



*Xidian University*

# C语言程序设计

---

## Lec 7 指针与动态存储管理





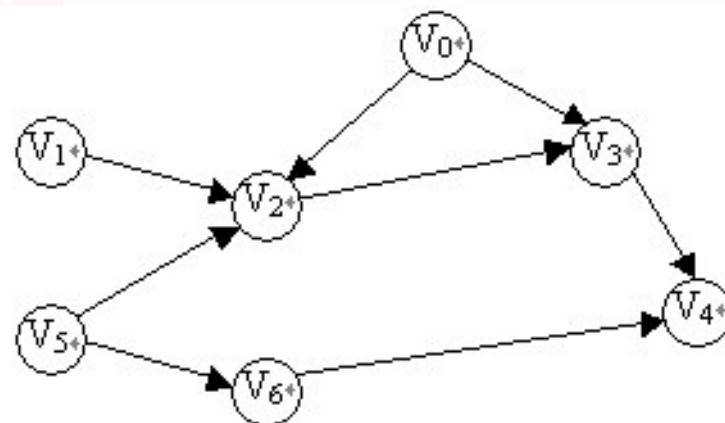
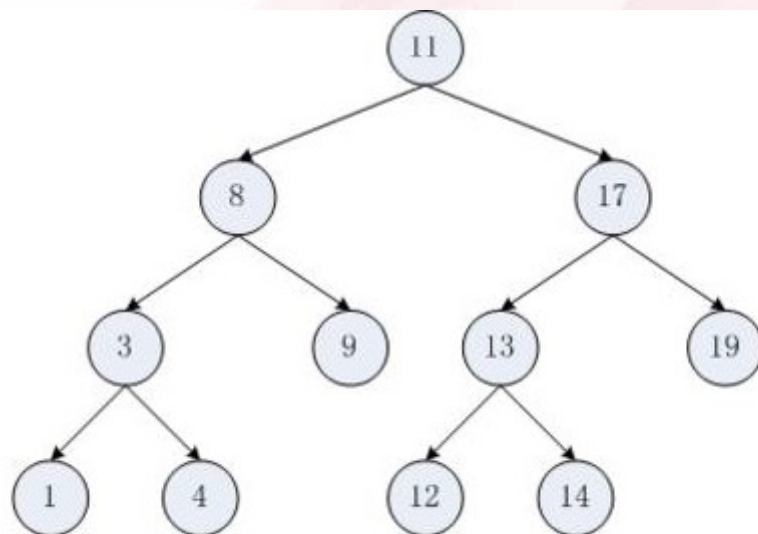
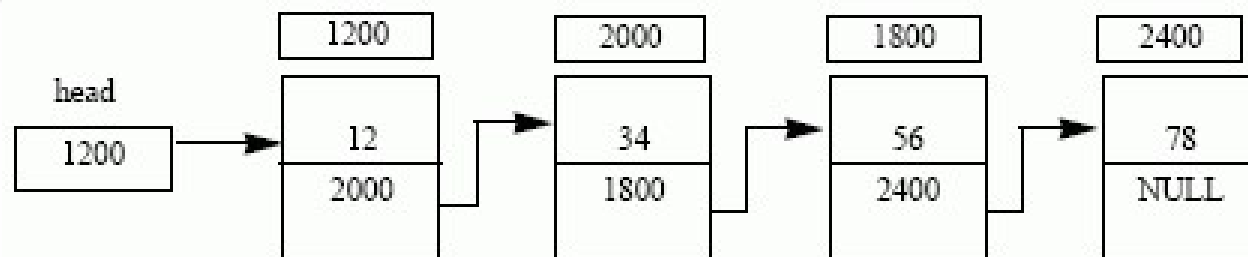
# 引言





## 目前的知识不能解决的问题

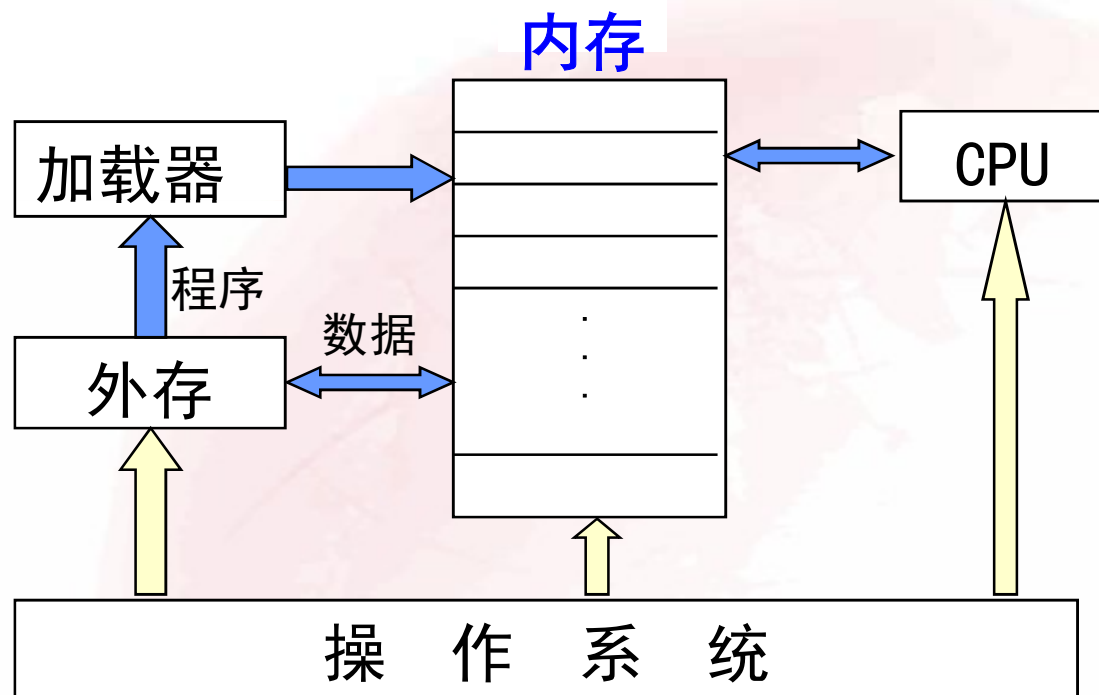
- ❖ 动态存储管理: `int a; int b[30]`
- ❖ 通过`return`语句返回多个数据: `return a;`
- ❖ 对函数参数按值传递的补充:  
`float max(float a, float b)`
- ❖ 一些复杂数据结构的实现
  - ❑ 链表
  - ❑ 树
  - ❑ 图





