

程序设计基础

(C Programming)

第五讲：程序设计方法（三）

复杂数据程序设计



本章目标

- 掌握二维（多维）数组的定义与初始化；
- 掌握指针说明与指针运算；
- 掌握指针与数组的关系；
- 掌握指针作为函数参数；
- 掌握指针数组；
- 掌握结构的定义和使用；
- 了解自引用结构。



问题5.1：旋转矩阵

【问题描述】

输入一个自然数 N ($2 \leq N \leq 9$)，要求输出如下的旋转矩阵，即边长为 $N \times N$ ，元素取值为1至 $N \times N$ ，1在左上角，呈顺时针方向依次放置各元素。

$N=3$ 时：

1	2	3
8	9	4
7	6	5

【输入形式】

从标准输入读取一个整数 N ($2 \leq N \leq 9$)。

【输出形式】

向标准输出打印结果。输出符合要求的方阵，每个数字占5个字符宽度，向右对齐，在每一行末均输出一个回车符。

【输入样例】

4

【输出样例】

1	2	3	4
12	13	14	5
11	16	15	6
10	9	8	7

问题5.1： 问题分析

■ 数据结构设计

- 显然可用一个 9×9 的二维整数数组来存放生成的旋转矩阵。

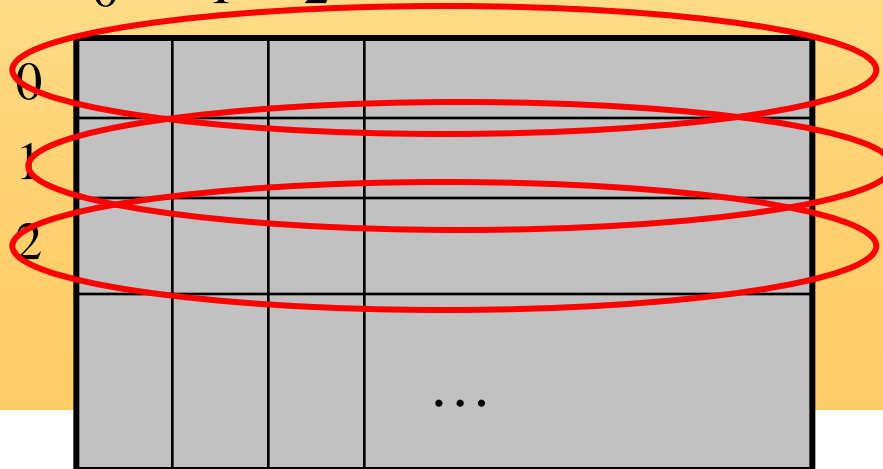
二维（多维）数组

■ 二维（多维）数组

如， `float y[4][3];`

在C语言中，二维数组可以看作是一个元素为另一个一维数组的一维数组；三维数组可以看作元素为二维数组的一维数组…。因此，在C语言中，下标变量应写作：`y[i][j]`，而不能写成：

`y[i, j]`





二维（多维）数组初始化

多维数组的初始化

```
int y[4][3] = {  
    { 1, 3, 5 },  
    { 2, 4, 6 },  
    { 3, 5, 7 },  
}
```

int y[4][3] = { 1, 3, 5, 2, 4, 6, 3, 5, 7 };
也同上，因为在C语言中，数组元素按行存贮。

```
int y[4][3] = {  
    {1}, {2}, {3}, {4}  
}
```

```
int y[4][3] = {0}; /* 初始化为0 */
```

1	3	5
2	4	6
3	5	7
0	0	0

1	0	0
2	0	0
3	0	0
4	0	0



二维（多维）数组使用*

例：对一个二维数组分别按行和按列求和

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

```
#define ROWS 4
```

```
#define COLS 4
```

```
int main()
```

```
{
```

```
    int matrix[ROWS][COLS], row, col, sum;
```

```
    for(row=0; row<ROWS; row++)
```

```
        for(col=0; col<COLS; col++)
```

```
            scanf("%d", &matrix[row][col]);
```

初始化数组元素

```
    for(row=0; row<ROWS; row++) {
```

```
        sum = 0;
```

```
        for(col=0; col<COLS; col++)
```

```
            sum += matrix[row][col];
```

```
        printf("Sum of row%d = %d\n", row, sum);
```

```
    }
```

按行求和

```
    for(col=0; col<COLS; col++) {
```

```
        sum = 0;
```

```
        for(row=0; row<ROWS; row++)
```

```
            sum += matrix[row][col];
```

```
        printf("Sum of col%d = %d\n", col, sum);
```

```
    }
```

```
    return 0;
```

```
}
```

按列求和

二维（多维）数组使用

注意：如果把一个二维数组作为参数，则在函数定义中，形参数组的说明中必须指明列的数目，而行的数目可空着不写。

如：

```
main( )
{
    float a[4][3], b[3][4], c[4][4];
    ...
    fun(a, b, c);
    ...
}

void fun(float x[ ][3], float y[ ][4], float z[ ][4])
{
    ...
}
```


问题5.1：算法设计

若N为所输入的整数（矩阵的阶），则旋转的层数：

$$L = N/2$$

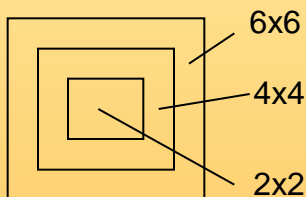
A

1	2	3	4
12	13	14	5
11	16	15	6
10	9	8	7

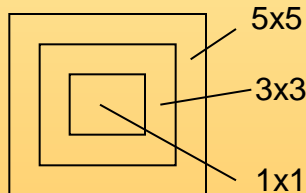
B

D

C



N=6



N=5

设m当前层(从0开始)，num为要填充的数字，则有：

A

```
for(i=m; i<N-m-1; i++)
```

```
    array[m][i] = num++;
```

B

```
for(i=m; i<N-m-1; i++)
```

```
    array[i][N-m-1] = num++;
```

C

```
for(i=m; i<N-m-1; i++)
```

```
    array[N-m-1][N-i-1] = num++;
```

D

```
for(i=m; i<N-m-1; i++)
```

```
    array[N-i-1][m] = num++;
```

注意：当N为奇数时，如N=3，L为1，则只填了一圈数字，中心数字（即最后一个数字）未填。因此，在填完所有层后有：

if(N%2 != 0) array[N/2][N/2] = N*N;



问题5.1： 算法设计

- 设`array[9][9]`用于存放旋转矩阵，`n`为旋转矩阵的阶，`m`为当前填写的层数，初值为0；
- 读入一个整数到`n`；
- `for (m=0; m<n/2; m++)`
 - 填充当前层（即分别填充A，B，C，D四段）；
- 若`n`为奇数，则
 - 填充最后一个数字；



问题5.1：代码实现

```
/* c5_1.c */
#include<stdio.h>
#define SIZE 9
int main() {
    int array[SIZE][SIZE];
    int n,m, num, i,j;
    num = 1;
    scanf("%d",&n);
    for (m=0;m<n/2;m++) {
        for (i=m;i<n-m-1;i++)
            array[m][i] = num++;
        for (i=m;i<n-m-1;i++)
            array[i][n-m-1] = num++;
        for (i=m;i<n-m-1;i++)
            array[n-m-1][n-i-1] = num++;
        for (i=m;i<n-m-1;i++)
            array[n-i-1][m] = num++;
    }
    if (n%2)
        array[n/2][n/2] = num;
    for (i=0;i<n;i++) {
        for (j=0;j<n;j++)
            printf("%5d",array[i][j]);
        printf("\n");
    }
    return 0;
}
```

测试应考虑：

N=3（正常）

N=4（正常）

N=2（边界）

N=9（边界）



问题5.1：常见问题分析

- 本问题一个常见的错误就是**数组下标表达式**出错，这也是应用数组时常犯的**错误**。如，本题在调试过程中出现如下问题：

C

```
for(i=m; i<N-m-1; i++)  
    array[N-m-1][N-m-i-1] = num++;
```

D

```
for(i=m; i<N-m-1; i++)  
    array[N-m-i-1][m] = num++;
```

- 没有考虑到N为奇数的情况（输入N=3时的现象）。

```
/* c5_1b.c */  
#include<stdio.h>  
#define SIZE 9  
int main() {  
    int array[SIZE][SIZE];  
    int n,m, num, i,j;  
  
    num = 1;  
    scanf("%d",&n);  
    for (m=0;m<n/2;m++) {  
        for (i=m;i<n-m-1;i++)  
            array[m][i] = num++;  
        for (i=m;i<n-m-1;i++)  
            array[i][n-m-1] = num++;  
        for (i=m;i<n-m-1;i++)  
            array[n-m-1][n-i-1] = num++;  
        for (i=m;i<n-m-1;i++)  
            array[n-i-1][m] = num++;  
    }  
    for (i=0;i<n;i++) {  
        for (j=0;j<n;j++)  
            printf("%5d",array[i][j]);  
        printf("\n");  
    }  
    return 0;  
}
```



问题5.1： 另一种解题思路

观察每层（圈）填充数据时**A**，**B**，**C**和**D**段数组下标变化规律，会发现什么？

其它方法？

A				B
1	2	3	4	
12	13	14	5	
11	16	15	6	
D	10	9	8	C
			7	

```
/* c5_1a.c */
#include<stdio.h>
#define SIZE 9
int main() {
    int array[SIZE][SIZE];
    int n,m, num, i,j;
    num = 1;
    scanf("%d",&n);
    for (m=0;m<n/2;m++) {
        for (i=m,j=m;j<n-m-1;j++)
            array[i][j] = num++;
        for (;j<n-m-1;i++)
            array[i][j] = num++;
        while (j>m)
            array[i][j--] = num++;
        while (i>m)
            array[i--][j] = num++;
    }
    if (n%2)
        array[n/2][n/2] = num;
    for (i=0;i<n;i++) {
        for (j=0;j<n;j++)
            printf("%5d",array[i][j]);
        printf("\n");
    }
    return 0;
}
```



指针

- 指针是用来确定另一个数据项地址的数据项。
- 指针变量是用来存放所指对象地址的变量。
- 在C语言中，允许指针指向任何类型的对象（可指向基本类型、构造类型），甚至可指向其它指针或指向函数。
- 在C语言里，当对象本身不能被直接传送的情况下，往往可以通过指针来进行传递。如函数的参数或返回结果通常是基本类型，但也可以是指向任何构造类型的指针。
- 指针应具有非零（无符号整数）值，如将0（通常#define NULL 0）赋给予指针，则该指针没有指向任何具体对象，即空指针。
- 在C语言中，指针使用得较多，指针用好了，可使程序表达能力大大加强，但用时需多加小心，要切实掌握指针的含义和用法，否则会使程序运行时乱套。

