



# 专题一 指针进阶

Part B

# 专题一 指针进阶

- C语言基础知识回顾
- 有序表的操作 (10.1)
- 内存动态分配 (8.5)
- 指针数组、二级指针、数组指针 (11.1)
  - 指针数组 (11.1.1-2, 11.1.4-5)
  - 二级指针 - 指向指针的指针 (11.1.3)
  - 数组指针
- 函数指针 (11.2.3)

# 构造数据类型： 指针、数组、结构

## ■ 数组(int a[10];) vs. 指针(int \*p;)

- 数组名代表了整个数组，指向首元素的指针
- 数组名是指针常量，不能被重新赋值
- 数组名算术运算或作为实参，数组名退化为指针

□ 细节1: `int a[10], *p = a, *q = &a[5];`

区分: `printf("%d", q - p); printf("%d", (int)q - (int)p);`

□ 细节2: `int a[10], *p = a; (&a vs. &p)`

区分: `a`是`a[0]`的起始地址, `&a`是整个数组的起始地址

□ 细节3: `int a[10], *p = a; (内存空间分配)`

区分: `sizeof(a)`和`sizeof(p)`

# 构造数据类型：指针、数组、结构

## ■ 数组 vs. 结构

- 都是构造类型，是多个变量的集合
- 数组成员和结构成员在内存中是连续存储
- 数组成员类型相同，结构成员类型不同
- 数组名是指针常量，不能被重新赋值
  - 即数组不能整体赋值，需要逐个拷贝，例如strcpy
- 结构可以整体赋值，即使结构成员包含数组
- 作为函数实参时，数组传递首元素地址，提高调用效率，结构整体赋值拷贝，影响调用效率

# 构造数据类型： 指针、数组、结构

类型	整型	浮点型	字符型	空型	指针	数组	结构
指针	int *	float * double *	char *	void *	?	?	struct data *
数组	int a[5]	float a[5] double a[5]	char s[5]		指针数组 char *p[5]	int a[5][10] char s[5][10]	struct data d[5]
结构							

```
struct data {  
    int a;  
    float b;  
    char s[10];  
    double *d;  
    int arr[10];  
    struct member m;  
};
```

```
void p[5];  
struct data {  
    void a;  
    .....  
}
```

# 指针数组

- C语言中的数组可以是任何类型，如果数组的各个元素都是指针类型，用于存放内存地址，那么这个数组就是指针数组
- 一维指针数组定义的一般格式为  
类型名 \*数组名[数组长度]

## 数组

- C语言中只有一维数组，而且数组的大小必须在编译时作为一个常数确定下来。
- 对于一个数组，我们只能够做两件事：确定该数组的大小，以及获得指向该数组下标为0的元素指针。其他操作都是通过指针进行。

# 指针数组

## ■ `int a[10];`

- `a`是一个数组，它有10个元素
- 每个元素的类型都是整型

## ■ `char *color[5];`

- `color`是一个数组，它有5个元素
- 每个元素的类型都是字符指针

# 指针数组

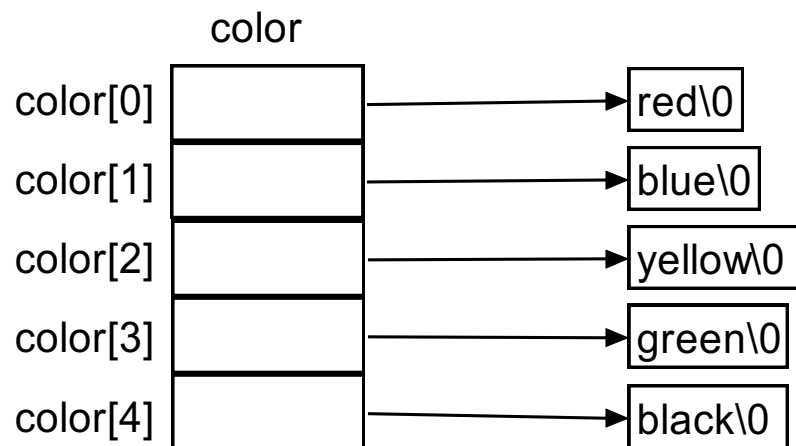
```
const char *color[5] = {"red", "blue", "yellow", "green", "black"};
```

- color是一个数组，它有5个元素
- 每个元素的类型都是字符指针
- 数组元素可以处理字符串
- const: 限定变量值不被改变
  - 指针color[i]，还是指针所指向的内容color[i][j]不能修改？
  - char \* const color[5] = {"red", "blue", "yellow", "green", "black"};

对指针数组元素的操作和对  
同类型指针变量的操作相同

## 对指针数组元素的操作

- printf ("%x %s\n", color[i], color[i]);

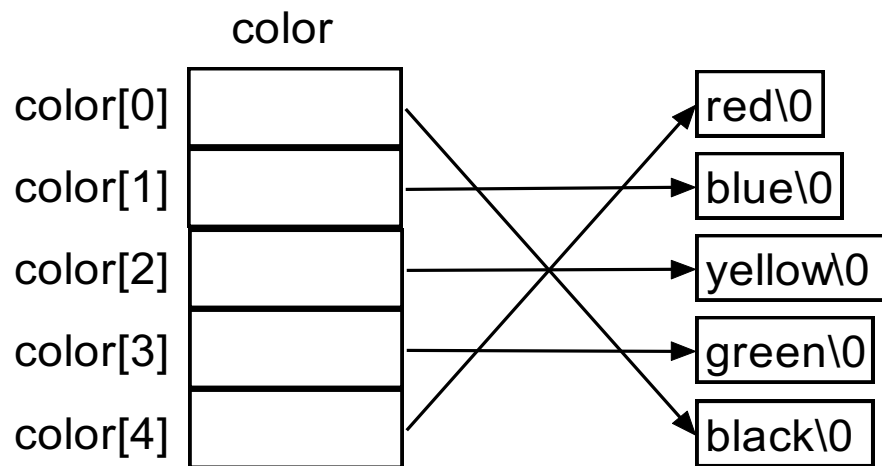




# 指针数组

## ■ 继续执行

```
char * tmp;  
tmp = color[0];  
color[0] = color[4];  
color[4] = tmp;
```



## ■ 指针数组操作

- 可以直接对数组元素进行引用操作

```
tmp = color[0];
```

- 也可以间接访问操作数组元素所指向的单元内容

```
printf ("%c", *(color[0]+1));
```

# 指针数组 vs. 二维数组

## ■ 多个字符串处理 - 常量

### □ 二维字符数组

```
char ccolor[ ][7] = {"red",  
"blue", "yellow", "green",  
"black"};
```

### □ 指针数组

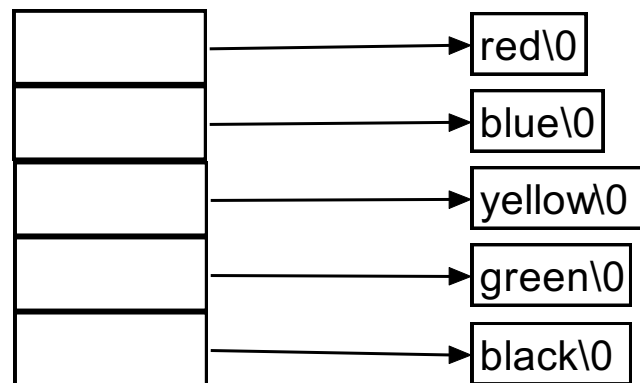
```
const char *pcolor[ ] = {"red",  
"blue", "yellow", "green",  
"black"};
```

