



中国科学技术大学
University of Science and Technology of China

计算机程序设计

Computer Programming



指针



主讲：李卫海

目录

CONTENTS

地址与寻址

指针类型

指针与数组

指向指针的指针

指针与函数

◎ 变量名到地址的映射

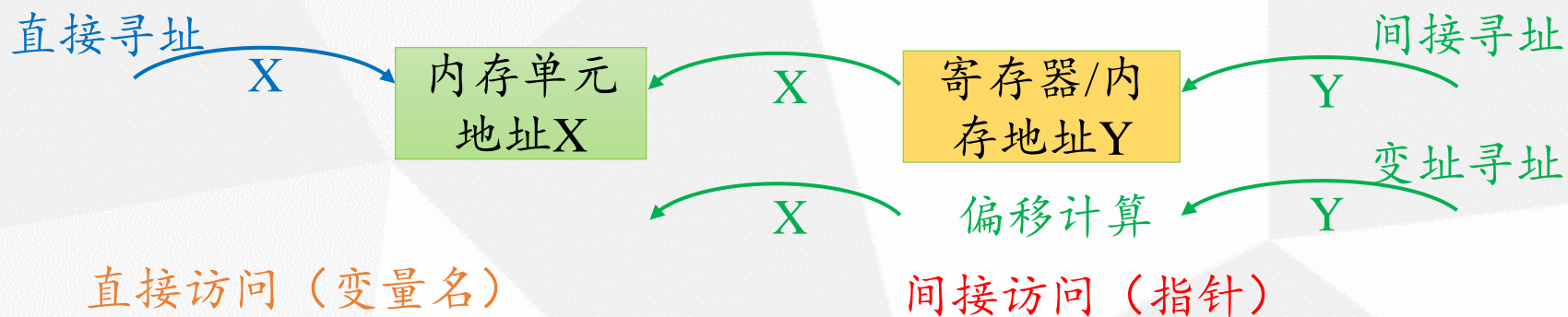
- **简单说**：程序运行过程中，CPU根据变量名与内存地址的映射关系，在该变量所在的**地址**进行数据操作
- **实际上**，从变量名到地址的映射
 - 程序编译后，变量名映射为逻辑地址
 - 程序加载到内存中时，逻辑地址映射为物理内存地址
 - 也有操作系统将逻辑地址映射为逻辑内存地址，再由操作系统维护逻辑内存地址到物理内存地址的映射
- 程序运行过程中，CPU通过**地址**对内存单元进行数据操作



◎ 寻址

• 汇编或机器语言的寻址

- 通过地址值读写，称为**直接寻址**
- 通过保存在寄存器或其它内存中的地址值读写，称为**间接寻址**
- 在间接寻址基础上，通过计算得到修改的地址，再进行读写，称为**变址寻址**



• 高级语言,

- 通过**变量名**读写变量值称为**直接访问**
- 通过**指针**(pointer)操作变量地址来读写，称为**间接访问**

◎ 指针的概念

• 地址、指针的概念

- **地址**：内存单元的编号
- **指针**：指向变量的地址，如 **&i**、**&j**、**&k**、**&h**、**&ptr**
- **指针变量**：存放其它变量地址的变量。如图中 **ptr** 指向变量 **i** 的地址 **i**

```
int i=3, j=6, k=9, h;  
int *ptr=&i;  
*ptr = 4;
```

上述定义中的*表示
ptr是一个指针变量

通过指针变量ptr可以
引用变量i的内容

指针变量 **ptr**
变量 **h**
变量 **k**
变量 **j**
变量 **i**

内存

内容	地址	
&i(0x2010)	0x2000	← &ptr
	0x2004	← &h
9	0x2008	← &k
6	0x200C	← &j
3	0x2010	← &i

◎ 指针类型变量的声明

- 指针变量也占据内存空间（在32/64位系统上占4/8字节），但由于指针变量的内容是其所指变量的（起始）地址，因此只有依托其所指向的地址才有实际意义
- 指针变量使用指针运算符“*”来声明，一般形式：

类型 *指针变量名;

如：int *ip; //一个可以指向整型变量的指针变量ip

char *next; //一个可以指向字符型变量的指针变量ip

float *fp; //一个可以指向float型变量的指针变量ip

此后需将该指针变量指向该类型的某个变量（或地址）



◎ 指针类型变量的声明

- 例,

```
int a, b;
```

```
int *pointer_1, *pointer_2; //定义指针变量
```

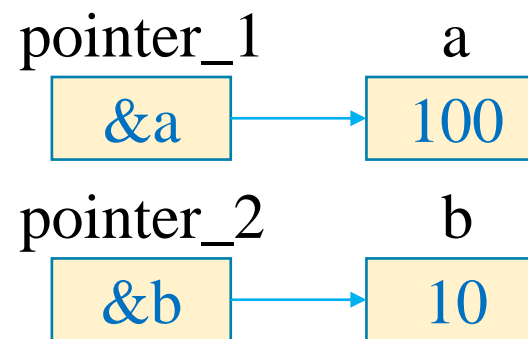
```
a=100; b=10;
```

```
pointer_1=&a; //指针变量赋值
```

```
pointer_2=&b;
```

```
printf("%d,%d\n",a,b);
```

```
printf("%d,%d\n",*pointer_1,*pointer_2); //指针间接取内容
```



◎ 指针类型变量的声明

- 指针变量声明的类型，是指针所指向变量的类型（如字符型指针应指向字符型变量）
- 编译器依据指针变量声明的类型，来读写、理解指针所指向地址的内容
 - 例如：

```
int *ip, i;  
float *fp, f;  
ip=&i; fp=&f;  
ip=&f; fp=&i;
```

正确

错误，但编译器未必报错
将按float型格式到 i 开始的地址读取

- 原则上，不同类型的指针既不能互相转换，也不能互相赋值(因不同类型数据在内存中的存放方式不同)
- 但不管指向何种类型的对象，指针变量本身占用内存的字节数都是一样的，必须能把程序中所用的最大地址表示出来（通常是一个机器字长）



