

C语言程序设计

The C Programming Language



第8章 指针

武汉光电国家研究中心

李春花



主要内容

- ◆ 指针的基本概念
- ◆ **指针参数**
- ◆ **一维数组和指针的关系**
- ◆ **指针数组**
- ◆ 带参数的main函数
- ◆ 指针函数
- ◆ **指向函数的指针**
- ◆ 指向数组的指针
- ◆ **综合应用**

8.1 指针的概念

- 变量占有一定数目(根据类型)的连续存储单元;

`short x;`

`char a[5];`

- 变量的连续存储单元首地址称为
变量的地址。

`&x, a <==> &a[0]`

- 思考: 用什么类型的变量来保存地址数据?

地址		变量名
0xFEBC		x
0xFEBC		
0xFEC0		a[0]
0xFEC1		a[1]
0xFEC2		a[2]
0xFEC3		a[3]
0xFEC4		a[4]

指针变量

➤ **指针**: 变量的地址

&x: 指针常量

p: 指针变量

➤ **指针变量**: 存放地址数据的变量。

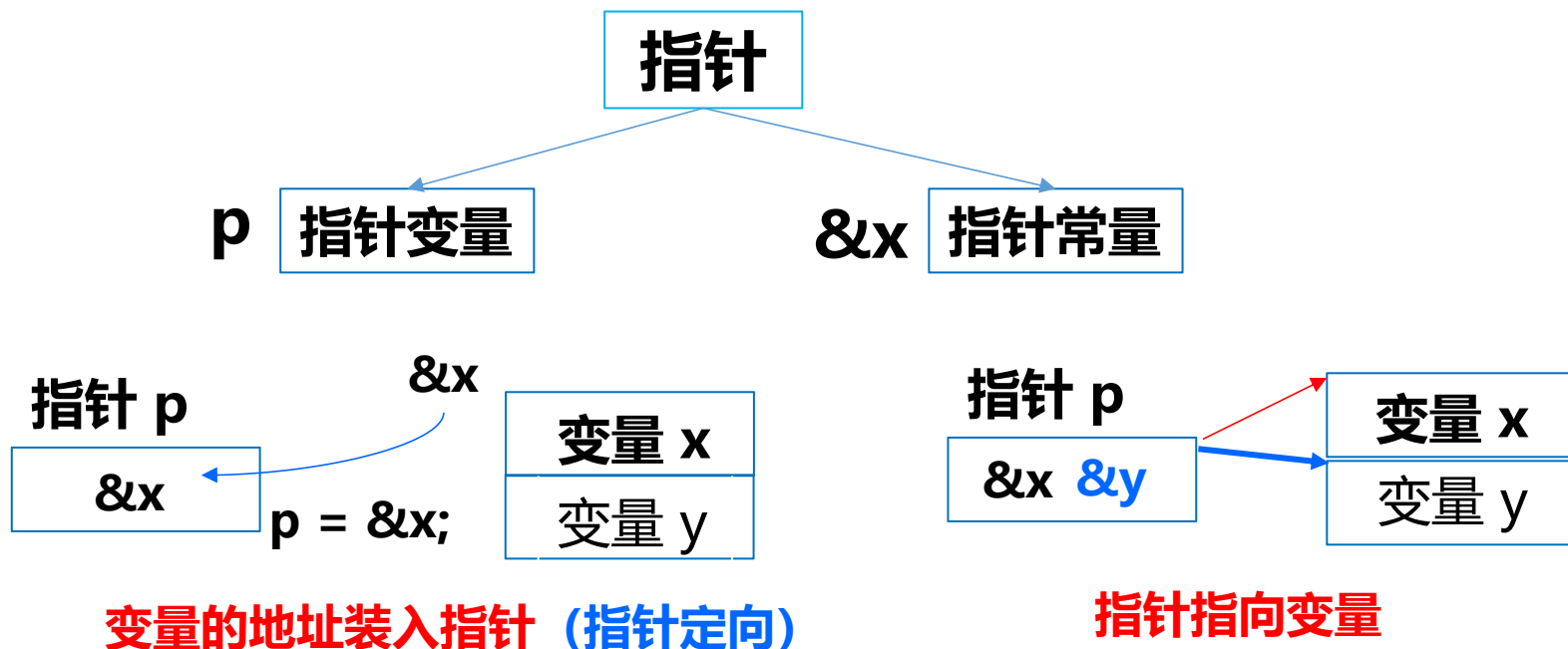
指针变量也是一种变量，也要占用一定的内存单元。

⚠ 指针变量所占存储空间一般等于机器字长

指针的特殊之处在于: 它存放的是另一个变量所占存储单元的起始地址。

地址		变量名
0xFEBC	1	x
0xFEBC	0	
	2	y
	0	
		p
0xFEC0		
0xFEC1		
0xFEC2		
0xFEC3		

指针变量



例: `int x, y;`
`int *p;`
`p = &x;`
`p = &y;`

变量 x,y称为指针p的目标变量

变量的两种访问方式

直接访问: 通过变量名存取变量 `x = 10;`

间接访问: 通过变量地址(**指针**)存取变量

`p = &x; *p = 10;`

指针变量的声明和初始化

数据类型 *指针名 [=初始地址值]; 地址

↓
所指 (即所指向) 变量的数据类型

```
short x=1, y=2, a[5];
```

```
short *p;
```

```
p=&x;    //可以将变量地址赋值给指针
```

```
p=a;    // 可以将数组名赋值给指针
```

```
p=a[0];    ×    不可用变量值给指针变量赋值
```

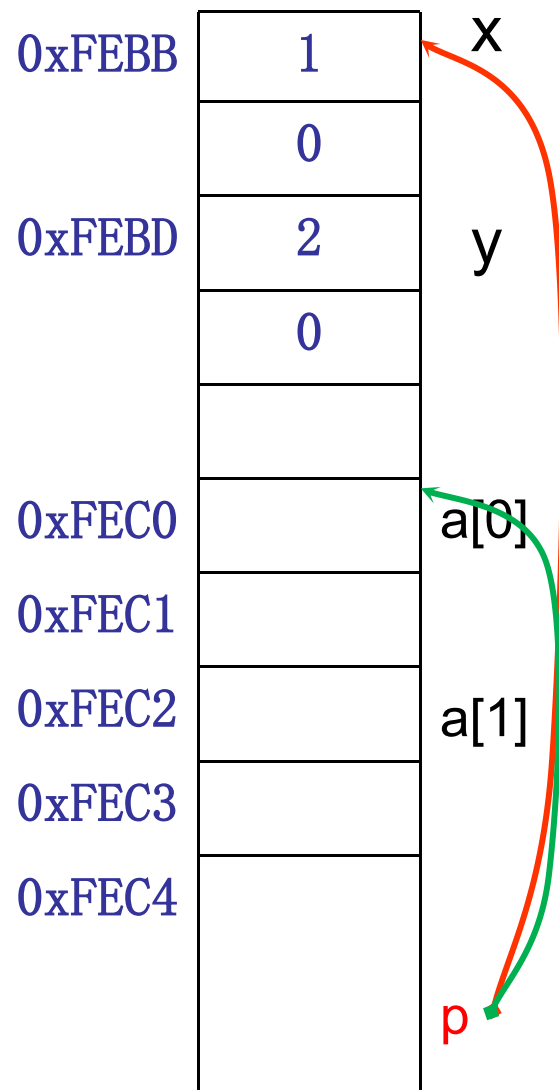
```
p=0xFEC0    ×    不可用常量给指针变量赋值
```

```
int n;
```

```
int *pn=&n;    //声明指针同时初始化
```

```
int *q=pn;    //可以用另一个指针变量给同类型的指针变量赋值
```

思考: 为什么使用指针?



变量的访问方式

- **直接访问：**通过变量名存取（或访问）变量

```
x=0x1234 ; printf( "%x" , x);
```

- **间接访问：**通过变量的地址（即指针）存取变量

```
int p=&x ;
```

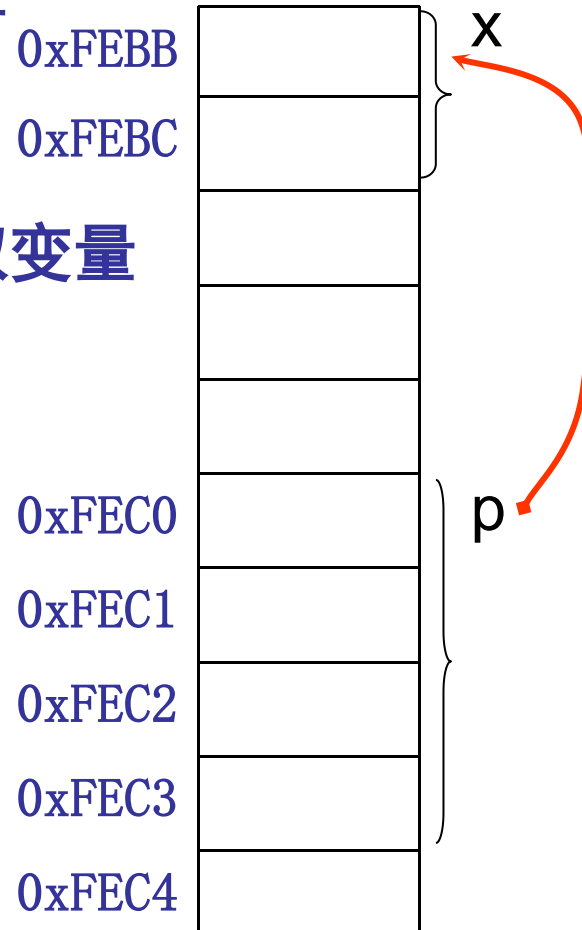
```
*p = 10; //通过运算符 * 取地址内容
```

```
printf( "%x" , *p);
```

先访问 p，得到x的地址，
再通过该地址找到它所指向的单元中的值。

地址

变量名



指针运算符 *

***操作数** /* 返回 操作数所指地址处的变量值。*/

short x=1, y=2, *p; // p是short型指针变量

p=&x; //将x的地址给p, 即指针 p指向 x

y=*p; // *p <=> x

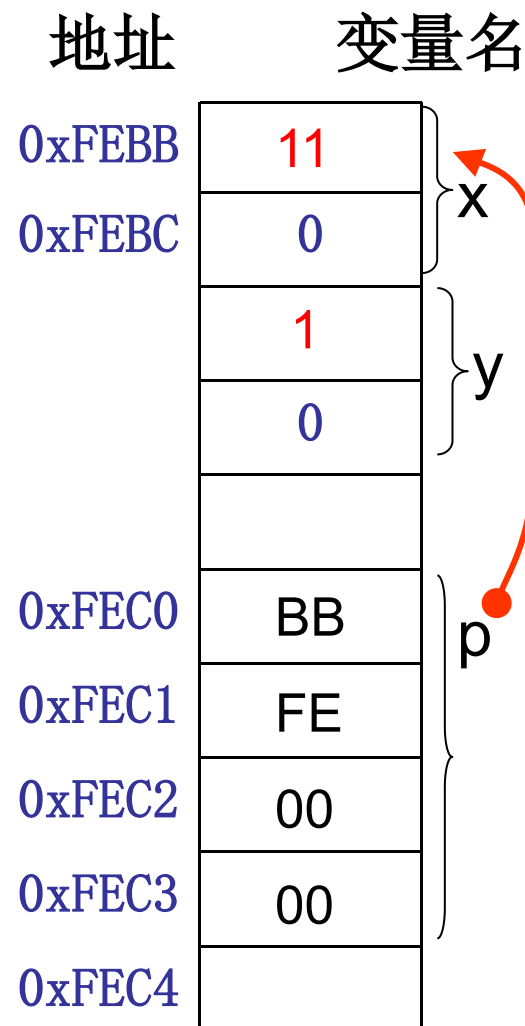
***p += 10; // 等价于 x += 10**

printf("x=%hd,y=%hd", x, y) ;

输出?

x=11, y=1

思考: 为什么指针有类型?



单目*和&的运算关系

单目*和 &互为逆运算

```
char c, *pc = &c;  
c = 'a';  
*pc = 'a';  
*(&c) = 'a';
```

注意区分*的不同含义

```
char *pc;
```

抽象指针说明符

```
*pc = 10;
```

间访运算符（单目*）

```
a = a*10;
```

乘运算符

***(& 左值表达式) = 左值表达式**

&(*地址表达式) = 地址表达式

```
int a [4];
```

```
&a[2]
```

int *

```
&a
```

错误

```
register int k;
```

```
&k
```

错误

C语言对寄存器变量直接寻址访问，因此不可显式取寄存器变量地址

野指针（悬挂指针）

- 指出下面程序段的错误:

```
int x,*p; /* 说明p是一个整型指针变量，其值不确定. */
```

```
x=25;
```

```
*p=x; /* 使用悬挂指针，错！ */
```

```
scanf("%d", p); /* 使用悬挂指针，错！ */
```

- 指针的声明只是创建了指针变量，获得了指针变量的存储，但并没有给出指针变量指向哪个具体的变量，此时指针的值是不确定的随机值，指针处于“无所指”的状态。称为悬挂指针（野指针），即未定向的指针。
- 不能使用悬挂指针

指针的正确使用

- (1) 首先，说明指针变量，声明其引用的变量类型；
- (2) 然后，给指针变量赋变量的地址值，使指针指向确定的目标对象；
- (3) 接着，使用指针间接访问所指向的变量。

例: `int *p;`

`*p = 5;` **×** 使用了未定向的指针
(野指针)

void型指针

所指向的数据类型不确定的指针

<存储类型> void *指针名;

```
int x, *px=&x;  
void *pp;  
pp = px;
```

可以将已定向的各种类型
指针**直接赋**给void型指针

```
int *px;  
void *pp = malloc(10*sizeof(int));  
px = (int *)pp;
```

void型指针给其它指针赋值时，必须采用强制类型
转换

NULL指针

- ANSI C++标准定义NULL指针，它作为指针变量的一个特殊状态，**表示不指向任何地址**
- 使一个指针变量为NULL，可以给它赋一个零值
- 测试一个指针变量是否为NULL，可以将它和零进行比较
- 对一个NULL指针进行引用操作是非法的

例

指针的输出

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```
    int a;
```

```
    int *pa = &a; // 指针pa指
```

```
    a = 10;
```

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

```
    printf("*pa:%d\n", *pa);
```

```
    printf("&a:0x%lx\n", &a);
```

```
    printf("pa:0x%lx\n", pa);
```

```
    printf("&pa:0x%lx\n", &pa);
```

```
    printf("&pa:%p\n", &pa);
```

```
    return 0;
```

```
}
```

E:\教学盘\未来学院\2022级\例程\指针.exe

a:10

*pa:10

&a:0x62fe1c

pa:0x62fe1c

&pa:0x62fe10

&pa:000000000062FE10



指针的运算

指针允许进行算术运算、赋值运算和关系运算。通过指针运算可以快速定位到指定的内存单元，提高程序的执行效率。



指针的赋值运算

(1) 同类型的指针可以直接赋值。如：