使用指针数组更节省内存空间  
不能修改ccolor[0]的值，不能修改pcolor[0][0]的值

ccolor

r	e	d	\0			
b	l	u	e	\0		
y	e	l	l	o	w	\0
g	r	e	e	n	\0	
b	l	a	c	k	\0	

pcolor



# 指针数组 vs. 二维数组

## ■ 多个字符串处理 - 运行时输入

### □ 指针数组

```
char *pcolor[10];  
for (i = 0; i < 10; i++)  
    scanf("%s", pcolor[i]);
```

段错误

### □ 二维字符数组

```
char ccolor[10][81];  
for (i = 0; i < 10; i++)  
    scanf("%s", ccolor[i]);
```

**常见错误:** pcolor[i]是指针，但未指向有效内存地址 (NULL不是有效内存地址)

pcolor[0]	?
pcolor[1]	?
.....	?
pcolor[9]	?

4/8 bytes

ccolor[0]				.....			
ccolor[1]				.....			
.....				.....			
ccolor[9]				.....			

81 bytes

# 指针数组 vs. 二维数组

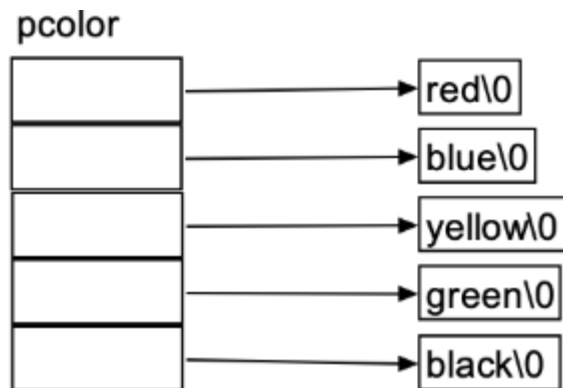
## ■ 多个字符串处理 - 运行时输入 (动态内存分配)

### □ 指针数组

```
char *pcolor[10];
char color[81];
for (i = 0; i < 10; i++) {
    scanf("%s", color);
    pcolor[i] = (char *)malloc(sizeof(char) * (strlen(color) + 1));
    strcpy(pcolor[i], color);
}
.....
for (i = 0; i < 10; i++)
    free(pcolor[i]);
```

### □ 二维字符数组

```
char ccolor[10][81];
for (i = 0; i < 10; i++)
    scanf("%s", ccolor[i]);
```



区别: `char *pcolor[ ]`  
= {"red", "blue",  
"yellow", "green",  
"black"};

# 指针数组应用1 - 单词索引

- [例11-1] 一个单词表存放了五个表示颜色的英文单词，输入一个字母，在单词表中查找并输出所有以此字母开头的单词，若没有找到，输出**Not Found**

# 例11-1 源程序

```
#include <stdio.h>
int main(void)
{
```

```
    int i, flag = 0;
```

```
    char ch;
```

```
    const char *color[5] = {"red", "blue", "yellow", "green", "black"};
```



```
    printf("Input a letter: ");
```

```
    ch = getchar();
```

```
    for (i = 0; i < 5; i++) {
```

```
        if(*color[i] == ch) {
```

```
            flag = 1;
```

```
            puts(color[i]);
```

```
        }
```

```
    }
```

```
    if(flag == 0)
```

```
        printf("Not Found\n");
```

```
    return 0;
```

```
}
```

**const**: C语言关键字，常量  
如 `const int a = 4;`  
    `a = 5;`      // 编译报错

Input a letter: **y**  
yellow

Input a letter: **a**  
Not Found

# 指针数组应用2 - 字符串排序

## ■ [例11-4] 将5个字符串从小到大排序后输出

```
void fsort(int a[ ], int n);  
int main( )  
{   int i;  
    int a[5] = {6, 5, 2, 8, 1};
```

```
    fsort(a, 5);  
    for(i = 0; i < 5; i++)  
        printf("%d ", a[i]);  
    return 0;
```

```
}
```

```
void fsort(const char*color[ ], int n);  
int main( )  
{   int i;  
    const char *pcolor[5] = {"red",  
    "blue", "yellow", "green", "black" };
```

```
    fsort(pcolor, 5);  
    for(i = 0; i < 5; i++)  
        printf("%s ", pcolor[i]);  
    return 0;
```

```
}
```

# 指针数组应用2 - 字符串排序

```
void fsort(int a[ ], int n)
```

```
{  int k, j;
```

```
    int temp;
```

```
    for(k = 1; k < n; k++)
```

```
        for(j = 0; j < n-k; j++)
```

```
            if(a[j] > a[j+1]) {
```

```
                temp = a[j];
```

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

```
                a[j+1] = temp;
```

```
            }
```

```
}
```

```
void fsort(const char *color[ ], int n)
```

```
{  int k, j;
```

```
    const char *temp;
```

```
    for(k = 1; k < n; k++)
```

```
        for(j = 0; j < n-k; j++)
```

```
            if(strcmp(color[j],color[j+1])>0) {
```

```
                temp = color[j];
```

```
                color[j] = color[j+1];
```

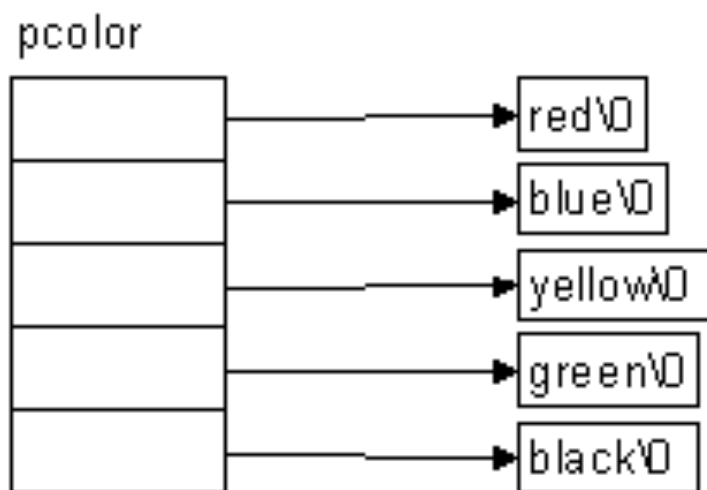
```
                color[j+1] = temp;
```

```
            }
```