◎ 指针类型变量的声明

- 指针变量声明的类型，是指针**所指向变量的类型**（如字符型指针应指向字符型变量）
- 编译器依据指针变量声明的类型，来**读写、理解指针所指向地址的内容**
- 指针变量的值是其所指向变量的地址，而地址是内存单元的编号，是**带有类型属性的特殊整数**
- 不同类型变量不仅占用内存的字节数不同，而且**对内存单元的起始地址也有不同要求**（与操作系统有关），如int、float、double要求偶数（或4的倍数）地址



◎ 指针类型变量的声明

- 指向复杂数据类型的指针（后续章节陆续介绍）

`char (*ptr)[5];` //ptr是指向由5个字符元素构成的数组的指针变量

`int **ip;` //ip是一个“指向整型量的指针变量”的指针变量

`int *fip();` //fip是返回整型指针的函数（函数声明）

`int (*pti)();` //pti是指向一个返回整数的函数的指针变量

`int *(*pfpi)();` //pfpi是一个指向函数的指针变量，该函数返回指向整数的指针

- 如何区分函数和指针？看运算符的优先级(前缀*优先的是指针，`int *fip()`中()`优先级更高`，因此不是指针)
- 需要注意的是，以上只是定义指针变量，并不对应实际的变量，必须指向实际变量后才
有意义



◎ 指针相关运算符

- 取地址运算符&

- 其操作对象必须是左值（在内存中有确切的单元地址），而不能是表达式或常量
- 例，scanf("%d",&a); 将键盘输入的串转换成十进制整数，送入变量a所在的地址单元

- 间接访问运算符*

- 操作对象必须是指针变量或指针表达式，即，对象已指向内存中确切的单元，如

int a,*p; //此处的*是定义符，不是运算符

p=&a; //指针p指向变量a

p是指针变量，*p则是指针所指向的变量(如a)

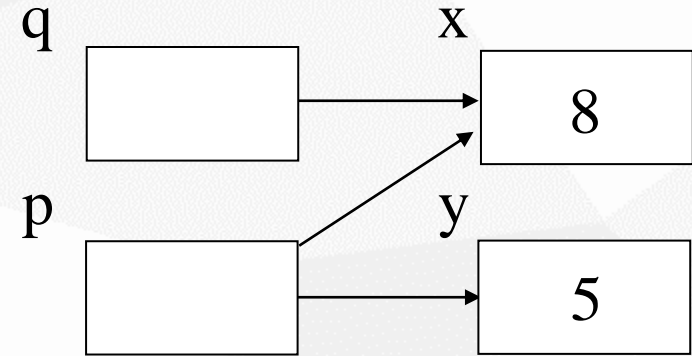
a=100;与*p=100;等效，后者先取出p的值（地址），再把100放入p值所表示的内存单元中

*p=*p+10也等价于a=a+10



◎ 指针相关运算符

```
int *p,*q, x, y;  
  
q=&x; *q=3;  
p=&y; *p=*q;  
*p=5;  
p=q; *p=8;  
printf("%d %d %d %d\n",*p,*q,x,y);  
//输出结果为8 8 8 5  
  
//p=NULL;      //将p指针置空  
//p=100; //报错
```



- 在变量定义时，*用来表示其后所跟的标识符是一个指针变量名，如`int *pi;`
`char *pc; float *pf;`
- 在表达式中，*用来访问地址表达式所指向的内存空间，其内容可变，可以作为左值，与变量等效

◎ 指针相关运算符

- 使用指针变量前，**务必要**将指针**指向确定的地址**
 - 通常要求该地址存放的数据的类型应与指针类型一致
 - **指针变量不能用一般常量值（不能表征该地址的数据类型）初始化**
- 指针变量是**存放地址量**的变量，地址量**并非普通的整数类型**，指针变量的加减并非整数的加减，而是依据**指针声明时的数据类型**来确定指针**单位增量**的长度，进行**地址的前后移动**
 - 例如：float *fp, x[]={1.0, 2.0, 3.0};
 - 若 fp=&x[1];
则*fp取x[1]的内容2.0, *(fp-1)取1.0, *(fp+1)取3.0, fp每次增减1都在内存中向后或向前移动4个字节，并按float类型访问该地址中的数据
 - 若 float (*fp)[3]=&x;
则fp每次增减1都在内存中向后或向前移动4*3个字节



◎ 指针相关运算符

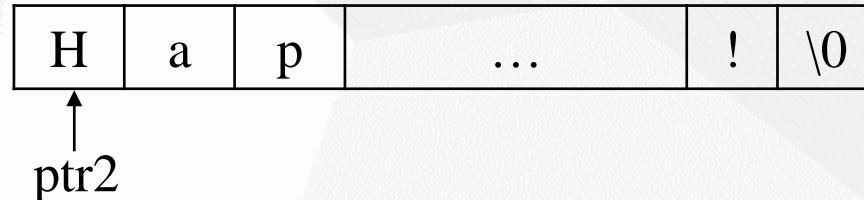
- 两个指针可以相减，或进行关系比较，但必须是**相容指针**（所指对象是同一类型，且**指向同一数据集合**）
 - 相减的结果是相差的**该类型数据的个数**
- 两个指针不可以相加（无意义）
- 例：int b, a[10]={1,2,3,4,5,6};
则 $\&(a[10])-\&(a[5])$ 的值为5， $\&(a[10])>\&(a[5])$ 的值为TRUE



◎ 指针相关运算符

- 例，按正向和反向顺序打印一个字符串

```
#include<stdio.h> //c8-e2.c
main() {
    char *ptr1, *ptr2;
    ptr1="Happy New Year!";
    ptr2=ptr1;
    while(*ptr2!=0) putchar(*ptr2++); //正向打印
    putchar('\n');
    while(--ptr2>=ptr1) putchar(*ptr2); //反向打印
    putchar('\n');
}
```



◎ 指针相关运算符

深入理解 “++”、“--”与指针变量的结合使用

- 注意*和&的优先级低于后缀++和--
 - *p++等价于*(p++), 表示先取值, 然后p再指向后继元素
 - *++p等价于*(++p), 表示p先指向后继元素, 然后再取值
- 例,

```
int a[10], *iptr, *q, u, v=5;
```

```
iptr=&a[v];
```

```
a[4]=40; a[5]=50; a[6]=60; //以下语句各自独立, 不相关
```