```
int a[3]={1, 2, 3}, x, *p, *q;  
p=a;   q=&x;
```

(2) 不同类型的指针必须使用类型强制转换再赋值。

```
long x;  
char *p;  
p = (char *) &x;
```


指针的移动

(+, -, ++, --, +=, -=)

指针的移动：指针在原有位置的基础上，通过加一个整数实现指针的**前移**（地址增大的方向）或者通过减一个整数实现指针的**后移**。

$p \pm k$
↑ ↑
指针 整型

p 前移/后移 k个元素

新的地址值是： $(p) \pm k \times \text{sizeof}(\text{数据类型})$ （字节）

指针的移动和类型有关

指针的移动操作

p ± n

前提：指针是指向一片存储单元
(如数组)

p1+n
p1-n

p1++

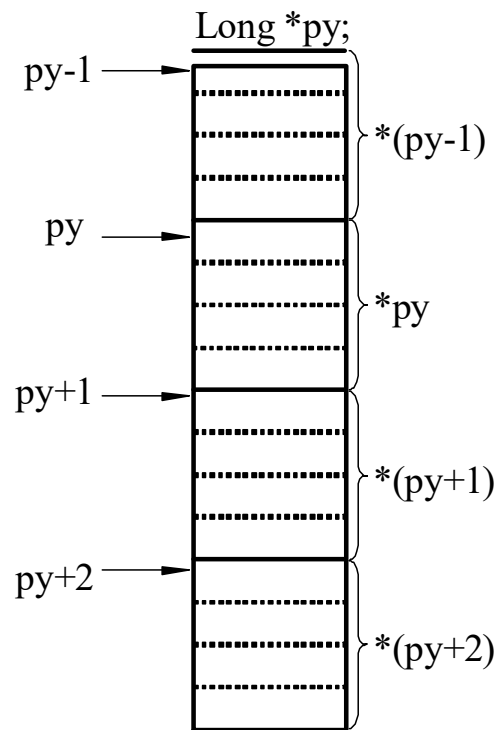
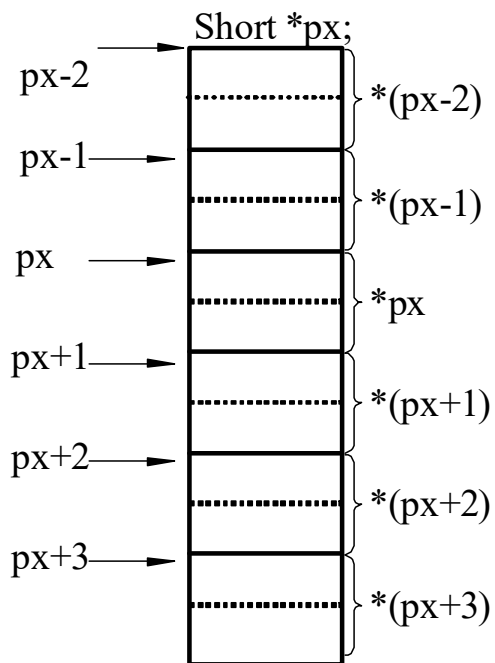
++p1

p1--

--p1

p1-p2

指针前移或后移n个数据位置，新的地址值是：
(p)±n×sizeof(数据类型) (字节)



指针的移动和类型有关

指针的移动操作

p++ / p--

前提：指针是指
向一片存储单元
(如数组)

p1+n

p1-n

p1++

++p1

p1--

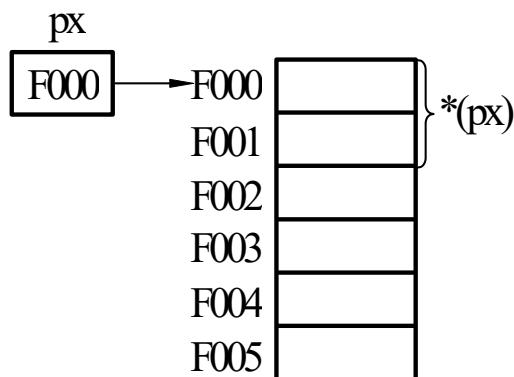
--p1

p1-p2

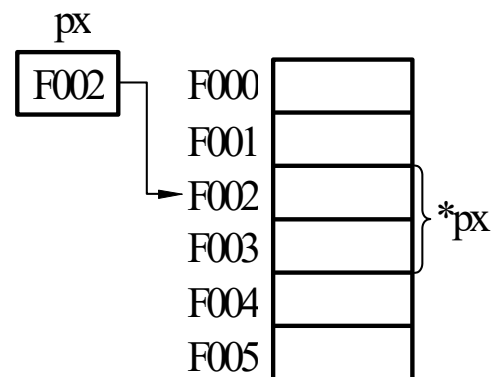


指针前移/后移一个数据位置，新的地址值是：
(p1) ± **sizeof(数据类型)** (字节)

int *px;



px++;



y = * px++; y = * (px++);

px的当前目标变量的值赋予y后，
px加1指向下一个目标

y = ++(*px); px的目标变量的值加一后赋予 y

与y = (*px)++ 等价吗?

两个指针相减

p1-p2

前提： 指针指向一片存储单元（如同一数组元素）

p1+n

p1-n

p1++

++p1

p1--

--p1

p1-p2

p1和p2指向**同一组**数据类型一致的数据

```
int a[100],n;  
int *p1=a;  
int *p2=&a[20];  
n = p2-p1; //20
```

```
int a[100],b,n;  
int *p1=a, *p2=&b;  
n = p2-p1; //100
```

结果为：两指针指向的地址位置之间的数据个数
 $((p1)-(p2))/sizeof(\text{数据类型})$

指针的移动操作

已知声明如下，指出下面表达式的值 (各题的表达式相互无关)

```
int x[5]={1,3,5,7,9}, *px=&x[1];
```

(1) ++*px; 4

(2) *++px; 5

(3) *--px; 1

(4) *(px--); 等价于 *px--; 3

(5) (*px--, *px); 1

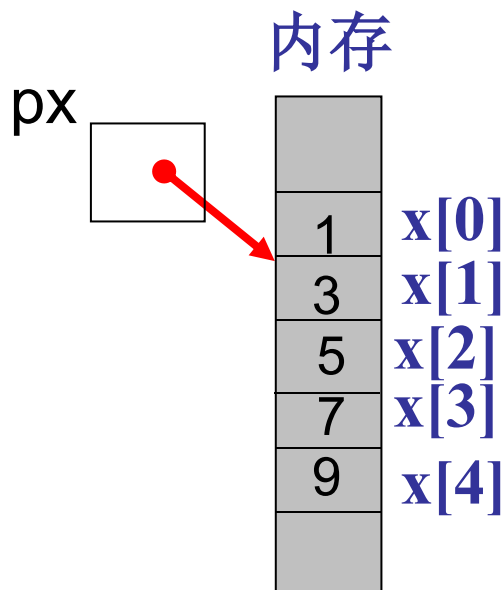
(6) *px++; 3

(7) (*px++, *px); 5

(8) (*px)++; 3

(9) ((*px)++, *px); 4

(10) (px+=2, *px); 7



指针的移动与数据类型有关

```
short x=0x1020, *p1=&x;
```

```
char *p2=(char *) &x;
```

```
int a,b;
```

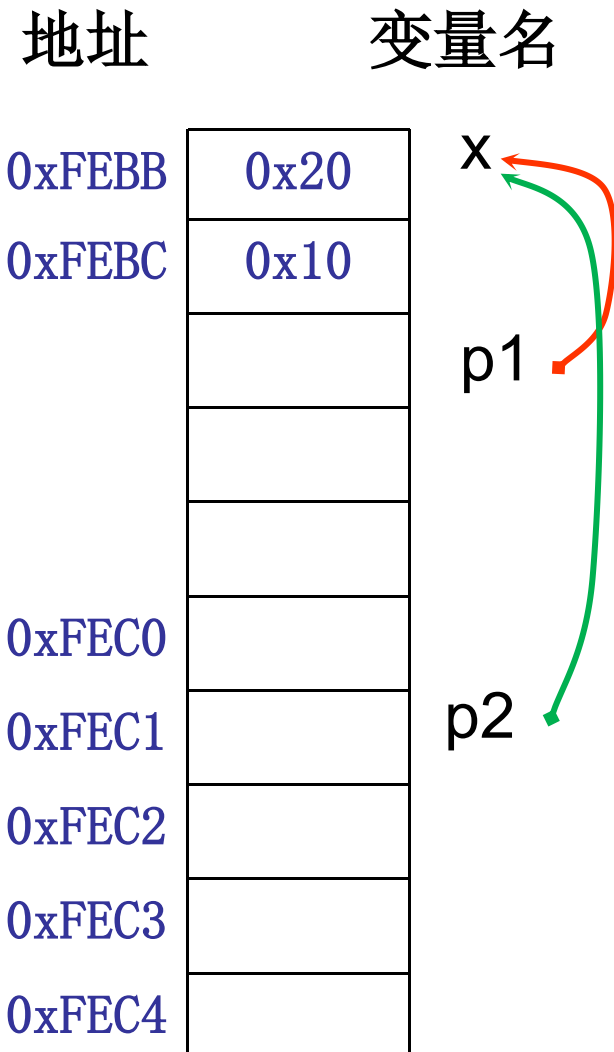
```
a=*p1;
```

```
b=*p2;
```

```
printf(“%x,%x”,a,b) ; // 1020, 20
```

```
b=*(p2+1);
```

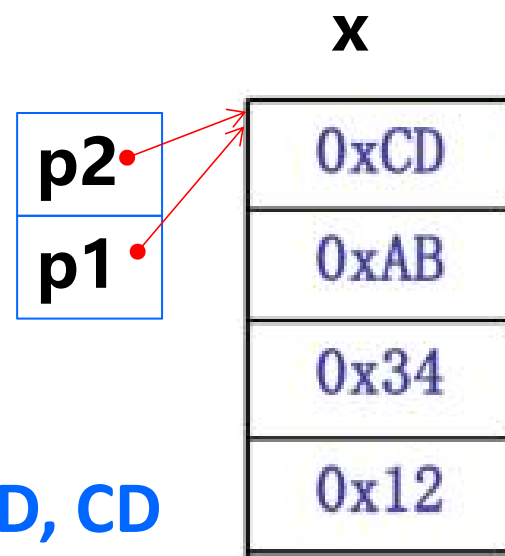
```
printf(“%x”,b); // 10
```



例

指针的移动与数据类型有关

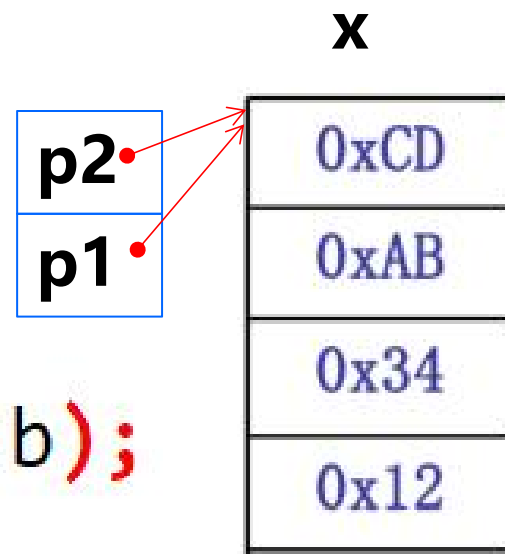
```
unsigned int x=0x1234ABcD, a, b, *p1;  
unsigned char *p2;  
p1=&x; p2=(char *)&x;  
a=*p1;  
b=*p2;  
printf("%x,%x\n", a, b);  
b=*(p2+1); // 1234ABCD, CD  
printf("%x", b); // AB
```



例

指针的移动与数据类型有关

```
int x=0x1234ABcD, a, b, *p1;  
char *p2;  
p1=&x; p2=(char *)&x;  
a=*p1;  
b=*p2;  
printf("%x,%x,%x\n", a, b, -b);  
b=*(p2+1);  
printf("%x,%x", b, -b);
```



```
1234abcd, ffffffffcd, 33  
ffffffab, 55
```


指针的赋值运算小结

向指针变量赋值时，赋的值必须是地址常量或变量，不能是普通整数

变量的地址赋给一个指向相同数据类型的指针

```
char c, *pc;  
pc=&c;
```

变量地址赋值

一个指针的值赋给同数据类型的另一个指针

```
int *p, *q;  
p=q;
```

指针变量赋值

数组的地址赋给同数据类型的指针

```
char name[20], *pname;  
pname=name;
```

数组名赋值

动态内存分配

必须使用类型强制转换再赋值

```
int *p, n=20;  
p=(int *)malloc(n*sizeof(int));  
if(p!=NULL)  
{ ... }
```

动态地址赋值

指针的关系运算

指针类型可以进行<、<=、>、>=、==和!=关系运算，运算结果为逻辑类型。

即：表达式成立“1”，不成立“0”

只限于同类型指针，
不同类型指针之间的关系运算被视为非法操作。

表示它们指向的地址位置之间的关系，指向后方的指针大于指向前方的指针[存储器的编号从小到大]

// 逆序输出数组a的全部元素

```
int a[10],*p=a ;
```

```
while(p<a+10) scanf("%d", p++);
```

```
while( p>a) printf("%d", *--p );
```

重点

动态内存分配/释放

动态内存分配: 在程序运行期间动态地分配存储空间,分配的内存空间放在数据区的堆 (heap) 中

指定所分配内存空间的大小

动态内存分配函数: `void * malloc(unsigned size);`

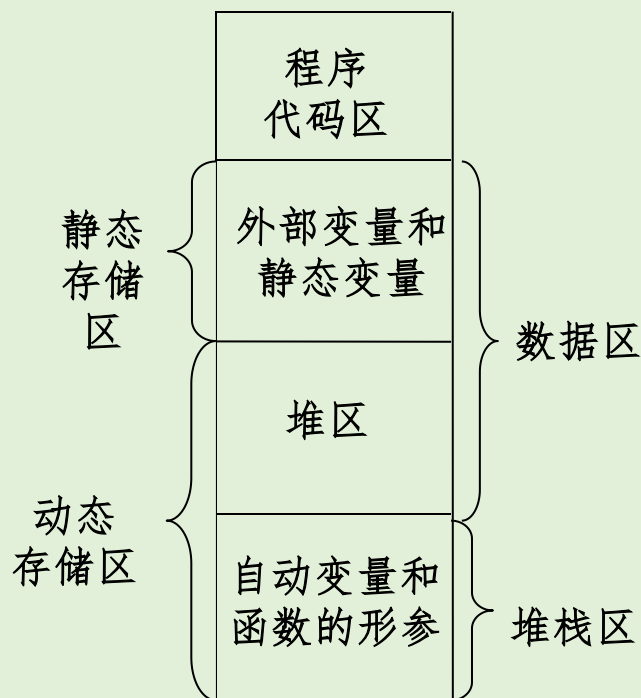
若分配成功, 则返回被分配内存空间的首地址, 否则, 返回空指针NULL。
当该内存不再使用时, 应使用`free()`函数将内存释放。

动态内存释放:

`malloc()`函数在堆区中所分配的内存空间首地址

`void free(void * ptr);`

**头文件: `#include <malloc.h>`或
`#include <stdlib.h>`**



动态内存分配/释放

malloc函数调用一般形式

```
#include <stdlib.h>
```

```
T *p=(T *)malloc(unsigned size)
```

//T为指针的数据类型

```
if(p == NULL) //
```

```
{
```

```
    出错处理操作;
```

```
    exit(1);
```

```
}
```

```
//.....
```

```
free(p);
```

```
p=NULL; //防止使用野指针
```

malloc函数使用示例

```
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>

int main(){ // 动态申请N个连续的int空间
    int i = 0, *p, N;
    printf( "Input array length:" );
    scanf( "%d" ,&N);
    p = (int*)malloc(N*sizeof(int));
    if(!p) return 1;
    for(i = 0;i<N;i++){
        p[i] = i+1; //等价于*(p+i)=i+1
        if((i+1)%10 == 0) printf( "\n" );
    }
    free(p);
    printf( "\n" );
    return 0;
}
```

运行结果:

Input array length:15

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15					

指针的应用一

通过指针变量输出a数组的5个元素

```
int main()
{
    int *p, i, a[5];
    p=a;
    printf("please input 5 numbers:\n");
    for (i=0; i<5; i++)
        scanf("%d",p++); // p+i, &a[i]
    printf ("/n");
    printf("the input array is:\n");
    for(p=a,i=0; i<5; i++)
        printf("%d",*p++);
    return 0;
}
```

通过p获得数组
a各元素的地址

通过p访问数组a
各元素的值

运行结果:

```
please input 5 numbers:
12 34 56 78 90
the input array is:
12 34 56 78 90
```

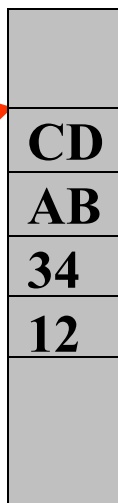
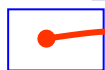
指针的应用二

例：一个长整型数占4个字节，其中每个字节又分成高4位和低4位。试从长整型数的低字节开始，依次取出每个字节的高4位和低4位，并以十六进制字符的形式进行显示。

char *p

内存

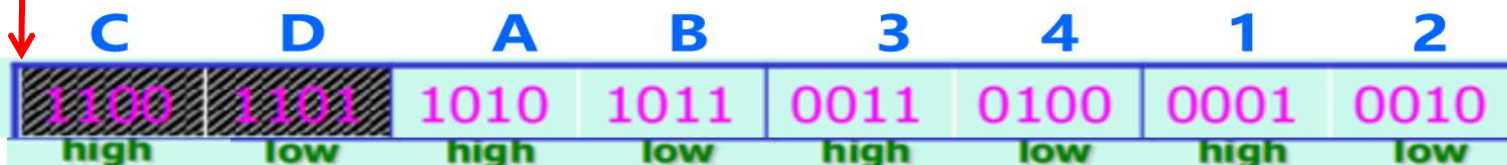
取低4位: $(*p) \& 0x0f$



$x = 0x1234ABCD$

取高4位: $(*p) \& 0xf0$??

$(*p >> 4) \& 0x0f$



```
#include <stdio.h>
```

```
int main(void)
```

```
{
```

```
    long x=0x1234ABCD,k;
```

```
    char *p=(char *)&x;
```

```
    char up_half,low_half;
```

```
    for(k=0;k<4;k++) { /* 依次取每字节 */
```

```
        low_half=(*p)&0x0f; // 取低4位
```

```
        if(low_half<10)
```

```
            low_half += '0'; // 低4位转成字符 '0'-'9'
```

```
        else
```

```
            low_half=(low_half-10)+'A'; //低4位转成字符 'A'-'F'
```

```
        up_half=(*p>>4)&0x0f; // 取高4位
```

```
        if(up_half<10)
```

```
            up_half |= '0'; // 等价于 low_half += '0';
```

```
        else
```

```
            up_half=(up_half-10)+'A';
```

```
        printf("%c  %c\n",up_half,low_half);
```

```
        p++; // 指向整型数x下一个字节
```

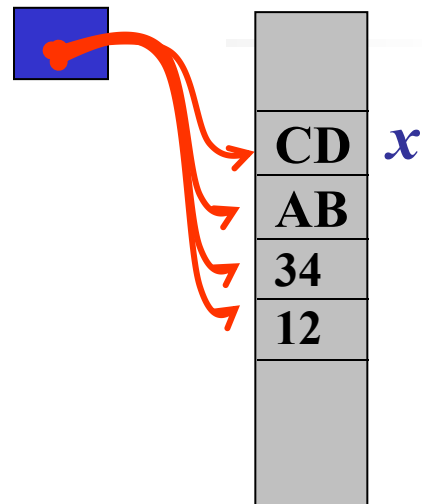
```
    }
```