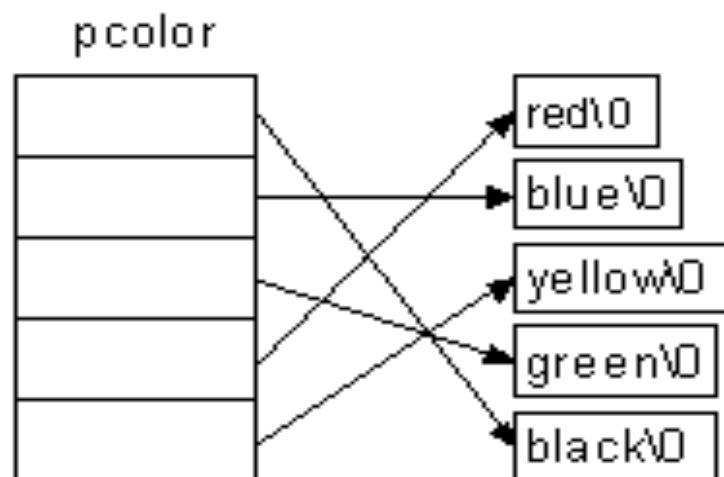
```
}
```



# 指针数组应用2 - 字符串排序



排序前



排序后

# 指针数组应用3 - 藏头诗

- [例11-5] 解密英文藏头诗。将一首诗每一句的第一个字连起来，所组成的内容是该诗的真正含义
  - 编写程序，输出一首英文藏头诗的真实含义。输入的藏头诗小于20行，每行不超过80个字符，以#作为输入结束标志，使用动态内存分配方法处理字符串的输入

**I come into a dream  
Leaves fall down but spring  
over a lake birds flying  
village have its nice morning  
everywhere can feel happiness  
Years have never been  
owners don't need anything  
until the sun bring another wind  
#  
ILoveYou**

# 例11-5 解密英文藏头诗

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(void)
{   int i, n = 0;
    char *poem[20], str[80], mean[20];
    gets(str);
    while(str[0] != '#') {
        poem[n] = (char *) malloc(sizeof(char) * (strlen(str)+1)); /* 动态分配 */
        strcpy(poem[n], str); /* 将输入的字符串赋值给动态内存单元 */
        n++;
        gets(str);
    }
    for(i = 0; i < n; i++) {
        mean[i] = *poem[i]; /* 每行取第一个字符 */
        free(poem[i]); /* 释放动态内存单元 */
    }
    mean[i] = '\0';
    puts(mean);
    return 0;
}
```

优点：能够根据实际输入数据的多少来申请和分配内存空间，从而提高了内存的使用率

中文：一个汉字2个字节

# 指针数组应用4 - 随机发牌

- [例11-6] 一副纸牌有52张，4种花色(黑桃Spade, 红桃Heart, 草花Club, 方块Diamond)，每种花色13张。用程序模拟随机发牌过程，将52张牌按轮转的方式发放给4人，并输出发牌结果

Player 1:	Player 2:	Player 3:	Player 4:
10 of Spade	6 of Heart	4 of Heart	7 of Heart
6 of Diamond	6 of Spade	9 of Diamond	J of Club
9 of Spade	2 of Heart	3 of Heart	9 of Heart
8 of Spade	2 of Spade	2 of Diamond	A of Diamond
J of Heart	J of Spade	A of Spade	4 of Spade
5 of Club	3 of Spade	10 of Club	Q of Spade
K of Spade	K of Club	10 of Heart	4 of Diamond
7 of Club	5 of Spade	5 of Diamond	A of Heart
8 of Diamond	3 of Club	Q of Heart	7 of Spade
9 of Club	7 of Diamond	Q of Club	8 of Heart
8 of Club	J of Diamond	6 of Club	Q of Diamond
3 of Diamond	5 of Heart	A of Club	4 of Club
K of Heart	2 of Club	10 of Diamond	K of Diamond

# 例11-6 分析

```
struct card {           /* 用结构表示一张牌 */
    int suit; /* 0~3对应花色Heart, Diamond, Club, Spade*/
    int face; /* 0~12对应点数A, K, Q, J, 10, 9, ... 3, 2 */
};
```

示例中第0张牌的花色和点数是？

```
struct card deck[52]; /*按顺序存放4个人的牌 */
    player1: deck[0]-deck[12]
    player2: deck[13]-deck[25]
    player3: deck[26]-deck[38]
    player4: deck[39]-deck[51]
```

以字符串的形式，输出每张牌的花色和点数：

```
const char *suit[] = {"Heart", "Diamond", "Club", "Spade"};
```

```
const char *face[] = {"A", "K", "Q", "J", "10", "9", "8", "7", "6", "5", "4", "3", "2"};
```

输出第i张牌的点数和花色：

```
printf("%s of %s\n", face[deck[i].face], suit[deck[i].suit]);
```

Player 1:  
10 of Spade  
6 of Diamond  
9 of Spade  
8 of Spade  
J of Heart  
5 of Club  
K of Spade  
7 of Club  
8 of Diamond  
9 of Club  
8 of Club  
3 of Diamond  
K of Heart

## 例11-6 源程序

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
struct card {
    int suit;
    int face;
};
void deal(struct card *deck);    /* 发牌函数声明 */
int main( void )
{
    int i;
    struct card deck[52];
    const char *suit[] = {"Heart", "Diamond", "Club", "Spade"};
    const char *face[]={"A", "K", "Q", "J", "10", "9", "8", "7", "6", "5", "4", "3", "2"};
    deal(deck);                  /* 调用函数，实现发牌 */
    for(i = 0; i < 52; i++) {
        if(i%13 == 0) printf("Player %d:\n", i / 13 + 1);
        printf("%s of %s\n", face[deck[i].face], suit[deck[i].suit]);
    }
    return 0;
}
```

```

void deal(struct card *deck)
{
    int i, m, t;
    int temp[52] = {0};

    srand(time(NULL));
    for(i = 0; i < 52; i++) {
        while(1) {
            m = rand() % 52;
            if(temp[m] == 0)
                break;
        }
        temp[m] = 1;

        t = (i % 4) * 13 + (i / 4);
        deck[t].suit = m / 13;
        deck[t].face = m % 13;
    }
}

```

/\* 发牌 \*/

## 例11-6 源程序

/\* 发牌标记 0:未发 1:已发 \*/

/\* 设定随机数的产生与系统时钟关联 \*/

/\* 使用随机数找到一张未发的牌 \*/

/\* 计算机随机产生一个0~51之间的数 \*/

/\* 随机数对应的牌未发 \*/

/\* 标识该牌已发 \*/

/\* 4人轮转发牌，i % 4第几人，i / 4第几张牌 \*/

/\* m表示第几张牌 \*/

课本中static int temp[52] = {0};  
能否多次发牌？即调用deal多次？

m	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	...	
m/13	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	3	...
m%13	0	1	2	3	4	5	6	7	8	9	10	11	12	0	1	2	3	4	5	6	7	8	9	10	11	12	0	1	2	3	4	5	6	7	8	9	10	11	12	0	...	