- q=iptr++; //q指向a[5], iptr指向a[6]
- u=*(iptr++); //u的值为50, iptr指向a[6]
- u=*iptr++; //同 u=*(iptr++);
- u=(*iptr)++; //u的值为50, iptr指向a[5], a[5]的值为51
- q=++iptr; //iptr指向a[6], q指向a[6]
- u=*(++iptr); //iptr指向a[6], u的值为60
- u=*++iptr; //同 u=*(++iptr);



◎ 特殊指针

- **空指针**NULL是C指针类型中的一个特殊值，表示指针变量的值为空，即不指向任何内存单元，常用于初始化指针变量

如： `int *p=NULL;` `//NULL的值为0，但不是地址0H`

- **无效指针**是一个指针变量无值，如指针变量未赋值或指针运算超出预想范围（数组越限），后果不确定
- **空指针**是用void声明的一种通用指针变量（万能指针）
 - 声明形式： `void *变量名;`
 - 其它类型指针可以直接赋值给void指针
 - void指针赋值给其它类型指针时，必须进行强制类型转换
 - void指针不能直接取值或运算，一般应将void指针强制转换后再进行有关操作



◎ 特殊指针

- 指针可以强制类型转换，实现不同类型指针变量间的交叉赋值，但程序员必须非常清楚自己在做什么
- 例，

```
int vi1, vi2, *ip;
```

```
float vf, *fp;
```

```
void *xp;
```

```
ip=&vi1;      //正确
```

```
fp=&vf;       //正确
```

```
ip=fp; fp=ip; //错误，类型不匹配（有的编译器只警告，可以运行，但结果错误）
```

```
xp=fp; xp=ip; //正确，使用时必须类型转换，比如：printf(" %d ", *(int *)xp );
```

```
ip=xp;        //错误，类型不匹配（有的编译器可通过编译，但结果错误）
```

```
ip=(int *)xp; //正确
```



◎ 指针与数组

- 声明变量时的**数组名**是一个**符号地址常量**，可以理解为指针常量，指向数组的首元素地址，是**右值**，不能对数组名赋值
- 除了数组名的值不能改变外，数组名和指针在很多地方可以通用
- 例如 `int a[5], *iptr=a;`
 - `iptr=a;` 等价于 `iptr=&a[0];` 即取元素地址
 - `a[3] ↔ *(a+3) ↔ *(iptr+3) ↔ iptr[3]`
 - **`*(iptr+3)`**的运算过程：取出指针变量iptr的值，加上 $3 \times \text{sizeof}(\text{int})$ ，得到新地址 `&a[3]`，再取值得到内容a[3]
 - **`a[3]`**的运算过程：取出数组首地址，加上 $3 \times \text{sizeof}(\text{int})$ ，得到新地址 `&a[3]`，再取值得到内容a[3]



◎ 指针与数组

- 例，指针与数组的效率比较

```
#include<stdio.h>
main() {
    int a[5]={ 1,2,3,4,5};
    int i,*p;
    for(i=0;i<5;i++) printf("%6d",a[i]); //下标法
    printf("\n");
    for(i=0;i<5;i++) printf("%6d",*(a+i)); //数组名法
    putchar('\n');
    for(p=a;p<a+5;p++) printf("%6d",*p); //指针变量法
    printf("\n");
} //3条printf运行结果一样
```

效率相同

效率略高

- 若是按递增或递减顺序访问数组，用指针，效率高
- 若不确定多维数组的高维大小，用指针，灵活
- 若随机访问数组，用下标，简洁明了



◎ 指针与数组

- 例，猜意图，找错误

```
#include<stdio.h>
main() {
    int i, a[10], *p;
    p=a;
    for(i=0; i<6; i++) scanf("%d", p++);
    for(i=0; i<6; i++) printf("%6d\t", *p++);
    printf("\n");
}
```

- 指针未复位
- 数组越界



◎ 例，利用指针访问二维数据，并输出

```
#include <stdio.h>
void main() {
    int a[3][4], i, j, *p, n=0;
    for (i=0; i<3; i++)
        for (j=0; j<4; j++)
            *(&a[i][j])=i*4+j;
    for (p=&a[0][0]; p<=&a[2][3]; p++) {
        printf("%6d", *p);
        if (++n%4 == 0) printf("\n");
    }
}
```

程序运行结果

0	1	2	3
4	5	6	7
8	9	10	11

◎ 指向数组的指针

- 定义形式：类型名 (*变量名)[数组大小]
 - 如：int (*pa)[4], pa是指向“由4个整型量构成的数组”的指针变量
 - 若有int a[2][4], 则可以pa=a;
 - 如：char (*next)[16], next是指向“由16个字符构成的数组”的指针变量
 - 若有char n[5][16], 则可以next=n;



◎ 指向数组的指针

- 例，被调函数的形参分别用指向基本类型的指针变量和指向数组的指针变量实现二维数组传递

```
void print_1(int *p, int n) {  
    ... *(p+i) ...}  
  
void print_2(int (*p)[4], int m, int n) {  
    ... *(*p+i)+j ...}  
  
void main() {  
    int a[3][4];  
    int (*pa)[4];  
    ...pa=a; print_1(*pa++, 4) ...  
    ...print_2(a, 3, 4) ...  
}
```



◎ 指针数组

- 可以定义由指针作为元素的数组
- 声明形式

类型说明符 *数组名[数组长度]

例如: `int *p[3]; char *name[]=...`; //注意`int (*p)[3]`与`int *p[3]`的区别

- 指针数组中的元素都是指向同一类型的指针
- 应用:
 - 若有大量数据（例如学生信息）需要按某个元素排序时（例如按姓名拼音顺序），使用指针数组排序，就可以避免大量数据的搬移
 - 当需要对多个已存在的对象进行批量处理时，也可以使用指针数据
 - 例如，游戏程序中各自独立行动的人物（不使用多线程），每人一个指针组成数组，就可以在一个循环中顺序处理所有人物
 - 例如，多线程游戏中，系统使用一个指针链表（类似数组）来顺序处理各个进程