```
    return 0;
```

```
}
```

char *p

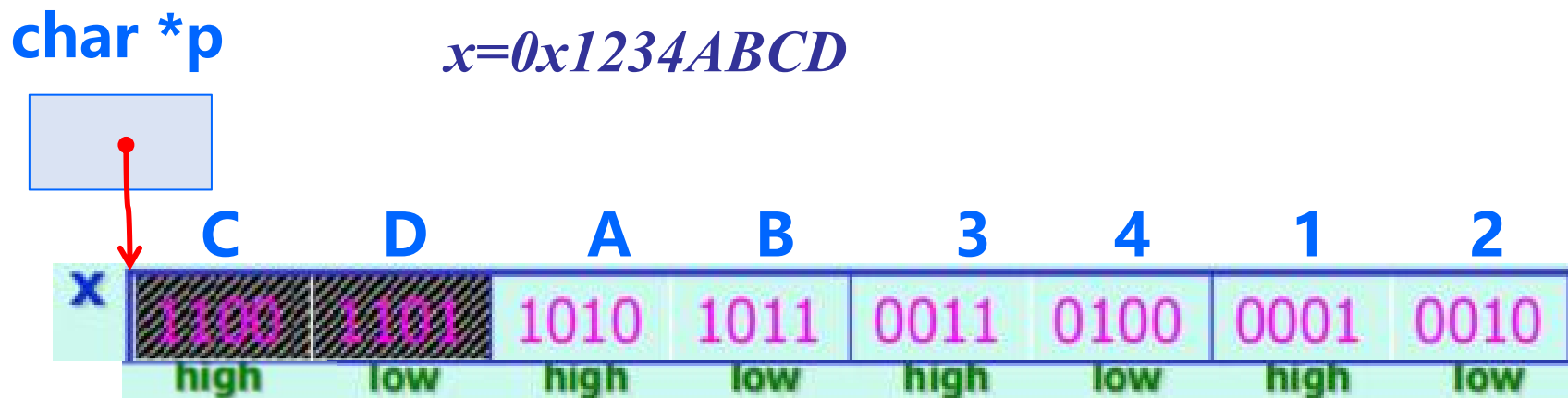
内存



0011 0000

指针的应用二

例：一个整型数占4个字节，其中每个字节又分成高4位和低4位。试从整型数的低字节开始，依次取出每个字节的高4位和低4位，并以十六进制字符的形式进行显示。



取低4位: $(*p) \& 0x0f$

取高4位: $(*p >> 4) \& 0x0f$

取高4位: $(*p) \& 0xf0$??

指针的应用二

```
#include<stdio.h>
```

```
int main(void){
```

```
    long x=0x1234ABCD,k;
```

```
    char * p =(char *)&x; //类型强制转换,p为字符指针
```

```
    char up_half,low_half; //up_half存高4位,low_half存低4位
```

```
    for(k=0;k<4;k++) {
```

```
        low_half=( *p )& 0x0f ; /* 取低4位 */
```

```
        if(low_half<10)
```

```
            low_half += '0'; // 低4位转换成字符'0'-'9'
```

```
        else
```

```
            low_half=(low_half-10)+'A'; // 低4位转换成字符'A'-'F'
```

```
        up_half=( *p >> 4 )& 0x0f; /* 取高4位 */
```

```
        if(up_half<10)
```

```
            up_half |= '0'; // 高4位转换成字符'0'-'9'
```

```
        else
```

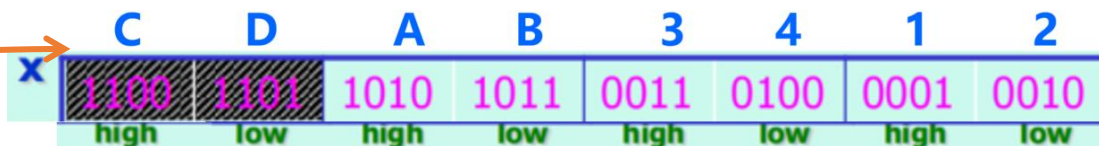
```
            up_half=(up_half-10)+'A'; // 高4位转换成字符'A'-'F'
```

```
        p++; /* 指向整型数x 下一个字节*/
```

```
        printf("%c %c ",up_half,low_half);
```

```
    }  
    return 0;
```

```
    low_half += '0'; 等价于 up_half |= '0';
```



0011 0000

CDAB3412_



指针练习一

1、定义宏提取一个**16**位数据的高八位和低八位

```
#define BYTE0(hdata) _____
```

```
#define BYTE1(hdata) _____
```

2、将**int**数的最高字节和最低字节交换。

3、将**32**位整数**IP**转为点分十进制输出（实验2中第6道编程题）

IPv4 地址	192.168.1.2
IPv4 子网掩码	255.255.255.0

4、**32**位本机字节序与网络字节序之间的转换，以 **host to network**为例，**hton(0x1234)**值为**0x34120000**

```
unsigned long  hton(unsigned long  h);
```

8.2 指针参数

传值调用：形参的修改无法改变实参变量的值。

```
#include <stdio.h>
```

```
void swap(int x, int y)
```

```
{
```

```
    int t;
```

```
    t=x; x=y; y=t;
```

```
}
```

```
int main(void)
```

```
{ int a=3, b=5;
```

```
    swap(a, b);
```

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

```
    return 0;
```

```
}
```

传址调用：以指针作为函数的参数实现实参值的改变。

指针作函数参数

- 改变主调函数中变量的值
- 使函数送回多个值

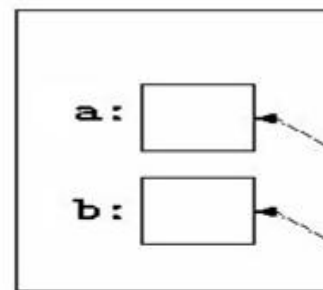
传址调用：以指针作为函数的参数实现变量值的改变。

```
#include <stdio.h>

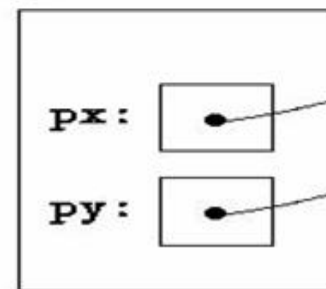
void swap(int *px, int *py)
{
    int t;
    t=*px; *px=*py; *py=t;
}

int main(void)
{
    int a=10, b=20;
    swap(&a, &b);
    printf("a=%d, b=%d\n", a, b);
    return 0;
}
```

in caller:



in swap:



引用作函数参数

传址调用：以引用作为函数的参数实现数据的双向传递

```
#include <stdio.h>
```

```
void swap(int &px, int &py)
```

```
{ int t;
```

```
    t=px; px=py; py=t;
```

```
}
```

```
int main(void)
```

```
{
```

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

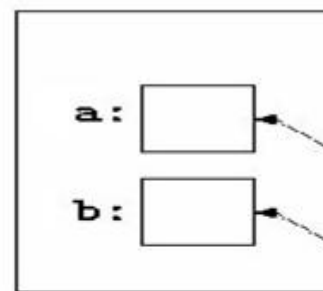
```
    swap(a, b); // int &px=a; int &py=b;
```

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

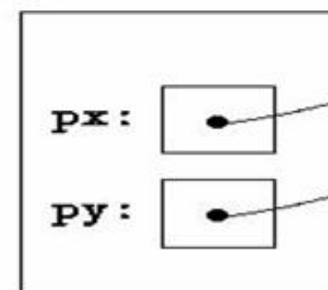
```
    return 0;
```

```
}
```

in caller:



in swap:





指针作输出参数

指针可作输出参数将值传回调用函数，使函数间接返回值，
解决函数返回多值的问题。

```
// implicit returned values:
```

```
void sum(int x,int y,int *result)
```

```
{
```

```
    *result=x+y;
```

```
}
```

```
// the caller
```

```
int s;
```

```
sum(3,4,&s);
```

指针参数：地址传递

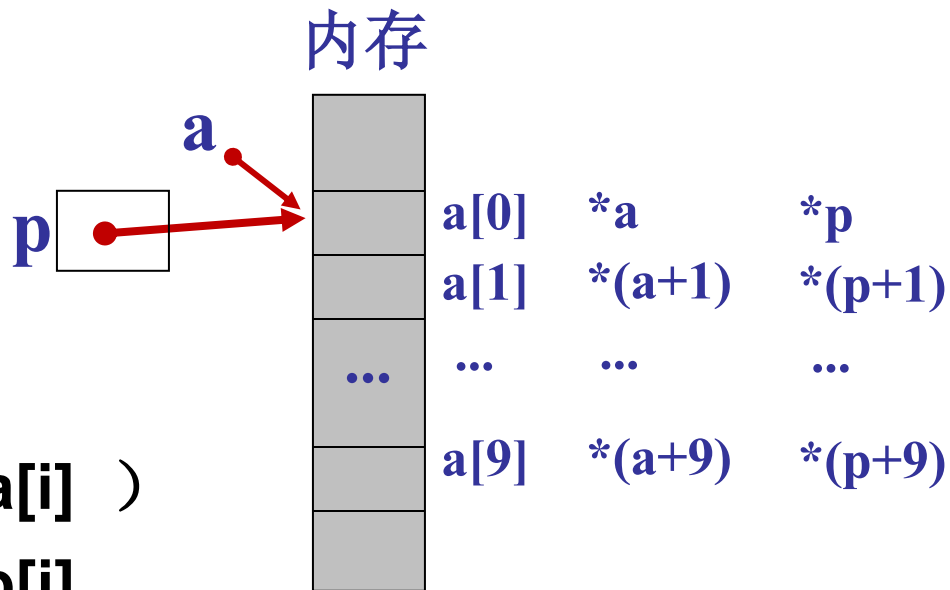
实现数据的双向传递，

将值传回主调函数

8.3 指针和一维数组

数组元素既可以用下标表示，也可以用指针表示。

```
int a[10], *p=a;
```



数组元素的表示

下标法

$a[i]$ (地址为 $\&a[i]$)

$p[i]$ (地址为 $\&p[i]$)

指针法

数组名 $*(a+i)$ (地址为 $a+i$)

指针变量 $*(p+i)$ (地址为 $p+i$)

指针表示效率高,
但不如下标表示
直观易读

数组元素的输入

已知: #define N 10

```
int a[N], *p, i;
```

```
p=a;
```

输入数组的全部元素

```
(1) for(i=0;i< N;i++)
```

```
scanf("%d", _____);
```

&a[i]

a+i

p+i

p++

a++ ✗

```
(2) for( ;p<a+ N;p++)
```

```
scanf("%d", _____);
```

p



数组元素的输出

已知: `int a[10], *p, i;`

`p=a;`

输出数组的全部元素

`for(i=0;i<10;i++)`

`printf(“%d”, _____) ;`

`a[i]`
`*(a+i)`
`*(p+i)`
`*p++`

`for(; p<a+10;p++)`

`printf(“%d”, *p _____) ;`

数组元素的输入和输出

输出数组*a*的全部元素(正确否 ?)

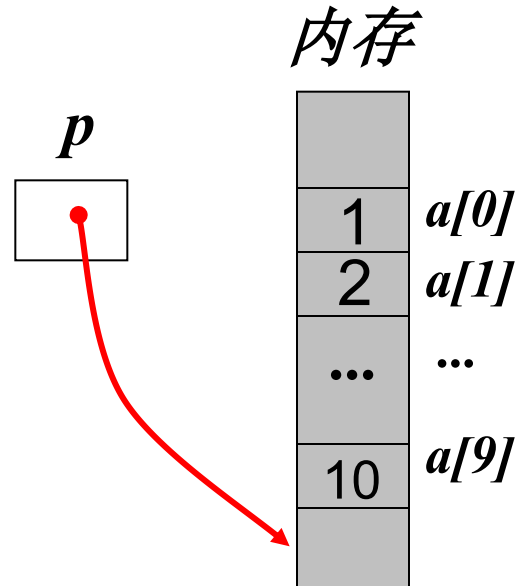
```
int a[10],*p , i;
```

```
for( p=a, i=0; i<10; i++)
```

```
    scanf("%d", p++) ;
```

```
while(p<a+10)
```

```
    printf("%d", *p++ ) ;
```





指针的关系运算

<, <=, >, >=, ==, !=

只限于同类型指针，

不同类型指针之间的关系运算被视为非法操作。

// 逆序输出数组a的全部元素

```
int a[10],*p=a ;
```

```
while(p<a+10)  scanf("%d", p++);
```

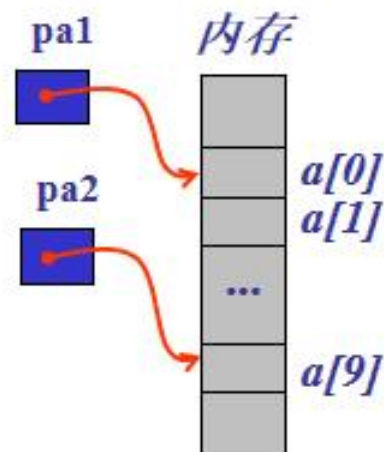
```
while( p>a)  printf("%d", *--p );
```

两个指针相减

两个指针可以相减

$pa2 - pa1$ /*等于所指元素的下标相减*/

指向同一数组的元素



```
int strlen(char s[ ])
```

```
{
```

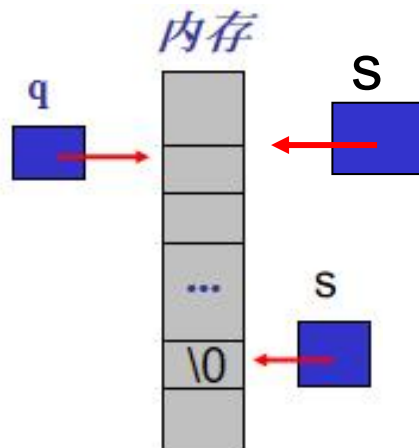
```
    char *q=s;
```

```
    while(*s) s++;
```

```
    return (s-q);
```

```
}
```

// 等价于 `int strlen(char *s)`





常量指针和指针常量

常量指针： 指针所指向的存储空间值是一个常量。

```
int a = 5, b = 6;
```

```
const int *p ; // 定义p为一个常量指针, 定义同时可不初始化
```

```
p = &a;        // 合法, 将p定向到a
```

```
*p= 10;        // 非法, *p为常量, 不能通过p修改所指变量的值
```

```
a = 8;         // 合法, 因为a不是常量
```

```
p = &b;         // 合法, p为变量, 可以将p重新定向到b
```

指针所指向的空间内容是常量, 不可修改



常量指针和指针常量

指针常量：数据类型是指针的常量。

```
int a = 5, b = 6;
```

```
int *const p = &a; // 定义p为指针常量，定义时必须初始化
```

```
*p = 10; // 合法，*p可修改
```

```
p = &b; // 非法，p不可修改，即不能重新定向到其它变量
```

```
int x[10]; /* x是指针常量 */
```

指针本身是常量，指针不可重新定向，但其所指向的空间内容可修改



指向常量的指针常量

```
int a = 5, b = 6;
```

```
const int * const p = &a ; /* p是指向常量的指针常量，  
                           必须初始化 */
```

```
*p= 10;    // 非法，不能通过p修改所指变量a的值，*p不可修改
```

```
p = &b;    // 非法，指针本身的值也不能修改，p不可修改
```

指针本身、指针所指向的空间内容，都是常量，都不可修改

一维数组参数的指针表示

```
void sort ( int a[ ], int n)
{
    .....
}
```

↑ 等价于 `int *a`

形参 { 不指定长度的数组
指针

实参 { 数组名（指针常量）
指向数组元素的指针变量

```
#include<stdio.h>
```

```
#define N 10
```

```
void BubbleSort ( int *a, int n) // 形参为指针
```

```
{
```

形参能否改成const int *a ??