指针定义

- ## ■ 指针变量的定义（说明）：

〈类型〉 *〈变量〉;

指针是用所指对象类型来表征的。如：

```
int    *px;                /* 指向整型的指针 */
```

```
char *pc;           /* 指向字符型的指针 */
```

```
char *acp[10]; /* 由指向字符的指针构成的数组，即指针数组 */
```

```
int f( ); /* 返回值为整型的函数 */
```

```
int *fpi( );    /* 返回值为指向整型的指针的函数, 指针函数 */
```

```
int (*pfi) (); /*指向一个返回值为整型的函数的指针, 函数指针*/
```

指针运算符

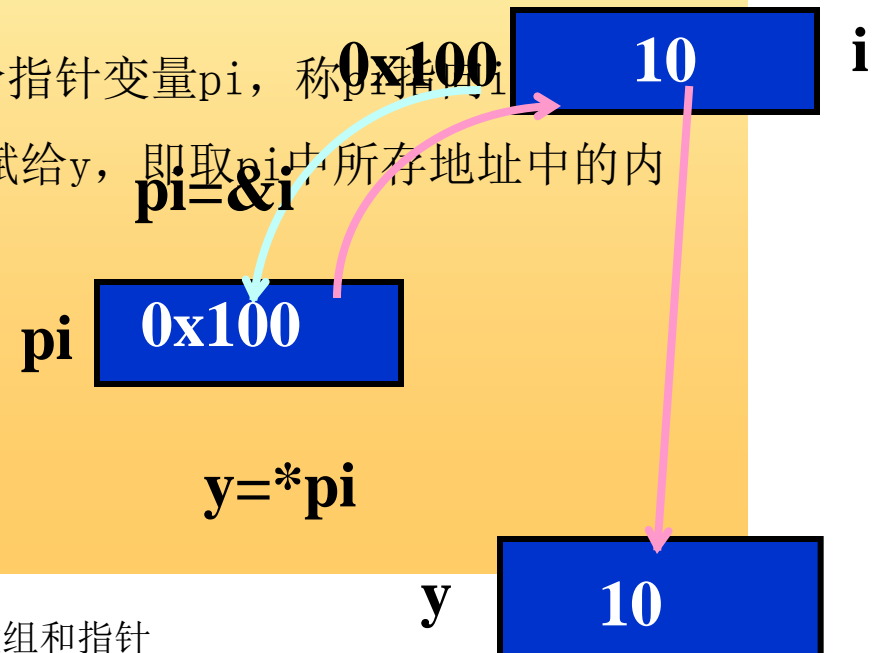
- 单目（取地址）运算符 **&**：用来取变量或数组成员地址的运算符。（获得某个变量的指针的运算符）
- 单目（间接引用，或递引用）运算符 *****：用来取某地址中内容的运算符。（取指针所指对象值的运算符）

例如：

```
int i = 10, y=20, *pi;
```

```
pi = &i; /*将变量i的地址赋给指针变量pi，称pi为指向i的指针变量*/
```

```
y = *pi; /*取pi所指对象的值赋给y，即取pi中所存地址中的内容*/
```



指针和地址（续）

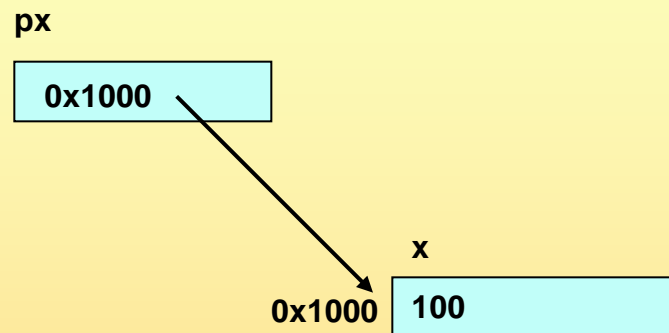
若定义：

```
int x = 100, y;
```

```
int *px;
```

则当：

```
px = &x;
```



则px指向具体对象x，*px则为px所指对象x的值，即100，以后凡是对x的引用，都可用*px来代替，如：

```
y = *px;
```

取px所指对象的值



指针和地址（续）

注意：使用任何指针变量之前必须先给它赋一个所指合法具体对象的地址值。

如下面用法是否有错？

1) `int x = 1;`

`int *px;`

`*px = x;`

错！

2) `char *string;`

`scanf("%s", string);`

`strcpy(string, "Hello");`

错！

上面是在编写C程序时常犯的错误！

指针和地址（续）

■ 如何使一个指针指向一个具体对象：

- 通过&运算符使指针指向某个对象。如：

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

10

- 将另一个同类型的（已指向具体对象的）指针赋给它以获得值，两指针指向同一对象。如，

```
int n=10, *px, *py;
px = &n;
py = px;
```

px

0x010f

py

0x010f

- 使用malloc或alloc等函数给指针分配一个具体空间（动态存储分配）。如：

```
p = (char *)malloc(strlen(s)+1);
```



动态内存管理（malloc与free）

- 在C中可以使用标准库函数malloc动态存储空间（以字节为单位）（用于初始

函数原型为: `void * malloc (size_t`

- 使用malloc初始化指针变量的常见用法

```
char *s;
```

```
int *intptr;
```

```
s = (char *)malloc(32); /* 申请大小  
    空间*/
```

```
s = (char *)malloc(strlen(p)+1); /* s指向能正好存放字符串p  
    的空间*/
```

运算符**sizeof**用来计算所在系统中某种类型或类型变量所占的长度（以字节为单位）。如：

`sizeof(int), sizeof(n), sizeof(double)`

具体值取决于系统，通常int或int变量长度为4，double为8

注意：由malloc等申请的动态空间，必须要用free函数来释放。

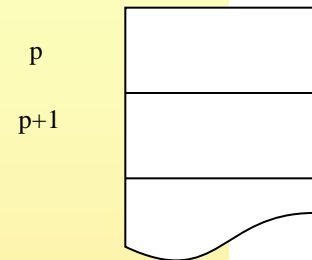
由malloc申请的动态空间不及时释放是造成许多软件出现内存泄漏的主要原因！

内存泄漏(memory leak)：指软件在长时间运行过程中造成内存越来越少，最终可能导致系统内存耗尽而导致软件性能下降或不能使用的现象。

指针运算

1. 指针和整型量可以进行加减。

若 p 为指针，则 $p+n$ 和 $p-n$ 是合法的，同样 $p++$ 也是合法的，它们的结果同指针所指对象类型相关。
如果 p 是指向数组某一元素的指针，则 $p+1$ 及 $p++$ 为数组下一元素的指针。



2. 当 P_1 和 P_2 指向同一类型时，可以进行赋值。

如： $py = px$ ，则 px ， py 指向同一对象。（注意与 $strcpy(py, px)$ 的不同。）

```
int array[N], *p;
for(p=&array[0]; p<=&array[N-1]; p++)
...
```

3. 两个指向同一类型的指针，可进行 $=$ ， $>$ ， $<$ 等关系运算，其实就是地址的比较。

4. 两个指向同一数组成员的指针可进行相减，其结果为两指针间相差元素的个数。

如， p 指向数组第一个元素， q 指向数组最后一个元素，则 $q - p + 1$ 表示数组长度。



指针运算（续）

- 注意：两指针不能相加。

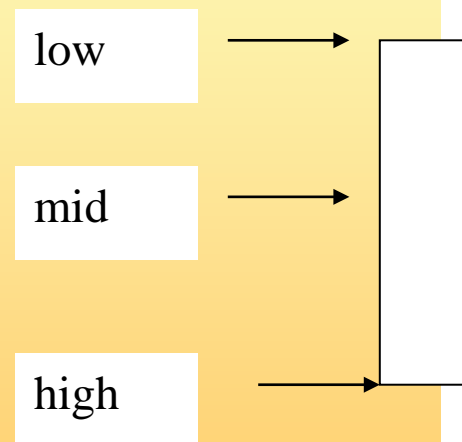
如右图，计算中间指针：

$$\text{mid} = (\text{low} + \text{high}) / 2$$

错！

正确方法是：

$$\text{mid} = \text{low} + (\text{high} - \text{low}) / 2$$



指针运算（续）

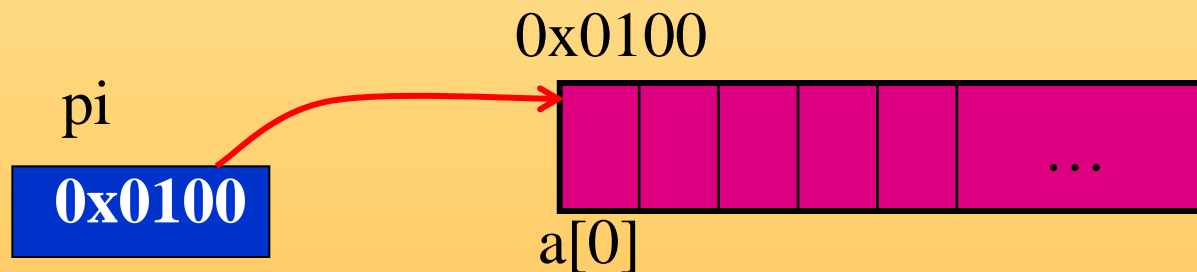
```
int a[5] = {0, 1, 2, 3, 4};
```

```
int *pi;
```

```
pi = &a[0];      /*这时pi指向a[0]*/
```