# 指针数组应用4 - 随机发牌[拓展]

## ■ 算法分析

- 生成0~51之间的随机数 $m$ ，如果 $m$ 之前出现过(该牌已发)，则继续生成随机数 $m$ ，直到 $m$ 未出现
  - 随着生成随机数数量(发牌张数)的增加，新的随机数与已经产生的随机数相同的可能性越来越大，有可能出现算法延迟问题

## ■ 高效算法

- 先将52张牌按照花色与点数顺序存放(`card[]`)
- 再将其随机打乱
  - 每次循环，生成0~51之间的随机数 $m$ ，然后将数组中当前牌`card[i]`与随机选出的牌`card[m]`进行交换



# 例11-6 源程序

```
void deal(struct card *deck)
{
```

```
    int i, m, t;
```

```
    int card[52];
```

```
    for (i = 0; i < 52; i++)
```

```
        card[i] = i;
```

```
    srand(time(NULL));
```

```
    for (i = 0; i < 52; i++) {
```

```
        m = rand() % 52;
```

```
        t = card[i];
```

```
        card[i] = card[m];
```

```
        card[m] = t;
```

```
    }
```

```
    for (i = 0; i < 52; i++) {
```

```
        t = (i % 4) * 13 + (i / 4);
```

```
        deck[t].suit = card[i] / 13;
```

```
        deck[t].face = card[i] % 13;
```

```
    }
```

```
}
```

/\* 发牌 \*/

/\* 顺序存放52张牌 \*/

/\* 设定随机数的产生与系统时钟关联 \*/

/\* 计算机随机产生一个0~51之间的数 \*/

/\* 交换card[i]和card[m]的牌 \*/

/\* 4人轮转发牌，i % 4第几人，i / 4第几张牌 \*/

/\* card[i]表示第几张牌 \*/

现在牌两两交换了52次，如果希望进一步打乱牌，如何修改代码？

# 指针数组应用5 - 命令行参数

- C语言源程序经编译和连接处理，生成可执行程序(例如**test.exe**)后，才能运行
- 在**DOS**环境的命令窗口中，输入可执行文件名，就以命令方式运行该程序
- 输入命令时，在可执行文件(命令)名的后面可以跟一些参数，这些参数被称为命令行参数

**test world**

(**test**是命令名， **world**是命令行参数)

# 指针数组应用5 - 命令行参数

- 命令行的一般形式为

命令名 参数1 参数2 ... 参数n

命令行和各个参数之间用空格分割，也可以没有参数

- 使用命令行的程序不能在编译器中执行，需要将源程序经编译、链接为相应的命令文件(一般以`.exe`为后缀)，然后回到命令行状态，再在该状态下直接输入命令文件名

# 指针数组应用5 - 命令行参数

## ■ 带参数的main()函数格式

```
int main(int argc, char *argv[ ])  
{  
    .....  
}
```

- 第1个参数`argc`接收命令行参数(包括命令名)的个数
- 第2个参数`argv`接收以字符串常量形式存放的命令行参数(包括命令名本身也作为一个参数)

# 指针数组应用5 - 命令行参数

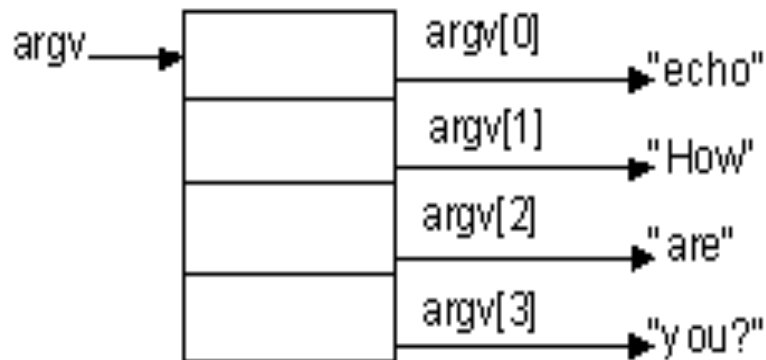
- [例11-7] 编写C程序echo，它的功能是将所有命令行参数在同一行上输出

```
#include <stdio.h>

int main(int argc, char *argv[ ])
{
    int k;
    /* 从第1个命令行参数开始 */
    for(k = 1; k < argc; k++)
        /* 打印命令行参数 */
        printf("%s ", argv[k]);

    printf("\n");
    return 0;
}
```

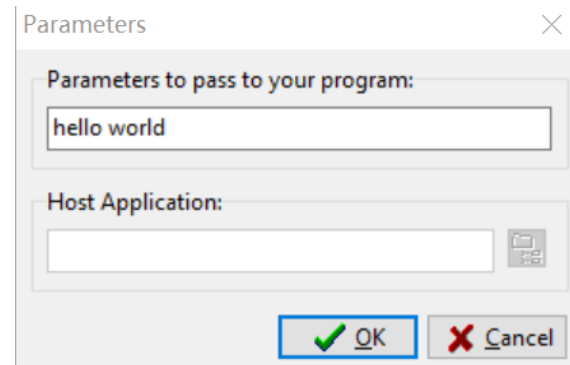
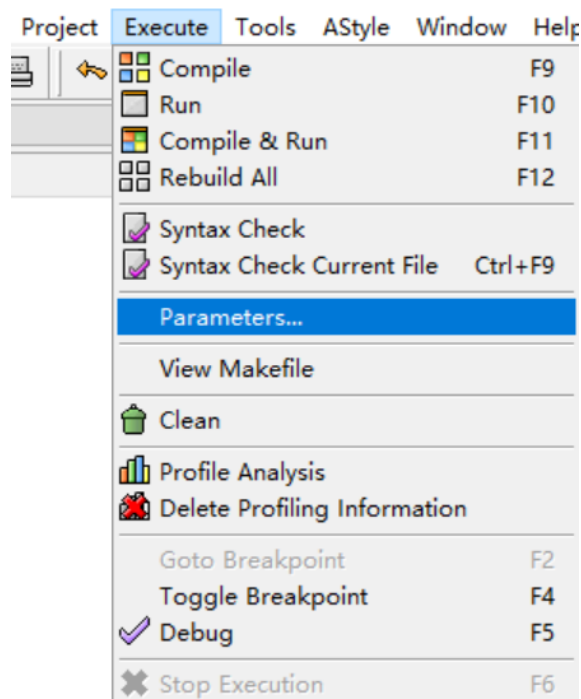
运行结果  
在命令行状态下输入：  
**echo How are you?**  
输出：  
How are you?



# 指针数组应用5 - 命令行参数

## ■ Dev-C++中命令行参数

□ Execute → Parameters...