## 地址与指针

### ❁ 程序执行原理图



数据对象在生存期间都有**存储位置**，占据一定数量的**存储单元**

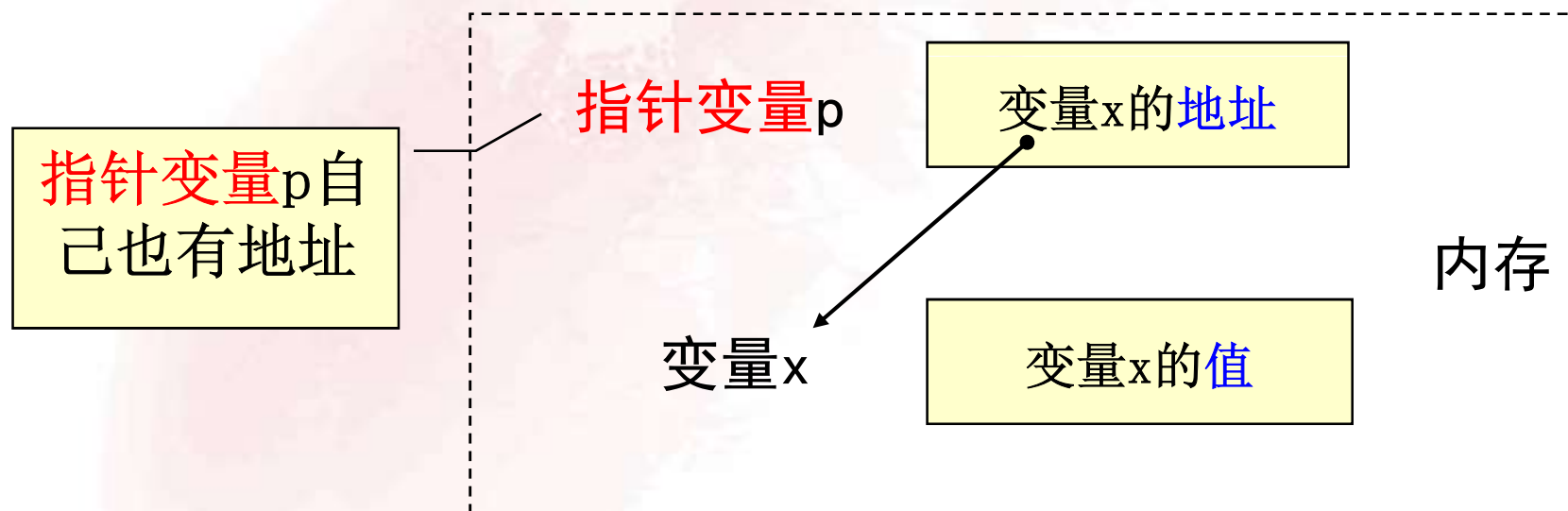
存储位置就是**内存地址**

存储单元的内容就是数据对象的值



## 地址与指针

- 如果把地址值也当作一种数据，存储地址值的变量就是**指针变量**，简称**指针** (pointer)





## 地址与指针

- ❖ 指针就是内存地址
- ❖ 指针没有**类型**，但指针指向的**对象具有类型****[注]**
- ❖ 无论指针变量指向什么类型的变量，指针自身只占据**4个字节**的内存单元(**32位计算机**)

**注：**通常会把指针指向的对象的类型称为指针的类型



# 地址与指针

## ✿ 指针的相关操作

- ✿ **指针赋值**：将程序对象的地址存入指针变量
- ✿ **间接访问**：通过指针变量存储的地址访问该地址对应的对象





## 主要内容

- ✿ 7.1 指针变量的定义和使用
- ✿ 7.2 指针作为函数参数
- ✿ 7.3 指针的相关问题
- ✿ 7.4 指针与数组
- ✿ 7.5 指针数组
- ✿ 7.6 动态存储管理





## 7.1 指针变量的定义和使用

---





## 定义指针变量

表示定义的是指针变量

❖ 格式：目标数据对象类型 \* 指针变量名称；

❖ 目标数据对象类型： 指针指向的数据对象的类型，用于确定指针指向的对象占据多少存储单元：char int long float double

❖ 指针变量名称： 任何合法标识符

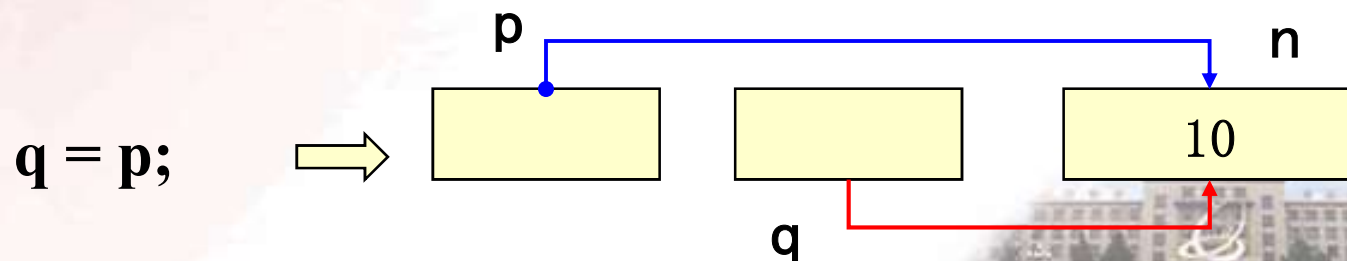
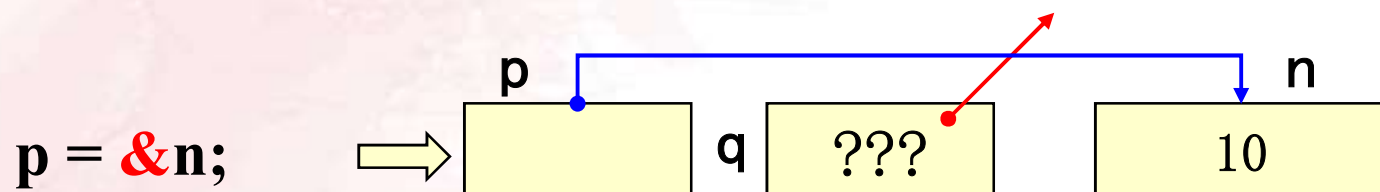
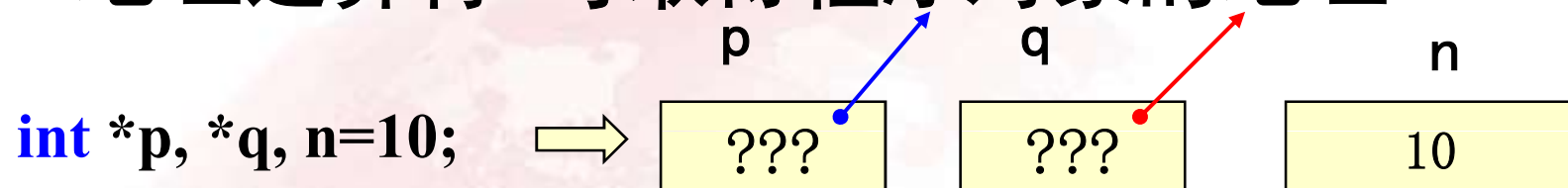
int \*p,\*q, n; //p和q是指针变量，n是整型变量

char \*s, ch; //s是指针变量，ch是字符变量



## 指针变量赋值

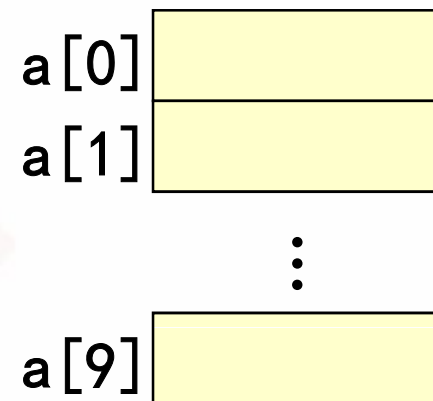
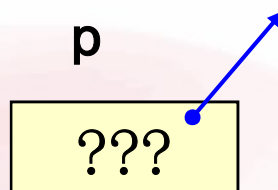
- 将程序对象的地址存入指针变量，通过取地址运算符`&`可取得程序对象的地址