◎ 指针数组

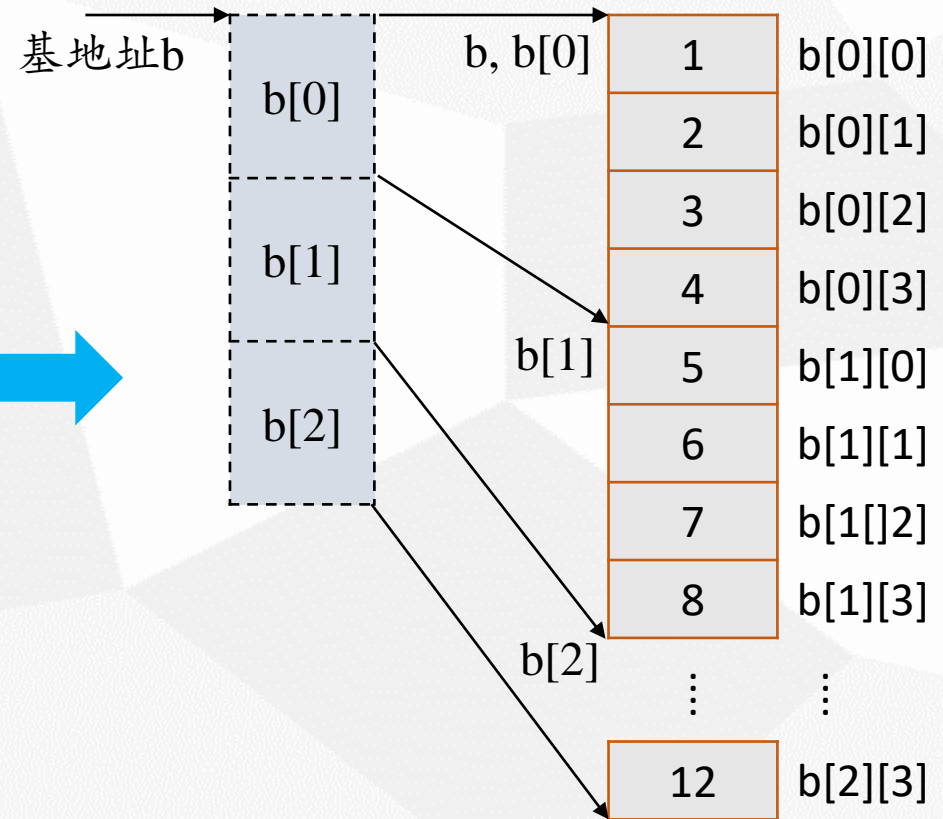
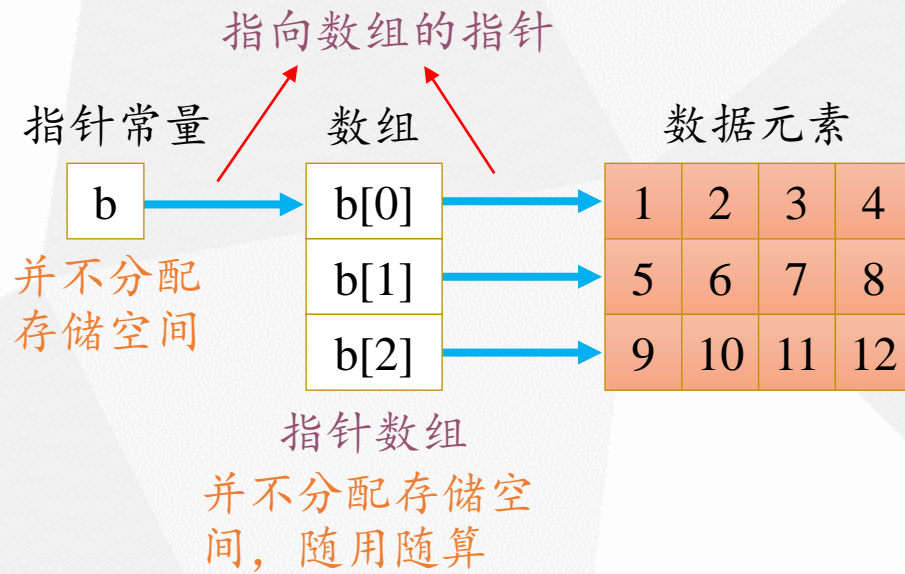
- 指针数组常用于处理一组字符串，比使用二维数组更方便，也节省内存空间
 - 例，`char *name[]={"allen", "word", "jones", "martin"};`
 - 字符串常量"allen"等又称为字符串字面量，它们被存储于常量区
 - 在代码中，字符串字面量可以等同于一个 `char *` 指针
 - 甚至允许 `"allen"[3]='a'`
 - 在这里，相当于在初始化器中列举了一系列字符指针，它们分别指向常数区的字符串
- 使用指针数组存储指向字符串的指针的优点
 - 允许不等长的字符串以相对规整的方式组织在一起，注意它们不一定存储在一起
 - 便于调整各字符串在数组内的位置：
修改指针即可

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



◎ 指针与数组的深入理解

- 多维数组虽然也是连续存储在内存中的元素集合，但语法上（编程使用时）是元素为数组的数组
- 例如：`int b[3][4]={ {1,2,3,4}, {5,6,7,8}, {9,10,11,12} };`



◎ 指针与数组的深入理解

- 数组取值时，每处理一个`[]`或`*`，相当于一次指针取值运算
 - 如 `int a[3][4];`
 - `a`、`a+i`，指向“由4个整型数构成的数组”的指针
 - `a[i]`、`a[i]+j`，指向整型数的指针，即 `*(a+i)`、`*(a+i)+j`
 - `a[i][j]`，整型数，即 `*(*(a+i)+j)` 或 `*(a+i*4+j)`
- 数组取地址时，`&`表示指向该类型元素的指针
 - 如 `int a[3][4];`
 - `&a[i][j]`，指向整型数的指针，值为 `*(a+i)+j` 或 `a[i]+j`
 - `&a[i]`，指向“由4个整型数构成的数组”的指针，值为 `a+i`
 - `&a`，指向二维数组的指针，值为 `a`
 - 注意`&a`与`&a[0]`的差异
 - `a[0]=a+0`，且类型一样，可以有`&(a+0)`吗？

注意，

- `a`的类型是二维数组；
- `a`的值是指向类型为“...数组”的地址`a[0]`的指针值
- `a[i]`的类型是“...数组”；
- `a[i]`的值是指向类型为整型的地址`a[i][0]`的指针值



