符串 "c8\_12"; argv[1]指向字符串 "tell"; argv[2]指向字符串 "me"; argv[3]指向字符串 "you"; argv[4]指向字符串 "love"; argv[5]指向字符串 "c"。

## 8.7 函数指针

除了变量、常量、数组等，指针指向的数据实体还可以是函数，这种类型的指针称之为函数指针。函数名的本质是函数体在代码区的地址，因此可以将一个函数名赋值给同类型的函数指针。

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

## 8.7.1 函数指针

函数指针变量定义的语法如下：

〈函数返回类型〉 (\*〈变量名〉) (〈参数类型说明〉);

其中〈参数类型说明〉是用逗号隔开的形参类型，或是 void（也可以省略），例如：

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

- 第 1 行，定义了一个函数指针 funPtr1，它所指向的函数原型为：double func (int, int)。
- 第 2 行，定义了一个函数指针 funPtr2，它所指向的函数原型为：void func (int, int, int)。
- 第 3 行，定义了一个函数指针 funPtr3，它所指向的函数原型为：int func (double, char\*)。
- 第 4 行，定义了一个函数指针 funPtr4，它所指向的函数原型为：int \* func ()。

形参类型后也可跟参数名，例如 int (\* funPtr1) (int x, int y)，但编译器会忽略参数名。

## 8.7.1 函数指针

- 函数指针应该指向与函数原型和函数指针定义一致的函数
- 通过将函数名赋值给函数指针，让它指向某个具体的函数
- 一旦函数指针指向一个具体的函数，就可以通过函数指针间接调用它所指向的函数。

```
#include <stdio.h>
int sum(int a, int b)
{
    return a + b;
}
int main()
{
    int (*funPtr)(int, int) = sum;
    int x = 3, y = 4;
    printf("%d + %d = %d\n", x, y, funPtr(x, y));
    return 0;
}
```

原型要匹配

将函数名赋值给函数指针

通过函数指针调用函数，也可写为：(\*funPtr)(x, y)

## 8.7.2 函数指针的用法与意义

- 函数指针的用法很简单，通过把函数名赋值给它让它指向该函数，再使用和函数调用完全一样的语法通过它间接调用原函数。那为什么不直接调用原函数非要间接转个弯呢？
- 函数指针的主要用途是将算法流程和其中某些具体实施动作分离，设计出通用、可扩展、可重用的计算框架，在计算框架下具体的函数动作由函数指针传递进来。计算框架是静态的，因为它的算法是确定的；函数指针是动态的，它可以在计算框架调用时指向不同的函数实体来完成具体的动作。
- 举个例子，排序算法可以看作是一个计算框架，对于每一种排序算法（例如冒泡排序、选择排序、插入排序、快速排序）来说，其核心的流程是确定的，不因排序对象是谁而改变。但是排序对象的类型有多种，比如有对 int 型数据排序的需求，也有对 double 型数据排序的需求，只要给出顺序的定义还可以对任意类型的数据进行排序（例如字符串的字典序，人按照身高/胖瘦/颜值来排序）。为每一种数据类型去实现特定的排序算法既不现实也不优雅。需要将抽象的算法从实现中剥离出来