```
    int i, j, t;
```

```
    for (i=1; i<n; i++) /* 共进行n-1轮"冒泡" */
```

```
        for (j=0; j<n-i ; j++) /* 对两两相邻的元素进行比较 */
```

```
            if (a[j]>a[j+1] ) { t=a[j]; a[j]=a[j+1]; a[j+1]=t; }
```

```
    }
```

```
int main ( )
```

```
{
```

```
    int x[N], i, *p=a;
```

```
    printf (" please input %d numbers: \n ", N);
```

```
    for (i=0; i<N; i++) scanf(" %d ", &x[i]);
```

```
    BubbleSort (x , N); // 实参为数组名
```

```
    // BubbleSort (p , N );
```

```
    ....
```

```
}
```

```
if (*(a+j)>*(a+j+1) )  
{ t=*(a+j); *(a+j)=*(a+j+1); *(a+j+1)=t; }
```

指针表示效率高,
但不如下标表示
直观易读

```

#include<stdio.h>
#define N 10
void BubbleSort ( int *a, int n) // 形参为指针
{
    int i, j, t;
    for (i=1; i<n; i++) /* 共进行n-1轮"冒泡" */
        for (j=0; j<n-i ; j++) /* 对两两相邻的元素进行比较 */
            if ( *(a+j)>*(a+j+1) )
                { t=*(a+j); *(a+j)=*(a+j+1); *(a+j+1)=t; }
}
int main ( )
{ int x[N], i, *p=a;
  printf (" please input %d numbers: \n ", N);
  for (i=0; i<N; i++) scanf(" %d ", &x[i]);
  BubbleSort (p , N ) ; // 实参为指针变量
  ..... ;
}

```

指针参数应用

编写三个函数分别完成指定一维数组元素的数据输入、求一维数组的平均值、求一维数组的最大值和最小值。由主函数完成这些函数的调用。

分析：采用地址传递的方式，把一维数组的存储首地址作为实参数调用函数。在被调用的函数中，以指针变量作为形式参数接收数组的地址。该指针被赋予数组的地址之后，使用这个指针就可以对数组中的所有数据进行处理。

函数原型：

```
void input(float *,int);  
float average(float *,int);  
void maxmin(float *,int, float *,float *);
```

float []

指针参数：地址传递

使函数送回多个值

最大值和最小值需要返回，采用地址传递方式

指针参数应用

```
#include <stdio.h>
void input(float *, int);
float average(float *, int);
void maxmin(float *, int, float *, float *);
int main( )
{
    float data[10]; //一维数组定义
    float aver,max,min;
    float *p=data;
    input(data,10);
    aver=average(data,10);
    maxmin(p,10,&max,&min);
    printf("aver=%f\n",aver); //输出平均值
    printf("max=%f,min=%f\n",max,min);
    return;
}
float average(float *pdata, int n) //数组平均值函数
{
    int i; float avg;
    for(avg=0,i=0;i<n;i++) avg+= pdata[i];
    avg /=n; //求平均值
    return(avg); //将平均值返回给被调用函数
}
```

求最大值和最小值，
以指针p以及max
与min的地址作为
函数实参

```
void input(float *pdata,int n) //输入数据函数
{
    int i;
    printf("please input array data: ");
    for(i=0;i<n;i++) //逐个输入数组的数据
        scanf("%f",pdata+i);
    //scanf("%f",&pdata[i]);
}
void maxmin(float *pdata,int n, float *pmax,
float *pmin)
{
    int i;
    *pmax=*pmin=pdata[0];
    for(i=1;i<n;i++)
    {
        if(*pmax<pdata[i])//求最大值
            *pmax=pdata[i];
        if(*pmin>pdata[i])//求最小值
            *pmin=pdata[i];
    }
}
```

指针形式

数组形式

//形参可否改成 const float *pdata?



高精度计算--- 超大整数的加法

高精度运算：是指参与运算的数(加数，减数，因子...) 范围大大超出了标准数据类型（整型，实型）能表示的范围的运算。例如，求两个**100**位的数的和、计算**100!**，都要用到高精度算法。



高精度计算主要解决的问题

高精度计算主要解决以下三个问题：

一、大数据（加数、结果）存储

二、运算过程

三、大数据的输入和输出



存储

用数组存储，每个数组元素存储1位（可以优化！）
，有多少位就需要多少个数组元素。个、十、百位等依次存储在 $x[0]$, $x[1]$, ...

用数组表示数的优点：每一位都是数的形式，可以直接加减；运算时非常方便。

求两个不超过200位的非负整数的和

```
#define N 200    /* 最大位数 */
int main(void)
{
    int x[N+1], y[N+1], z[N+2], len, i;

    printf("输入被加数: ");
    getBigNum(x, N);    /* 输入被加数 x */
    printf("输入加数: ");
    getBigNum(y, N);    /* 输入加数 y */
    addBigNum(z, x, y); /* z=x+y */
    putBigNum(z);       /* 输出z */
    putchar('\n');
    return 0;
}
```



输入大数: `getBigNum`

- ✓ 用字符一位一位输入，先输入高位再输入低位
- ✓ 放于数组中，从**0**号元素开始放，即高位放于低地址
- ✓ 实际存储：低位在低地址（也可以低位在高地址）
- ✓ 反转数组元素

函数getBigNum

- **函数功能：** 输入一个大整数，放于数组中，数组的每个元素存放一个十进制数，低位在低地址，即个位在`x[1]`，十位在`x[2]`,...，**`x[0]`放总位数。**
- **函数参数：**
 - 存放大整数的数组`x`
 - 能输入的最多位数`lim`
- **函数返回值：**
`void`
- **函数原型**
`void getBigNum(int *x, int lim);`

函数getBigNum

```
/*输入一个最多 lim位的大整数，存于x指向的单元，
x[0]保存总位数，个位在x[1]、十位在x[2] ,...*/
void getBigNum( int *x,int lim)
{
    int i,t,c;
    int *p1,*p2;
    for(i=1;i<=lim;i++) *(x+i)=0;        /* 先清零 */
    for(i=1;i<=lim && isdigit(c=getchar());i++)/*最先输的高位在x[1],...*/
        *(x+i)=c-'0';
    *x=i-1;        /* x[0]保存整数的位数 */
    for(p1=x+1,p2=x+i-1;p1<p2;p1++,p2--) /* 反转，低位在x[1]... */
        t=*p1,*p1=*p2,*p2=t;
}
```

输出大数: putBigNum

- 从高位按运算结果的实际位数输出每一位（数组元素）

```
/* 输出x指向的整数 */  
void putBigNum(int *x)  
{  
    int *p;  
    int n=*x;    /* 整数的位数 */  
    for(p=x+n;p>x;p--) { /* 从高位开始输出 */  
        printf("%d",*p);  
    }  
}
```



运算过程：addBigNum

模拟列竖式计算两数相加（如**45+96**）。注意：

（1）**运算顺序**：从低位向高位运算；先低位后高位；

（2）**运算规则**：相同位的两个数相加再加上**进位**，成为该位的和；这个和去掉向高位的进位就成为该位的值；如上例：

5+6+0=11，向前进一位1，本位的值是1；可借助%、/运算完成这一步；

4+9+1=14，向前进一位1，本位的值是4

（3）**最后一位的进位**：如果完成两个数的相加后，进位位值不为0，则应添加一位；

（4）如两个加数位数不一样多，则按位数多的一个进行计算；



函数addBigNum

- 函数功能：两个大整数相加

- 函数参数：

输入 **x**--被加数

y--加数

输出 **z**--和数

- 函数返回值：

void

- ⑩ 函数原型

void addBigNum(int *z,int *x,int *y);

函数addBigNum

/* 将x指向的被加数与y指向的加数相加, 结果放到z指向的单元 */

```
void addBigNum(int *z,int *x,int *y)
```

```
{
```

```
    int i,carry,n;
```

```
    for(i=1;i<=N;i++) /* 和数清零 */
```

```
        *(z+i)=0;
```

```
    n = max(*x,*y); /* n为两个加数中较长的位数 */
```

```
    for(carry=0,i=1;i<=n;i++) {
```

```
        *(z+i) = *(x+i) + *(y+i) + carry; /* 带进位的加法运算 */
```

```
        carry = *(z+i) /10; /* 计算新的进位 */
```

```
        *(z+i) %= 10; /* 计算本位 */
```

```
    }
```

```
    if(carry) /* 最后一位的进位 */
```

```
    {
```

```
        *(z+i)=carry;
```

```
        i++;
```

```
    }
```

```
    *z = i-1; /* z[0]保存和数的实际位数 */
```

```
}
```




分治法

高精度计算是基于分治法。

分治法是一种很重要的算法。“分而治之”就是把一个复杂的问题分成两个或更多的相同或相似的子问题，再把子问题分成更小的子问题……直到最后子问题可以简单的直接求解，原问题的解即子问题的解的合并。这个技巧是很多高效算法的基础。

采用的分治策略：

将“一个大数分为多个1位整数（0~9）”。



该分治策略的缺点

- (1) **浪费空间**: 一个整型数组的每个元素
(-32768~32767) 只存放一位 (0~9) ;
- (2) **浪费时间**: 一次加法只处理一位;



优化

- (1) 分治策略：将 “一个大数分为多个4位整数（0~9999）”（考虑2B整数的最大范围是32767，5位的话可能导致出界）
- (2) 一个数组元素存放四位数；将标准数组改为紧缩数组。
- (3) 运算时：逢万进位；将 “十进制运算” 改为 “万进制运算”。
- (4) 输出时：最高位直接输出，其余各位，要判断是否足够4位，不足部分要补0；例如：1，23，2345这样三段的数，输出时，应该是100232345而不是1232345。



思考练习

- 编程计算 $n!$ (n 可达100)



8.4 指针和字符串

字符串是一维字符数组，要访问和操纵字符串，
可以用字符数组，也可用字符指针。欲用指针变量
访问字符串，需：

- **声明**字符指针变量；
- 通过初始化或者赋值，使字符指针变量**指向**字符串；
- 通过字符指针来**访问**字符串中对应的字符。

字符串的指针表示

(1) 声明一个字符指针变量和一个字符数组

```
char s[80]= "This is a string." ;
```

```
char *p=s; //将p指向数组s
```

```
puts(p);           // 输出: This is a string.
```

```
putchar(*p);      // 输出: T
```

```
s= "That is a book." ; // 错, 数组名s是指针常量
```

```
strcpy(s, "That is a book." ); ✓
```

```
puts(s);           // 输出: That is a book.
```

字符指针形参: 实参可传入指针, 也可传入数组名

字符串的指针表示

(2) 声明和初始化一个字符指针变量

```
char *p=" This is a string." ;
```

- 1) 分配内存给字符指针p
- 2) 分配内存（静态区）给字符串常量（只读）
- 3) 将字符串首地址赋值给字符指针

```
puts(p);           // 输出: This is a string.
```

```
putchar(p[1]); // 输出: h
```

```
p="This is a book."; //√, p是指针变量, 但不提倡指向常量
```

```
*p='t'; // 运行错 //改成 p="t"; 或 p='t'; 呢?
```

√

X

声明为:

```
const char *p=" This is a string." ;
```

用char s[]和 char *s 初始化字符串的差异

若将p指向常量，建议采用：

const char *p = "123456789";

```
int main()
```

```
{
```

```
    char s[ ] = "123456789";
```

```
    char *p = "123456789"; //p指向常量字符串后，*p只能读
```

```
    puts(p);
```

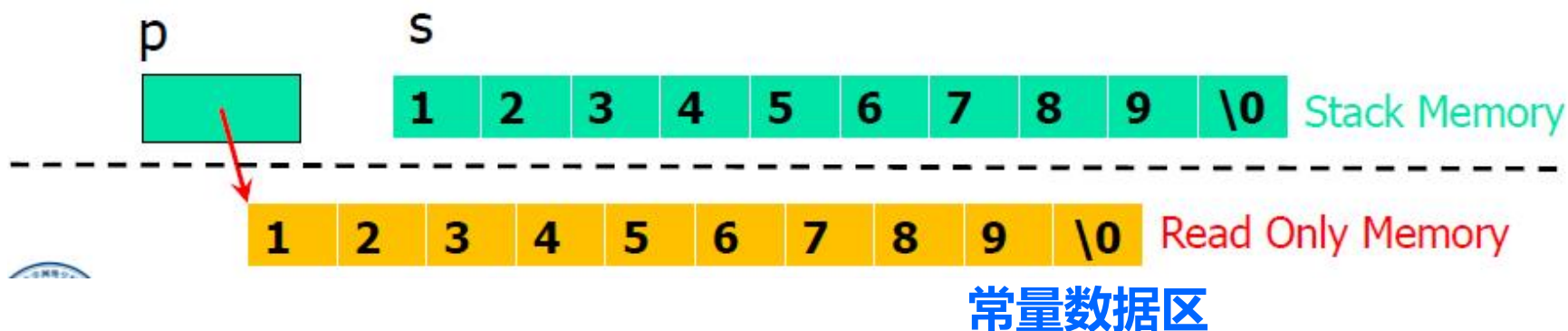
```
    s[0] = '2'; //✓
```

```
    p[0] = '2'; //✗， p指向常量字符串，不能修改*p内容
```

```
    return 0;
```

```
}
```

Thread 1: EXC_BAD_ACCESS (code=2, address=0x100001fad)



字符串作函数参数

/* 字符串拷贝 */

```
void mystrcpy(char *t, const char *s)
{
    while((*t=*s) != '\0')
        t++, s++;
}
```

```
void mystrcpy(char *t, const char *s)
{
    while((*t++=*s++) != '\0');
}
```

```
void mystrcpy(char *t, const char *s)
{
    while( *t++=*s++ );
}
```

```
char t[30], s[ ]="world";
char *p;
```

```
mystrcpy(t, "hello"); //√
mystrcpy(t, s); //√
mystrcpy(p, s);
// X, 指针未定向
```

```
p=t;
mystrcpy(p, s); //√
```

程序员要确保目标字符串t的空间足够大

目标字符串空间不够，结果会怎样？

```
int main()
{
    char s[]="12345", t[4];
    printf("t=%lx, s=%lx\n", t, s);
    mystrcpy(t, s);
    puts(t);
    puts(s);
    putchar(t[4]);
    //puts(s+1);
    return 0;
}
```

E:\教学盘\未来学院\2022级\例程\指针.exe

```
t=62fe00, s=62fe10
12345
12345
5
```

目标字符 t 空间小于源字符串 s 空间时，
越界操作，给系统带来安全隐患！

```
int main()
{
    char s[]="123456789ABCDEFGH", t[4];
    printf("t=%lx, s=%lx\n", t, s);
    mystrcpy(t, s);
    puts(t);
    puts(s);
    puts(s+1);
    return 0;
}
```

采用 `strncpy(t, s, 4);`;
限制拷贝的字符数

E:\教学盘\未来学院\2022级\例程\指针.exe

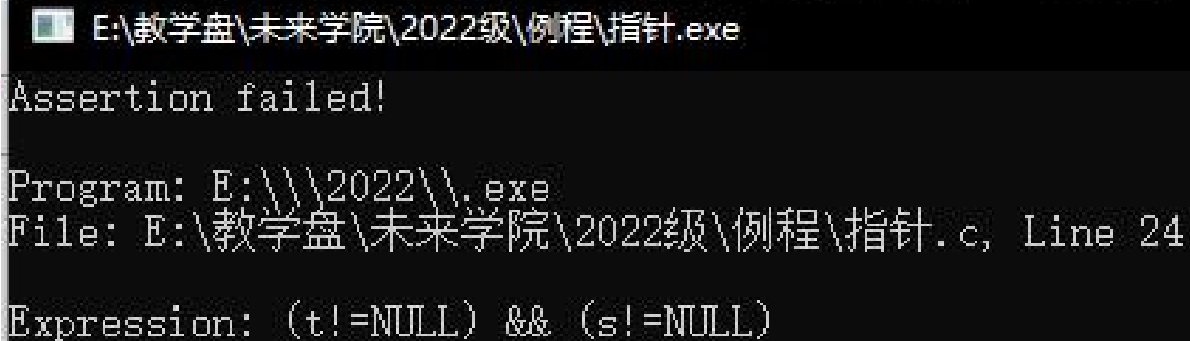
```
t=62fdf0, s=62fe00
123456789ABCDEFGH
23456789ABCDEFGH
```

如何避免传入空指针实参

```
#include <assert.h>

//加非0断言, 头文件<assert.h>
void mystrcpy(char *t, const char *s)
{
    assert((t!=NULL) && (s!=NULL));
    while((*t++=*s++) != '\0');
}

int main()
{
    char s[]="12345", t[4];
    char *p=NULL;
    mystrcpy(p, s);
    puts(t);
    puts(s);
    return 0;
}
```



E:\教学盘\未来学院\2022级\例程\指针.exe
Assertion failed!
Program: E:\\\\2022\\.exe
File: E:\教学盘\未来学院\2022级\例程\指针.c, Line 24
Expression: (t!=NULL) && (s!=NULL)



如何实现链式拷贝操作呢

```
#include <assert.h>
```

//为了实现链式操作, 将目标地址返回

```
char* mystrcpy(char *t, const char *s)
{
    assert((t!=NULL) && (s!=NULL));

    char *add=t;

    while((*t++=*s++) != '\0');

    return add;
}
```

8.5 指针数组

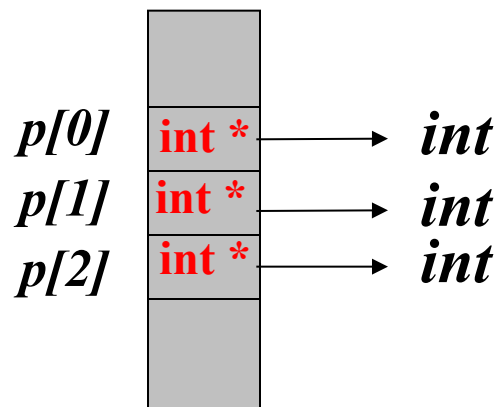
指针数组：指针的集合，每一个元素都是一个指针变量，并且每一个指针变量具有相同的存储类型和数据类型。

`int a[3];` **//整型数组**

`int *p[3];` /* **p**是一个有 3 个元素的指针数组，
每个元素是指向**int**变量的指针 */

`int *[3]`

p → **[3]** → * → **int**





8.5 指针数组

对指针数组中元素可以进行同类型指针允许的全部操作。

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