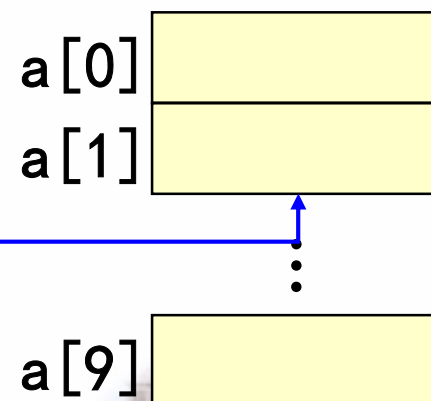
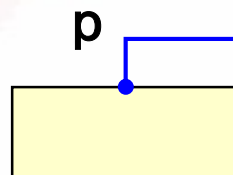
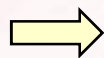
## 指针变量赋值

`int *p, a[10];`



&优先级和其他一元运算符  
相同，从右向左结合

`p = &a[1];`





## 间接访问

### 访问指针指向的对象

访问方法：\* 指针变量

```
int *p, *q, n=10, m, a[10];
```

```
p=&n;
```

```
q=p;
```

```
m=*p+*q*n;  $\Rightarrow$  m=n+n*n=110; //m=110
```

```
++*p;  $\Rightarrow$  ++n; //n=11
```

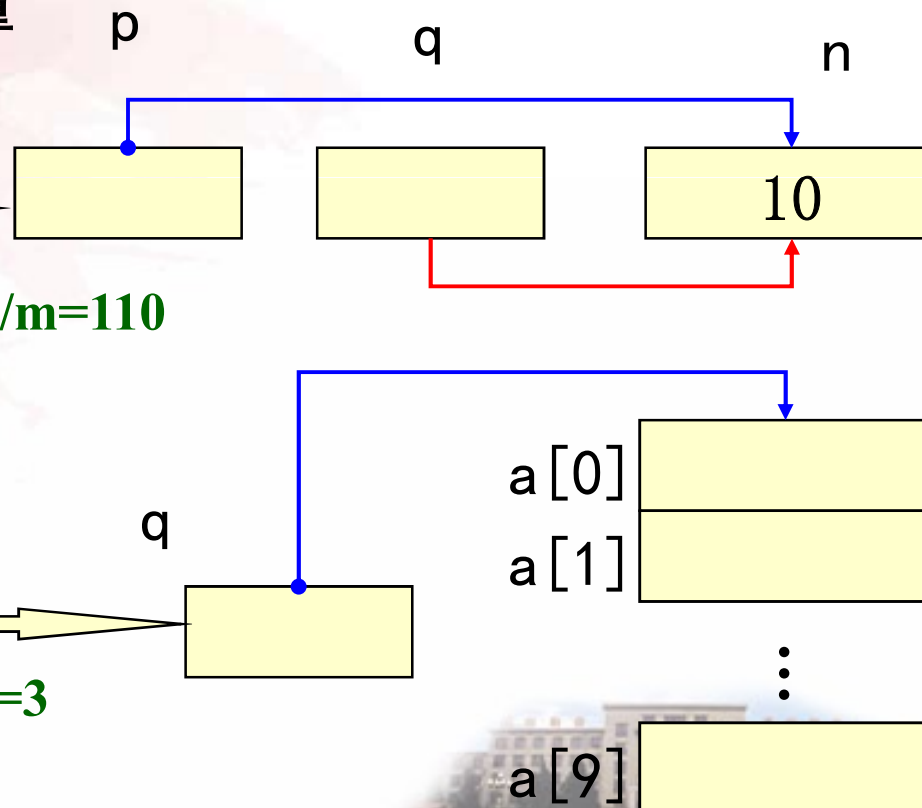
```
(*p)++;  $\Rightarrow$  n++; //n=12
```

```
*p+=*q+n;  $\Rightarrow$  n+=n+n; //n=36
```

```
q=&a[0];
```

```
*q=*p/12;  $\Rightarrow$  a[0]=n/12; //a[0]=3
```

\*优先级和其他一元运算符相同，从右向左结合





## 间接访问

### 间接访问的过程

- ❖ **step1.** 通过指针的值（地址）找到对应的内存单元
- ❖ **step2.** 通过指针的类型，确定数据占据多少内存单元
- ❖ **step3.** 将找到的内存单元内存储的数据按照指针的类型转换为相应数据

```
int *p,n=0x00000130;
```

```
char *q;
```

```
p=&n;
```

```
q=p;
```

```
printf("%d %c", *p, *q); //???
```

p

0x0012ff78

q

0x0012ff78

0x0012ff78

0x0012ff7b

30

01

00

00

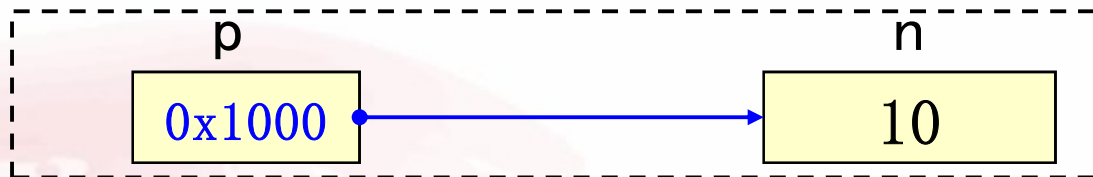
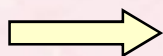


## 指针变量使用的一个小问题

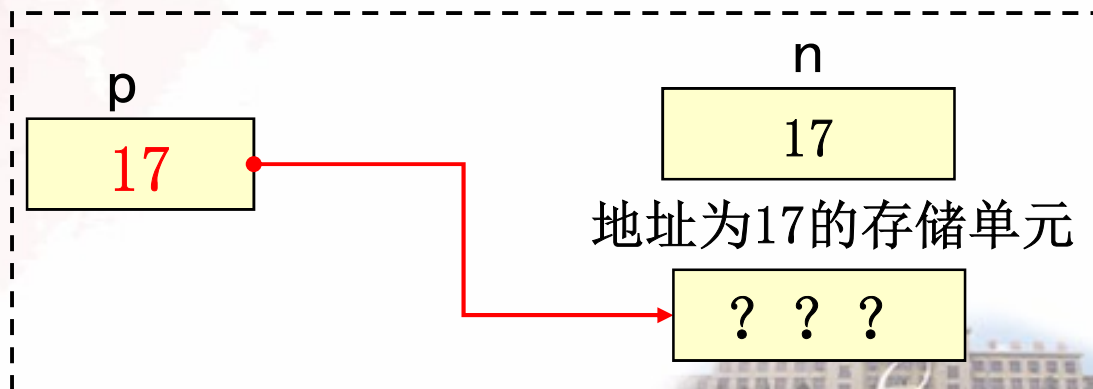
`int n=10, *p=&n;`

假设n存储在地址为0x1000的内存单元

`*p=17;`



`p=17;`







## 7.2 指针作为函数参数



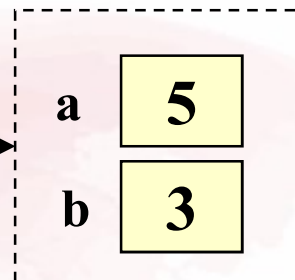


## 函数调用时实参按值传递

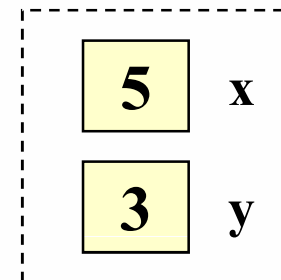
```
void swap( int x, int y ){  
    int t;  
    t = x; x = y; y = t;  
}
```

```
int main(){  
    int a=5, b=3;  
  
    printf( "a= %d; b= %d\n", a, b );  
    swap(a, b);  
    printf( "a= %d; b=%d\n", a, b );  
    return 0;  
}
```