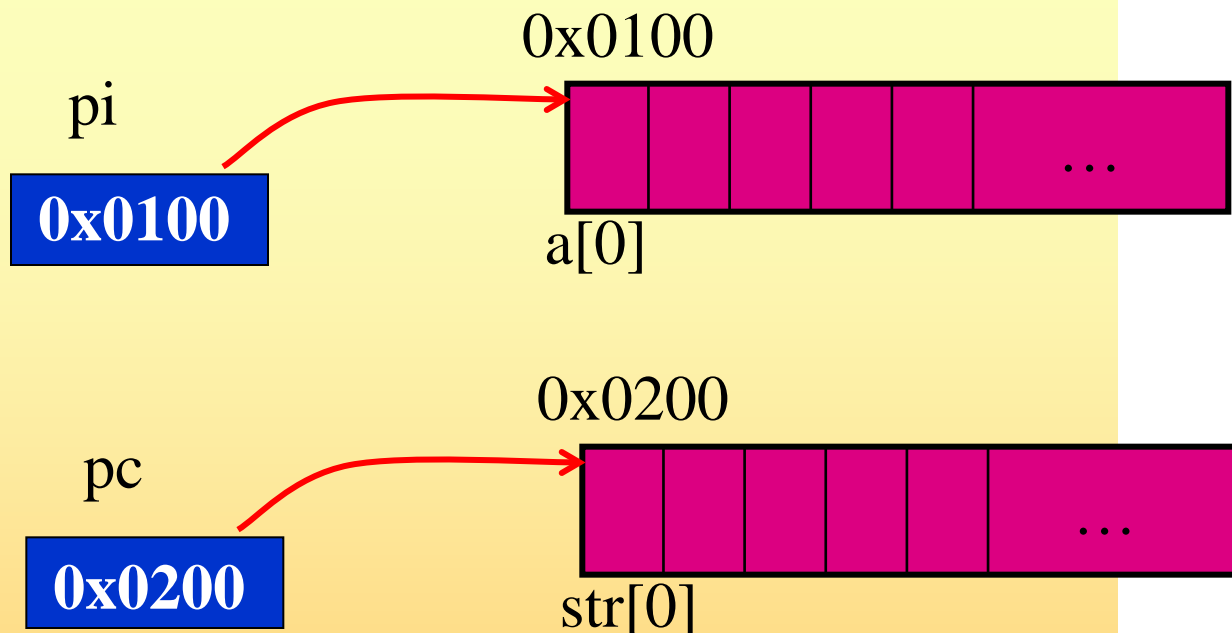
```
pi++;           /*这时pi指向a[1], *pi为a[1], 即1*/
```

```
pi+=2;          /*这时pi指向a[3], *pi为a[3], 即3*/
```



指针运算（续）

```
int a[10];  
int *pi= &a[0];  
char str[10];  
char *pc= &str[0];
```



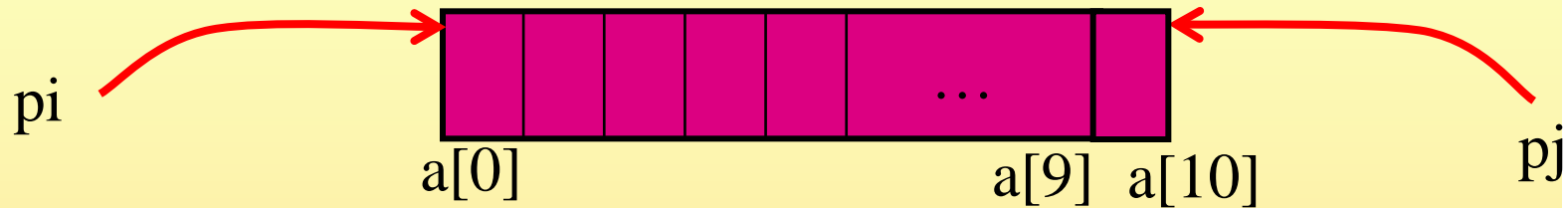
pi++;

/*若int占4个字节, 此时pi增加一个单位, 即四个字节, 结果为0x0104*/

pc++;

/*若char占1个字节, 此时, pc增加一个单位, 即一个字节, 结果为0x0201*/

指针运算（续）



```
for (pi=&a[0],pj=&a[N-1]; pi<=pj; pi++)
```

...

用来遍历一个数组

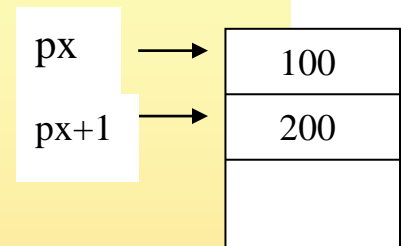
指针运算（续）

■ 指针运算分析：

- $p++$ 和 $p+1$ 的区别；
- $y = *px + 1$ 和 $y = *(px + 1)$ 的区别；
- $y = (*px)++$ 和 $y = *px++$ 的区别；

指针运算（续）

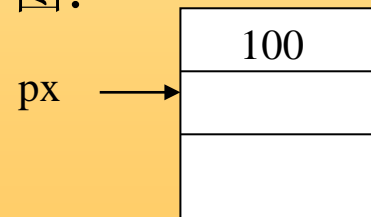
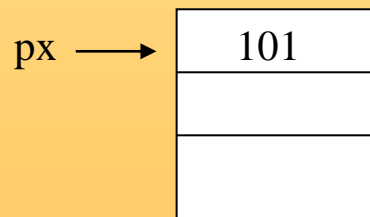
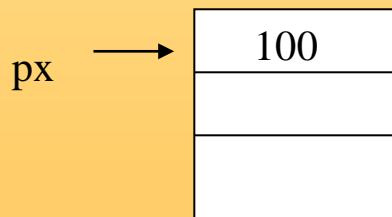
- $p++$ 结果为 p 指向下一元素； $p+1$ 结果为下一元素的指针，但 p 本身不变。
- $*px+1$ 为取 px 所指对象内容加1； $*(px+1)$ 为 px 指针加1，并取结果指针所指对象内容；如右图：



$y = *px+1 = 101$

$y = *(px+1) = 200$

- $(*px)++$ 为先取 px 所指对象内容进行运算，然后对其加1； $*px++$ 为先取 px 所指对象内容进行运算，然后指针 px 加1，其等价于 $*(p++)$ 。如对下图：



$Y = (*px)++ = 100$

$Y = *px++ = 100$



问题5.3

■ 问题：输出输入行中的最长行

■ 数据结构设计

- 设两个一维字符数组来存储新输入行

如何从标准输入中输入一行？
如何判断输入结束？
`while(gets(s)!=NULL)`

...

■ 主算法设计

While(还有新输入行)

If(新行比以前保存的最长行更长，

保存新行及其长度；

输出所保存的最长行；

如何比较两个字符串长度大小？
需要计算字符串长度：
`int str_len(char s[]);`

如何保存一个字符串？需要拷贝一个串至另一个串：
`int str_copy(char s[], char t[]);`

问题5.3： 算法设计

■ 计算字符串（即输入行）长度

- 函数 `str_len(char s[])`

```
i = 0;
```

```
while (s[i] != '\0')
```

```
    i++;
```

■ 保存字符串（即输入行）

- 函数 `str_copy(char s[], char t[])`

```
i = 0;
```

```
while ((s[i] = t[i]) != '\0')
```

```
    i++;
```

问题5.3： 代码实现

```
int str_len(char s[ ])
{
    int i = 0;
    while(s[i] != '\0' ) i++;
    return i;
}

void str_copy(char s[ ], char t[ ])
{
    int i = 0;
    while((s[i] =t[i] )!= '\0' )    i++;
}
```

问题5.3： 代码实现

```
/* c5_2.c */
#include <stdio.h>
#define MAXLINE      1024
int str_len(char s[ ]);
void str_copy(char s[ ], char t[ ]);
int main( )    /* find longest line */
{
    int len;           /* current line length */
    int max;           /* max length seen so far */
    char line[MAXLINE]; /* current input line */
    char save[MAXLINE]; /* longest line saved */
    max = 0;
    while( gets(line) != NULL ){
        len = str_len(line);
        if( len > max ) {
            max = len;
            str_copy(save, line);
        }
    }
    if( max > 0)
        printf("%s", save);
    return 0;
}
```

char* gets(char s[])从标准输入中读入一行到数组s中，但换行符不读入，数组以‘\0’结束。若输入结束或发生错误，则返回NULL



问题5.3：常见问题分析

- 一个错误的str_copy函数实现案例：

```
void str_copy(char s[], char t[])  
{  
    int i = 0;  
    while(t[i] != '\0') {  
        s[i] = t[i];  
        i++;  
    }  
}
```

错误原因：字符串结束符
('\0')没有拷贝到字符串s中。

常用标准字符串库函数

使用**strcpy**、**strcat**函数之前，必须保证**s**有足够的空间容纳操作后的字符串！

■ `#include <string.h>`

```
int strlen(char s[]);    /*计算字符串s长度*/
```

```
char *strcpy(char s[], char t[]); /*将字符串t拷贝到字符串s中*/
```

```
char *strcat(char s[], char t[]); /*将字符串t拷贝到字符串s尾部*/
```

```
int strcmp(char s[], char t[]); /*比较两个字符串, 若s>t, 则返回大于0的数; 若s<t, 则返回小于0的数; 若相等, 返回0 */
```

子曰：工欲善其事，必先利其器。...

问题5.3：指针方式实现

- 可用指针方式实现问题5.2（输出输入行中的最长行）。
- 算法设计：

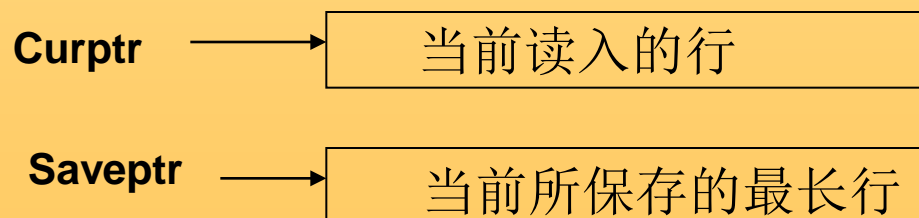
设指针变量Curptr和Saveptr分别指向当前行（新行）和当前最长行

While(还有新输入行)

 If (Curptr所指向的行比Saveptr所指向的行长)

 交换Curptr和Saveptr指针并保存新行长度；

输出Saveptr所指内容



问题5.3： 代码实现（指针方式）（续）

```
#include <stdio.h>
#define MAXLINE      1024
int str_len(char s[]);
int main()          /* find longest line */
{
    int len, max;      /* current line length */
    char *curptr, *saveptr, *tmp; /* current pointer */
    char save1[MAXLINE], save2[MAXLINE];
    curptr = &save1[0];
    saveptr = &save2[0];
    max = 0;
    while( gets(curptr) != NULL ){
        len = str_len(curptr);
        if( len > max ){
            max = len;
            tmp = curptr;
            curptr = saveptr;
            saveptr = tmp;
        }
    }
    if( max > 0 )
        printf("%s", saveptr);
    return 0;
}
```

初始
分别
间

保存

交换
和
指针

```
#include <stdio.h>
#define MAXLINE      1024
int str_len(char s[ ]);
void str_copy(char s[ ], char t[ ]);
int main()          /* find longest line */
{
    int len;          /* current line length */
    int max;          /* maximum length seen so far */
    char line[MAXLINE]; /* current input line */
    char save[MAXLINE]; /* longest line saved */
    max = 0;
    while( gets(line) != NULL ){
        len = str_len(line);
        if( len > max ){
            max = len;
            str_copy(save, line);
        }
    }
    if( max > 0 )
        printf("%s", save);
    return 0;
}
```