## 8.7.2 函数指针的用法与意义

**例 8-13 通用冒泡排序函数** 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，具体的数组元素比较行为通过函数指针实现。

➤ 因为数组类型不确定，所以采用通用型指针 `void*` 作为函数接口，设计通用冒泡排序函数原型为：

```
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）。

## 8.7.2 函数指针的用法与意义

**例 8-13 通用冒泡排序函数** 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，具体的数组元素比较行为通过函数指针实现。

(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




## 8.7.2 函数指针的用法与意义

**例 8-13 通用冒泡排序函数** 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，具体的数组元素比较行为通过函数指针实现。

(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。

## 8.7.2 函数指针的用法与意义

**例 8-13 通用冒泡排序函数** 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，具体的数组元素比较行为通过函数指针实现。

(3) 比较函数示例（整数型升序）

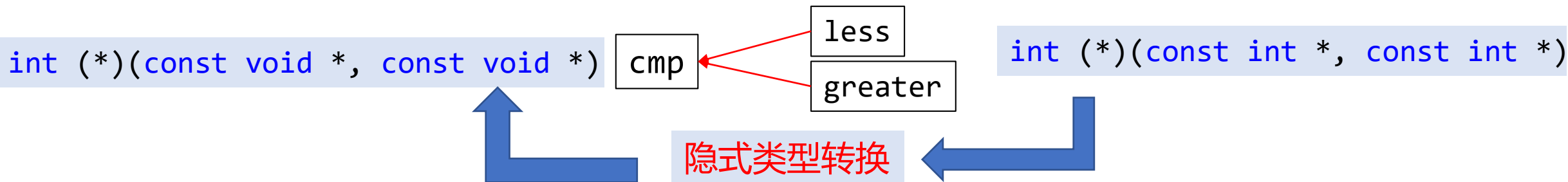
```
int less(const int *e1, const int *e2)
{
    return *e1 < *e2;
}
```

(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` 从大到小排序。



## 8.7.2 函数指针的用法与意义

**例 8-13 通用冒泡排序函数** 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，具体的数组元素比较行为通过函数指针实现。

(5) 通用冒泡排序函数的具体应用

```
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("%.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。

## 8.7.2 函数指针的用法与意义

**例 8-13 通用冒泡排序函数** 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，具体的数组元素比较行为通过函数指针实现。

- 比较函数需要满足**严格有序**条件。即，设 `comp(e1, e2)` 是一个比较函数，则它必须满足：
  - ① 若 `e1` 先于 `e2` 则 `comp(e1, e2)` 的值为真；
  - ② 若 `e1` 后于 `e2` 则 `comp(e2, e1)` 的值为真；
  - ③ 若 `e1` 和 `e2` 顺序相当，则 `comp(e1, e2)` 和 `comp(e2, e1)` 的值同时为假。
- 可以验证，`less` 和 `greater` 函数均满足严格有序条件。否则，可能造成不稳定排序。如果将 `less` 函数中的小于号写成小于等于，则不满足严格有序条件中的第③条。在进行冒泡排序时，如果 `j` 元素和 `j+1` 元素相等，`g_bub_sort` 函数在调用 `less` 函数会返回真，从而进行元素值的交换。这显然是一种**不稳定排序**。

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

不满足严格有序条件

## 8.7.3 使用函数指针的库函数

### (1) qsort() 快速排序函数（标准库函数）

包含头文件<stdlib.h>

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

- base: 指向所要排序的数组的指针（void\*指向任意类型的数组，准确地说是指向数组的首位置）；
- num: 是数组中元素（作为整体进行排序的单元）的个数；
- wid: 是每个元素所占用的字节数；
- cmp: 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位置的指针，const表示指针指向的内容是只读的，在cmp所指向的函数中不可被修改。

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

(\*cmp): 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。

**注意与例 8-13 通用冒泡排序函数中cmp函数的约定不同！！**

## 8.7.3 使用函数指针的库函数

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);
```

## 8.7.3 使用函数指针的库函数

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);
}
```

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。

## 8.7.3 使用函数指针的库函数

**例 8-14 点集排序** 已知一个  $N \times 2$  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，再根据  $y$  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。首先分析数组内存空间中数据的构成和含义：`pt` 中每一行两个 `double` 元素代表一个点的坐标，将这 2 个 `double` 元素看作一个单元，那么数组 `pt` 中有  $N$  个连续单元；其次要**明确比较函数中元素指针指向的内容**：因为要把 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