main函数数据区

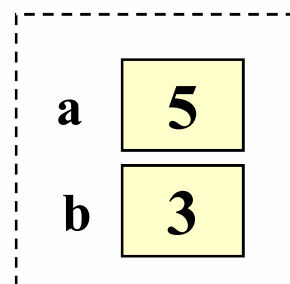


swap函数数据区

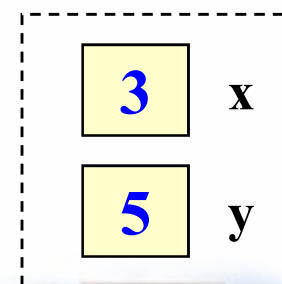


执行后

main函数数据区



swap函数数据区





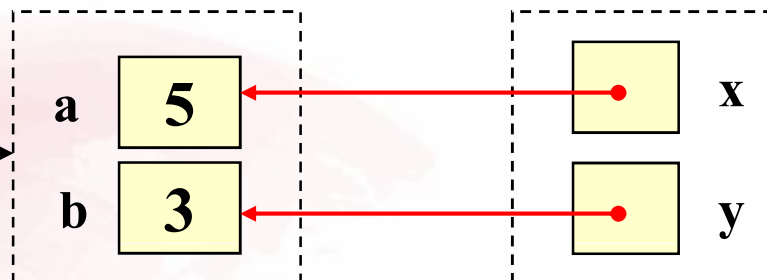
## 如果参数传递的值是变量地址

```
void swap( int *x, int *y ){  
    int t;  
    t = *x; *x = *y; *y = t;  
}
```

main函数数据区

swap函数数据区

执行前

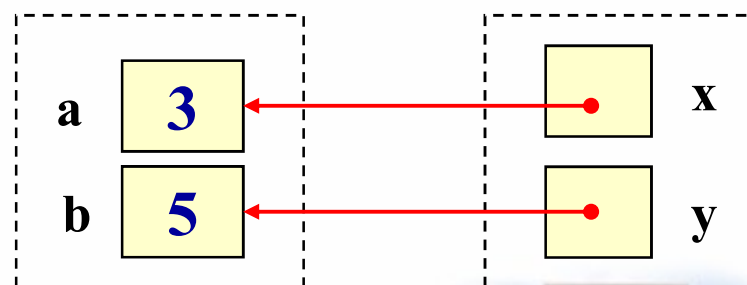


```
int main(){  
    int a=5, b=3;  
  
    printf( "a= %d; b= %d\n", a, b );  
    swap( &a, &b);  
    printf( "a= %d; b=%d\n", a, b );  
    return 0;  
}
```

执行后

main函数数据区

swap函数数据区





## 指针作为函数参数示例

- ✿ 用一个函数实现两个整数除法，要求返回整除的商和余数

- ❑ 用return语句不能返回两个值
- ❑ 用参数返回值，参数必须是指针形式

```
void division(int dividend,
int divisor,
int *quotientp,
int *remainderp)
{
    *quotientp = dividend / divisor;
    *remainderp = dividend % divisor;
}
```

```
int main(){
    int quot, rem;
    division(40, 3, &quot, &rem);
    printf("40/3 yield ");
    printf("quotient= %d ", quot);
    printf("remainder =%d\n", rem);
    return 0;
}
```



## 7.3 指针的相关问题





## 野指针

demo\_7\_pointer\_error.c

- ❖ 指针在未初始化时指向哪里不确定, 不能用\*间接引用指针指向的对象, 这样的指针称为野指针

```
int *p;  
*p=2;  
printf("%d",*p);
```



编译时警告p未初始化,  
执行时出错





## 空指针（NULL）

demo\_7\_pointer\_error.c

- ❖ 定义指针变量时如果暂时不知道要指向哪里，应该先给指针变量赋予空指针NULL（数值0），表示哪里也不指向
- ❖ 当指针变量被赋予空指针，不能用\*间接引用指针指向的对象

```
int *q=NULL;  
*q=2;  
printf("%d", *q);
```



编译器不会警告，  
执行时出错





# 指针作为函数参数的常见错误

## 一错误传递

```
void swap( int *x, int *y ){  
    int t;  
    t=*x;*x=*y;*y=t;  
}  
int main(){  
    int p,n;  
    scanf("%d", n);  
    scanf("%d", p);  
    swap(p, &n);  
    return 0;  
}
```

n和p的值会被当作地址传入函数，但由于n和p的值不确定，因此函数调用可能出错，即使不出错，输入的值也不会保存在n和p中





# 指针作为函数参数的常见错误

## —不判断指针的合法性

```
void swap( int *x, int *y ){  
    int t;  
    t=*x;  
    *x=*y;  
    *y=t;  
}
```

万一传进来的指针值为NULL (0) 怎么办?

demo\_7\_swap\_ok.c



## 通用指针与指针类型转换

- 通用指针指向的对象类型不确定，此时用 **void** 作为指针的类型

```
void *gp;
```

- 可以把任何类型的指针赋值给通用类型指针

```
int n=10,*p=&n;
```

```
float f=2.5;
```

```
void *gp1=p,*gp2=&f;
```

- 通用类型指针不能够用\*运算符间接引用被指对象。

```
int n=10;
```

```
float f=2.5;
```

```
void *gp1=&n,*gp2=&f;
```

```
printf("%d %f",*gp1,*gp2); //编译错误
```



## 通用指针与指针类型转换

- 为了访问通用指针指向的对象需要进行指针类型转换

转换方法: (类型 \*) 通用指针

```
int n=10, *p1;  
float f=2.5, *p2;  
void *gp1=&n, *gp2=&f;  
p1=(int *)gp1; p2=(float *)gp2;  
printf("%d %f",*p1, *p2); //正确
```

```
p1=(int *)gp2; p2=(float *)gp1;  
printf("%d %f",*p1, *p2); //运行结果不正确
```