# 构造数据类型： 指针、数组、结构

类型	整型	浮点型	字符型	空型	指针	数组	结构
指针	int *	float * double *	char *	void *	二级指针 int **p float **p void **p	?	struct data *
数组	int a[5]	float a[5] double a[5]	char s[5]		指针数组 char *p[5]	int a[5][10] char s[5][10]	struct data d[5]
结构							

```

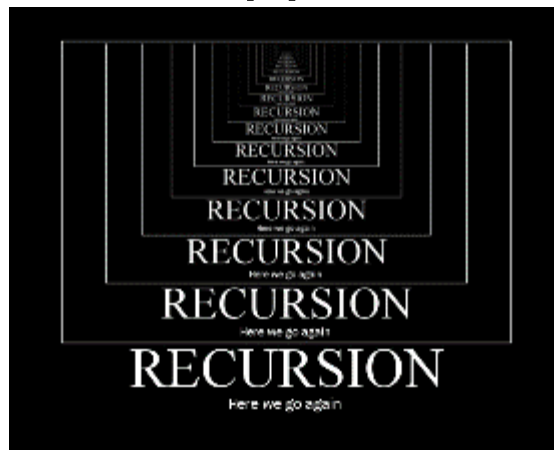
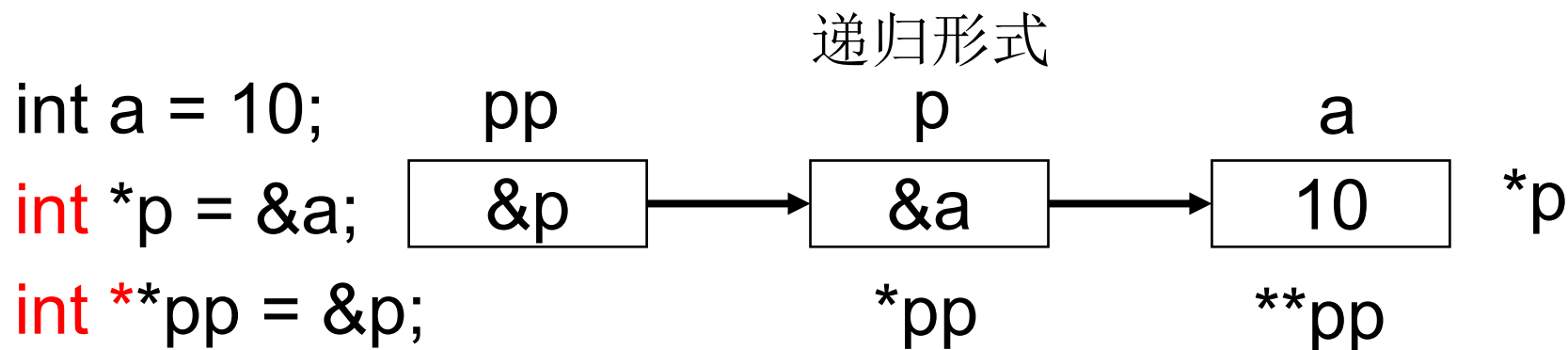
struct data {
    int a;
    float b;
    char s[10];
    double *d;
    int arr[10];
    struct member m;
};
    
```

```

void p[5];
struct data {
    void a;
    .....
}
    
```

# 二级指针 - 指向指针的指针

- 指向指针的指针(二级指针)一般定义为  
类型名 \*\*变量名



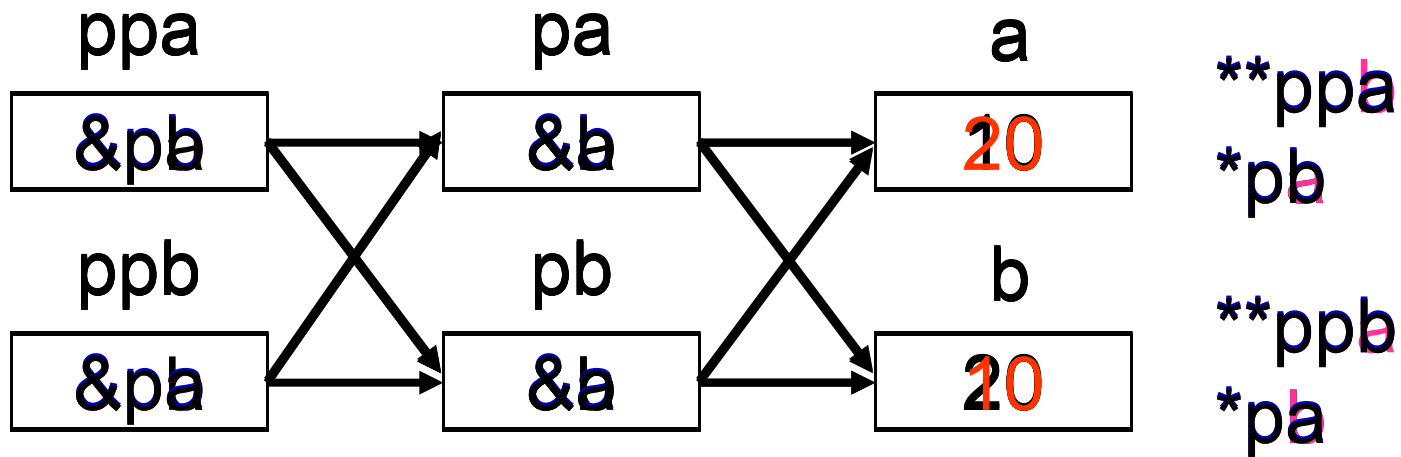


## 例11-2 解析

```
int a = 10, b = 20, t;
```

```
int *pa = &a, *pb = &b, *pt;
```

```
int **ppa = &pa, **ppb = &pb, **ppt;
```



操作(1): `ppt = ppb; ppb = ppa; ppa = ppt;`

操作(2): `pt = pb; pb = pa; pa = pt;`

操作(3): `t = b; b = a; a = t;`

# 例11-3 单词索引 二级指针实现

```
int main(void)
{ int i, flag = 0;
  char ch;
  const char *color[5] = {"red", "blue", "yellow", "green", "black" };
  const char **pc = color; ← 二级指针
  printf("Input a letter:");
  ch = getchar();
  for(i = 0; i < 5; i++) {
    if(**(pc + i) == ch) { ← 使用二级指针操作数据
      flag = 1;
      puts( *(pc + i) );
    }
  }
  if(flag == 0) printf("Not Found\n");
  return 0;
}
```

int a[10];

整型数组的数组名 → 指向整型的指针常量  
一级指针

int \*b[10];

指针数组的数组名 → 指向指针的指针常量  
二级指针

## 例11-3 单词索引 二级指针实现

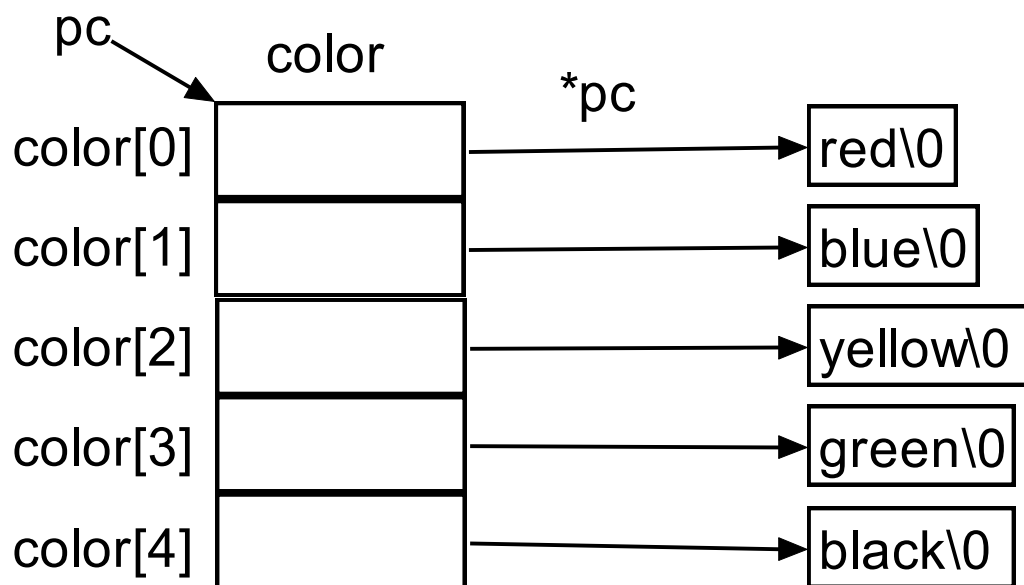
$pc \Leftrightarrow color \Leftrightarrow \&color[0]$