```
p[0]=a; /* 对指针数组中的元素p[0]赋值，使之指向a数组*/
```

```
p[1]=b; /* 对指针数组中的元素p[1]赋值，使之指向b数组*/
```

```
*p[0]=12; /* 使a[0]=12 */
```

```
++*p[1]; /* 访问b[0]并使其自增，结果为5，p[1]仍然指向b[0] */
```

```
*p[1]++; /* 访问b[0]，然后p[1]自增，使p[1]指向b[1] */
```

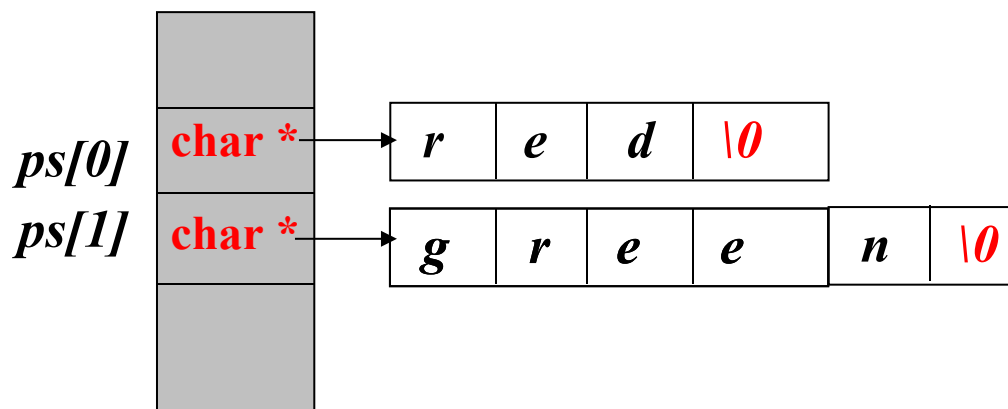
字符指针数组

```
char *ps[2]={ “red”, “green” };
```

/* ps是一个有 2个元素的字符指针数组(char *[2])

ps[0]指向字符串 “red”

ps[1]指字符串 “green” */



```
const char *ps[2]={ “red”, “green” };
```




指针数组的应用

指针数组可以用来处理多维数组

描述二维数组 (数值型二维数组, 字符串数组)

尤其是每行元素个数不相同的二维数组

(如三角矩阵,

由不同长度的字符串组成的数组)

利用指针数组描述杨辉三角形

```
#define N 5
```

```
#define SIZE N*(N+1)/2
```

```
int main( )
```

```
{
```

```
    int *p[N], a[SIZE];
```

```
    int sum=0, i;
```

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

```
        p[i]=a+sum; //指针元素初始化
```

```
        sum += i+1; //累积当前行元素个数
```

```
    } // p[i]=a+i*(i+1)/2;
```

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

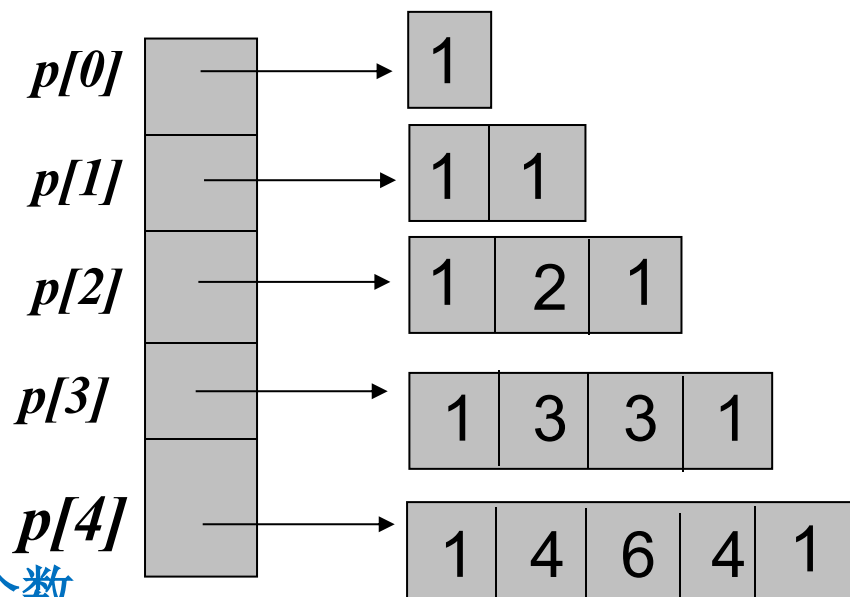
```
        *p[i]=1;
```

```
        *(p[i]+i) = 1;
```

```
    } //每行首尾元素赋值 1
```

```
}
```

int *p[N] 如何确定各指针指向的位置



```
    for(i=2;i<N;i++) //计算中间元素值
```

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

```
            p[i][k]=p[i-1][k-1]+p[i-1][k];
```

```
    return 0;
```

利用指针数组描述杨辉三角形

用二维数组和指针数组的存储空间情况：

对于 `int *p[5], a[15];`

$$5*4 + 15*4 = 80\text{B}$$

若采用数组 `int x[5][5];`

$$25*4 = 100\text{B}$$

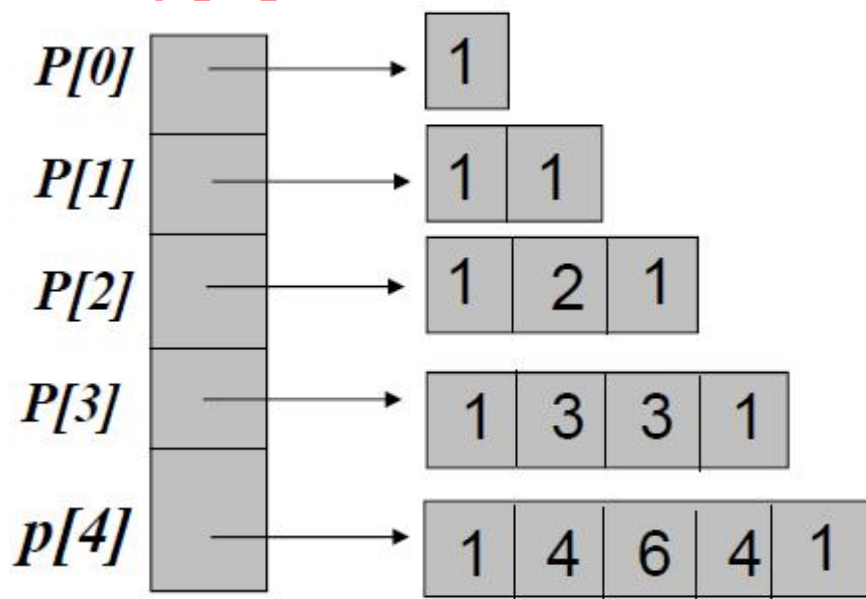
对于 `int *p[6], a[21];`

$$6*4 + 21*4 = 108\text{B}$$

若采用数组 `int x[6][6];`

$$36*4 = 144\text{B}$$

`int *p[N]`



N越大，浪费空间越多

用指针方式存取数组元素比用下标速度快，并节省存储空间。

利用字符指针数组实现菜单函数

```
int selectMenu(void ) //函数返回值是整型，代表所选的菜单项
{
    static char *menu[]={
        "1:Enter record",
        "2:Search record on name",
        "3>Delete a record",
        "4:Add a record",
        "0:Quit",
        NULL
    };
    int i,ch;
    system("cls");    /* 清屏 */
    do {
        for(i=0;menu[i]!=NULL;i++)
            puts(menu[i]);
        printf("\n Enter your choice:");
        scanf("%d",&ch);    /* 输入选择项 */
    } while( ch<0 || ch>i ); /* 选择项不在1-5之间重输 */
    return ch; /* 返回选择项，调用函数根据该数调用相应的函数 */
}
```

菜单程序框架

```
#include<stdio.h>
#include<stdlib.h>
int selectMenu(void);
int main(void)
{
    int choice;
    while((choice=selectMenu())){
        switch(choice) {
            case 1: // 功能1
                break;
            case 2: // 功能2
                break;
            case 3: // 功能3
                break;
            case 4: // 功能4
                break;
        }
    }
    return 0;
}
```

利用字符指针数组识别关键字

```
/* 若s是关键字, 函数返回1; 否则, 返回0 */
int iskey(char *s)
{
    static char *keyword[]={ /* 关键字表 */
        "auto", "_Bool", "break", "case", "char", "_Complex",
        "const", "continue", "default", "restrict", "do", "double",
        "else", "enum", "extern", "float", "for", "goto",
        "if", "_Imaginary", "inline", "int", "long", "register",
        "return", "short", "signed", "sizeof", "static", "struct",
        "switch", "typedef", "union", "unsigned", "void", "volatile",
        "while", NULL};
    int i;
    for(i=0; keyword[i]!=NULL; i++) /* 将标识符s 依次与每个关键字比较 */
        if(!strcmp(s, keyword[i]))
            return 1;
    return 0; /* 否返回0 */
}
```

用指针数组实现字符串排序

// 字符串排序函数 ---- *strsort*

```
void strsort ( char *s[ ], int n )  
{
```

```
    int i, j ;
```

```
    char *temp;
```

```
    for(i=0; i<n-1; i++) //选择法排序
```

```
        for(j=i+1; j<n; j++)
```

```
            if ( strcmp(s[i],s[j] ) >0 )  
            {
```

```
                temp=s[i];
```

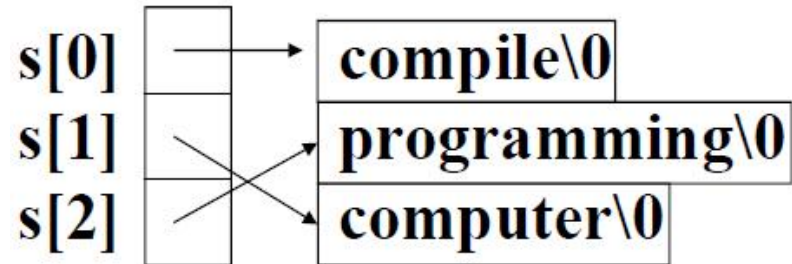
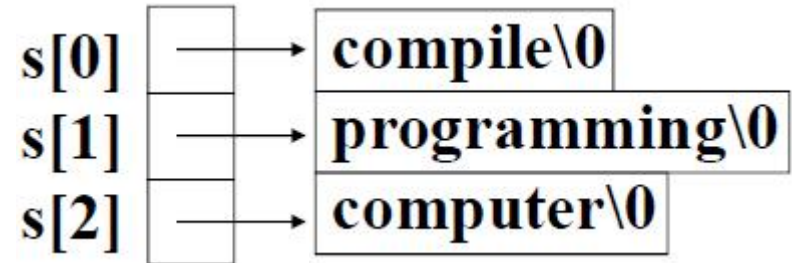
```
                s[i]=s[j]; /* 交换指针变量 */
```

```
                s[j]=temp;
```

```
            }
```

```
    }
```

char *s[3]




字符串排序：输入N本书名，排序后输出

```
#define N 3
#include<stdio.h>
int main( )
{
    int i;
    char *s[N];
    for(i=0;i<N;i++)
        fgets(s[i],80,stdin);

    strsort(s,N);
    for(i=0;i<N; i++) puts(s[i]);
    return 0;
}
```

读入的N个字符串存哪里？



如何解决



错误，使用了悬挂指针

动态分配存储字符串的空间

```
#define N 3
#include<stdio.h>
#include<stdlib.h>
int main( ){
    int i;
    char *s[N], t[80];
    for(i=0;i<N;i++) {
        fgets(t,80,stdin);
        s[i] = (char *)malloc(strlen(t)+1);
        strcpy(s[i],t);
    }
    strsort(s,N);
    for(i=0;i<N;i++) puts(s[i]);
}
```


二级指针

```
char *s[4];
```

问: $s[0][0]$, $s[0]$, s 类型分别是什么?

$s[0][0] \langle\langle == \rangle\rangle *(s[0]+0)$ 类型 `char`

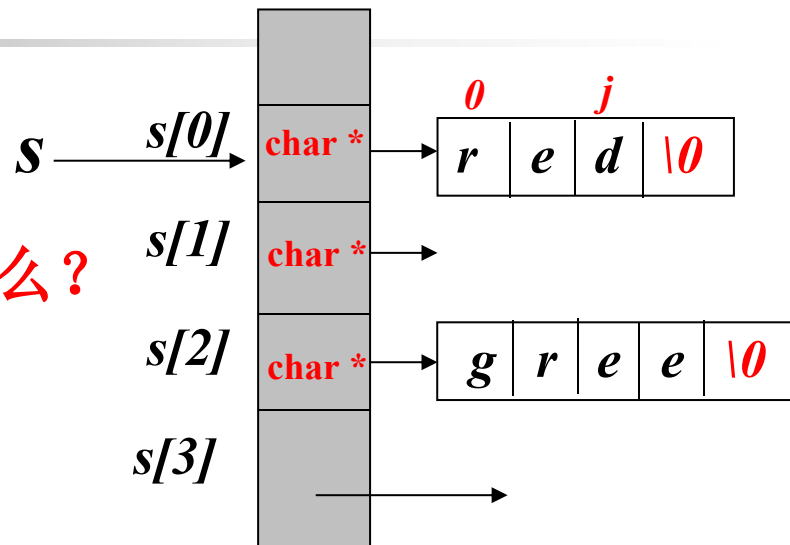
$s[0]$, 指针数组的元素, 类型 `char *`

s , 指针数组名, 指向 $s[0]$, 类型 `char **`

(指向字符指针的指针, 即字符型的二级指针)

$*s \langle\langle == \rangle\rangle s[0]$ $*(s+1) \langle\langle == \rangle\rangle s[1]$ $*(s+i) \langle\langle == \rangle\rangle s[i]$

要访问 $s[i][j]$, 指针表达方式: $*(s[i]+j) \langle\langle == \rangle\rangle *(*s+i)+j)$



二级指针的应用: 作函数的形参, 与指针数组实参进行虚实结合

字符串排序函数----*strsort*

```
void strsort ( char *s[ ], int n )
```

```
{
```

```
    char *temp;
```

```
    int i, j ;
```

```
    for(i=0; i<n-1; i++)
```

```
        for(j=0; j<n-i-1; j++)
```

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

```
                temp=s[j];
```

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

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

```
            }
```

```
    }
```

char **s

```
if ( strcmp(*(s+j),*(s+j+1)) >0 ) {
```

```
    temp=*(s+j);    /* 移动指针变量 */
```

```
    *(s+j)=*(s+j+1);
```

```
    *(s+j+1)=temp;
```

```
}
```

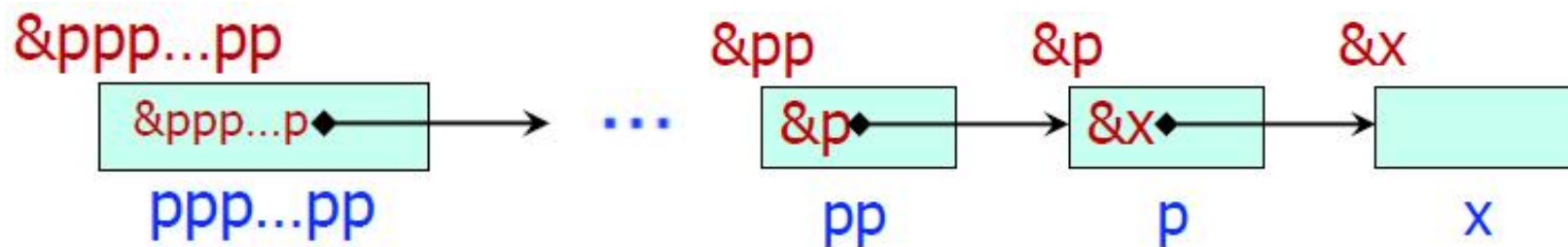
二级指针

如果p是指向T类型变量x的指针，则p的地址 或 存储p的地址的变量pp被称为T类型的**二级指针**。

n 重指针

pp为二级指针

p为指针变量



`int **...*pointer;`

错 `int *pp=&p;` `int *p=&x;` `int x;`

`int **pp=&p;`

- 二级指针的声明形式: `T **pointer;` // T类型的二级指针

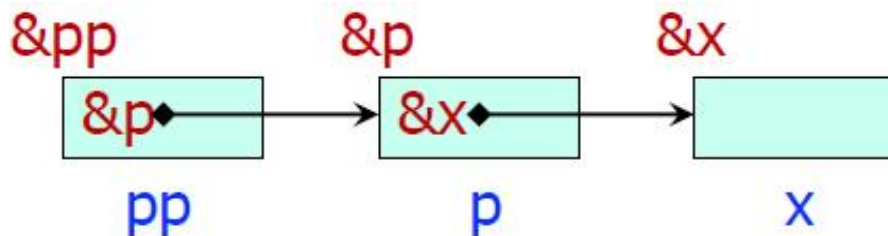
`int **pp; pp=&p;`

- n重指针，就是以n-1重指针变量的地址为其值的指针变量。

`*pp=?`
`**pp=?`

二级指针如何获取最终目标的值