一个指针可以转换为任何类型的指针，但是否具有正确的意义，取决于该指针实际指向的对象类型



## 理解指针操作

```
int a=0x04030201
```

```
int *p=&a;
```

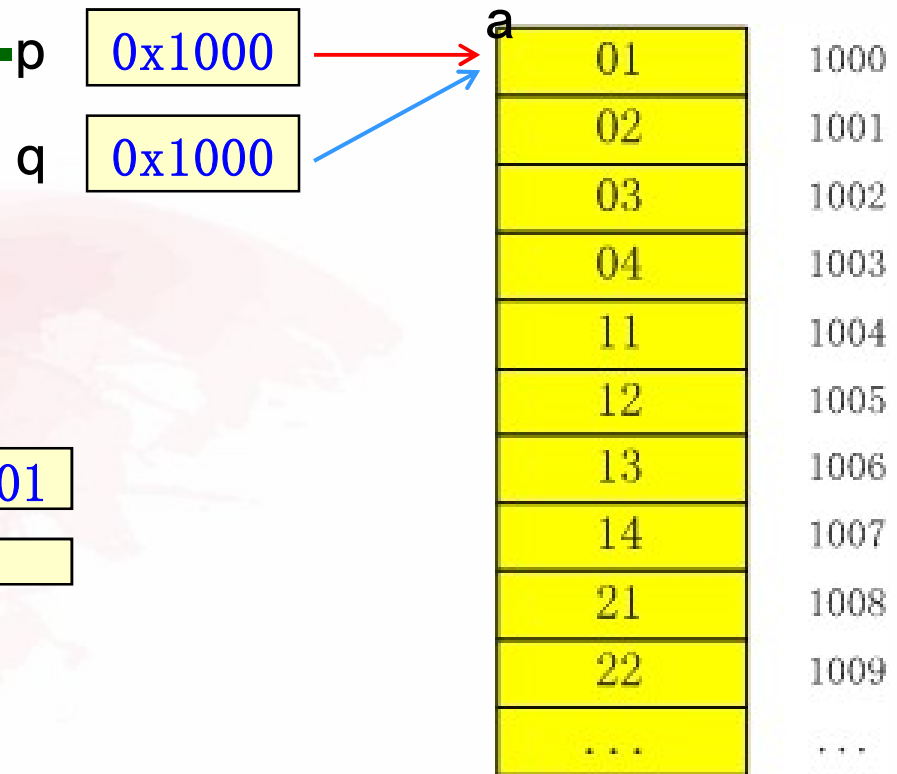
```
char *q=p;
```

```
int c=*p;
```

```
char d=*q;
```

c    0x04030201

d    0x01





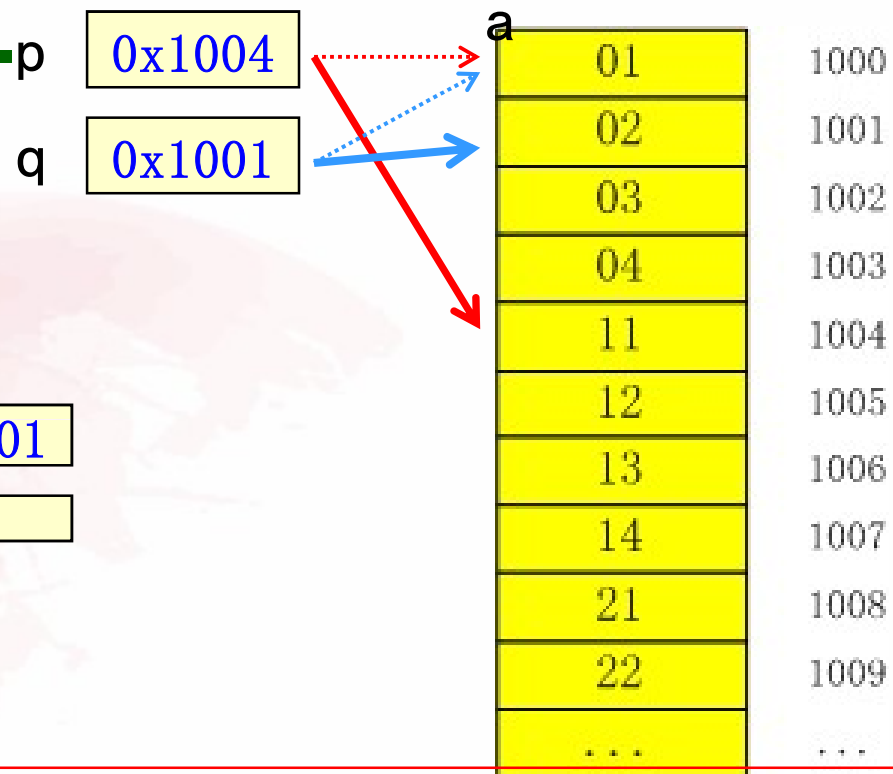
## 理解指针操作

```
int a=0x04030201;
int *p=&a //假设为1000
char *q=p;
```

```
int c=*p;      c  0x04030201
char d=*q;     d  0x01
```

```
p=p+1; //跳一个int单元, 4个字节
c=*p;   c  0x14131211
```

```
q=q+1; //跳一个char单元, 1个字节
d=*q;   d  0x02
```



```
int a;
void *tempP=&a;
tempP++; ???
//编译不通过, 提示 error C2036: 'void *' : unknown size
```



## 7.4 指针与数组





## 数组在内存中的实现

- ❖ 数组名是一个内存地址，称为数组首地址
- ❖ 数组元素从首地址开始连续存放
- ❖ 访问数组元素的方法：数组名[下标]

**int a[4];**

地址	数组元素
0x1000	?
0x1004	?
0x1008	?
0x100C	?

通过数组名和下标引用数组元素的本质是：

$\&a[i] = a + i \times \text{sizeof}(a[0])$

既然数组名是内存地址，就可以用一个指针变量来表示





## 取数组首地址的三种方法

- 取数组名对应的值  $a$
- 取数组名的地址  $\&a$
- 取第一个元素的地址  $\&a[0]$

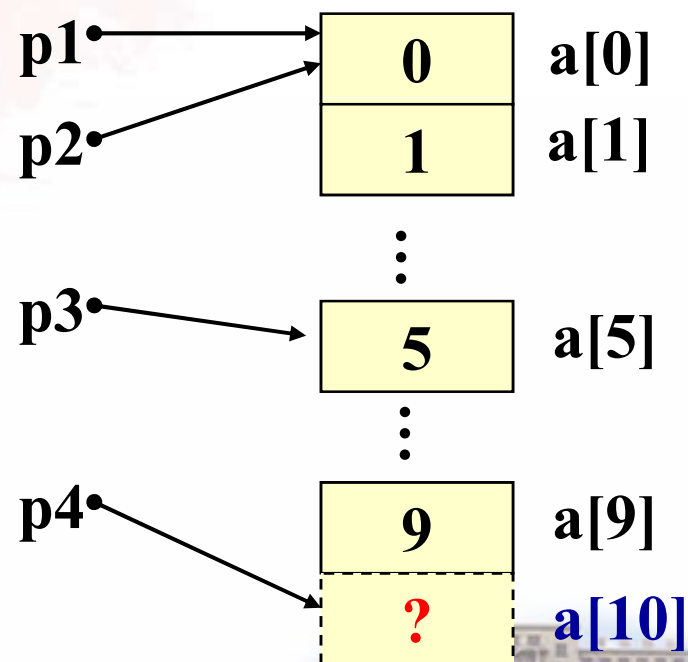
## 指向数组元素的指针

- 数组元素是一个普通变量，具有唯一地址
- 与数组同类型的**指针变量**可以指向数组的任一元素

```
int *p1,*p2,*p3,*p4;  
int a[10]={0,1,2,3,4,5,6,7,8,9};
```

```
p1 = &a[0];  
p2 = p1;  
p3 = &a[5];  
p4 = &a[10];
```