问题5.3：代码实现（指针方式）（续）

- 与数组实现方式相比，指针实现方式减少了每当发现新的更长行时所进行的字符数组拷贝（通过调用函数str_copy）。显然指针实现方式代码执行速度要快。

指针作为函数参数

- 在C中函数参数传递方式为“**传值**”。这种方式的最大好处是函数调用不会改变实参变量的值。如何通过函数调用来改变实参变量的值？如通过函数swap(a,b)来交换两个变量的内容。
- 可以通过将指针作为函数参数来改变实参内容。例如：

指针作为函数参数（续）

形参定义为指针

如何通过函数swap交换实参变量a和b？正确的做法应为：

```
void swap ( int *px, int *py)
{
    int temp;
    temp = *px; /*间接取*/
    *px = *py; /*间接取，间接存*/
    *py = temp; /*间接存*/
}

main()
{
    int a =2, b = 3;
    swap ( &a, &b);
}
```

实参传递的是变量的地址

&a

数组和指

在前面介绍函数时，说明不能通过调用下面函数swap(a,b)调用达到交换两个变量的值的目的。

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

```
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```

```
main( )
```

```
{
```

```
    int a=2,b=3;
```

下面用法亦不能达到交换两个变量的值的目的。

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

```
{
```

```
    int *temp;
    temp = px;
    px = py;
    py = temp;
}
```

```
main( )
```

```
{
```

```
    int a=2,b=3;
    swap(&a,&b);
}
```



指针作为函数参数（续）

因此，在一定要改变实参变量内容时，应把函数的形参显式地说明为指向实参变量（类型）的指针，相应地调用时应该用变量的地址值作为参数。

提示：这也是为什么用scanf读int, char, double类型数据时要取变量地址。因为需要改变变量内容。

尽管C的函数参数和函数返回值一般应为基本类型，但它们却可以是指向任何类型（包括复杂的结构类型，甚至其它函数）的指针，这就大大扩充了C的功能和应用范围。



指针和数组

- 在C语言中，数组的名字就是指向该数组第一个元素（下标为0）的指针，即该数组第一个元素的地址，也即数组的首地址。



指针和数组（续）

例如：

```
int a[10], x;
```

```
int *pa;
```

若：

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

则：

```
x = *pa;
```

```
x = a[0];
```

```
x = *a;
```

```
x = *(pa + 1);
```

```
x = a[1];
```

```
x = *(a+1);
```

```
x = *(pa+i);
```

```
x = a[i];
```

```
x = *(a+i);
```

其实 `pa = &a[0]` 可以写成 `pa = a;`

一般有： `a[i] = *(a+i)`

但特别注意：数组名和指针（变量）是有区别的，前者是常量，而后者是变量。因此，尽管我们可写 `pa = a;` 但决不能写： `a = pa` ； `a++`； `pa = &a`； 等。

指针和数组（续）

数组名可作为参数进行传递。当将数组名传给函数时，实际上所传递的是数组的开始地址。（即数组第一个元素的地址）

为什么要使用指针？

- 扩展了语言的功能，如通过传递指针来修改实参变量、或通过返回指针来返回数组等；
- 能够更方便地组织和操作数据，如，离散数据的组织和访问（链表，树等）；

指针和数组（续）

对于字符串常量，可以把它看成一个无名字符数组，C编译程序会自动为它分配一个空间来存放这个常量，字符串常量的值是指向这个无名数组的第一个字符的指针，其类型是字符指针。

所以，`printf("a constant character string\n");` 传递给函数的是字符串第一个字符的指针。

```
char a[]="hello", *p="hello";
```

```
a[0] = 'b';           /*正确*/
```

注意：字符数组和字符指针使用时

```
*p = 'b'; /*错误，不能修改常量值*/
```

例：

```
char *char_ptr, word[20];
```

```
char_ptr = "point to me";
```

正确，把字符串常量第一个字符指针赋给变量。

```
word = "you can 't do this";
```

正确做法为：`strcpy(word, "...");`

指针和数组（续）

假设：int values[100], *intptr = values, i;

表：指针与数组的关系

表达式	值
&values[0] values intptr	指向values数组第一个元素的指针
values[0] *values *intptr	values数组的第一个元素
&values[i] values+i intptr+i	指向values数组第i+1个元素的指针
values[i] *(values+i) *(intptr+i), intptr[i]	values数组的第i+1个元素



指针和数组（续）

例：用指针和数组两种方式实现strlen函数

1) 数组方式

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

2) 指针方式

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

```
main( )
{
    char st[100];
    scanf("%s", st);
    printf("%d\n", strlen(st));
}
或
main( )
```

在函数定义中形参形式char s[]和char *s完全等价，即指向某类型的指针与该类型没有指明长度的数组是同一回事。用哪个取决于在函数里表达式的写法。

建议：由于指针方式可读性明显不如数组方式，而且容易出错，对于初学者来说建议使用数组方式

指针和数组（续）

例：用指针和数组两种方式实现strcpy函数

1) 数组方式

```
void strcpy(char s[ ], char t[ ])  
{  
    int i = 0;  
    while((s[i] = t[i]) != '\0')  
        i++;  
}
```

2) 指针方式

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

或

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

指针和数组（续）*

- 数组就是一个（常量）指针，数组的名就是指向数据第一个元素的指针。

- 数组可以按指针方式使用，如：

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

```
for (i=0; i<n; i++) ...*(a+i)...
```

- 指针亦可按数组形式访问，如：

```
void fun(int p[]) {
```

```
    ...p[i]...
```

- 对于初学者来说，将指针用数组形式访问不容易出错。



指针数组

- 指针数组就是由指针组成的数组，即该数组的每一个元素都是指向某一类型对象的指针。

如：

```
char *lineptr[100];           //由字符指针构成的数组
```

```
int  *iptr[50];               //由整型指针构成的数组
```


指针数组（续）

■ 指针数组与二维数组的区别：

1) 二维数组：

```
char days[7][10] = {
    "Sunday", "Monday", "Tuesday", "Wednesday",
    "Thursday", "Friday", "Saturday"
};
```

存贮形式：

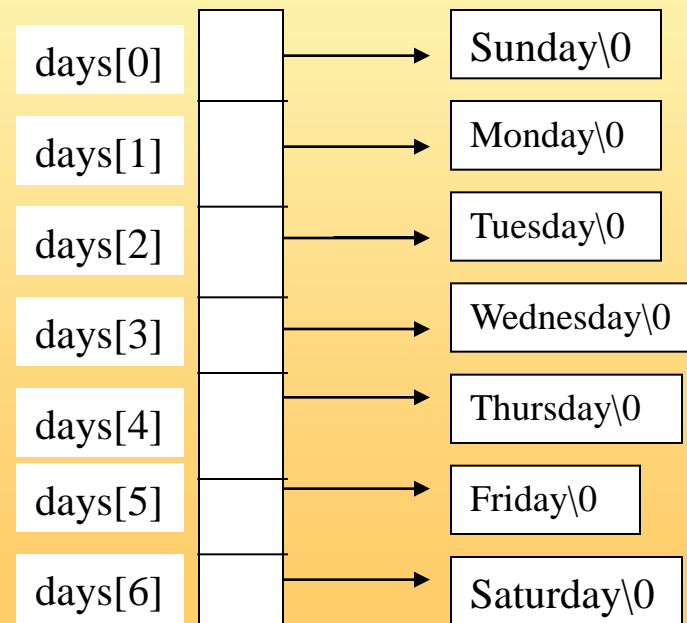
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
days[0]	S	u	n	d	a	y	\0			
days[1]	M	o	n	d	a	y	\0			
days[2]	T	u	e	s	d	a	y	\0		
days[3]	W	e	d	n	e	s	d	a	y	\0
days[4]	T	h	u	r	s	d	a	y	\0	
days[5]	F	r	i	d	a	y	\0			
days[6]	S	a	t	u	r	d	a	y	\0	

指针数组（续）

2) 指针数组

```
char *days[7] = {
    "Sunday", "Monday", "Tuesday", "Wednesday",
    "Thursday", "Friday", "Saturday"
};
```

存贮形式:



char *days[7]



指针数组（续）

- 比较上面两个例子，可以看出尽管二维字符数组与字符指针数组在存储形式上不同，但它们在初始化形式以及访问元素方式上却是相同的。例如，无论是指针数组，还是二维数组，下面两种形式访问的都是同一个元素，结果都是字符串“Friday”中的字符’y’。

$\text{*(days[5]+5)} = \text{days[5][5]} = \text{'y'}$

- 使用指针数组来存放不同长度的字符串可以节省存贮空间，如，存放多个单词串、行。例如，如果要保存从标准输入或文件中读入的行，字符指针数组是一个好的选择。因为读入的行可能长短差异很大。下面程序片段即为保存从标准输入中读入的多行：



指针数组（续）

```
/* read lines from input */
#define MAXLENGTH      512
#define MAXLINES        1000
...
char *lineptr[MAXLINES], buf[MAXLENGTH];
int i;
...
i = 0;
while(gets(buf) != NULL){
    lineptr[i] = (char *)malloc(strlen(buf)+1);
    strcpy(lineptr[i], buf);
    i++;
}
...
```

二维数组指针运算*

二维数组指针运算的理解:

例:

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

```
char a[4][5] = { "abcd", "efgh", "ijkl",
```

```
main()
```

```
{
```

```
    printf("a=%x, a[0]=%x, &a[0][0]=%x\n", a, a[0], &a[0][0]);
```

```
    printf("a+1=%x, a[0]+1=%x, &a[0][1]=%x\n", a+1, a[0]+1, &a[0][1]);
```

```
    printf("**(a+1)=%c, *(a[0]+1)=%c\n", **(a+1), *(a[0]+1));
```

```
}
```

从该例中可以看出， a ， $a[0]$ ， $\&a[0][0]$ 虽然值相同，但含义却不一样， $a+1$ 指向数组的下一行（即组成二维数组的一维数组的下一个元素），而 $a[0]+1$ 指向下一个元素。

一次运行结果:

$a = 0x194$, $a[0] = 0x194$, $\&a[0][0] = 0x194$

$a+1 = 0x199$, $a[0]+1 = 0x195$, $\&a[0][1] = 0x195$

$**(a+1) = 'e'$, $*(a[0]+1) = 'b'$

$a[0]$	$a[0]+1$			
a	b	c	d	\0
e	f	g	h	\0
i	j	k	l	\0
m	n	o	p	\0



指针数组（续）

例：将数字表示的月分转换成英文表示的月分（指针数组的初始化）。