```
int x=100;  
int *p=&x;  
int **pp=&p;
```



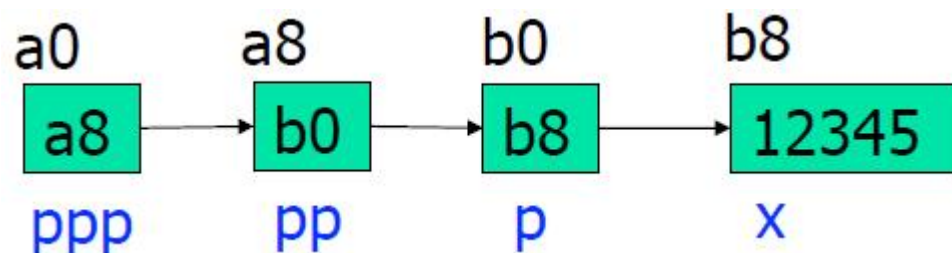
运算符*具有右结合性

$**pp \leftrightarrow *(*pp) = *(&x) = x$

*pp, 取pp指向的空间内容, 即&x

所以, $**pp = *p = x = 100;$

例：多重指针之间的关系



```
#include "stdio.h"
```

```
int main(void)
```

```
{
```

```
    int x=12345, *p=&x, **pp=&p, ***ppp=&pp;
```

```
    printf("&x=%lx\t x=%d\n",&x, x);
```

```
    printf("&p=%lx\t p=%lx\n",&p, p);
```

```
    printf("&pp=%lx\t pp=%lx\n",&pp, pp);
```

```
    printf("&ppp=%lx\t ppp=%lx\n",&ppp,ppp);
```

```
    return 0;
```

```
}
```

E:\教学盘\未来学院\2022级\例程\指针数组.exe

&x=62fe1c

x=12345

&p=62fe10

p=62fe1c

&pp=62fe08

pp=62fe10

&ppp=62fe00

ppp=62fe08

! 多重指针主要用作函数的形参，与指针数组实参进行虚实结合。



8.6 带参数的main函数

```
int main(int argc, char* argv[])
{
    printf("hello, HUST!\n")
    return 0;
}
```

```
int main()
{
    printf("hello, HUST!\n")
    return 0;
}
```

运行程序时操作系统将命令行参数传给main的形参

8.6 带参数的main函数

命令行：是一个术语，指的是通过输入文本命令来与计算机操作系统交互的一种方式。在命令行界面下，可以直接输入命令并按下回车键，计算机将执行相应的操作或返回结果。

```
e:\教学盘\未来学院\C例程>copy 8-01.cpp 8-02.cpp  
已复制          1 个文件。
```

在Windows命令行下执行copy程序，将文件8-01.cpp内容复制到文件8-02.cpp中。命令行有三个参数：copy, 8-01.cpp, 8-02.cpp

命令行参数：指在操作系统环境下**执行一个可执行程序时为其所提供的实际参数。**

```
e:\教学盘\未来学院\C例程>sum 10  
1+2+...+10=55
```

在Windows命令行下执行sum程序，求前n项的整数和。n由命令行参数给出

命令行提供了一种强大而灵活的方式来管理计算机。通过命令行，用户可以执行各种任务，例如创建、复制和删除文件/目录，运行程序，配置系统设置等等。



8.6 带参数的main函数

命令行参数：指在操作系统环境下**执行一个可执行程序时为其所提供的实际参数。**

命令行一般格式如下：**参数之间以空格隔开**

可执行文件名 参数₁ 参数₂ ... 参数_n

对于**C**语言，命令行参数就是**main**函数的参数。

8.6 带参数的main函数

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

```
{
```

c:\>程序名 参数₁ 参数₂ ... 参数_n

...

```
}
```

argc n+1



实参由命令行提供。

argc: 表示命令行中字符串的个数 (包括命令名, 也即**程序名**)

argv: 字符指针数组, **argv[i]**分别指向命令行中的字符串,

argv[0], 指向程序名字串, **argv[1]**指向第1个实参字串,

命令行方式传参给main函数

```
#include<stdio.h>
#include<stdlib.h>
```

```
int main(int argc, char *argv[])
{
    int n,sum,i;
    if(argc!=2) {
        printf("Command line error!\n");
        return -1;
    }
    n=atoi(argv[1]);

    for(sum=0,i=1;i<=n;i++)
        sum+=i;
    printf("1+2+...+%d=%d\n",n,sum);

    return 0;
}
```

C:\> **sum 11**

```
argc=2;
char *argv[]={
    "sum",
    "11",
};
```

长度为argc的字符指针数组
每个参数是一个字符串，
有argc个字符串

命令行中参数的个数（包括程序名）

命令行参数的传递

C:\> sum 11

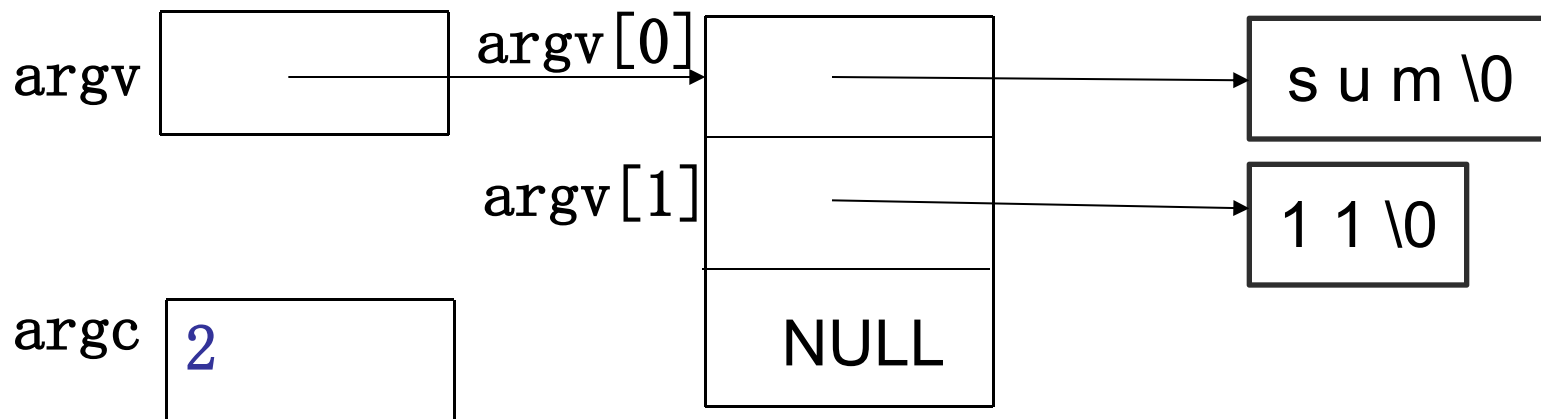
```
int main(int argc, char *argv[])
```

```
{
```

```
.....
```

```
}
```

char **argv





argv的类型实质为 char **

```
#include<stdio.h>
#include<stdlib.h>
int main(int argc, char **argv)
{
    int n,sum,i;
    if(argc!=2) {
        printf("Command line error!\n");
        return 1;
    }
    n=atoi(*++argv);
    .....
}
```

回显命令行参数示例

//show.cpp

```
#include <stdio.h>
```

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

```
{
```

```
    int n=0;
```

```
    while(n<argc){
```

```
        printf("%s", argv[n]);
```

```
        (n<argc-1) ? printf(" ") : printf("\n");
```

```
        n++;
```

```
    }
```

```
    printf("the number of command-line arguments are %d\n",argc);
```

```
    return 0;
```

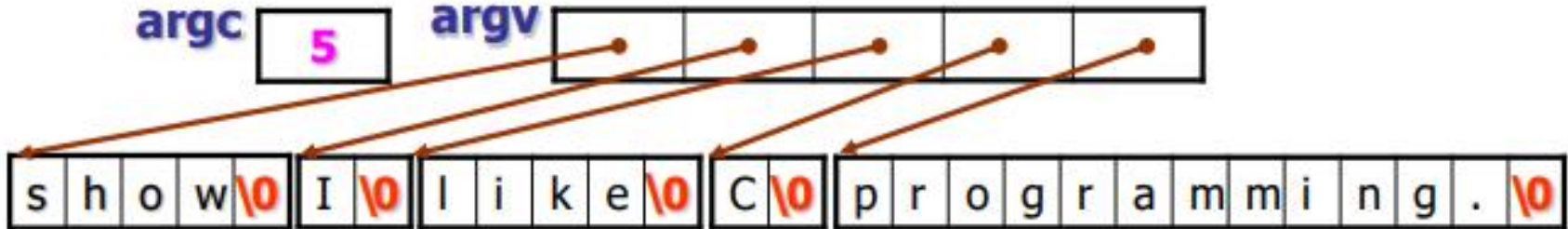
```
}
```

```
E:\教学盘\未来学院\C例程>show I like C programming.  
show I like C programming.  
the number of command-line arguments are 5
```

argc

5

argv





指定main的参数

在集成开发环境下调试程序时命令行所带参数如何输入？(只输入文件名后的参数，文件名不输入)

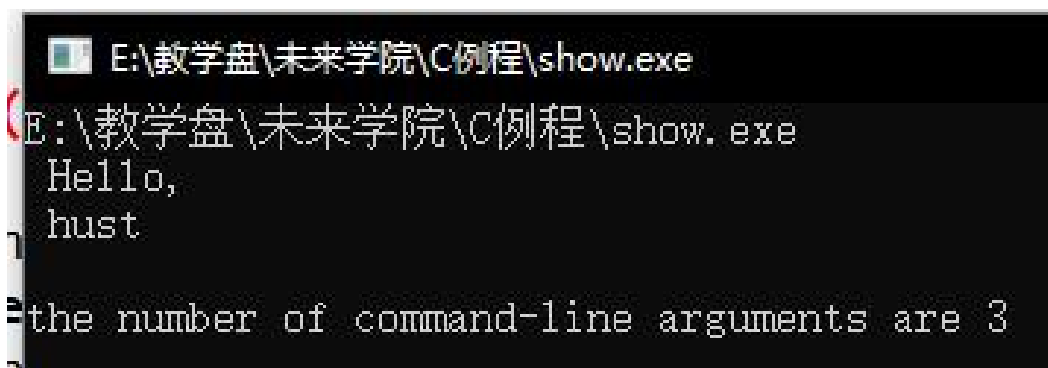
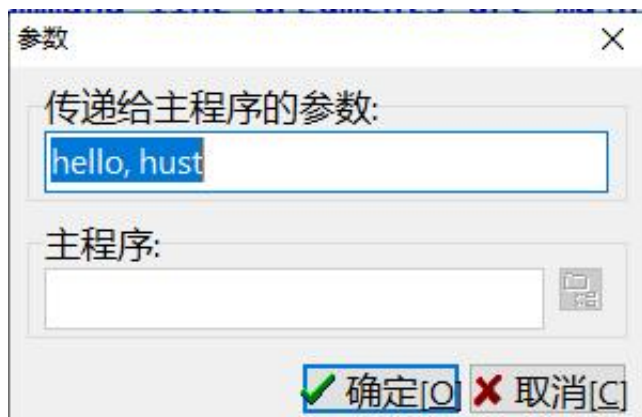
见 实验指导书

- (1) CodeBlocks: P7 1.1.7
- (2) Dev: P10 1.2.6
- (3) VS: P18 1.3.6

集成环境下如何指定main的参数

```
#include <stdio.h>
int main(int argc, char* argv[]) //打印命令行参数的程序
{
    int n=0;
    while(n<argc){
        printf("%s", argv[n]);
        (n<argc-1) ? printf(" ") : printf("\n");
        n++;
    }
    printf("the number of command-line arguments are %d\n",argc);
    return 0;
}
```

运行-->参数



集成环境下如何指定main的参数

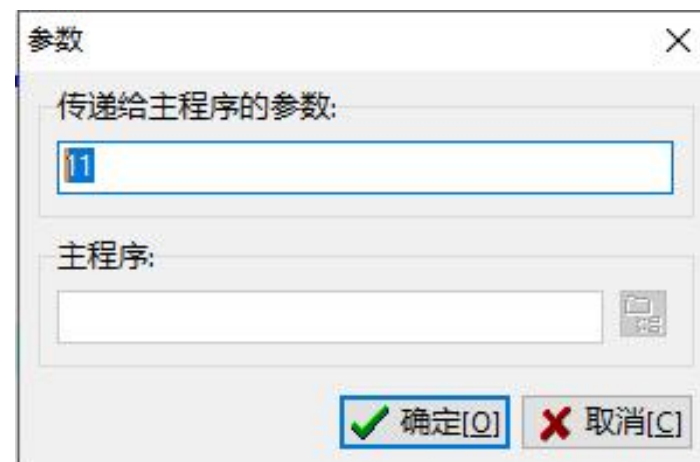
```
#include<stdio.h>
#include<stdlib.h>
```

```
int main(int argc, char *argv[])
{
    int n,sum,i;
    if(argc!=2) {
        printf("Command line error!\n");
        return -1;
    }
    n=atoi(argv[1]);

    for(sum=0,i=1;i<=n;i++)
        sum+=i;
    printf("1+2+...+%d=%d\n",n,sum);

    return 0;
}
```

运行-->参数



```
E:\教学盘\未来学院\C例程\sum.exe
1+2+...+11=66
-----
Process exited after 0.1197 seconds
请按任意键继续. . .
```


8.7 指针函数

如果函数的返回值是指针类型的值，该函数称为指针函数。

声明形式： **类型 *函数名 (形参表) ;**

// **类型**为任意数据类型

如： **char *strcpy(char *t, const char *s);**

函数strcpy是一个字符指针函数。即：该函数的返回值是字符指针。



返回的变量地址必在主调函数中可访问 (如, **全局或静态型**), 不能把被调用函数内的自动变量的地址和自动型数组的首地址作为指针函数的返回值

指针函数的返回值必须是可访问的

```
#include <stdio.h>
```

```
char *getString( )  
{  
    char p[]="Hello world";  
    return p;  
}
```

改为:

```
static char p[]="Hello world";
```

//永远不要从子函数中返回局部自动变量的地址

```
int main()  
{  
    char *str=NULL;  
    str = getString();  
    puts(str);  
  
    return 0;  
}
```

字符串拷贝的指针函数的定义

/* 将指针s所指向的字符串拷贝到指针t所指向的内存区域中，
返回 t 所指向的字符串的首地址 */

```
char *strcpy( char *t, const char *s )
```

```
{
```

```
    char *p=t ;
```

```
    while(*t++ = *s++);
```

```
    return(p) ; /* 返回第1个串的首地址 */
```

```
}
```

```
int main( )
```

```
{
```

```
    char st1[40]=" abcd" , st2[ ]=" hijklmn" ;
```

```
    printf( "%s" , strcpy( st1,st2));
```

```
    return 0;
```

```
}
```



字符串拷贝的指针函数定义

```
char *strcpy( char *t, const char *s )
{
    char *p=t ;
    while(*t++ = *s++);
    return(p) ; /* 返回第1个串的首地址 */
}

int main( )                                // 哪里有错
{
    char st1[ ]=" abcd" , st2[ ]=" hijklmn" ;
    printf( "%s" , strcpy( st1,st2));
    return 0;
}
```

查找子串的指针函数 【例8.18】

/* 在 s 中查找整个 t 第一次出现的起始位置。找到完全匹配子串，
返回一个指向该位置的指针；否则返回一个空指针 */

char *strstr (char *s, char *t)

{

char *ps=s, *pt, *pc;

查找最后一次匹配位置?

while(*ps!='\0'){

for(pt=t,pc=ps; *pt!='\0' && *pt==*pc; pt++, pc++);

if(*pt=='\0') return ps;

ps++;

}

return NULL; /* NULL : 0 */

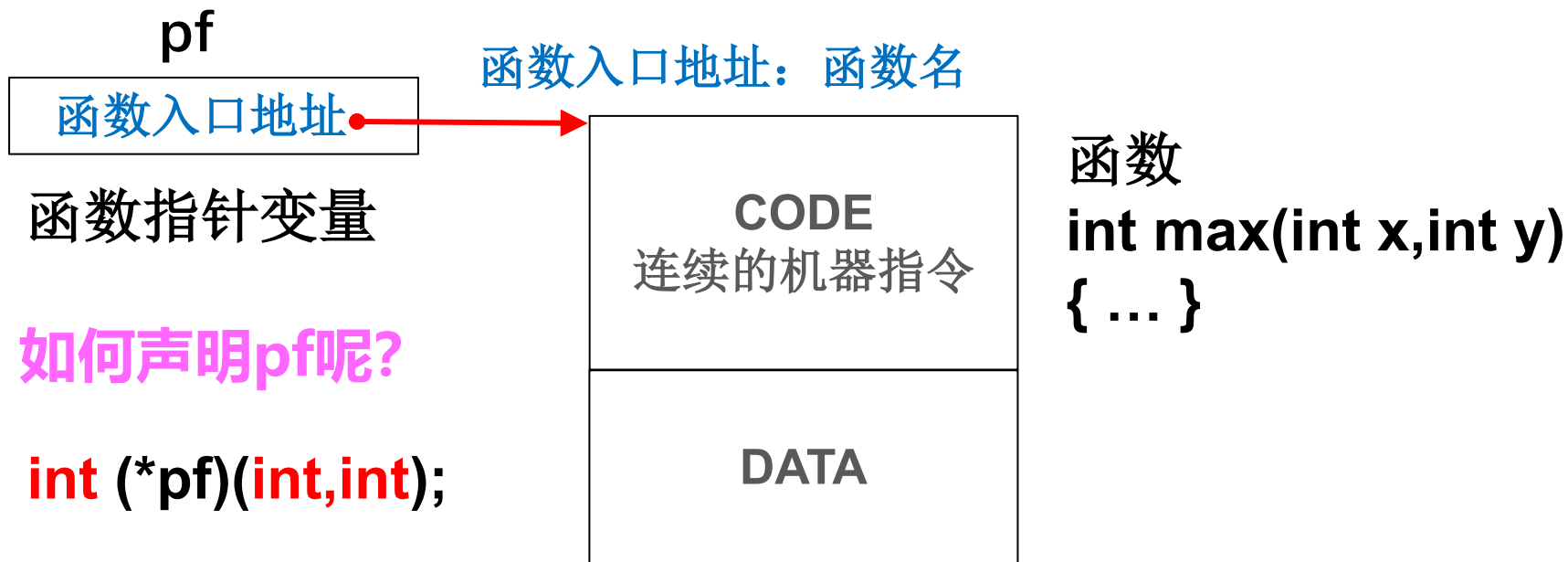
}

如果函数返回的指针指向一个数组的首元素，就间接解决了函数返回多值的问题。

8.8 指向函数的指针（函数指针）

每个函数都占用一段内存单元，有一个起始地址。

如果一个指针存放的是函数的入口地址，则该指针称为**函数指针**。



如同数组名，函数名也是地址常量，其值即为函数的入口地址。

函数