等价于  $p1=a;$







## 指针运算

指针不指向  
数组时没有  
任何意义

### 1. 指针和数值n（变量、常量）的加减法

- 前提：指针变量指向数组中的某个元素
- 意义：表示将指针向前或向后移动n个元素

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

```
int *p1=a,*p2;      →      p1指向a[0]
```

p1+=2;	→	p1指向a[2]
*p1=5;	→	给a[2]赋值
*(p1+6)=0;	→	给a[8]赋值
p2=p1+3;	→	p2指向a[5]
p2--;	→	p2指向a[4]



## 指针运算

两个指针变量相加没有任何意义

### ❖ 2. 两个指针变量相减

❖ 前提：两个指针变量指向同一个数组

❖ 意义：得到两个指针之间的元素个数

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

```
int *p1=a,*p2; → p1指向a[0]
```

```
int n;
```

```
p2=&a[5]; → p2指向a[5]
```

```
n=p2-p1; → n=5
```

```
p1++; → p1指向a[1]
```

```
n=p1-p2; → n=-4
```



## 指针运算

两个指针变量不在同一数组，比较没有任何意义

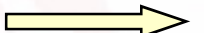
### ❖ 3. 两个指针变量比较大小

❖ 前提：两个指针变量指向同一个数组

❖ 意义：得到两个指针之间的位置关系

```
int a[10];
```

```
int *p1=a,*p2=&a[10];
```



p1指向a[0], p2指向a[10]

```
int i=0;
```

```
while( p1<p2 ){
```

```
    *p1=i;
```

```
    p1++;
```

```
    i++;
```

```
}
```

通过指针p1依次给a的每个元素赋值i，循环结束条件是p1指向最后一个元素之后

demo\_7\_compare.c



## 指针运算

指针变量不指向数组时没有任何意义

### 4. 对指针变量使用下标

前提：指针变量指向数组

意义：得到对应数组元素

```
int a[10];
```

```
int *p1=a;
```

→ p1指向a[0]

```
int i=0;
```

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

```
    p1[i]=i;
```

```
}
```

通过指针p1依次给a的每个元素赋值i，循环结束条件是i不超出数组元素个数

$p1[i] = p1 + i \times \text{sizeof}(a[0])$



## 访问数组元素的几种模式

```
int a[10];  
int i;  
for(i=0;i<10;i++){  
    a[i]=i;  
}
```

```
int a[10], i=0;  
int *p1,*p2;  
for(p1=a,p2=a+10;p1<p2; p1++){  
    *p1=i;  i++;  
}
```

```
int a[10],i=0;  
int *p1;  
for( p1=a;p1<a+10; p1++){  
    *p1=i;  i++;  
}
```

```
int a[10], i=0;  
int *p1,*p2;  
for( p1=a,p2=a;p1-p2<10; p1++){  
    *p1=i;  i++;  
}
```



## 7.4 指针与数组

### ✿ 取值运算符(\*)和递增运算符(++)的混合使用

- ❏ \*和++优先级相同
- ❏ \*和++都是右结合: `*++p; ++*p`
- ❏ ++有前缀和后缀两种形式
  - 前缀方式先自增再取值参与运算
  - 后缀方式先取值参与运算再自增





## 7.4 指针与数组

### ✿ 取值运算符(\*)和递增运算符(++)的混合使用

```
int a[10]={0,1,2,3,4,5,6,7,8,9}, *p=a, n;
```

<code>n = *p++;</code>	→	<code>n=*p; p++;</code>	→	<code>n=a[0] p指向a[1]</code>	等效 <code>n=*(p++);</code>
<code>n = *++p;</code>	→	<code>++p; n=*p;</code>	→	<code>p指向a[2] n=a[2]</code>	
<code>++*p;</code>	→	<code>a[2]++;</code>			

demo\_7\_zuhe.c



## 数组参数与指针

- 数组作为函数形式参数时，实际是以指针的形式实现

```
double average(  
    double a[], int n){  
    double sum=0.0;  
    int i;  
    for(i=0;i<n;i++) sum+=a[i];  
    return sum/n;  
}
```

```
double average(  
    double *a, int n){  
    double sum=0.0;  
    int i;  
    for(i=0;i<n;i++) sum+=a[i];  
    return sum/n;  
}
```

```
int main(){  
    int b[20]={...}  
    average(b,20);  
    return 0;  
}
```

传递的是数组  
首地址

只对b[5]-b[10]  
求平均值如何处  
理??





## 数组参数与指针

```
double average(  
    double a[],  
    int n){  
    double sum=0.0;  
    int i;  
    for(i=0;i<n;i++) sum+=a[i];  
    return sum/n;  
}
```



```
double average(  
    double *a,  
    int n){  
    double sum=0.0;  
    int i;  
    for(i=0;i<n;i++) sum+=a[i];  
    return sum/n;  
}
```

```
int main(){  
    double b[20]={...}  
    average( b+5, 6);  
    return 0;  
}
```

传递的是元素  
b[5]的地址



## 示例1：用指针实现求字符串长度的函数

```
int strlen( char *s ){
    int n = 0;
    while ( *s != '\0' ) {
        ++s; ++n;
    }
    return n;
}
```

```
int strlen( char *s ){
    char *p = s;
    while ( *p != '\0' )
        ++p;
    return p - s;
}
```

```
int main(){
    char s1[]="hello,world!"; *ps=s1;
    int len;
    len = strlen(s1);
    len = strlen(ps);
    len = strlen("c language");
    return 0;
}
```

调用以字符指针为形式参数的函数，实参可以是

- 字符串常量: `strlen( "lgx" );`
- 字符数组（其中应存着字符串）: `char a[4];`  
`strlen(a);`
- 指向字符串的指针: `char a[10]="lgx", *p=a;`  
`strlen(p);`



## 示例2：用指针实现复制字符串的函数

```
void strcpy (char *d, char *s) {  
    while ( (*d = *s) != '\0' ) {  
        d++; s++;  
    }  
}
```

```
void strcpy (char *d, char *t) {  
    while ( *d = *t ) {  
        d++; t++;  
    }  
}
```

```
void strcpy (char *d, char *s) {  
    while ( *d++ = *s++ );  
}
```