```
char *month_name(int n)
{
    static char *name[] = {
        "illegal month",
        "January",
        "February",
        "March",
        "April",
        "May",
        "June",
        "July",
        "August",
        "September",
        "October",
        "November",
        "December"
    };
    return ((n<1 || n>12) ? name[0] : name[n]);
}
```

指针数组（续）

■ 命令行参数

在C语言中，主函数main还可以，形式如下：

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

或

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

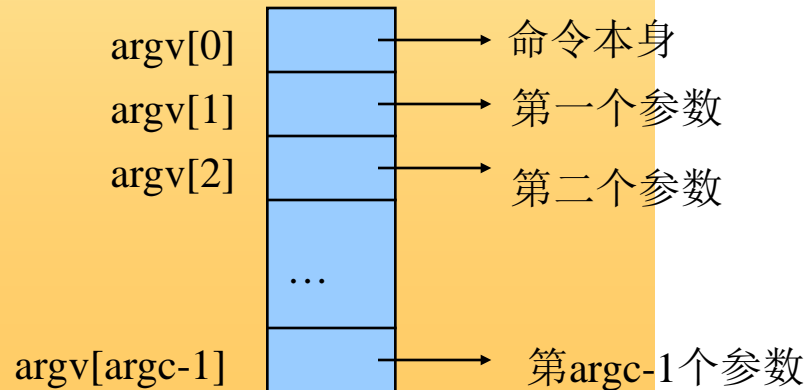
其中：

- argc包含命令本身在内的参数个数
- argv指针数组，数组元素为指向各参数（包含命令本身）的指针。

许多命令在执行时除了提供命令名之外，还要给出一定的参数，如在Windows命令行窗口中执行命令：

```
C>copy file1 file2
```

在此，file1和file2被称为**命令行参数**。在实际应用时，经常会需要编写带命令行参数的程序。





问题5.4

- 问题：实现一个命令echo，其将命令后的正文串显示在屏幕上，如：

```
C> echo  hello world
```

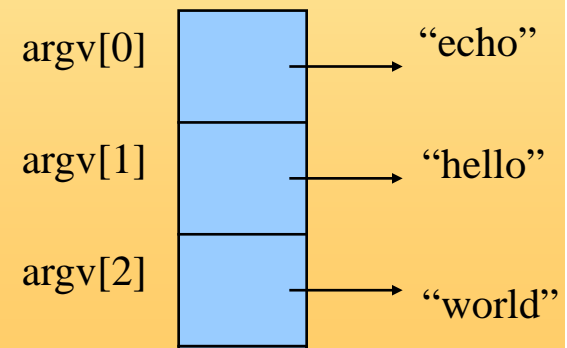
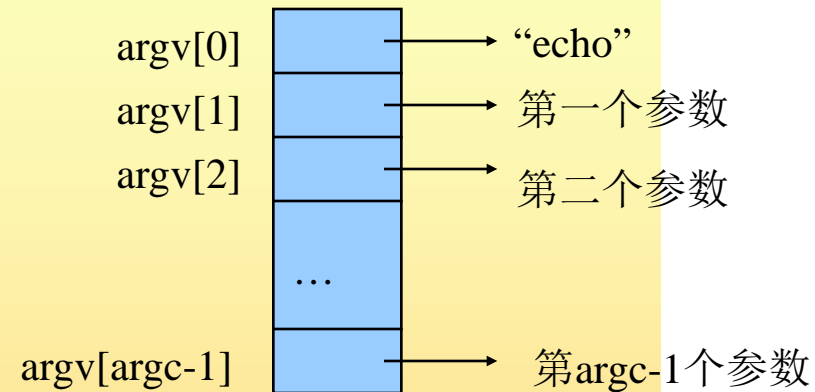
屏幕输出：

```
hello world
```


问题5.4： 算法分析

- 从右图可知，使用下面循环就可输出所有命令行参数：

```
for(i=1; i<argc; i++)  
    printf( "%s  ", argv[i]);
```



C> echo hello world

问题5.4：代码实现

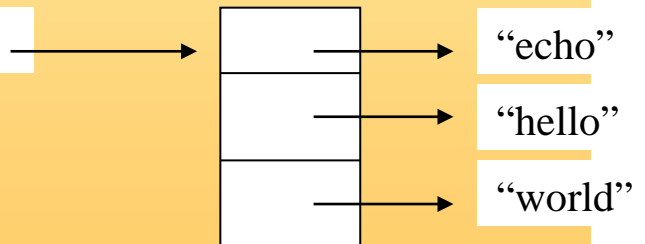
■ 实现一：

```
main( int argc, char *argv[ ])  
{  
    int i;  
    for(i=1; i<argc; i++)  
        printf("%s%c", argv[i], (i< argc-1)? ' ': '\n');  
}
```

■ 实现二：

```
main(int argc, char *argv[ ])  
{  
    while(--argc > 0)  
        printf((argc > 1)? "%s " : "%s\n", *++argv);  
}
```

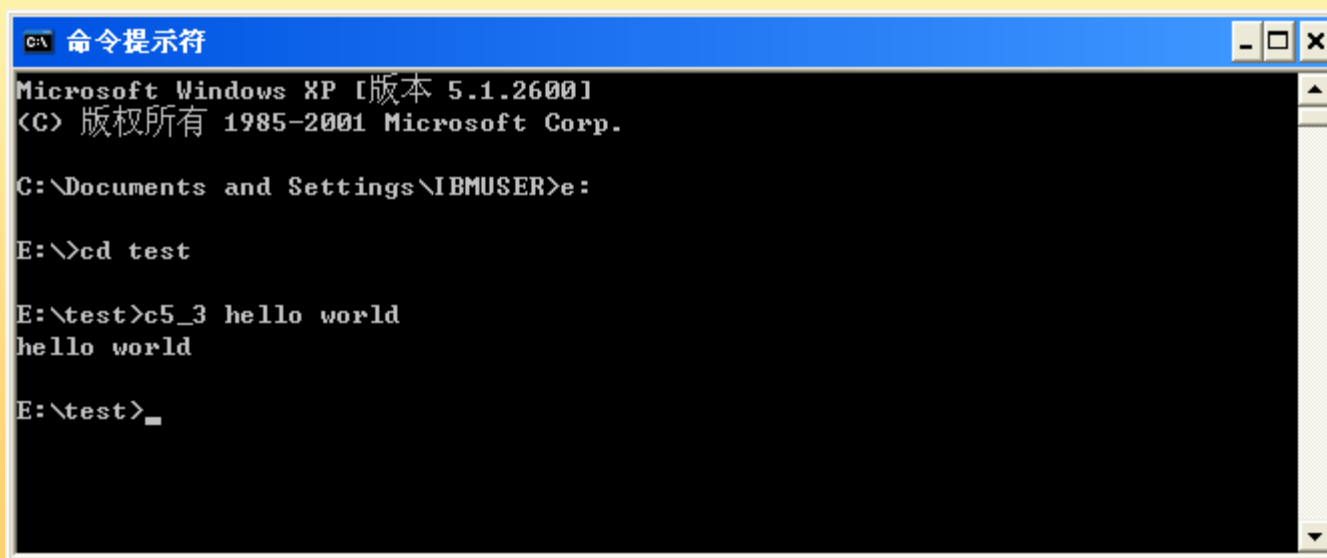
argv



问题5.4：如何运行命令行程序

■ 方式一：在命令窗口（DOS窗口）中直接运行；

若c5_4.exe执行文件在\test目录下，则从<开始>菜单中<附件>中找到<命令提示符>，并执行。然后，转到test目录下执行c5_4.exe文件。



```
C:\ 命令提示符
Microsoft Windows XP [版本 5.1.2600]
(C) 版权所有 1985-2001 Microsoft Corp.

C:\Documents and Settings\IBMUSER>e:

E:\>cd test

E:\test>c5_3 hello world
hello world

E:\test>_
```

问题5.4：如何运行命令行程序（续）

- 在VC环境下运行





函数指针*

- 函数指针

即指向函数的指针。

函数指针说明形式为：

*类型 (*标识符) () ;*

例：int (*fp) () ; 注意：与int *fp () ; 的不同

对函数指针赋值，可通过赋值语句或参数传递。

函数指针 = 函数名;

(在C语言中，函数名是作为该函数的指针来处理。)



函数指针*（续）

例：

```
int leapyear( int year);  
main( )  
{  
    int (*fnptr)();  
    fnptr = leapyear;  
    result = (*fnptr)(2000);    /* 与调用leapyear(2000)完全等价*/  
}  
  
int leapyear( int year)  
{  
    if(((year % 4 == 0) && (year % 100 != 0)) || (year % 400) == 0)  
        return ( 1);  
    else  
        return ( 0);  
}
```

函数指针*（续）

函数指针的作用：

■ 函数派遣表

指向函数的指针（函数指针）通常用于设计所谓函数派遣表（dispatch），即通过下标数字来访问某个函数，这常见于菜单系统的设计中。如，index变量为0~9，当index为0时，调用函数fn0；为1时调用函数fn1...，则可设计如下：

```
int fn0(), fn1(), fn2(), ... fn9();
int (*dispatch[ ])( ) = {
    fn0, fn1, fn2, ... fn9
};
```

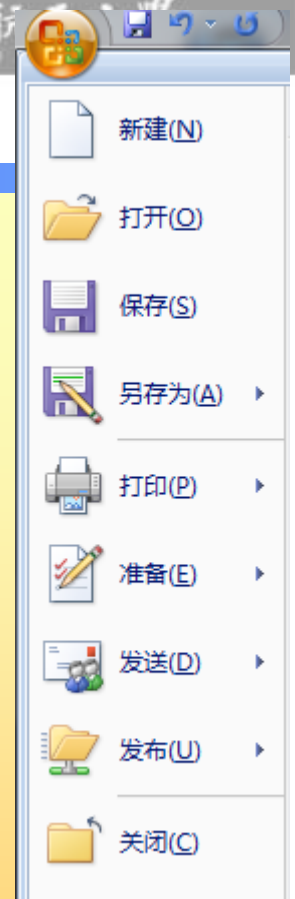
调用形式为：

```
(*dispatch[index])( );
```

■ 作为函数参数

函数作为参数传递，可扩展一个函数的功能。

```
int (*menu[ ])( ) = {
    New, Open, Save, ... Close };
```





典型错误案例分析

```
int main()
{
    char *string, c;
    scanf ("%s\n", string);
    scanf ("%c", &c);
    insert (*string, c);
    return 0;
}
```




典型错误案例分析（续）

```
#include <stdio.h>
char *insert(char *string,char c)
{
    int i;
    char s[50];
    for(i=0;*string<c;i++)
        s[i]=*string++;
    s[i++]=c;
    for(;*string!='\0';i++)
        s[i]=*string++;
    return s;
}

void main()
{
    char s1[50],c;
    scanf("%s\n",s1);
    scanf("%c",&c);
    printf("%s",insert(s1,c));
}
```

定义了一个局部数组

该循环没有复制
'\0'。字符数组s没有'\0'

不能返回一个局部数组。因为它的生存期为当前函数。

修改正确后:

```
char *insert(char *string,char c)
{
    int i;
    char *s;
    s = (char *)malloc(50);
    for(i=0;*string<c;i++)
        s[i]=*string++;
    s[i++]=c;
    for(;(s[i]=*string++)!='\0';i++)
        ;
    return s;
}
```



问题5.5

- 问题：编写一个程序，统计输入中C语言每个关键字的出现次数。

问题5.5：问题分析

- 按目前掌握的知识，需要设两个数组分别存储C关键字列表和相应出现次数：

```
char *keywordlist[N];    /*存储关键字*/
```

```
int count[N];            /* 存储关键字出现次数 */
```

其中N为C中关键字的数目

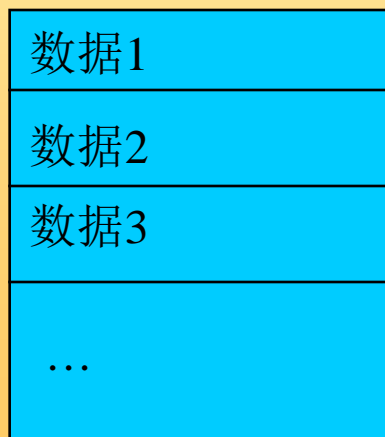
- 这样处理相互关联的数据（如某一关键字和其出现次数是相关的）的问题：
 - 数据的处理不方便，如插入/删除一个关键字及出现次数要分别操作两个数组。
- 如何组织不同类型的相关数据？

结构

结构说明

■ 结构（struct）的表示和含义

结构是由若干分量组成的一种构造类型。然而，组成结构的各个分量可以具有不同的类型（这和数组情况相异），并且，对结构变量的访问必须通过它的分量名字（亦称为成员名），而不像数组是通过下标来访问它的成员。





结构说明（续）

说明形式：

```
struct 结构类型名 {  
    成员类型 成员名;  
    成员类型 成员名;  
    ...  
    ...  
};
```



结构说明（续）

如下面为用以表示日期相关信息的结构说明：

```
struct date {  
    int day;  
    int month;  
    int year;  
    int yearday;  
    char mon_name[4];  
};
```

结构名

结构成员

上面的结构说明，只是定义了一个结构的模板（template）或称为结构的框架，而并未定义结构的对象，也不为它分配存储空间。有了这样的结构模板说明后，一个结构变量可定义为：

```
struct date d1, d2;
```

注意：在此，关键字struct和结构名date都不可少，可以把struct date一起看作是某种类型说明符。

结构变量说明（续）

在说明结构模板时，一般都要有结构名，但也可不用结构名，直接把结构模板和变量定义（或说明）放在一起。下面是结构变量的几种说明方式：

1) 无结构名

```
struct {  
    ...  
} d;
```

一般适用于说明本地变量。

2) 有结构名

```
struct date {  
    ...  
} d;
```

通常用来说明外部结构变量，或需要在多个函数中用到的相同的结构的变量。

3) 使用typedef

```
typedef struct {  
    ...  
} DATE;
```

则变量定义为：DATE d, *pd, ad[10];

使用typedef定义结构类型名后，结构变量的定义（或说明）就更简洁了。

类型定义



类型定义 (typedef)

类型定义的语法格式为:

typedef 原类型名 新类型名

如:

```
typedef int LENGTH;  
typedef char *STRING;
```

变量说明为:

```
LENGTH len, maxlen;  
STRING lineptr[LINES], alloc( );
```

这与如下直接说明等价:

```
int len, maxlen;  
char *lineptr[LINES], *alloc( );
```

typedef常见用法:

1. 一些安全关键的软件中需要在程序中明确运行环境的数据类型长度, 如:
 typedef int INT32;
 typedef short INT16
 INT32 port0, port1;
 ...
2. 用来定义结构类型, 如FILE就是一个用typedef定义的结构类型。

类型定义（续）

必须强调，typedef说明均不产生新的数据类型，也不定义存储单元，它只是给已有的类型又增添了新的类型名，没有产生新的语义，即用这种方法所说明的变量与明确指出说明的那些变量有相同的性质。

类型定义的必要性：

- 将程序参数化，便于移植；
- 为程序提供较好的说明信息，便于理解；

类型定义的一个常见用法是用来定义结构类型，如常用的文件类型FILE，就是结构类型定义。

结构说明（续）：结构嵌套

- 结构成员可以具有各种类型，当然它也可以是其它的结构类型，即结构可以嵌套定义。如下面为描述个人信息（姓名、住址、单位、薪水、生日）的结构说明：

```
struct person {
    int ID;
    char name[32];
    char address[64];
    char department[64];
    double salary;
```

```
    struct date {
        int year;
        int month;
        int day;
    };
};
```

```
struct person
```

ID	姓名	单位	住址	工资	出生年月		
					年	月	日
1	张三	北航计算机学院	北航家属楼xxx	3000	1980	5	18
...

结构说明（续）

- 注意：结构成员的类型不能是该结构本身，因为它无法确定此结构的边界。但它可以是指向本身结构的指针。例如：

```
struct keyword {  
    char *name;  
    int count;  
    struct keyword next;  
}  
*base;
```

错误

```
struct keyword {  
    char *name;  
    int count;  
    struct keyword *next;  
}  
*base;
```

正确

结构变量初始化

- 结构变量在定义时可以初始化，如：

```
struct date d = { 27, 12, 1984, 361,  
    "Dec" };
```

- 结构变量亦可通过整体赋值来初始化，如：

```
struct date d1, d2 = { 27, 12, 1984, 361,  
    "Dec" };  
  
d1 = d2;
```



结构成员的引用

通过

结构变量名.成员名

来访问结构成员，如：

若定义了结构变量：`struct person emp;`

则给emp的出生年、月赋值可写成：

`emp.birthdate.year = 1970;`

`emp.birthdate.month = 8;`

如定义：`struct date *pd, d;`（pd是指向struct date的指针）

则：`pd = &d` （指向一个已有结构变量）

或：`pd = (struct date *) malloc(sizeof(struct date));` （指向一个新的结构空间）

注意：与其它指针变量一样，定义了pd并不表示pd有了它所指对象。

运算符`->`专门用于存取指向结构的指针所指对象成员，如：

`pd->year = 1958;`

`strcpy(pd->mon_name, "OCT");` 等等。

实际上，`pd->year`与`(*pd).year`是完全等价的，这是由于在C语言中指向结构的指针使用非常频繁，因此，特为此设立了一个新运算符（`->`）。

构造类型 – 数组和指针

结构成员的引用（续）

- 结构变量可进行如下操作：
 - 访问结构成员，如： `emp.name`
 - 作为一个整体复制、赋值，如， `e1 = emp;`
 - 取地址运算（&），如： `pd = &d;`



结构成员的引用（续）

例：复数（加）运算

```
#include <stdio.h>

struct complex {
    float  real;
    float  imag;
};

struct complex addComplex(struct complex c1, struct complex c2);

int main()
{
    struct complex c1, c2, c3;
    scanf( "%f %f %f %f" , &c1.real, &c1.imag, &c2.real,
    &c2.imag);
    c3 = addComplex(c1, c2);
    printf( "(%.2f, %.2f) + (%.2f, %.2f) = (%.2f, %.2f)\n" ,
    c1.real, c1.imag, c2.real, c2.imag, c3.real, c3.imag);
    return 0;
}
```

结构成员的引用（续）

```
struct complex addComplex(struct complex c1, struct complex c2)
{
    struct complex tmp;
    tmp.real = c1.real + c2.real;
    tmp.imag = c1.imag + c2.imag;
    return tmp;
}
```


结构数组

- 当数组中的每一个元素都是同一结构类型的变量时，则称该数组为**结构数组**。例如：

```
struct person table[100];
```

ID	姓名	单位	住址	工资	出生年月		
					年	月	日
1	张三	北航计算机学院	北航家属楼xxx	3000	1980	5	18
...



问题5.5：问题分析

- 问题：编写一个程序，统计输入中C语言每个关键字的出现次数。
- 定义一个结构说明用以表示关键字与其出现次数：

```
struct Key {  
    char *keyword;  
    int count;  
};
```

将关键字有序存放能提高关键字的查找效率。

- 关键字表的组织：使用一个**有序**的结构数组来存放关键字表及关键字出现次数：

```
struct Key Keytab[ ] = {  
    "auto", 0,  
    "break", 0,  
    "case", 0,  
    ...  
    "while", 0  
};
```

问题5.5： 算法设计

■ 主要算法：

While (仍有新单词读入)

 If(在关键字表中查找并找到输入的单词)

 相应关键字次数加1;

输出关键字及出现次数;

设函数

```
char getWord(char word[],int lim)
```

从标准输入中读入一个长度不超过lim-1的单词，并返回单词类型。

为何不用scanf的%s来读？

如：

```
for(i=0; i<n; i++)...
```

设函数

```
struct Key *binary(char *word,  
struct Key tab[ ], int n)
```

在关键字表tab中查找单词word是否存在。如果找到，则返回其出现位置。n为关键字表的长度（关键字个数）。

设函数

```
void printKey(struct Key tab[ ], int n)
```

输出关键字及出现次数。



顺序查找算法

- 在有序数据集中查找指定元素的最简单方法是顺序查找, 即指定数据依次与数据集中数据比较, 直到找到或查到数据集结束。



折半查找算法 (binary search)

- 在有序数据集中查找指定数据项最常用及最快的算法是折半查找算法。
 - 假设数据集按由小到大排列，折半查找算法的核心思想是：
 1. 将要查找的有序数据集的中间元素与指定数据项相比较；
 2. 如果指定数据项小于该中间元素，则将数据集的前半部分指定为要查找的数据集，然后转步骤1；
 3. 如果指定数据项大于该中间元素，则将数据集的后半部分指定为要查找的数据集，然后转步骤1；
 4. 如果指定数据项等于中间元素，则查找成功结束。
 5. 最后如果数据集中没有元素再可进行查找，则查找失败。
- 下面以在一个有序整型数据集中查找给定整数为例来说明折半查找。

折半查找算法（续）

例：在下面有序数据集中查找数据项62

0	1	2	3	4	5	6	7	8	9
5	7	16	24	25	50	45	50	62	65

↑ ← 查找范围 → ↑
low $item > data[mid]$, 即 $62 > 25$ high

0	1	2	3	4	5	6	7	8	9
5	7	16	24	25	50	45	50	62	65

↑ ← 查找范围 → ↑
low $item > data[mid]$, 即 $62 > 50$ high

0	1	2	3	4	5	6	7	8	9
5	7	16	24	25	50	45	50	62	65

↑ ← 查找范围 → ↑
 $item = data[mid]$, 即 $62 = 62$ low high

item = 62(查找项)

low = 0(查找范围开始)

high = 9(查找范围结束)

mid = $(low+high)/2=4$ (查找范围中间)

low = mid+1=5

high = 9

mid = $(low+high)/2=7$

low = mid+1=8

high = 9

mid = $(low+high)/2=8$



折半查找算法（续）

在有序整数数组中查找给定元素的折半查找算法如下:

```
int bsearch(int item, int array[ ], int len)
{
    int low=0, high=len-1, mid;
    while(low <= high) {
        mid = (high + low) / 2;
        if(( item < array[mid])
            high = mid - 1;
        else if ( item > array[mid])
            low = mid + 1;
        else
            return (mid);
    }
    return -1;
}
```



问题5.5：代码实现

```
/*c5_4.c
#include <stdio.h>
struct Key {
    char *keyword;
    int  keycount;
} Keytab[] = {
    "auto", 0,
    "break", 0,
    "case", 0,
    ...
    "while", 0
};
#define MAXWORD 20
#define NKEYS (sizeof(Keytab) / sizeof(struct Key))
#define LETTER 'a'
#define DIGIT '0'
struct Key *binary(char *word, struct Key tab[], int n);
char getword(char *w, int lim);
char type(int c);
void printKey(struct Key tab[], int n);
```

关键字表长度（即关键字个数）



问题5.5：代码实现（续）

```
int main()          /* count C keyword */
{
    int t;
    char word[MAXWORD];
    struct Key *p;

    while((t = getword(word, MAXWORD)) != EOF)
        if(t == LETTER)
            if((p = binary(word, Keytab, NKEYS)) != NULL)
                p->keycount++;
    printKey(keytab, NKEYS);
    return 0;
}
```



问题5.5：代码实现（续）

```
struct Key *binary(char *word, struct Key tab[], int n)
{
    int cond;
    struct Key *low = &tab[0];
    struct Key *high = &tab[n-1];
    struct Key *mid;

    while(low <= high) {
        mid = low + (high - low) / 2;
        if((cond = strcmp(word, mid->keyword)) < 0)
            high = mid - 1;
        else if (cond > 0)
            low = mid + 1;
        else
            return (mid);
    }
    return (NULL);
}
```

折半查找

注意：基于指针运算的折半查找算法在计算中间元素位置时与前面的不同。



问题5.5：代码实现（续）

```
char getword(char *w, int lim)
{
    int c, t;

    if(type(c = *w++ = getchar( ))
        *w = '\0';
        return (c);
    }

    while(--lim > 0) {
        t = type(c = *w++ = getchar());
        if(t != LETTER && t != DIGIT){
            ungetc(c);
            break;
        }
    }

    *(w-1) = '\0' ;
    return ( LETTER);
}
```

如何在一行中识别出标识符？

```
...
j = 0;
for(i=0; line[i]!='\0'; i++) {
    if(type(line[i]) != LETTER)
        continue;
    s[j++] = line[i++];
    while(line[i]!='\0'&&(type(line[i])==LETTER
    || type(line[i])==DIGTH))
        s[j++] = line[i++];
    s[j]='\0';
    ...
}
...
```



问题5.5：代码实现（续）

```
char type(int c)      /* return type of ASCII character */
{
    if( c >= 'a' && c <= 'z' || c >= 'A' && c <= 'Z' )
        return ( LETTER );
    else if ( c >= '0' && c <= '9' )
        return ( DIGIT );
    else return (c);
}

void printKey(struct Key tab[], int n)
{
    struct Key *p;
    for(p=Keytab, p < Keytab+n; p++)
        if(p->keycount > 0)
            printf("%4d%s\n", p->keycount, p->keyword);
}
```



自引用结构

■ 问题1：谁是幸存者？

海盗们曾经玩一种游戏：每当捕获一艘船，船上船员只有一人能幸存。规则如下：船上船员分别编上号，站成一圈；然后从1号船员开始循环数数，每数到13，该船员将被推到海里，直到剩下一个船员。谁是最后的幸存者？

■ 问题2：显示文件最后N行。

■ 问题3：实现任意两个的多项式相加

由于打印前我们无法知道文件的总行数，如何在程序中组织这类数据？



自引用结构（续）

■ （大）数据的组织与存储方式

- 顺序存放，如数组
 - 需要连续空间
 - 数据项的插入或删除操作需要移动大量数据
- 非顺序（链式）存放，如链表，二叉树等
 - 不需要连续空间
 - 数据项的插入或删除操作非常简单

如何构造？

自引用结构

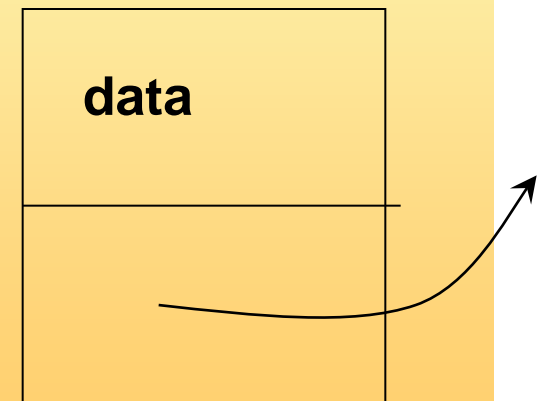
优点：

- 1) 动态管理（增减数据）
- 2) 不需要连续空间

自引用结构（续）

- 自引用结构其成员分为两部分：
 1. 各种实际数据成员；
 2. 一个或几个指向自身结构的指针；

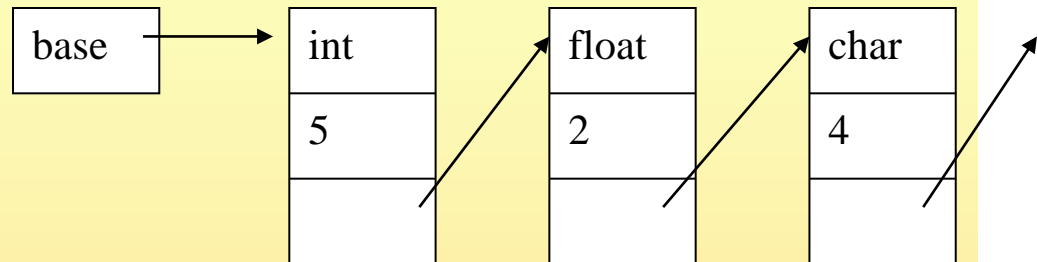
```
struct Type {  
    data_member;    // 如 int n;  
    struct Type *link;  
};
```



自引用结构（续）

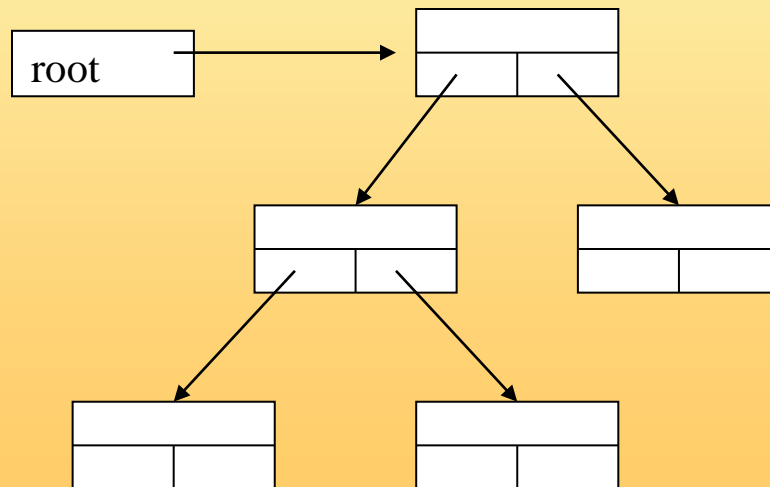
■ 链表结构

```
struct word {
    char *name;
    int count;
    struct word *next;
} *base;
```



■ 二叉树结构

```
struct tnode {
    char *word;
    int count;
    struct tnode *left;
    struct tnode *right;
} *root;
```



自引用结构（续）

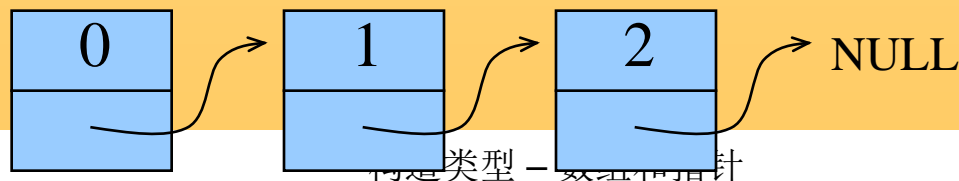
下面通过示例说明如何构造、使用一个链表：

链表结构：