◎ 指针与数组的深入理解

- 多维数组的指针进行赋值时，应特别注意类型的匹配
- 例，

```
int a[2][2]={ {1,2},{3,4} };  
int b[3][3]={ 2,3,4,5,6,7,8,9,10};  
int c[4]={ 12,13,14,15}  
int (*pa)[2];  
pa=a;  
pa=b;    //警告但通过编译  
pa=c;    //警告但通过编译
```

- pa将以自己的类型（指向包含两个整型数的数组的指针）去理解b和c数组中的数据



◎ 指针与字符数组

- 字符串可以视为一个字符序列
 - 字符数组存放在程序活动的栈中，可以修改
 - 字符串常数存放在常量区，不可以修改。
- 可以定义一个字符指针变量，让其指向一个字符串常量
 - 例如：char *p='abcde';
 - 实际上就是把字符串常量在内存中的首地址赋给该指针变量
 - 其地位等价于指向一维字符数组第一个元素的指针
 - 注意这个指针指向常量区，指针值可以改，指针指向的内容不可以改



◎ 指针与字符数组

- 可以用字符串常量对字符指针变量和字符数组初始化
- 可以用字符串常量对字符指针变量赋值
- 不可以用字符串常量对数组名赋值（试图修改数组名常量）

```
char *pstr1= "Hello", str1[40]= "Okay";
```

```
char *pstr2, str2[40];
```

```
pstr2= "world";
```

```
str2= "world"; //错误！
```

- 只能逐个元素地对数组赋值（for循环等方法）

◎ 指针与字符数组

- 当 **字符指针变量** 指向一个字符串（不需要是首字符）后，可以用 **数组下标的形式** 来引用该字符串中的字符

```
char *pstr= "Hello, world!";
```

```
char str[40];
```

```
int i;
```

```
for(i=0; *(pstr+i)!='\0'; i++) str[i]=pstr[i];
```

```
str[i]= '\0'; //此句不能少
```

//注意，*(pstr+i)等价于pstr[i]，即访问字符串中第i个字符



◎ 指向指针的指针

- 指向指针变量的指针变量：两次寻址/间接寻址
- 定义的一般形式

类型名 **指针变量名;

◦ 如:

```
int **p, *s, v;
```

```
p=&s; s=&v; v=300;
```

则 **p的值为300, 与*s和v的值相同



◎ 指向指针的指针

- 例， 5个学生， 每人所修课程门数不同（成绩存放在数组中， 以-1表示结束）， 编写程序输出他们的各项成绩

```
int stu1[]={...}, stu2[]={...}, stu3[]={...}, stu4[]={...}, stu5[]={...};  
int *gread[]={stu1, stu2, stu3, stu4, stu5};  
int **p=gread, i;
```

- 每人的成绩各用一个一维数组保存
- 指针数组 int *gread[], 每个元素指向一个学生的数组
- 指向指针的指针int **p=gread, 用于指向gread的元素

◎ 指向指针的指针

- 例， 5个学生， 每人所修课程门数不同（成绩存放在数组中， 以-1表示结束）， 编写程序输出他们的各项成绩

```
for (i=1; i<=5; i++) {  
    printf("student %d gread:", i);  
    while (**p > 0) {  
        printf("%4d", **p);  
        (*p)++;  
    }  
    p++;  
    printf("\n");  
}
```

```
int stu1[]={...}, stu2[]={...}, stu3[]={...},  
    stu4[]={...}, stu5[]={...};  
int *gread[]={stu1, stu2, stu3, stu4, stu5};  
int **p=gread, i;
```

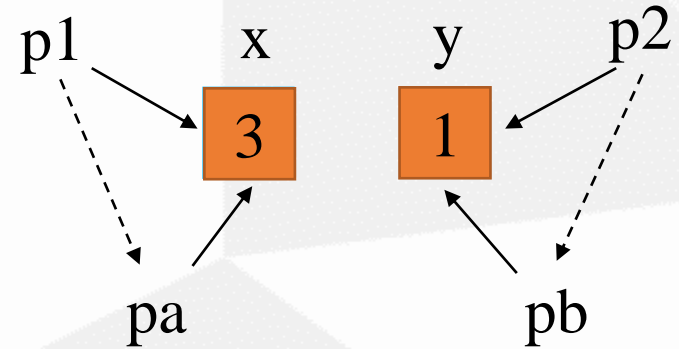
- **p取出一个成绩
- (*p)++修改了gread数组中某一个元素的值， 以指向该学生的下一项成绩
- p++使得p指向gread中的下一个元素



◎ 指针作为函数参数

- 指针作为函数参数时，传的是值，其内容是地址（有人也称它传地址调用）
 - 被调函数通过指针访问函数外部的数据——具有数据双向传递的功效

```
#include<stdio.h>
void swap2(int *pa, int *pb) {
    int t;
    t=*pa; *pa=*pb; *pb=t;
}
main() {
    int x=1, y=3, *p1=&x, *p2=&y;
    swap2(p1, p2);
    printf("%d,%d,%d,%d\n", x, y, *p1, *p2);
}
```



④ 指针作为函数参数

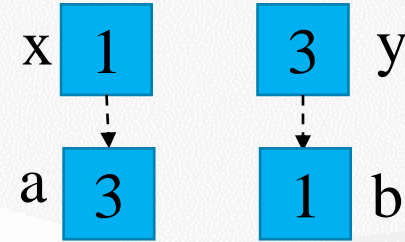
- 例，三种变量交换写法的区别

```
void swap1(int a, int b) {  
    int t;  
    t=a; a=b; b=t;  
}
```

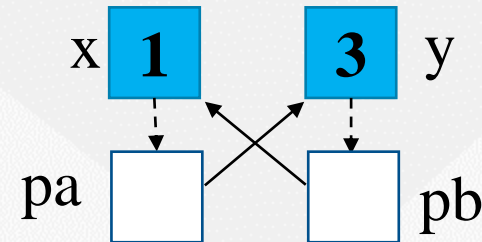
```
void swap2(int *pa, int *pb) {  
    int t;  
    t=*pa; *pa=*pb; *pb=t;  
}
```

```
void swap3(int *pa,int *pb) {  
    int *pt;  
    pt=pa; pa=pb; pb=pt;  
}
```