$*pc \Leftrightarrow color[0]$

$*(pc+i) \Leftrightarrow color[i]$

$**pc \Leftrightarrow *(*pc) \Leftrightarrow *color[0]: 'r'$

$** (pc+i) \Leftrightarrow *(* (pc+i)) \Leftrightarrow *color[i]: '?'$



$**pc + i$   
 $*( *pc + i)$   
 $** (pc + i)$   
 $*pc[i]$   
 $( *pc)[i]$   
 $pc[i][j]$

# 构造数据类型： 指针、数组、结构

类型	整型	浮点型	字符型	空型	指针	数组	结构
指针	int *	float * double *	char *	void *	二级指针 int **p float **p void **p	数组指针 int (*p)[10] float (*p)[10] char (*p)[10]	struct data *
数组	int a[5]	float a[5] double a[5]	char s[5]		指针数组 char *p[5]	int a[5][10] char s[5][10]	struct data d[5]
结构							

```

struct data {
    int a;
    float b;
    char s[10];
    double *d;
    int arr[10];
    struct member m;
};
    
```

```

void p[5];
struct data {
    void a;
    .....
}
    
```

# 数组指针

## ■ 指针数组 - 一个以指针为元素的数组

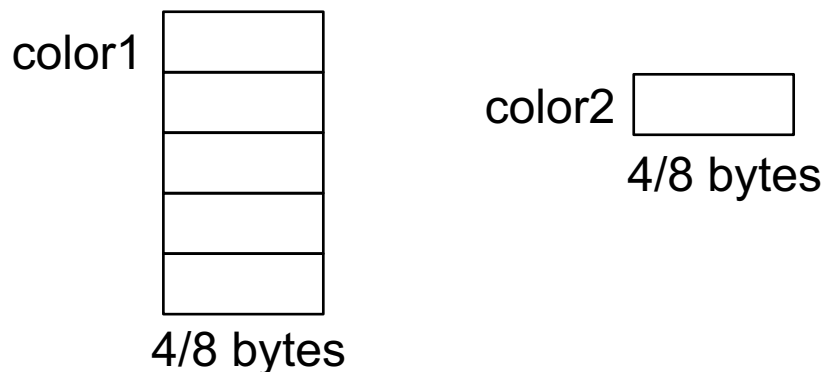
□ `char *color1[5]`

- 一个数组，包含5个`char *`，即5个指针
- `[]`优先级高于`*`

## ■ 数组指针 - 一个指向数组的指针

□ `char (*color2)[5];`

- 一个指针，指向具有5个`char`元素的数组



# 一维数组的指针形式

int a[3];

- 看成是由a[0]、a[1]、a[2]组成的一维数组
- a[0]、a[1]、a[2]各自是一个整数 (int)
- int \*p = a;

DataType a[3];

$a + i \rightarrow a + \text{sizeof}(\text{DataType}) * i$   
第i个元素    内存地址(单位字节)

- a:                    第0个元素的地址 (int \*)
- a+i:                第i 个元素的地址 (int \*)
- \*(a+i) / a[i]: 第i 个元素的值 (int)
- p+i:                第i 个元素的地址 (int \*)
- \*(p+i) / p[i]: 第i 个元素的值 (int)

# 二维数组的指针形式

递归形式:  
二级指针  
二维数组

`int a[3][4];`

- 看成是由`a[0]`、`a[1]`、`a[2]`组成的一维数组
- `a[0]`、`a[1]`、`a[2]`各自又是一个一维数组 (`int *`)
- 二维数组是数组元素为一维数组的一维数组 (数组的数组)
- `int (*a)[4]`是一个指向数组的指针 vs. `int *a[4]`;

- `a`: 第0行地址 (行地址) (`int (*)[4]`)
- `a+i`: 第*i* 行地址 (行地址) (`int (*)[4]`)
- `*(a+i)` / `a[i]` : 第*i*行首元素的地址 (`int *`)
- `*(a+i)+j` / `a[i]+j` : 第*i*行第*j*个元素的地址 (`int *`)
- `** (a+i)` / `a[i][0]` : 第*i*行首元素的值 (`int`)
- `*(*(a+i)+j)` / `a[i][j]` : 第*i*行第*j*个元素的值 (`int`)

# 二维数组的指针形式

■ `int days[12][31];`

□ 数组级指针

■ `int (*pmonth)[31] = days + 2;`

■ `(long)(pmonth + 1) - (long)pmonth = ?`

□ `sizeof(days[0]) = 31 * sizeof(int)`

□ 元素级指针

■ `int *pday = days[2];`

■ `int *pday = *pmonth;`

□ 元素

■ `int day = days[2][15];`

■ `int day = *(days[2] + 15);`

■ `int day = *(*days + 2) + 15);`

	<code>c[11][30]</code>
	...
	<code>c[2][30]</code>
	...
	<code>c[2][2]</code>
<code>c[2]</code>	<code>c[2][1]</code>
	<code>c[2][0]</code>
	<code>c[1][30]</code>
	...
	<code>c[1][2]</code>
	<code>c[1][1]</code>
	<code>c[1][0]</code>
	<code>c[0][30]</code>
	...
	<code>c[0][2]</code>
	<code>c[0][1]</code>
	<code>c[0][0]</code>
<code>c[0]</code>	



# 一维数组作为函数参数

- `int a[10]; func(a);`

- 函数

- ☐ `void func(int *ap);`



- ☐ `void func(int a[]);`



- ☐ `void func(int a[10]);`



# 二维数组作为函数参数

■ `int a[10][20]; func(a);`

■ 函数

☐ `void func(int *ap);`

✗

☐ `void func(int **ap);`

✗

☐ `void func(int a[][]);`

✗

☐ `void func(int a[10][]);`

✗

☐ `void func(int a[][20]);`

✓

☐ `void func(int a[10][20]);`

✓

☐ `void func(int (*ap)[20]);`

✓

只能忽略第一维。忽略后面的维度，将导致编译器无法正确寻址

`int a[m][n];`

`a[i][j]: a + i * n * sizeof(int) + j * sizeof(int)`

# 数组作为函数参数

- 编译器改写规则：数组名被改成一个指针参数
  - 非递归定义
  - 数组的数组会被改写为“数组的指针”，而非指针的指针

实参		所匹配的形式参数	
数组的数组	<code>char c[8][10]</code>	<code>char (*)[10]</code>	数组指针
指针数组	<code>char *c[15]</code>	<code>char **c</code>	指针的指针
数组指针(行指针)	<code>char (*c)[64]</code>	<code>char (*c)[64]</code>	不改变
指针的指针	<code>char **c</code>	<code>char **c</code>	不改变

# 数组指针应用 - 字符串排序

## ■ 指针数组 vs. 二维字符数组 (数组指针)

```
void fsort(const char *color[ ], int n);  
int main( )
```

```
{ int i;
```

```
    const char *pcolor[5] = {"red",  
    "blue", "yellow", "green", "black" };
```

```
    fsort(pcolor, 5);
```

```
    for(i = 0; i < 5; i++)
```

```
        printf("%s ", pcolor[i]);
```

```
    return 0;
```

```
}
```

```
void fsort(const char (*color)[8], int n);  
int main( )
```

```
{ int i;
```

```
    char pcolor[][8] = {"red", "blue",  
    "yellow", "green", "black" };
```

```
    fsort(pcolor, 5);
```

```
    for(i = 0; i < 5; i++)
```

```
        printf("%s ", pcolor[i]);
```

```
    return 0;
```

```
}
```