## 8.7.3 使用函数指针的库函数

**例 8-14 点集排序** 已知一个  $N \times 2$  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，再根据  $y$  坐标值从大到小排序。

根据排序依据写出比较函数如下

```
int x_ascending(const double *pt1, const double *pt2)
{
    if (pt1[0] < pt2[0])
        return -1;
    else if (pt1[0] > pt2[0])
        return 1;
    else
        return 0;
}
```

x坐标比较

```
int y_descending(const double *pt1, const double *pt2)
{
    if (pt1[1] < pt2[1])
        return 1;
    else if (pt1[1] > pt2[1])
        return -1;
    else
        return 0;
}
```

y坐标比较

pt1, pt2指向的是排序单元，包含两个double元素，分别是x, y坐标

## 8.7.3 使用函数指针的库函数

**例 8-14 点集排序** 已知一个  $N \times 2$  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，再根据  $y$  坐标值从大到小排序。

- 注意 qsort 函数的第一个参数应该是数组的起始地址，只要数值上等于数组的起始地址即可，与类型无关，因为它会被隐式转换为 void\* 类型从而丢掉具体的类型信息。
- 这种情况下传入数组名 pt 也行，传入数组首元素的地址 &pt[0][0] 也可，它们在数值上是相等的，都等于数组空间的起始地址。

### qsort的使用

```
#include <stdio.h>
#include <stdlib.h>
#define LINE 6
#define ROW 2
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), x_ascending);
    qsort(pt, LINE, 2 * sizeof(double), y_descending);
    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;
}
```

## 8.7.3 使用函数指针的库函数

### 例 8-15 按字符串长度排序 (qsort 版) 题意见例8-5。

问题分析：与例 8-10 不同，本例算法不打算使用指针数组。同样定义一个全局二维数组 `char array[100][1002]` 保存输入的所有行。一旦输入结束，每行的字符串由其在 `array` 数组中的行号（从 0 开始）唯一决定。从这种意义上说，行号类似与指针，都是一种数据索引。因此定义一个全局数组 `int lines[100]` 保存每行的行号，采用 `qsort` 函数对 `lines` 数组进行排序。在比较函数中，根据 `lines` 数组中的行号，找到对应的字符串，比较大小返回顺序。`qsort` 对 `lines` 数组排序后的结果是，`lines` 数组中的行号将根据要求变得有序，最后依次输出行号所代表的原始字符串即可。

## 8.7.3 使用函数指针的库函数

**例 8-15 按字符串长度排序 (qsort 版)** 题意见例8-5。

```
int cmp(const int *e1, const int *e2)
{
    int i = *e1, j = *e2; // get line number
    int iLen = strlen(array[i]);
    int jLen = strlen(array[j]);
    if (iLen > jLen || iLen == jLen && strcmp(array[i], array[j]) > 0)
        return 1;
    else if (iLen == jLen && strcmp(array[i], array[j]) == 0)
        return 0;
    else
        return -1;
}
```

e1和e2指向的是行号 (对应array数组)

长度长, 或者长度一样时字典序大的往后排, 返回1

长度相等, 并且字符串内容一样时, 无所谓顺序, 返回0 (能否去掉iLen==jLen这个条件?)

## 8.7.3 使用函数指针的库函数

**例 8-15 按字符串长度排序 (qsort 版)** 题意见例8-5。

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
char array[100][1002];
int lines[100]; // line number
int main()
{
    int i, k = 0;
    while (fgets(array[k], 1000, stdin) != NULL)
    {
        lines[k] = k; // record line number
        k++;
    }
    qsort(lines, k, sizeof(int), cmp);
    for (i = 0; i < k; ++i)
        fputs(array[lines[i]], stdout);
    return 0;
}
```

qsort 不一定要对原始数组内容进行排序，因为原始数组的元素可能会很大，内存交换内容过多会导致效率低下。通过对原始数组内容的索引数组进行排序，可以非常有效地提高排序效率。

qsort后，lines里的行号变得有序

根据lines里的有序行号，检索array输出对应的字符串

## 8.7.3 使用函数指针的库函数

### (2) bsearch() 函数 (标准库函数)

包含头文件<stdlib.h>

```
void *bsearch (const void *key, const void *base, size_t num,  
              size_t wid, int (*cmp) (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首地址;
- num: 数组中元素 (查找单元) 的个数;
- wid: 每一个元素 (查找单元) 所占用的字节数;
- cmp: 指向比较函数的指针, 规则与qsort相同
- 当 base 所指向的数组中有与 key 所指向的数据相等的元素时 (在cmp函数规则下两者相等, cmp 返回0), bsearch()返回该元素的地址, 否则返回NULL。
- 数组必须根据cmp规则是有序的