只交换形参a,b的值,
未修改main函数中的
变量值



交换了形参pa,pb指
针的值, 未修改pa,
pb所指变量的值



◎ 指针作为函数参数

- 当形参中声明了数组时，ANSI C规定，形参中的数组名作为指针变量使用（传递的是数组的首地址），是可以被赋值的

- 如 `int func(int a[])` 相当于 `int func(int *a);`

`int func(int a[][10])` 相当于 `int func(int *a[10])`

- 例，编写字符串复制函数 `udf_strcpy()`

```
void udf_strcpy1(char s[], char t[]) { //数组方式
    int i=0;
    while ((s[i]=t[i])!='\0') i++;
}
```

```
void udf_strcpy2(char *s, char *t) { //指针方式
//也可声明为void udf_strcpy2(char s[], char t[])
    while ((*s=*t)!='\0') {
        s++;
        t++;
    }
}
```

```
void udf_strcpy4(char *s, char *t) { //指针方式
    while (*s++=*t++);
}
```

注意，这里实现的函数是不安全的！没有考虑拷贝过程中字符串数组越界的错误。

◎ 指针作为函数参数

- 多维数组的指针可以通过参数传递，需要注意形参和实参的 **类型应匹配**
- 例，把某月的几号转换为这一年的第几天

```
int days(int (*p)[13], int year, int month, int day) {  
    ... *(p+leap) ... //leap=1(闰年), 0(平年)  
}  
void main() {  
    int day_tab[2][13]={ {31,28,31,30,...},{31,29,31,30,...}};  
    ... days(day_tab, y, m, d) ...  
}
```



◎ 指针作为函数参数

- 字符串常量本身也可以作为函数参数，其对应的形参是字符指针变量
- 如字符串处理函数，函数原型声明
 - 字符串连接函数 `extern char*strcat(char *dest, char *src);`
// src可以是字符串、字符数组、字符指针
// dest必须是字符数组或字符指针，不能是字符串
 - 字符串比较函数 `extern int strcmp(char *s1, char*s2);`
// s1和s2都可以是字符串、字符数组、字符指针
 - 格式化输出函数 `int printf(char *format, ...);`
// format可以是字符串、字符数组、字符指针
// ...表示该函数可以接收不定数量的参数，或称为可变参数，其实现使用了C语言的VA_LIST宏定义



◎ 指针作为函数参数

- VA_LIST宏定义的例子：在任意个整数参数中找出最大值（有兴趣的自学）

```
#include <stdarg.h>

int max_int(int n, ...) {    //...表示参数可变
    va_list ap;    //定义参数列表，实质是一个指针
    int i, current, largest;
    va_start(ap, n);    //初始化，指出可变长度部分开始的位置
    largest=va_arg(ap, int); //取回单个变量
    for (i=1; i<n; i++) {
        current=va_arg(ap, int);    //取回单个变量
        if (current>largest) largest=current;
    }
    va_end(ap);    //结束获取
    return largest;
}
```



◎ 指针作为函数参数

- VA_LIST宏定义的例子：自定义打印接口的实现（有兴趣的自学）

```
int my_printf(const char *fmt, ...) { //...表示参数可变
    va_list args; //定义参数列表，实质是一个指针
    static char gc_PrintfOut[1000];
    va_start(args, fmt); //初始化，指出可变长度部分开始的位置
    vsnprintf((char *) gc_PrintfOut, 1000, (char *) fmt, args); //使用vsnprintf一次性读取
    va_end(args); //结束获取
    puts("%s", (const char *)gc_PrintfOut); //使用
    return 0;
}
```

◎ 指针作为函数参数

- 有时传指针的目的不是为了让函数修改原变量，而是为了节省传递的数据量，实际只是读取变量的值
 - 用`const`来保护指针所指向的变量不被修改
 - 例如，`void f(const int *p);`函数内不得修改`p`所指向的对象，有时称`p`是指向常整数的指针
 - 对比，`void f(int * const p);`是指函数内不得修改指针`p`，但对`p`指向的对象没有约束
 - 对比，`void f(const int * const p);`在函数内`p`和`p`所指向的对象都不得修改
- 帮助编译器检查函数内部是否有非法的（模块设计时禁止的）操作



◎ 指针作为函数参数

- 例，用指针做函数参数实现字符串比较

```
#include<stdio.h>
int udf_strcmp(char *s, char *t);
main() {
    printf("%d\n",udf_strcmp("a string", "a strange"));
    printf("%d\n",udf_strcmp("a strange", "a string"));
    printf("%d\n",udf_strcmp("a string", "a string"));
}
int udf_strcmp(char *s, char *t) {
    for(;*s==*t;s++,t++);
    if(*s=='\0') return(0);
    return(*s-*t);
}
```



◎ 指针作为函数参数

- 例，借助void实现编写通用的数据交换函数

```
void genswap(void *a, void *b, int size) {  
    char t, *ac, *bc;  
    int i;  
    ac=(char *)a; bc=(char *)b;  
    for (i=0; i<size; i++, ac++, bc++) {  
        t=*ac;  
        *ac=*bc;  
        *bc=t;  
    }  
}
```