# 数组指针应用 - 字符串排序

## ■ 指针数组 vs. 二维字符数组 (数组指针)

```
void fsort(const char *color[ ], int n)
{   int k, j;
    const char *temp;
```

```
    for(k = 1; k < n; k++)
        for(j = 0; j < n-k; j++)
            if(strcmp(color[j],color[j+1])>0) {
                temp = color[j];
                color[j] = color[j+1];
                color[j+1] = temp;
            }
}
```

```
void fsort(char (*color)[8], int n)
{   int k, j;
    char temp[8];
```

```
    for(k = 1; k < n; k++)
        for(j = 0; j < n-k; j++)
            if(strcmp(color[j],color[j+1])>0) {
                strcpy(temp, color[j]);
                strcpy(color[j], color[j+1]);
                strcpy(color[j+1], temp);
            }
}
```

# 总结

## ■ 二级指针

- 定义，指针数组和数组指针都是二级指针

## ■ 指针数组

- 定义 `int *p[5];`
- 指针数组(`char *s[5]`)与二维数组(`char s[5][81]`)
- 指针数组与动态内存分配

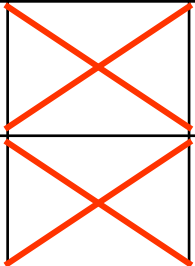
## ■ 数组指针

- 定义 `int (*p)[5];`
- 二维数组的指针形式
- 二维数组作为函数参数

# 专题一 指针进阶

- C语言基础知识回顾
- 有序表的操作 (10.1)
- 内存动态分配 (8.5)
- 指针数组、二级指针、数组指针 (11.1)
- 函数指针 (11.2.3)
  - 指向函数的指针
  - 函数指针应用

# 构造数据类型： 指针、数组、结构

类型	整型	浮点型	字符型	空型	指针	数组	结构
指针	int *	float * double *	char *	void *	二级指针 int **p float **p void **p	数组指针 int (*p)[10] float (*p)[10] char (*p)[10]	struct data *
数组	int a[5]	float a[5] double a[5]	char s[5]		指针数组 char *p[5]	int a[5][10] char s[5][10]	struct data d[5]
结构							

## ■ 指针变量 – 存储内存地址

- 程序内存模型： Stack, Heap, Globals, Constants, Code

- 指向数据的指针 vs. 指向函数的指针



# 指向函数的指针

函数名与数组名相似

入口地址  
→

- 每个函数都占用一段内存单元，它们有一个入口地址(起始地址)
- 在C语言中，函数名代表函数的入口地址(常量)，是一个指针
- 可以定义一个指针变量，接收函数的入口地址，让它指向函数，这就是指向函数的指针，也称为函数指针
- 通过函数指针可以调用函数，它也可以作为函数的参数

指令1
指令2
指令3
...
指令n

# 指向函数的指针

## 1. 函数指针的定义

- 函数指针定义的一般格式为

类型名 (\*变量名) (参数类型表);

- 类型名指定函数返回值的类型，变量名是指向函数的指针变量的名称

- 例如

```
int (*funptr) (int, int);
```

- 定义一个函数指针funptr，它可以指向有两个整型参数且返回值类型为int的函数

```
int *funptr(int a, int b);
```



()优先级高于\*

# 指向函数的指针

## 2. 通过函数指针调用函数

- 通过函数指针调用函数的一般格式为：  
(\*函数指针名) (参数表)

- 例如

```
int fun (int x, int y);  
int (*funptr) (int, int);  
funptr = fun;  
(*funptr) (3, 5);  
fun (3, 5)
```

与数组类比

```
int a[10];  
int *p;  
p = a;  
(*p)++;  
a[0]++;
```

# 指向函数的指针

## 3. 函数指针作为函数的参数

- C语言的函数调用中，函数名或已赋值的函数指针也能作为实参，此时，形参就是函数指针，它指向实参所代表函数的入口地址

□ 例如

```
void f(int (*funptr)(int, int))
{...}

void main()
{
    ...
    int (*funptr)(int, int);
    funptr = fun;
    f( funptr );
    ...
}
```

# 指向函数的指针

## ■ 自定义类型typedef (12.1)

- 定义变量 `int num[10]`
- 变量名  $\rightarrow$  新类型名 `num  $\rightarrow$  IntArray`
- 加上typedef `typedef int IntArray[10]`
- 用新类型名定义变量 `IntArray a  $\leftrightarrow$  int a[10]`

## ■ 指向函数的指针

- `void (*funptr)(int x, int y)`
- `funptr  $\rightarrow$  FunPtr`
- `typedef void (*FunPtr)(int x, int y)`
- `FunPtr fptr  $\leftrightarrow$  void (*fptr)(int x, int y)`

# 函数指针应用1 - 数值积分

[例11-9] 编写一个函数`calc(f, a, b)`，用梯形公式求函数`f(x)`在`[a, b]`上的数值积分

$$\int_a^b f(x)dx = (b-a)/2 * (f(a) + f(b))$$

然后调用该函数计算下列数值积分。(函数指针作为函数参数示例)

$$\int_0^1 x^2 dx \quad \int_1^2 \sin x / x dx$$

**分析：**`calc()`是一个通用函数，用梯形公式求解数值积分。它和被积函数`f(x)`以及积分区间`[a, b]`有关，相应的形参包括函数指针、积分区间上下限参数。在函数调用时，把被积函数的名称(或函数指针)和积分区间的上下限作为实参

# 函数指针应用1 - 数值积分

/\* 计算数值积分 (函数指针作为函数参数示例) \*/

```
int main(void)
{
    double result;
    double (*funp)(double);
    result = calc(f1, 0.0, 1.0);
    printf("1: result=%.4f\n", result);
    funp = f2;
    result = calc(funp, 1.0, 2.0);
    printf("2: result=%.4f\n", result);
    return 0;
}
```

/\* 函数指针funp作为函数的形参 \*/

```
double calc(double (*funp)(double), double a, double b)
```

```
{
    double z;
    z = (b - a) / 2 * ((*funp)(a) + (*funp)(b)); /* 调用funp指向的函数 */
    return(z);
}
```

```
double f1(double x) { return x * x; }
```

```
double f2(double x) { return sin(x) / x; }
```

1: result=0.5000

2: result=0.6481

/\* 函数名f1作为函数calc的实参 \*/

/\* 对函数指针funp赋值 \*/

/\* 函数指针funp作为函数calc的实参 \*/

# 函数指针应用2 - 值调用

## ■ 根据数值调用相应的函数 - 函数指针数组

```
if (a == 0)
    f();
else if (a == 1)
    g();
else if (a == 2)
    h();
```

```
switch (a) {
    case 0: f(); break;
    case 1: g(); break;
    case 2: h(); break;
}
```

```
void (*fun[])() = {f, g, h};    函数指针数组
if (a >= 0 && a < sizeof(fun) / sizeof(fun[0]))
    (*fun[a})();
```