注意：

- 运算符的优先级与结合顺序
- 增量运算的作用与计算出的值
- 赋值表达式的值



## 字符指针与字符数组

- ❖ 字符指针赋值为字符数组首地址时可操作整个字符串
- ❖ 用字符指针定义字符串，在赋值时比较方便



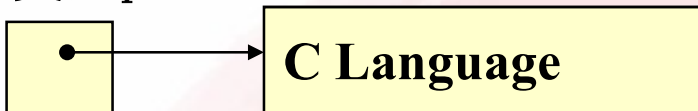


# 字符指针与字符数组

```
char *p="C Language";
```

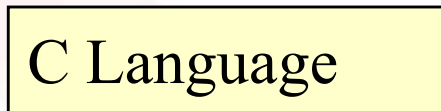
指针变量p

常量字符数组s



```
char s[]="C Language";
```

字符数组s



```
p="B Language";
```

正确，s指向另一个常量字符数组

```
s="B Language";
```

编译错误，不能给数组整体赋值

```
*p='D';
```

错误，不能修改常量字符数组的元素  
能编译，但运行错误

```
s[0]='D';
```

正确，给s[0]赋值



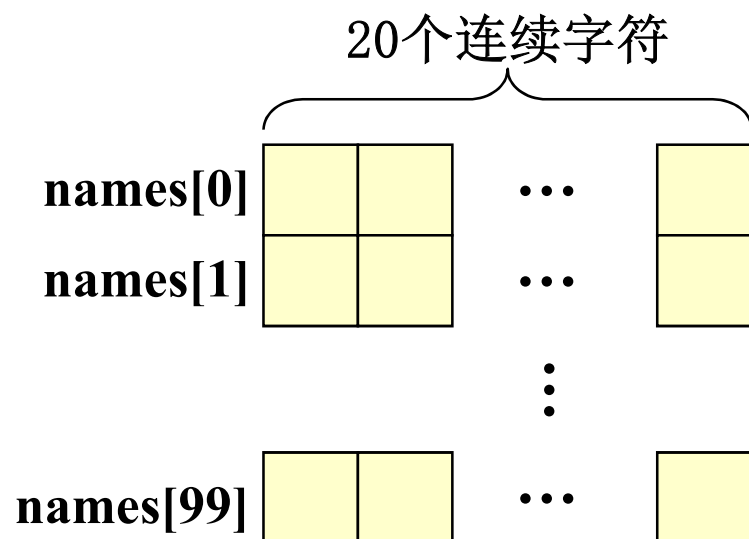
## 7.5 指针数组

- ✿ 假设一个班有100人，如何表示每个人的姓名？

## ❁ 方法1：二维数组

❁ `char names[100][20];`

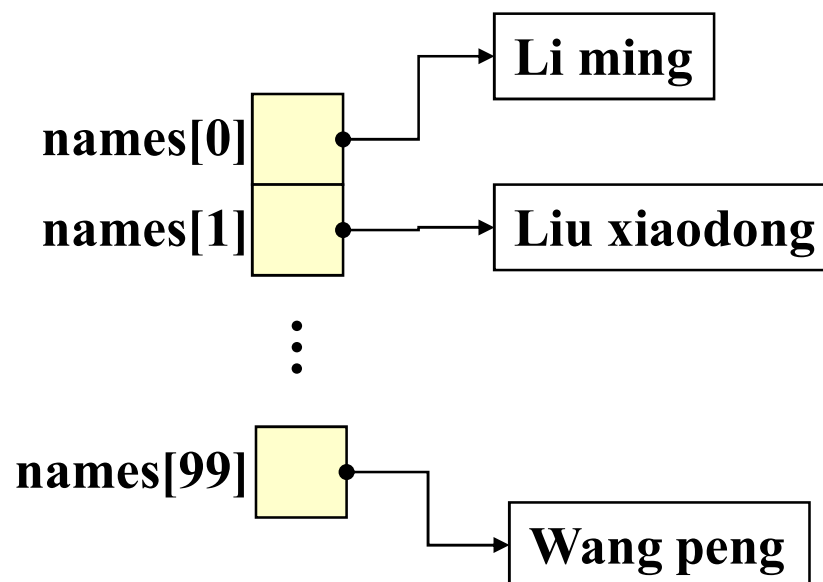
- ❁ 每个人的名字长度限定为不超过20个字符



## ❁ 方法2：一维指针数组

❁ `char *names[100]`

- ❁ 每个人的名字长度不固定







## 二维数组与一维指针数组的区别

- ❖ 二维数组的**所有元素**在内存中**连续存放**，要求一块较大的连续内存（行数 $\times$ 列数）
- ❖ 一维指针数组**各个指针连续存放**，但指针指向的内存区地址并不连续，只需要一块能连续存放指针的内存（行数）



## 二维数组与一维指针数组的区别

//以下为程序片段

```
char names1[100][20]={"Li ming","Liu xiaodong",...};  
char *names2[100]={"Li ming","Liu xiaodong",...};  
int i=0;  
for(i=0;i<5;i++)  
    printf("address of names1[%d][0] is %d\n",i,&names1[i][0]);  
  
for(i=0;i<5;i++)  
    printf("address of names2[%d][0] is %d\n",i,&names2[i][0]);
```

demo\_7\_pointer\_array.c

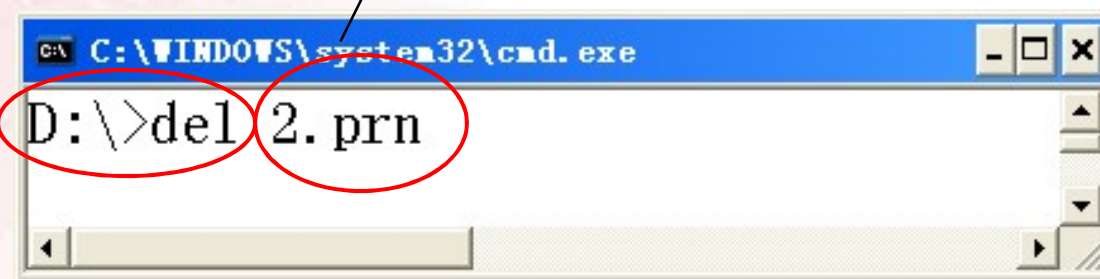


## 命令行参数

- ❁ DOS下的命令行程序都可以有**命令行参数**，用于控制程序执行，或者给程序提供输入信息

命令行参数，可以有多个

命令或  
可执行程序





## 命令行参数

- ❁ C语言的程序也可以提供命令行参数，该参数在main函数中以指针数组形式给出

```
int main( int argc, char *argv[] ){  
    ...  
    return 0;  
}
```

命令行参数个数，  
大于等于1

指针数组形式给出的  
命令行参数，第一个  
参数总是程序名



## 指针数组示例

cmd\_param.c

- 从命令行参数得到两个字符串，输出相同前缀，如果没有则输出“no same prefix”。

argv[0] c m d \_ p a r a m '0'

argv[1] a a a c '0'

argv[2] a a a b '0'

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [版本 5.1.2600]
(C) 版权所有 1985-2001 Microsoft Corp.