◎ 命令行参数的实现

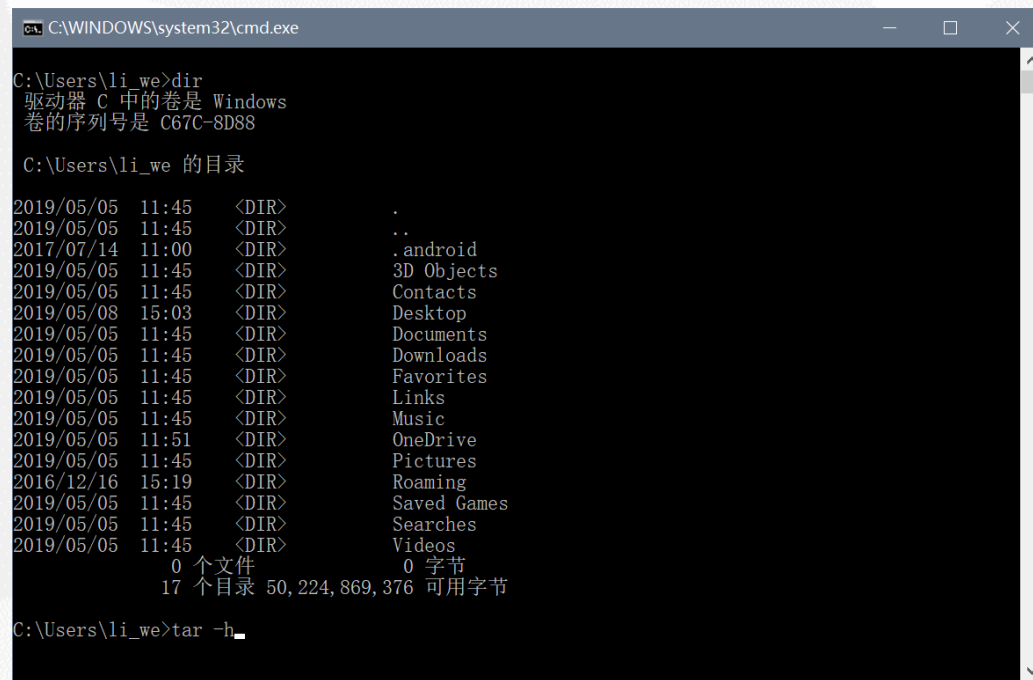
• 命令行参数

- 非图形人机交互界面下输入参数的一种方式
- 通过命令行将参数传递给程序，而不是在运行时输入
- 可以实现“静默”工作
- 例如：

`notepad.exe abc.txt`

`tar -h`

- 多个参数间用空格分隔



```
C:\WINDOWS\system32\cmd.exe

C:\Users\li_we>dir
驱动器 C 中的卷是 Windows
卷的序列号是 C67C-8D88

C:\Users\li_we 的目录
2019/05/05 11:45 <DIR>      .
2019/05/05 11:45 <DIR>      ..
2017/07/14 11:00 <DIR>      .android
2019/05/05 11:45 <DIR>      3D Objects
2019/05/05 11:45 <DIR>      Contacts
2019/05/08 15:03 <DIR>      Desktop
2019/05/05 11:45 <DIR>      Documents
2019/05/05 11:45 <DIR>      Downloads
2019/05/05 11:45 <DIR>      Favorites
2019/05/05 11:45 <DIR>      Links
2019/05/05 11:45 <DIR>      Music
2019/05/05 11:51 <DIR>      OneDrive
2019/05/05 11:45 <DIR>      Pictures
2016/12/16 15:19 <DIR>      Roaming
2019/05/05 11:45 <DIR>      Saved Games
2019/05/05 11:45 <DIR>      Searches
2019/05/05 11:45 <DIR>      Videos
                0 个文件          0 字节
                17 个目录 50,224,869,376 可用字节

C:\Users\li_we>tar -h_
```

◎ 命令行参数的实现

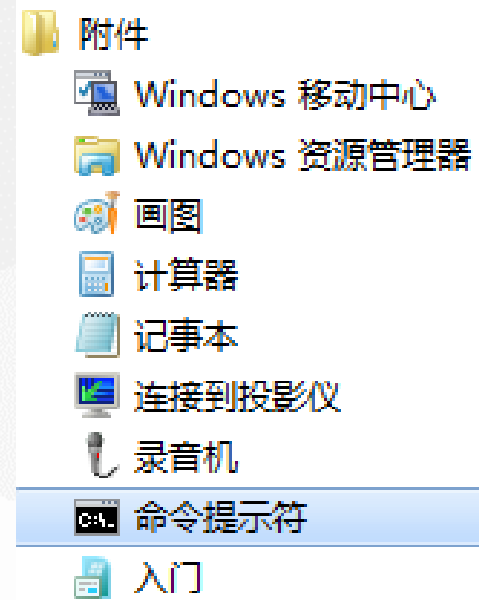
- windows操作系统下使用命令行

- 进入命令行界面（模拟DOS系统）

- “开始” → “所有程序” → “附件” → “命令提示符”
 - WIN-R，在输入框内输入“cmd”

- 一些常用DOS命令

- 改变当前目录：**cd** 目的目录
 - 显示当前目录内容：**dir**
 - 创建新目录：**mkdir** 目录名
 - 拷贝文件：**copy** 源文件 目标文件
 - 删除文件：**del** 文件名
 - 删除目录：**rmdir** 目录名



◎ 命令行参数的实现

- 给main函数添加参数，可以实现命令行参数

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

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

- argc代表传递给main的形参的个数，其值至少为1
- argv是指向字符串的指针数组。第一个字符串是程序执行文件名本身，第二个开始是真实参数
- 这两个参数由操作系统传递给main函数



◎ 命令行参数的实现

- 例，命令行参数获取

```
#include <stdio.h>
void main(int argc, char *argv[]) {
    int i=1;
    while (i<argc)
        printf("%s%c", argv[i++], (i+1<argc ? ' ' : '\n'));
}
```

```
#include <stdio.h>
void main(int argc, char **argv) {
    int i=1;
    while (i<argc)
        printf("%s%c", *(++argv), (i+1<argc ? ' ' : '\n'));
}
```



◎ 指针作为函数返回值

- 函数的返回值可以是**指针**（即某个对象的地址），又称为指针函数

声明形式：**类型名 *函数名(参数表);**

如：**double *func(int x, double y);**

- 指针函数的用法同一般函数
- 指针指向谁？
 - 字符串常数，**OK**，存在静态数据区
 - 外部变量，**OK**，存在全局数据区
 - 内部静态变量，**OK**，存在全局数据区
 - 调用函数的局部动态变量，**OK**，存在栈的深处
 - 被调用函数的局部动态变量，**NO**，将变为悬空引用



◎ 指针作为函数返回值

- 例，比较实参大小
 - `int larger1()` 返回较大的数值
 - `int *larger2()` 返回指向较大数值的变量的指针
- 例，输入月份后输出其英文名
 - 在被调函数 `char *month_name(int month)` 中定义英文名数组 `={"January", "February", ...}`;
 - 使用指向字符串的指针数组 `char *name[]`，字符串常量被存在静态数据区，返回的指针指向的内容有效
 - 使用二维字符数组 `char name[][16]`，在函数调用时分配临时内存，字符串常量被用于初始化，作为一个个字符存在临时内容中。调用结束后，`name` 占据的空间被释放，返回的指针指向的空间无效
 - 但有的时候结果仍会是对的！神奇吧？