# 函数指针应用3 - 冒泡排序

## ■ 冒泡排序(8.3): 每轮两两比较, 大值调前/后

```
void BubbleA(int array[], int n);      /* 升序排序函数 */
void BubbleB(int array[], int n);      /* 降序排序函数 */
int main(void)
{
    int k;
    int array[10]={6,3,5,7,4,2,9,8,0,1};
    printf("Ascending (0) or Descending (1) order?");
    scanf("%d", &k);
    if (k == 0)                          /* 值调用 */
        BubbleA(array, 10);
    else if (k == 1)
        BubbleB(array, 10);

    for (k = 0; k < 10; k++)
        printf("%d ", array[k]);
    return 0;
}
```

# 函数指针应用3 - 冒泡排序

```
void BubbleA(int array[ ], int n)
{
    int i, j, t;

    for (j = 0; j < n-1; j++) {
        for (i = 0; i < n-1-j; i++) {
            if (array[i] > array[i+1]){
                t = array[i];
                array[i] = array[i+1];
                array[i+1] = t;
            }
        }
    }
}
```

从小到大

```
void BubbleB(int array[ ], int n)
{
    int i, j, t;

    for (j = 0; j < n-1; j++) {
        for (i = 0; i < n-1-j; i++) {
            if (array[i] < array[i+1]){
                t = array[i];
                array[i] = array[i+1];
                array[i+1] = t;
            }
        }
    }
}
```

从大到小

# 函数指针应用3 - 冒泡排序

```
void Bubble(int array[ ], int n, .....)  
{  
    int i, j, t;  
  
    for (j = 0; j < n-1; j++) {  
        for (i = 0; i < n-1-j; i++) {  
            if (compare(array[i], array[i+1])) {  
                t = array[i];  
                array[i] = array[i+1];  
                array[i+1] = t;  
            }  
        }  
    }  
}
```

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

```
int Less(int a, int b)  
{    return (a<b); }
```

```
int main()  
{  
    .....  
    if (k == 0)  
        Bubble(array, 10, Large);  
    else if (k == 1)  
        Bubble(array, 10, Less);  
    .....  
}
```

# 函数指针应用3 - 冒泡排序

```
void Bubble(int array[ ], int n,  
            int (*compare)(int a, int b))  
{  
    // 回调函数  
    int i, j, t;  
  
    for (j = 0; j < n-1; j++) {  
        for (i = 0; i < n-1-j; i++) {  
            if (compare(array[i], array[i+1])) {  
                t = array[i];  
                array[i] = array[i+1];  
                array[i+1] = t;  
            }  
        }  
    }  
}
```

回调函数就是一个通过函数指针调用的函数。如果把函数的指针(地址)作为参数传递给另一个函数，当这个指针被用来调用其所指向的函数时，就说这是回调函数

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

```
int Less(int a, int b)  
{ return (a<b); }
```

```
int main()  
{  
    .....  
    int (*fun[])(int a, int b) =  
        {Large, Less};  
    if (k >= 0 && k <= 1)  
        Bubble(array, 10, fun[k]);  
    .....  
}
```

# 函数指针应用3 - 冒泡排序

```
int Large(int a, int b);
int Less(int a, int b);
void Bubble(int array[ ], int n,
    int (*compare)(int a, int b));
typedef struct {
    char* name;
    void (*cmd)();
} SC;

int main()
{
    SC cmds[] = {
        {"Ascending", Large},
        {"Descending", Less}
    };
```

```
int k;
char cmdstring[20];
int array[10]={6,3,5,7,4,2,9,8,0,1};
printf("Ascending or Descending?");
scanf("%s", cmdstring);
for (k = 0; k < sizeof(cmds)/sizeof(cmds[0]);
    k++) {
    if (strcmp(cmdstring, cmds[k].name) == 0)
        Bubble(array, 10, cmds[k].cmd);
}
for (k=0; k<10; k++)
    printf("%d ", array[k]);
printf("\n");
return 0;
}
```

# 函数指针应用4 - 通用排序

- 冒泡排序：相同数据类型，不同比较函数
  - 函数指针
- 通用排序：不同数据类型，不同比较函数
  - 通用指针 `void *`，函数指针 (比较、交换)
- C语言函数库自带的排序函数 `qsort (stdlib.h)`

```
void qsort (void* base, size_t num, size_t size,  
            int (*compar)(const void*, const void*));
```

- `base` - 指向要排序的数组的第一个元素的指针
- `num` - 由 `base` 指向的数组中元素的个数
- `size` - 数组中每个元素的大小，以字节为单位
- `compar` - 用来比较两个元素的函数，即函数指针 (回调函数)

# 函数指针应用4 - 通用排序

- **qsort**是通用的快速排序函数，只知道数组个数和每个元素的字节数，(类库设计者)对数据类型和排序方式未知
- (类库使用者)提供**compar**比较函数，供**qsort**函数回调，(类库使用者)知道具体数据类型和排序方式

`int (*compar)(const void *p1, const void *p2);`

- 返回值小于0: p1所指向元素会被排在p2所指向元素的左面
- 返回值等于0: p1所指向元素与p2所指向元素的顺序不确定
- 返回值大于0: p1所指向元素会被排在p2所指向元素的右面

```
// 一维整数数组从小到大
```

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

```
int a[10] = {6,3,5,7,4,2,9,8,0,1};  
qsort(a, 10, sizeof(int), comp);
```

# 函数指针应用4 - 通用排序

// 一维双精度浮点数数组从大到小

```
int comp(const void *p1, const void *p2)
{   return *(double*)p2 - *(double*)p1; }
qsort(a, 10, sizeof(double), comp); // double a[10];
```

// 二维整数数组按a[1]从小到大

```
int comp(const void *p1, const void *p2)
{   return ((int*)p1)[1] - ((int*)p2)[1];   }
qsort(a, 10, sizeof(a[0]), comp); // int a[10][2];
```

// 字符串从小到大

```
int comp(const void *p1, const void *p2)
{   return strcmp((char *)p1, (char *)p2); }
qsort(a, M1, sizeof(a[0]), comp); // char a[M1][M2];
```



# 函数指针应用4 - 通用排序

```
struct Node {  
    double data;  
    int other;  
} s[100];  
// 结构数组按某个属性排序 (一级排序)  
int comp1(const void *p1, const void *p2) {  
    return (*(Node*)p1).data - (*(Node*)p2).data;  
}  
qsort(s, 100, sizeof(s[0]), comp1);  
// 结构数组按多个属性排序 (二级排序)  
int comp2(const void *p1, const void *p2) {  
    Node *m = (Node*)p1, *n = (Node*)p2;  
    if (m->data != n->data) return m->data - n->data;  
    else return m->other - n->other;  
}
```

# 函数指针

- 指向函数的指针
  - 定义
- 作用
  - 简化代码，减少重复代码
  - 提高代码的通用性和可扩展性
- 应用
  - 回调函数 (广泛应用于**GUI**框架)
  - 通用代码 (例如**C++**的**template**)

# 专题一 指针进阶

- C语言基础知识回顾
- 有序表的操作 (10.1)
- 内存动态分配 (8.5)
- 指针数组、二级指针、数组指针 (11.1)
- 函数指针 (11.2.3)