C:\Documents and Settings\Administrator>d:
D:\>cmd_param aaa aaab
```

argv[1] a a a '0'

s1  
↓



## 7.6 动态内存管理





## 示例

### ✿ 写一个处理一组（数量不确定）学生成绩数据的程序

#### ❏ 数组的局限性—必须事先指定数组大小

- 数组太小不能满足需求
- 数组太大浪费存储空间

#### ❏ 如何才能按需分配内存？

//程序框架

**int** n; //表示数据项数

**scanf**("%d",&n); // 读入数据项数

... //根据数据项数分配存储单元

这里需要用到  
动态内存管理





## 动态内存管理机制

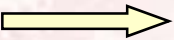
- ❖ 指针可以和数组发生联系，根据指针也能访问连续的存储单元
- ❖ 如果可以动态分配一块连续内存单元，并得到其首地址，则可以通过指针访问这块连续内存单元中的任一单元
- ❖ 动态分配的内存位于堆（heap）中，只要不释放就会一直存在。
- ❖ 动态分配的内存必须要释放，否则会一直占据内存导致内存泄漏

注：动态内存（heap），局部变量（stack），全局、静态变量（static），文字常量区，程序区



## 动态内存管理流程

- 分配内存—在内存中分配指定大小的连续存储单元

`malloc`  
`calloc`  
`realloc` }  `#include <stdlib.h>`

- 使用内存—以数组或指针方式使用分配的内存

`memset`  
`memcpy` }  `#include <memory.h>`

- 释放内存—释放已分配的内存

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



# 分配内存

❁ `void *malloc(size_t n);`

- ❁ 分配n字节连续内存
- ❁ 返回值是分配好的连续内存起始地址，如果分配失败返回NULL
- ❁ 分配的内存未初始化为0，每个单元的值都不确定
- ❁ 返回值是通用指针类型，需要做指针类型转换



## 分配内存

❁ `void *calloc(size_t n, size_t size);`

- ❁ 分配 `n*size` 字节连续内存，并将每个字节都清零，即全部初始化为0
- ❁ 返回值是分配好的连续内存起始地址，如果分配失败返回NULL
- ❁ 返回值是通用指针类型，需要做指针类型转换

```

int n=10;
int *p = (int *) malloc( n * sizeof( int ) );
if( p!=NULL ){
    //分配成功
}
else{
    //分配失败
}

```

//.....其它操作.....

```

if(p!=NULL)
{
    free(p) ;
    p=NULL;
}

```

## malloc V.S. calloc

```

int n=10;
int *p = (int *) calloc( n , sizeof( int ) );
if( p!=NULL ){
    //分配成功
}
else{
    //分配失败
}
if(p!=NULL)
{
    free(p) ;
    p=NULL;
}

```



# 分配内存

❖ `void *realloc(void *p, size_t n);`

❖ 将原来已分配的内存大小 $m$ 调整为 $n$

- 如果 $n < m$ ，前 $n$ 个字节内容不变，多余的单元被释放
- 如果 $n > m$ ，尝试保持原地址不变，在后面追加 $n - m$ 个字节，如果在原起始地址开始不能分配连续 $n$ 个字节，则重新在其他地方分配 $n$ 字节内存，并将原来的 $m$ 个字节复制到新的存储区

❖ **返回值**是调整后的连续内存起始地址，如果调整失败返回NULL，即使调整成功返回的起始地址可能不是原来的地址，因此需要将**返回值赋值给原来的指针变量**

❖ **返回值是通用指针类型，需要做指针类型转换**



# 分配内存

## realloc使用框架

```
int n=10;
int *p = (int *) malloc( n * sizeof( int ) );
//改变n的大小后重新分配
n=20;
p = (int *) realloc( p, n * sizeof( int ) );
if( p!=NULL ){
    //分配成功
}
else{
    //分配失败，不应该再继续后续操作
}
...
if(p!=NULL)
{
    free(p);
    p=NULL;
}
```





# 使用内存

## ✿ 以数组或指针方式使用分配的内存

```
int n=10,i;
int *p = (int *) malloc( n * sizeof( int ) );
if( p!=NULL ){ //分配成功
    for( i=0;i<10;i++)
        *p++=i;
}
else{
    //分配失败
}
...
if(p!=NULL)
{
    free(p);
    p=NULL;
}
```

```
int n=10,i;
int *p = (int *) malloc( n * sizeof( int ) );
if( p!=NULL ){ //分配成功
    for( i=0;i<10;i++)
        p[i]=i;
}
else{
    //分配失败
}
...
if(p!=NULL)
{
    free(p);
    p=NULL;
}
```



# 使用内存

❁ `void *memset( void *dest, int c, size_t count );`

- ❁ 将分配好的内存从dest开始的连续count个字节设定为同一个值c
- ❁ 该函数也可以应用于数组

❁ `void *memcpy( void *dest, const void *src, size_t count );`

- ❁ 将从src开始的连续count个字节复制到从dest开始的内存单元
- ❁ 该函数也可以应用于数组



## 使用内存

```
int n=10,a[5]={1, 2, 3, 4, 5};
int *p = (int *) malloc( n * sizeof( int ) );
if( p!=NULL ){ //分配成功
    memset(p, 0, n*sizeof(int) ); //将分配好的内存清零
    memcpy(p, a, 5*sizeof(int) ); //复制数组a的前5个数据
}
else{ //分配失败，不应该再继续后续操作
}
。 。 。
if(p!=NULL)
{
    free(p); //释放指针指向的内存，但P中保存的地址值不变
    p=NULL; //让P变为空指针，预防后面继续使用导致Bug
}

```



# 释放内存

- ✚ 所有用`malloc`、`calloc`、`realloc`分配的内存都必须释放
- ✚ 释放内存函数：`void free( void *memblock );`
  - ✚ 将指针`memblock`指向的内存块释放
    - 释放多少个单元由操作系统内存管理机制确定
    - 如果传入的指针不是原来分配的内存区起始地址会导致错误
    - 释放后`p`的值不会改变，但`p`所指向的内存单元均已无效不能再使用



# 释放内存

```
int n=10;  
int *p = (int *) malloc( n * sizeof( int ) );  
if( p!=NULL ){ //分配成功  
    ...  
    free(p);  
    // *p=10; //错误, p已经无效  
    p=NULL;  
}
```



## 示例

- ❁ 输入学生的个数 $n$ ，然后申请内存，输入并保存 $n$ 个学生的成绩，最后求 $n$ 个学生的平均分。

demo\_7\_mem.c



## 示例

- 用函数实现读入n个整数，在函数内分配存储单元并通过返回值返回存储单元起始地址

```
int *input( int n){  
    int *p=NULL,i;  
    p=(int *)malloc(sizeof(int) * n);  
    if( p!=NULL ){  
        for(i=0;i<n;i++){  
            scanf("%d", p+i );  
        }  
    }  
    return p;  
}
```





## 示例

```
int main(){
    int *data, a[10], n, i;
    scanf("%d",&n);
    data = input(n);
    printf("alloc address is %x\n",data);
    if(data!=NULL){
        memset(a, 0, sizeof(int)*10 );
        memcpy(a, data, n*sizeof(int) );
        for(i=0;i<10;i++)
            printf("a[%d]=%d\n", i, a[i]);
        free(data);
    }
    return 0;
}
```



## 关于返回指针的函数

- ❖ 函数的返回值可以是指针类型
- ❖ 函数返回值是指针类型时可以返回任何一个同类型地址值，但必须保证这个地址所对应的存储单元在函数结束后仍然有效

```
int *input( int n){  
    int a[100],i;  
    for(i=0;i<n&& i<100;i++){  
        scanf("%d", p+i );  
    }  
    return a;  
}
```

a确实可以当作一个int类型的指针值返回，但a是局部变量，对应的内存单元在函数结束后无效



## 小结

- ✿ 指针和地址的关系
- ✿ 指针和数组的联系
- ✿ 指针作为函数参数
- ✿ 字符指针和字符串
- ✿ 指针数组和命令行参数
- ✿ 动态内存管理