◎ 指针作为函数返回值

- 地址值（数组名）参数传递方式是把数组传给函数的唯一方法？
- 一个函数只返回一个值，当需要返回多个值时，
 - 使用地址值参数传递方式，即调用者提供实参变量的指针；被调函数把返回结果放入相应单元中，如：

```
void fun(int *a, int *b, int n) {  
    b[0]=a[0]; b[1]=a[n-1];  
}
```

- 或者用指针返回多个值所在存储空间地址，如：

与外部变量的
区别是作用域
不同

```
int *fun(int *a, int n) {  
    static b[2];  
    b[0]=a[0]; b[1]=a[n-1];  
    return b;  
}
```


◎ 指针作为函数返回值

- 例，读入正文行并打印输出其中最长行
- 解题思路（算法描述）

`int getline(char *s, int lim);`

读入一行并返回其内容和长度
若长度为零，表示正文输入完毕

正文结束读入0个字符

while(还有其他行)

if(它比以前最长行还长)

保存它和它的长度

最后输出最长行

主函数main()

主程序预申请足够的空间，
即一行的长度上限

`void udf_strcpy(char *s, char *t);`



◎ 函数指针

- 函数语句存储在一片连续的内存单元中（代码区），其首地址就是函数的入口地址，函数名则代表该地址
- 主函数在调用子函数时，就是让程序转移到函数的入口地址去执行
- 程序也可以自定义指针，指向函数的入口地址，即指向函数，以此实现函数调用
- 定义形式：类型 (*指针变量名)(参数表);
 - 其中类型即函数返回值类型，参数表即函数形参
 - 如：float (*func)(int x, char y, float z);
 - 注意与 float *func(int x, char y, float z); 的区别



◎ 函数指针

- 和任何一个指针一样，指向函数的指针也**必须先初始化**（指向某个函数），然后才能使用

如：double square(double x); //函数声明

double (*p)(double x); //函数型指针声明

p=square; //指针初始化

s=p(1.6); 或 s=(*p)(1.6) //函数调用

- 注意，指向的函数的返回值类型和形参必须与指针变量中的定义相一致
- C语言规定，函数定义是不能嵌套的，整个函数也不能作为参数在函数间进行传送，如果需要进行传递，就必须使用指向函数的指针变量作为参数
 - 函数指针的主要作用是编写通用程序



◎ 函数指针

- 例，计算定积分

- 积分计算函数 `double collect(double a, double b, double prec, double(*p)(double x))`

- a、b是积分上下限，prec是精度，*p是被积分函数

- 被积分函数 `double func(double x)`

- 对自变量x求值

- 在collect中需要计算被积分函数值时，调用 `(*p)(x)`



- 任何一个傻子都能写出让电脑能懂的代码，而只有好的程序员可以写出让人能看懂的代码。

— Martin Fowler

- 好代码本身就是最好的文档。当你需要添加一个注释时，你应该考虑如何修改代码才能不需要注释。

——Steve McConnell, Code Complete 作者



A17. 编写一个函数，实现两个字符串的比较，替代strcmp()函数。

- 新函数应能防止字符串末尾的'\0'缺失

A18. 编写一个函数，实现两个字符串相减。

- 字符串相减，定义为从第一个串中，删除完整的第二个串（如果有的话）。如果第二个串在第一个串中出现多次，则只删除最后一个

A19. 输入一条消息，检查该消息是否是“回文”。



B9. 取胜之道：Program 国度的人，喜欢玩这样一个游戏，在一块板上写着一行数，共 n 个。两个游戏者，轮流从最右或最左取一个数。刚开始，每个游戏者的得分均为零。如果一个游戏者取下一个数，则将该数的值加到该游戏者的得分上，最后谁的得分最高谁就赢了游戏。输入 n 个数（从左往右），假设游戏者都是非常聪明的，问最后两个人的得分（假设第一个人首先取数）。

◦ 例如：

【输入】

6

4 7 2 9 5 2

【输出】

18 11

B10. 计算 a/b 的小数值， $0 < a < b < 32767$ 。若能除尽，则直接显示结果，例如 $3/5=0.6$ 。若不能除尽，则用循环节显示其精确结果，例如 $116/165=0.703(03)$ ，其中括号内标示出循环节。

- 这里指的是数学中的除法，不是计算机中的整数除法，因此结果不是0。
- 不得使用数组或动态分配空间等技术。



C4. 用Miller Rabin测试获取大素数（二进制形式）

- ① 输入大素数位数N
 - ② 产生N位随机数，确保末位为1（奇数）
 - ③ 测试该随机数是否能通过Miller Rabin测试，若不通过，则返回第(1)步
 - ④ 输出产生的大素数
- 尽可能提高你程序的效率，产生一个长度1024（或256）位的大素数

◎ Miller Rabin测试算法

- 对奇数 $n \geq 3$, $(n-1)$ 可表示为 $n-1=2^kq$, $k > 0$, q 是奇数
- 若 n 是素数, 则序列 $(a^q, a^{2q}, \dots, a^{2^{k-1}q}) \bmod n$ 中或者第一个元素为1, 或者某个元素为 $n-1$; 否则 n 是合数。
- 证明:
 - 考察序列 $a^q, a^{2q}, \dots, a^{2^{k-1}q}, a^{2^kq}$
 - 若 n 是素数, 则由费马定理可知, $a^{n-1} \bmod n = a^{2^kq} \bmod n = 1$ 。即此序列中至少会有一个1
 - 序列的最后一项是前一项的平方, 则第一个1或者是第一项, 或者其前面一项为-1



◎ Miller Rabin测试算法

- 测试 n 是否为素数：
 1. 计算奇数 q 和整数 k , 使得 $(n-1)=2^kq$
 2. 任选随机整数 a , $1 < a < n-1$
 3. if $a^q \bmod n = 1$ then return ("maybe prime");
 4. for $j = 0$ to $k-1$ do
 5. if $(a^{2^j q} \bmod n = n-1)$ then return("maybe prime")
 6. return ("composite")
- 若Miller-Rabin测试算法返回“合数”，则该数确定不是素数
- Miller-Rabin测试算法返回“不确定”，但不是素数的概率小于 $1/4$
- 重复算法 t (t 足够大) 次，若检测结果均不确定，则 n 是素数的概率为 $1-4^{-t}$
 - 例, $t=10$, 则此概率 >0.999999