## 8.7.3 使用函数指针的库函数

### 例 8-16 快速排序与查找

- 输入：多个整数，第一个是正整数  $n$  ( $n < 1000000$ )；接着输入  $n$  个 `int` 型整数，每一个元素  $x$  记为数组 `data` 的数据元素；然后再输入多个整数（不超过 1000000 个），每一个数记为  $t$ 。输入的数之间用空白符号（空格、换行、或制表符）分隔。
- 输出：对每一个  $t$ ，若  $t$  出现在数组 `data` 中，则输出它出现的位置（出现在 `data` 中第几个输入的数），否则输出 NO。每一个输出占一行。

问题分析：从效率的角度而言，每次都遍历所有数据去暴力寻找目标是不合适的，对本题，这种暴力方法最多可能需要比较  $10^{12}$  次，即便对于每秒能比较 10 亿次的计算机，也需要 1000 秒才能完成任务。对于查找频率较高的操作，应该先对数据进行排序，然后再用二分法快速寻找：即一次排序，多次查找。题目要求找到  $t$  时还要输出它的位置，因此在输入数据时同时需要保存它的序号，故采用二维数组保存  $x$  以及  $x$  的序号（输入的次序），并将它们视为一个排序单元。然后采用 `qsort` 函数对输入数据按照大小连同序号一起进行排序。在查找时采用效率很高的 `bsearch` 函数，如果找到则输出对应的序号，否则输出 NO。

## 8.7.3 使用函数指针的库函数

```
int data[1000000][2];
int main()
{
    int n, i, t, *ret;
    scanf("%d", &n);
    for (i = 0; i < n; i++)
    {
        scanf("%d", data[i]); //等价于 scanf("%d", &data[i][0]);
        data[i][1] = i + 1;    //保存序号
    }
    qsort(data, n, 2 * sizeof(int), cmp); // n 个单元, 每个单元存放数据和序号
    while (scanf("%d", &t) != EOF)
    {
        ret = (int *)bsearch(&t, data, n, 2 * sizeof(int), cmp);
        if (ret)
            printf("%d\n", ret[1]);
        else
            printf("NO\n");
    }
    return 0;
}
```

每一行是一个排序单元，两个元素，第一个是数据，第二个是序号（输入顺序）

```
int cmp(const int *e1, const int *e2)
{
    if (*e1 < *e2)
        return -1;
    else if (*e1 > *e2)
        return 1;
    else
        return 0;
}
```

cmp只根据第一个关键字，即数据大小进行排序

ret[0]是数据, ret[1]是序号

**qsort是不稳定排序，当输入的数据有重复时，输出结果会有什么问题？**



## 8.7.3 使用函数指针的库函数

### 例 8-16 快速排序与查找

- 输入：多个整数，第一个是正整数  $n$  ( $n < 1000000$ )；接着输入  $n$  个 `int` 型整数，每一个元素  $x$  记为数组 `data` 的数据元素；然后再输入多个整数（不超过 1000000 个），每一个数记为  $t$ 。输入的数之间用空白符号（空格、换行、或制表符）分隔。
- 输出：对每一个  $t$ ，若  $t$  出现在数组 `data` 中，则输出它出现的位置（出现在 `data` 中第几个输入的数），否则输出 `NO`。每一个输出占一行。

**思考题：如果  $t$  在 `data` 中有重复，输出在输入时第一次出现的位置！如何修改程序？**

## 8.8 本章小结

---

本章介绍了指针的更深层次的原理和更高级的用法。首先，重点解释了指针与数组之间的联系和区别；强调了数组名作为表达式可以隐式转换为指向数组首元素的指针。其次，从数组的地址引出了数组指针的概念，讲解了数组指针和数组元素指针之间的区别；将数组的概念扩展到多维数组，揭示了二维数组中的行指针和列指针的本质。随后，讲解了多重指针和指针数组的概念及其应用。最后，介绍了函数指针及其意义，阐述了利用函数指针进行泛型编程的设计模式与初步思想。