```
int max(int x,int y)
{ ... }
```


所指函数的形参的类型与个数

// pf 是指向有两个int参数的int型函数指针

或 `int (*pf) (int, int);`

pf=max; //先声明再初始化

如, `int (*pf) (int, int)=max;`

(*函数指针名)(实参表); 或
(函数指针名)(实参表);

所指函数的类型和参数，
必须与指针对象类型即
类型(形参表)一致！



函数指针的使用

```
void f1(int x)
```

```
{  
    printf("x=%d\n",x);  
}
```

```
void f2(int x,int y)
```

```
{  
    printf("x=%d\n",x);  
    printf("y=%d\n",y);  
}
```

通过函数指针来调用
它所指的函数：

```
int main(void)
```

```
{  
    void (*pf1)(int x); //声明函数指针  
    void (*pf2)(int x, int y);  
    pf1=f1; //初始化  
    pf2=f2;  
    pf1(5); //等价于(*pf1)(5);  
    pf2(10,20); //等价于(*pf2)(10,20);  
    return 0;  
}
```

(*函数指针名)(实参表); 或
(函数指针名)(实参表);

函数指针的应用--实现函数的回调

1. 作为函数的参数传递给其它函数，实现函数的回调

【例8.20】使用回调函数实现数据查找。30个学生成绩，分别用函数输出<60 和 >=90 的成绩，要求代码灵活，可复用性强。

■ 分析：

需求类似，代码重复多，尤其将来有其它类似需求时

✓ 如何用一个函数实现所有类似的需求？

void findNumbers()实现不同条件的查找功能

✓ 如何描述不同的条件呢？

✓ **使用函数指针作为参数**，通过函数指针调用不同的函数，完成数据的不同比较任务。

回调：一个函数通过由运行时决定的指针来调用另一个函数的行为。

回调函数：通过函数指针参数调用的函数。

函数指针的应用

```
#include <stdio.h>
#define N 30

//小于被比较数的回调函数
int less(int num,int compNum)
{
    return (num<compNum);
}

//>=被比较数的回调函数
int GreaterOrEqual(int num,int compNum)
{
    return (num>=compNum);
}
```

/*查找x指向的n个整数中满足指定条件的数;
p是函数指针,它指向的函数用于将数组中的
数与compNum比较*/

```
void findNumbers(int *x,int n,int compNum,int (*p)(int,int))
{
    for(int i=0;i<n;i++)
    {
        if (p(x[i],compNum)) printf("%d\n",x[i]);
    }
}
```

```
int main()
{
    int score[N];
    //输入N个学生成绩
    printf("低于60分:\n");
    findNumbers(score,N,60,less);

    printf("不低于90分:\n");
    findNumbers(score,N,90,GreaterOrEqual);

    return 0;
}
```

通过函数回调,实现使用同一接口
对不同类型数据、不同功能的处理

函数回调为使用者和开发者提供了灵活性,也增强了函数的通用性。 114



函数指针的应用

通用的整数排序函数

- 既能实现升序排序，也能实现降序排序

函数名称：**sort**

函数参数：

v--待排序数组的首地址

n--数组中待排序元素数量

comp--指向函数的指针，用于确定排序的规则

函数返回值：无

```
void sort ( int *v, int n, int (*comp)(int, int) );
```



通用的整数排序函数

```
/*对指针 v 指向的n个整数按comp规则排序 */
void sort ( int *v, int n, int (*comp)(int, int) )
{
    int i, j ;

    for(i=1; i<n; i++)        /*冒泡法*/
        for(j=0; j<n-i; j++)
            /*对v[j]和v[j+1]按照comp的规则进行比较*/
            if ( (*comp)(v[j],v[j+1]) )
                swap( v+j, v+j+1) ;        /* 交换 */
}
```

规则：按升序排序

//回调函数，通过函数指针参数调用的函数

```
int asc(int x, int y)
```

```
{
```

```
    if(x>y) return 1;
```

```
    else return 0;
```

```
}
```

```
return (x>y?1:0);
```

// caller

```
int a[6]={4,6,3,9,7,2};
```

```
sort(a,6,asc); // void sort ( int *v, int n, int (*comp)(int, int) );
```

// 思考：如果要降序，如何定义回调函数？

进阶：定义更通用的排序函数

- 能够对int、char、double、字符串、struct类型的数据排序。
- 既能实现升序排序，也能实现降序排序

- 函数参数

```
// void sort ( int *v, int n, int (*comp)(int, int) );
```

void *v, 待排序数组首地址

int n, 数组中待排序元素数量

int size, 各元素的占用空间大小（字节）

int (*fcmp)(const void *,const void *),

指向函数的指针，用于确定排序的规则

- **stdlib.h**中的标准库函数**qsort**----万能数组排序函数

void qsort(void *base, int nelem, int width,

int (*fcmp)(const void *,const void *));

- **思考**：如何调用**qsort**对字符串数组排序。

P147（快速排序）

函数指针的应用--求积分

编写函数实现积分

```
#include <stdio.h>
#include <math.h>
float sin2(float x);
float f1(float x);
float integrate( float a , float b, int n,
                 float (*f)(float) );
//积分函数定义: a,b为积分边界, n为积分区
//间分割数, f 指向被积分函数*/
float integrate( float a , float b, int n,
                 float (*f)(float) )
{
    float s, d;
    int i;
    d=(b-a)/n; //微元精度d
    s=(*f)(a) * d; //微元面积
    for( i=1; i<=n-1; i++)
        s=s+(*f)(a+i*d) * d; //微元面积累加
    return s;
}
```

//定义被积函数

```
float sin2(float x)
```

```
{ return sin(x)*sin(x); }
```

```
float f1(float x)
```

```
{ return x*x+x/2; }
```

```
int main( )
```

```
{
```

//定义指针指向函数sin2

```
float (*p)( float ) = sin2;
```

```
printf("%f", integrate(0,1,100,p));
```

```
printf("%f", integrate(-1,2,100,f1));
```

```
return 0;
```

```
}
```

运行结果:

0.269143

3.682950

函数指针的应用—多分支函数处理

2. 通过函数指针数组，建立转移表，实现多分支函数处理

- 借助函数指针数组，设计分支转移。每个数组元素依次存放不同函数的入口地址，通过下标操作实现多分支函数处理问题，从而省去了大量的if或switch语句。
- 如：设有10个函数 $f_0(x)$, $f_1(x)$, ..., $f_9(x)$ ，其原型为：
`double f0(int), f1(int), ..., f9(int); // 函数原型`
- 定义一个函数指针数组，其元素分别指向各函数：
`double (*pf[])(int) = {f0, f1, ..., f9}; // 函数指针数组`
- 要调用数组中的第i个函数，可写成如下形式：
`pf[i](x); // (*pf[i])(x);`
- 采用该方法编写的源程序，结构清晰、易读、易修改。更具有C语言的风格。

用函数指针数组控制菜单的驱动

```
void (*cmd[4])(void)={ f1, f2, f3, f4 };
```

/ cmd 是有4个元素的函数指针数组，指向的函数无返回值，无参数 */*

```
int main( void )
```

```
{ .....
```

```
do {
```

```
    printf( " menu1 \n " );
```

```
    printf( " menu2 \n " );
```

```
    printf( " menu3 \n " );
```

```
    printf( " menu4 \n " );
```

```
    printf( " exit(0) \n " );
```

```
    printf( " Enter your choice: \n " );
```

```
    scanf( " %d " ,&choice);
```

```
    if(choice>=1&&choice<=4)
```

```
        cmd [choice-1]( );
```

```
    } while(choice);
```

```
}
```

```
void f1( )
```

```
{ ... }
```

```
void f2( )
```

```
{ ... }
```

```
void f3( )
```

```
{ ... }
```

```
void f4( )
```

```
{ ... }
```

8.9 指向数组的指针（数组指针）

回忆一下指向函数的指针：

```
int (*pf) (int, int);
```

指针函数： `int *pf(int, int);`

// pf是一个函数指针，所指向的函数具有两个int参数，返回类型为int

类似地，如何定义指向数组的指针呢？

```
int (*pa) [3];
```

指针数组： `int *pa [3];`

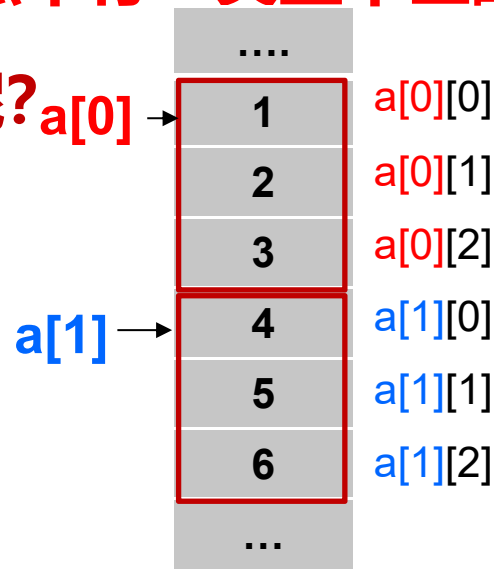
// pa 是一个指向数组的指针，所指向的数组包含3个int型元素

`int a[3]; pa=a; //?` 显然不行！ 类型不匹配

那么，数组指针是指向什么样的数组呢？

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

<code>a[0]</code>	<code>a[0][0]</code>	<code>a[0][1]</code>	<code>a[0][2]</code>
<code>a[1]</code>	<code>a[1][0]</code>	<code>a[1][1]</code>	<code>a[1][2]</code>



行地址与元素地址

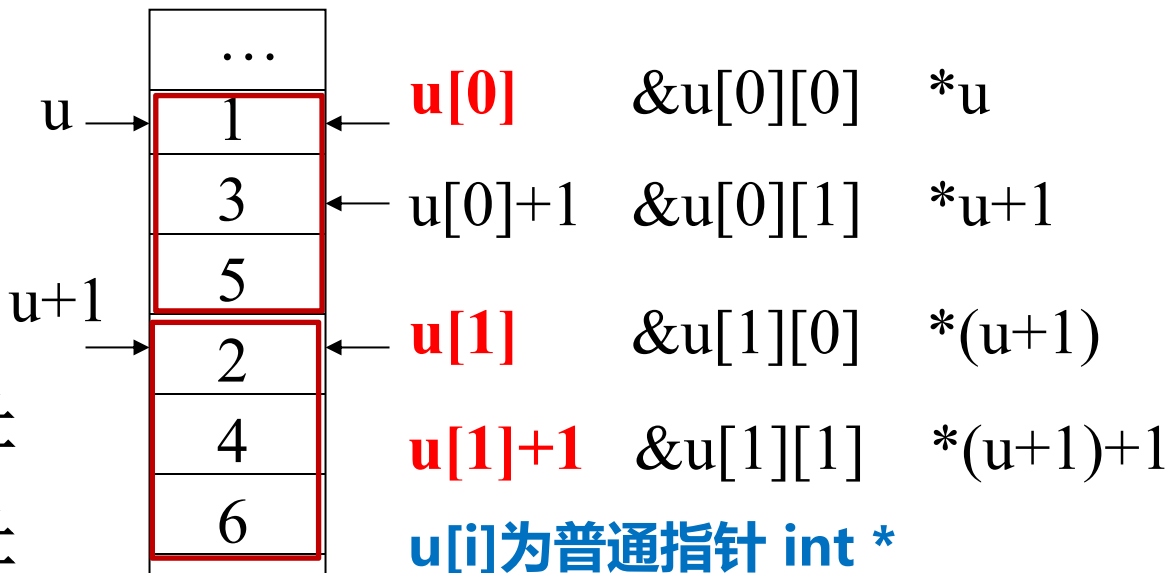
二维数组被看成以1维数组(行)为元素的数组。

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

u[0] 第0行首元素地址

u[1] 第1行首元素地址

u 第0行地址



u为指向数组的指针, 类型为 `int (*)[3]`

***u** 第0行首元素地址 即 ***u == u[0] == &u[0][0]**

***(u+1)** 第1行首元素地址 即 ***(u+1) == u[1] == &u[1][0]**

i行j列元素的地址: `&u[i][j]`、`u[i]+j`、`*(u+i)+j`

i行j列元素的表示: `u[i][j]`、`*(u[i]+j)`、`*(*(u+i)+j)`

指向数组的指针

指向数组的指针又称为**数组(的)指针**。

类型名 (*标识符) [数组元素个数];



所指数组元素的类型



指针变量名

注意与指针数组的区别
`int (*p) [];` 数组指针
`int *p [];` 指针数组

`int (*p) [3];` // `p`为数组指针, 类型为 `int (*)[3]`

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

`p=u;`

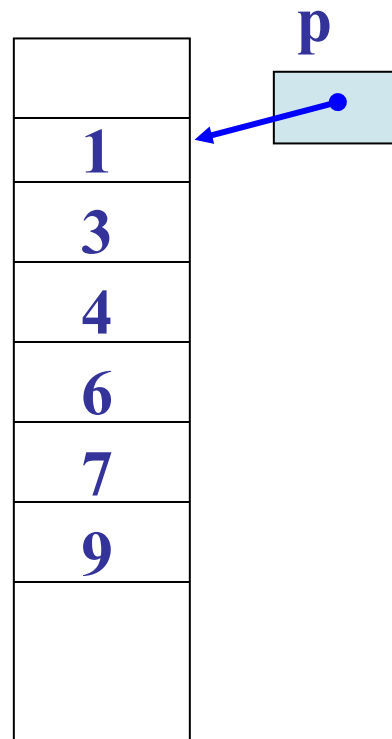
用指针表示二维数组元素

- (1) 用指向数组元素的指针
- (2) 用指向数组（即行）的指针

```
int a[3][2]={  
    {1, 3},  
    {4, 6},  
    {7, 9}  
};  
int *p;
```

将指针p指向数组的首元素

```
p=&a[0][0];    ✓  
p=a[0];        ✓  
p=a;          ✗  
p=(int *)a;    ✓  
p= *a;         ✓
```



(2) 用指向数组的指针表示二维数组元素

```
int a[3][2]={ {1, 3}, {4, 6}, {7, 9} };
```

```
int (*p)[2]; // p是指向数组的指针, 该数组有2个int 型元素
```

```
p=a; // p[i][j] : a[i][j]
```

教材表8-1、8-2 (P260)

<code>*(<i>*p</i>)</code> 或 <code>(<i>*p</i>)[0]</code>	<code>*(<i>*p</i>+1)</code> 或 <code>(<i>*p</i>)[1]</code>
<code>*(<i>*(p+1)</i>)</code> 或 <code>(<i>*(p+1)</i>)[0]</code>	<code>*(<i>*(p+1)</i>+1)</code> 或 <code>(<i>*(p+1)</i>)[1]</code>
<code>*(<i>*(p+2)</i>)</code> 或 <code>(<i>*(p+2)</i>)[0]</code>	<code>*(<i>*(p+2)</i>+1)</code> 或 <code>(<i>*(p+2)</i>)[1]</code>

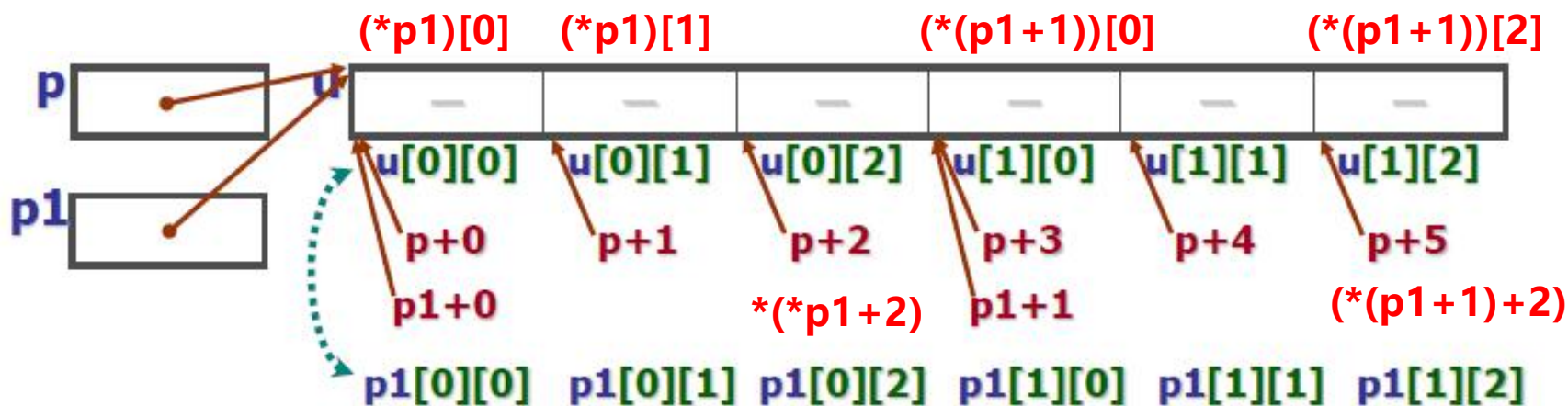
用指针表示二维数组元素

```
int u[2][3], *p=&u[0][0];
```

```
int (*p1)[3]=u; //p1为数组指针，指向一维数组的首地址
```

p 数据类型为 **int ***，其指向的存储单元的数据类型为 **int**。

p1 数据类型为 **int(*)[3]**，其指向的存储单元的数据类型为 **int[3]**。



使用数组指针时注意

(1) 不能把一个一维数组的首地址(即一维数组名)直接赋给一个数组指针
如:

```
int a[10], (*ap)[10];
```

```
ap = a; //错误
```

类型不一致, a的类型为int *,
ap的类型为int (*)[10]。

```
ap = (int (*)[10])a; //正确
```

强制类型转换, 使数组指针
指向了一维数组a

或者用初始化操作写成:

```
int a[10], (*ap)[10] = (int ( * )[10])a;
```

(2) 数组指针即使指向了一维数组a, 也不能用“ap[i]”或“*ap[i]”等形式访问一维数组a的第i号元素。用(*ap)[i]或*(*ap+i)访问a[i]元素

```
(*ap) = a
```

```
(*ap)+i = a+i
```

```
(*ap)[i] = *((*ap)+i) = a[i] = *(a+i)
```


二维数组元素的输入/输出

```
#include <stdio.h>
#define I 2
#define J 3
int main(void)
{   int u[I][J], (*p)[J]=u;
    int j;
    for(j=0;j<J;j++)    /* 用指向数组元素的指针完成第0行元素的输入 */
        scanf("%d",<b>u[0]+j</b>));
    for(<b>p++</b>,j=0;j<J;j++) /* 用指向数组的指针完成第1行元素的输入 */
        scanf("%d",<b>*p+j</b>));
    for(j=0;j<J;j++) /* 用指向数组的指针完成第0行元素的输出 */
        printf("%6d",<b>*(<b>*u+j</b></b>));
    printf("\n");
    for(j=0;j<J;j++)    /* 用指向数组元素的指针完成第1行元素的输出 */
        printf("%6d",<b>*(<b>u[1]+j</b></b>));
    return 0;
}
```

二维数组作函数参数

- 形参说明为数组
 - 形参说明为指针
- 指向数组元素的指针
指向下一级数组的指针

```
fun(int x[ ][4], int row)  
{ ... }
```

```
fun(int *x,int row,int col)  
{ ... }
```

```
fun(int (*x)[4], int row )  
{ ... }
```

```
fun(int row,int col,int (*x)[col])  
{ ... }
```

变长数组（C99）

二维数组作函数参数

1. 形参为指向数组元素的指针

形参说明为指向数组元素的指针时，实参应为数组元素的地址，或为指向数组元素的指针变量。

```
void fun(int *p,int n); /* 形参说明为指向数组元素的指针 */
void main(void){
    int u[2][3];
    ...
    fun(u[0],2*3); /* 实参u[0]是元素u[0][0]的地址 */
    ...
}
```

2. 形参为指向下一级数组的指针

二维数组作函数参数

1. 形参为指向数组元素的指针

形参说明为指向数组元素的指针时，实参应为数组元素的地址，或为指向数组元素的指针变量。

2. 形参为指向下一级数组的指针

形参说明为指向下一级数组的指针时，实参应为数组名，或指向下一级数组的指针。

```
void f1(int (*p1)[3],int n); /* 指向一维数组的指针p1为形参 */
void main(void){
    int u[2][3],v[2][3][4];
    ...
    f1(u,2*3);    /* 数组名u作为实参 */
    ...
}
```

三维数组的指针表示

类推:

三维数组被看成以二维数组（页）为元素的一维数组
一个n维数组被看成以n-1维数组为元素的一维数组。

```
int x[2][3][4];
```

```
int (*p)[3][4]=x; /*p是指向3行4列的二维整型数组的指针*/
```

```
*(*(p+i)+j)+k)
```

```
*(*(p[i]+j)+k)
```

```
*(p[i][j]+k)
```

```
p[i][j][k]
```

x[i][j][k]

8.10 用typedef定义类型

typedef是关键字，为一个已有类型定义一个别名。

typedef <已有类型名> <新类型名>;

(1) **typedef unsigned int size_t;**

size_t 被定义为 **unsigned int**类型

size_t x, y; //等价于 **unsigned int x,y;**

(2) **typedef char * string ;**

string 被定义为 **char ***

string p, s[10]; //等价于 **char *p, *s[10];**

意义更明确
可读性更强

(3) **typedef int (*p_to_fun)(int ,int);**

p_to_fun 被定义为 **int (*)(int ,int)**

p_to_fun pf; //等价于 **int (*pf)(int ,int);**

void fun(int *v, int n, p_to_fun cmp);

书写简洁方便
易于理解记忆

//形参**cmp**指向函数的指针

类型定义语句

typedef <已有类型名> <新类型名>;

使用typedef给已定义的结构类型赋予新的类型名，大大简化了对结构变量的说明

(4) **typedef struct Student Stu;**

Stu 被定义为struct Student结构类型

Stu stu1, *pstu; 等价于 **struct Student stu, *pstu;**

```
typedef struct{  
    double real;  
    double image;  
} COMPLEX;
```

COMPLEX被定义为struct{}的新类型名

COMPLEX c1, c2;

COMPLEX *pc1, *pc2;

注意：1、typedef可以定义类型，但不能定义变量

2、typedef只能对已存在的类型重新定义一个别名，而不能创建一个新的类型名。

用typedef声明一个新类型的方法

```
typedef char * string;
```

```
string ps, s[10];    等价于: char *ps, *s[10];
```

```
typedef char STRING[81];
```

```
STRING s, s2;    等价于: char s[81], s2[81];
```

1、先按定义变量的方法写出定义体 **char s[81];**

2、把变量名换成新类型名 **char **STRING**[81];**

3、在前面加typedef **typedef char **STRING**[81];**

4、再用新类型名定义变量 ****STRING** s;**

typedef与#define的区别

- #define语句, 在预处理阶段进行简单的串替换
typedef则是在编译阶段进行类型识别, 然后替换。

如, 定义一个STRING类型为具有81个字符的数组

```
typedef char STRING[81];
```

```
STRING text; //解释为: char text[81];
```

若, #define STRING char[81]

```
STRING text; //替换为: char[81] text; 显然错误
```

又如, #define string char*

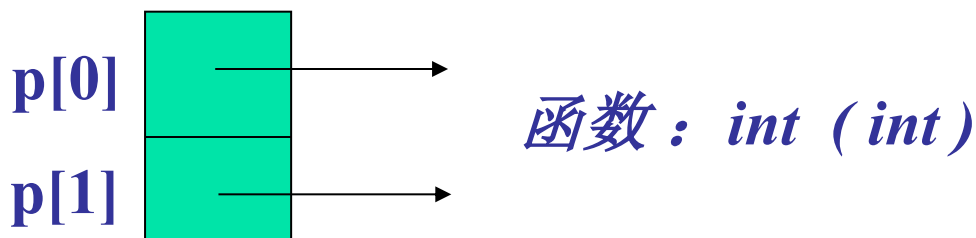
```
string ps, s[10]; //替换为: char *ps, s[10];  
显然, s解释错误
```

8.11 复杂声明

```
int (*p[2])(int);
```

p是含有2个指针元素的数组，每个指针指向有一个整型参数，返回值为整型的函数。

p是指向函数的指针数组。



函数指针数组的使用

```
int fun1(int x)
{ return 2*x; }
```

```
int fun2(int y)
{ return 3*y; }
```

```
int main(void)
```

```
{ int i,(*fp[2])(int); /* fp 是有2个元素的函数指针数组,
                        每个元素指向的函数有一个整型形参, 返回整型值*/
```

```
    fp[0]=fun1; /* 0号元素指向 fun1函数 */
```

```
    fp[1]=fun2; /* 1号元素指向 fun2 函数 */
```

```
    for(i=0;i<2;i++)
```

```
        printf("%d\n",fp[i]((i+1)*5)); /*依次调用 fp[0]、fp[1]所指函数 */
```

```
    return 0;
```

```
}
```

```
int (*fp[2])(int)={fun1,fun2};
```

指向函数的指针函数

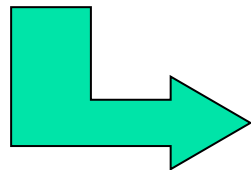
```
int (*f(char *, char *)) (int, int) ;
```

f是一个指针函数，

f函数有2个char * 类型的形参，其返回值是指向有2个int参数且返回值为int的函数的指针。

f()函数的形参： (char *, char *)

*f()是一个指针函数



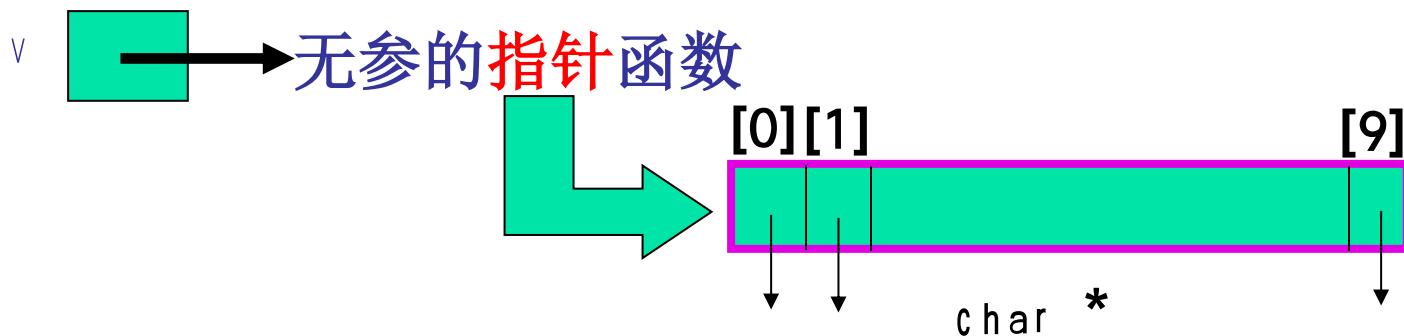
函数： int (int, int)

指针函数的指针

```
char * (*(*v)(void))[10];
```

- (1) $(*v)$, v 是指针变量
- (2) $(*v)(\text{void})$, v 是指向函数的指针, 所指函数无参数
- (3) $(*(*v)(\text{void}))$, v 所指函数的返回值是指针类型
- (4) $(*(*v)(\text{void}))[10]$, 返回的指针指向有10个元素的数组
- (5) **char** $*(*(*v)(\text{void}))[10]$, 数组元素的类型是**char** *

v 是指向函数的指针, 所指函数没有参数, 返回值是指向有10个元素的字符指针数组的指针。





复杂声明练习

一、解释下面的声明语句

- (1) `float (*p)[2];`
- (2) `char *p[5];`
- (3) `char(*fp)(char *, int *);`
- (4) `int *pf(float(*a)(int));`
- (5) `int(*pf(char*))[5];`

二、依据描述写相应的声明语句

- (1) 设p是3个元素的函数指针数组，函数指针数组中每个元素所指向的函数是无参字符指针函数，请写出相应的声明语句。
- (2) 设p是9个元素的指针数组，数组中每个元素是指向有一个整形参数，返回值为双精度浮点型数的函数的指针，请写出相应的声明语句。
- (3) 设p是函数指针，所指的函数无参，且返回一个指向有3个元素的字符数组的指针，请写出相应的声明语句。

知识点小结

一维数组的指针表示

若a为一维数组名，指针p=a或p=&a[0]，一维数组中元素及元素的地址可以归纳如下表所示。

表示方式	元 素	元 素 地 址
用数组名的下标表示	a[i]	&a[i]
用指针p的下标表示	p[i]	&p[i]
用数组名的指针表示	*(a+i)	(a+i)
用指针p的指针表示	*(p+i)	(p+i)
用数组名构成的地址表达式和下标表示	(a+i)[0]	(a+i)
用指针构成的地址表达式和下标表示	(p+i)[0]	(p+i)

二维数组元素的访问方式

对于二维整型数组u，其u[i][j]元的地址用指向数组元素的指针表示为：

$$u[i]+j$$

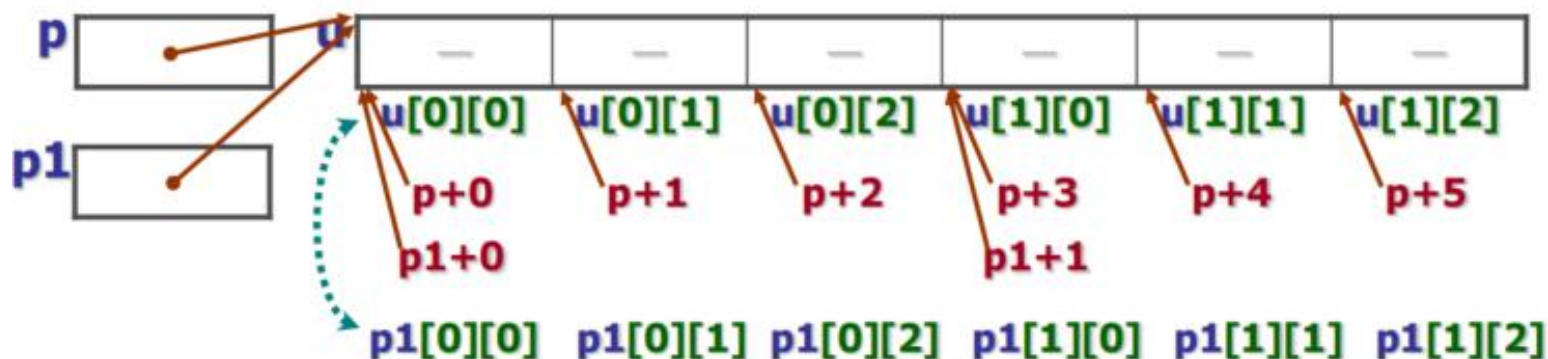
用指向一维数组（下一级数组）的指针表示则为：

$$*(u+i)+j$$

用间访形式访问其u[i][j]元可以表示为：

$$*(u[i]+j) \text{ 和 } *(*u+i)+j$$

```
int u[2][3], *p=&u[0][0], (*p1)[3]=u;
```



p数据类型为`int *`，其指向的存储单元之数据类型为`int`。

p1数据类型为`int(*)[3]`，其指向的存储单元之数据类型为`int[3]`。

指针移动操作

```
char s[6]={0,1,2,3,4,5} , *pc ;  int i = 2;
```

```
pc=s;    //对字符指针变量pc赋值，使其指向字符数组s(即s[0])
```

```
pc+i;    //结果为pc后面第i个元素的地址
```

```
*(pc+i); //结果为pc后面第i个元素
```

```
*++pc;   //结果为pc加1之后所指元素
```

```
++*pc;   //结果为pc所指元素加1
```

```
*pc++;   //结果为pc所指元素，然后pc加1，指向后面一个元素的地址
```

```
(*pc)++; //结果为pc所指元素，然后pc所指元素加1
```

```
&pc;     //结果为字符指针变量pc的地址，类型为char **
```

特别提示： 注意运算符的优先级和结合性！

数组与指针的不同点

```
char s[6]={0,1,2,3,4,5} , *pc=&s[1] ;  int i = 2;
```

s+i; //结果为数组s中第i个元素的地址(即s[i]的地址)

***(s+i);** //结果为数组s中第i个元素s[i]

++*s; //结果为元素s[0]加1

(*s)++; //结果为元素s[0]的值, 然后s[0]加1

s=pc; //非法! 数组名s是地址常量, 不能进行赋值操作

&s; //非法! 数组名s是地址常量, 不能取地址 **&s类型为char(*)[6]**

***++s;** //非法! 不能对数组名s进行前缀++操作

***s++;** //非法! 不能对数组名s进行后缀++操作

特别提示: 注意运算符的优先级和结合性!

知识点小结

几种等价的一维数组作为函数参数时的函数原型

```
int fun(int x[], int n);
```

```
int fun(int [], int );
```

```
int fun(int *x, int n);
```

```
int fun(int *, int );
```



当一维数组做函数参数时，编译器总是把它解析成一个指向其首元素的指针。