```
struct link {
    int n;
    struct link *next;
};
```

1) 创建链表：

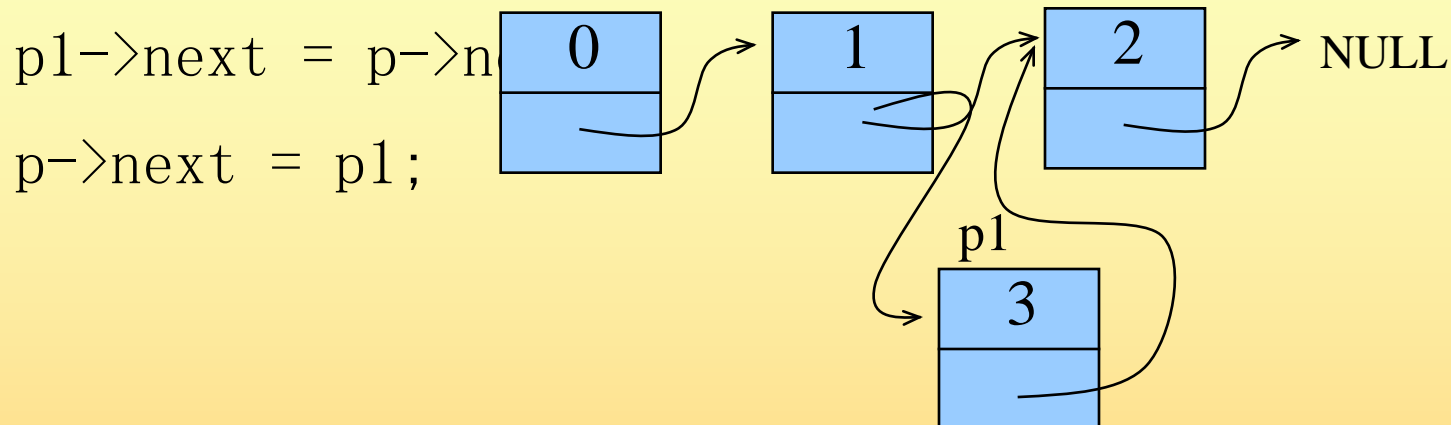
```
struct link *first, *p;
p = first = (struct link *)malloc(sizeof(struct link));
p->n = 0;
p->next = NULL;
for(i=1; i< 10; i++) {
    p->next = (struct link *)malloc(sizeof(struct link));
    p = p->next;
    p->n = i;
    p->next = NULL;
}
```



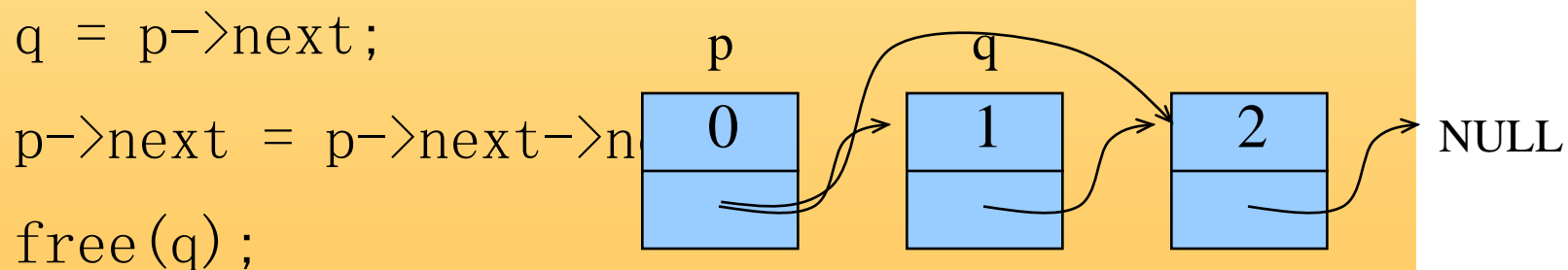
构造类型 - 数组和指针

自引用结构（续）

2) 插入一个节点:



3) 删除一个节点:



问题5.7

- 问题：命令tail用来显示一个文件的最后n行。其格式为：

```
tail [-n] filename
```

其中：

-n : n表示需要显示的行数，省略时n的值为10。

filename : 给定文件名。

如，命令tail -20 example.txt 表示显示文件example.txt的最后20行。实现该程序，该程序应具有一定的错误处理能力，如能处理非法命令参数和非法文件名。

问题5.7： 问题分析

若从命令行输入：

tail -20 test.txt

以显示文件test.txt的最后20行。

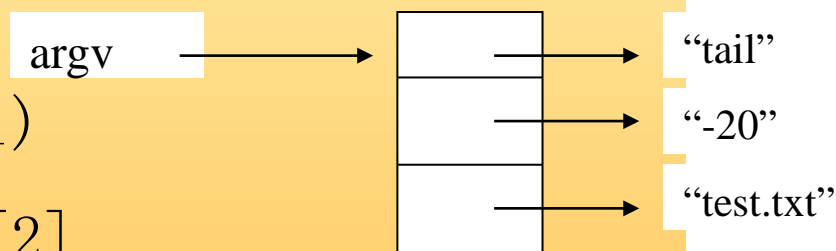
■ 如何得到需要显示的行数和文件名？

- 使用命令行参数

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

- 行数 $n = \text{atoi}(\text{argv}[1] + 1)$

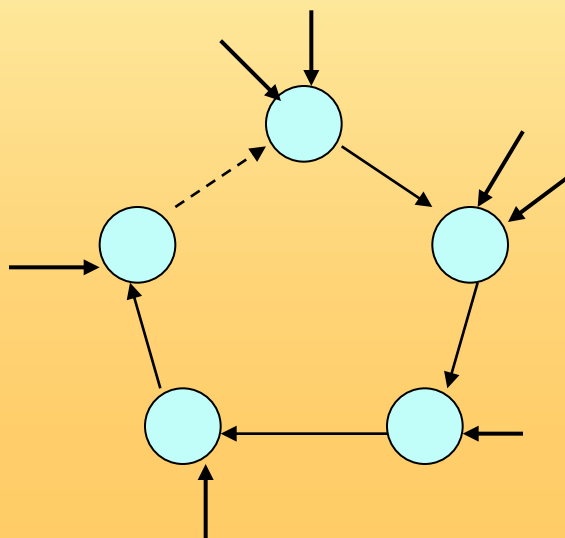
- 文件名 $\text{filename} = \text{argv}[2]$



■ 如何得到最后n行？

问题5.7： 算法设计

- 方法一： 使用 n 个节点的循环链表。链表中始终存放最近读入的 n 行。
 1. 首先创建一个空的循环链表；
 2. 然后再依次读入文件的每一行挂在链表上，最后链表上即为最后 n 行。





问题5.7： 算法设计（续）

- 方法二： 使用一个n个元素的指针数组。
 - 依次读入每一行，然后循环挂到指针数组上。

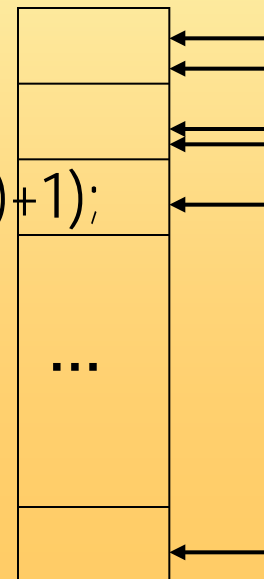
```
char *lineptr[N]; /*存入所读入的行*/
```

```
char line[LEN]; /*当前读入行*/
```

```
int i; /*读入的行数*/
```

```
lineptr[i % n] = (char *)malloc(strlen(line)+1);
```

```
strcpy(lineptr[i%n], line);
```





问题5.7：算法设计（续）

■ 方法三：两次扫描文件。

- 第一遍扫描文件，用于统计文件的总行数 N ；
- 第二遍扫描文件时，首先跳过前面 $N-n$ 行，只读取最后 n 行。

■ 如何开始第二遍扫描？

`fseek(fp, 0, SEEK_SET);` -- 将文件读写位置移至文件头

或（关闭后）再打开同一个文件

请同学们考虑是否还有其它方法？甚至更好的方法？



问题5.7： 代码实现（循环链表）

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define DEFLINES 10
#define MAXLEN 81
struct Node {
    char *line;
    struct Node *next;
};
```

```
int main(int argc, char *argv[ ])
{
    char curline[MAXLEN], *filename;
    int n = DEFLINES, i;
    struct Node *first, *ptr;
    FILE *fp;
    if( argc == 3 && argv[1][0] == '-' ) {
        n = atoi(argv[1]+1);
        filename = argv[2];
    }
    else if( argc == 2 )
        filename = argv[1];
    else {
        printf("Usage: tail [-n] filename\n");
        return (1);
    }
}
```

命令行输入中指定打印行数时，获取行数及文件名。如 **tail -20 test.txt**

命令行输入中没有指定打印行数时，获取文件名，此时行数为缺省**10**。如 **tail test.txt**



问题5.7： 代码实现

```
if((fp = fopen(filename, "r")) == NULL){  
    printf(" Can't open file: %s !\n", filename);  
    return (-1);  
}  
  
first = ptr = (struct Node *)malloc(sizeof ( struct Node));  
first->line = NULL;  
for(i=1; i<n; i++){  
    ptr->next = (struct Node *)malloc(sizeof ( struct Node));  
    ptr = ptr->next;  
    ptr->line = NULL;  
}  
ptr->next = first;  
ptr = first;
```

创建循环链表

将链表的最后一个节点指向头节点，以构成一个循环链表。



问题5.7： 代码实现

```
while(fgets(curline, MAXLEN, fp) != NULL){
    if(ptr->line != NULL)
        free(ptr->line);
    ptr->line = (char *) malloc ( strlen(curline)+1);
    strcpy(ptr->line, curline);
    ptr = ptr->next;
}
for(i=0; i<n; i++) {
    if(ptr->line != NULL)
        printf("%s", ptr->line);
    ptr = ptr->next;
}
fclose(fp);
return 0;
}
```

测试考虑点：

准备一个包含内容（如11~20行）
的正文文件test.txt

- 1) tail -5 test.txt (正常)
- 2) tail test.txt (正常)
- 3) tail -30 test.txt (非正常)
- 4) tail -0 test.txt (非正常)
- 5) tail -1 test.txt (边界)

自引用结构使用总结*

- 自引用结构是实现常用数据结构，如链表、树等的主要手段。

- 自引用结构有如下特点：

- 定义结构时，应包含一个或多个指向自身结构的指针，如：

```
struct node {  
    普通成员;  
    struct node *next;  
};
```

单向或循环链表

```
struct node {  
    普通成员;  
    struct node *left,*right;  
};
```

双向链表、二叉树

- 使用时应有一个头指针（first, head, root），并使用malloc为每个节点分配空间，如：

```
first = (struct node)malloc(sizeof(struct node));
```

- 使用时，通常使用如下方式将节点链接起来：

```
p->next = (struct node)malloc(sizeof(struct node));
```

```
p = p->next;
```

构造类型 – 数组和指针

- 使用时要注意节点丢失问题，不要轻易移动头节点位置。

联合 (union) *

- 联合是一种数据类型，该类型变量可以在不同时间内维持定义它的不同类型和不同长度的对象，也就是说提供单独的变量，以便合理地保存几种类型中的任何一类变量。

定义形式：

union 联合名 {分量表} 联合变量名；



联合 (union) (续) *

例:

```
union v_tag {  
    int ival;  
    float fval;  
    char *pval;  
} uval;
```

下面是联合的使用:

```
if(utype == INT)  
    printf("%d\n", uval.ival);  
else if (utype == FLOAT)  
    printf("%f\n", uval.fval);  
else if(utype == STRING)  
    printf("%s\n", uval.pval);  
else  
    printf("bad type\n");
```

[注意]: 使用联合, 用法必须一致, 即取出的类型必须是最近存入的类型。因此, 在使用联合时, 要记住当前存于联合中的类型是什么, 不允许存入是一种类型, 而取出是另一种类型。



联合（union）（续）*

联合可以出现在结构和数组中，数组和结构也可以出现在联合中，下例是在编译中常见到的符号表：

例：

```
struct {  
    char *name;  
    int flags;  
    int utype;  
    union {  
        int ival;  
        float fval;  
        char *pval;  
    } uval;  
} symtab[NSYM];
```

使用：symtab[i].uval.ival

联合（union）（续）*

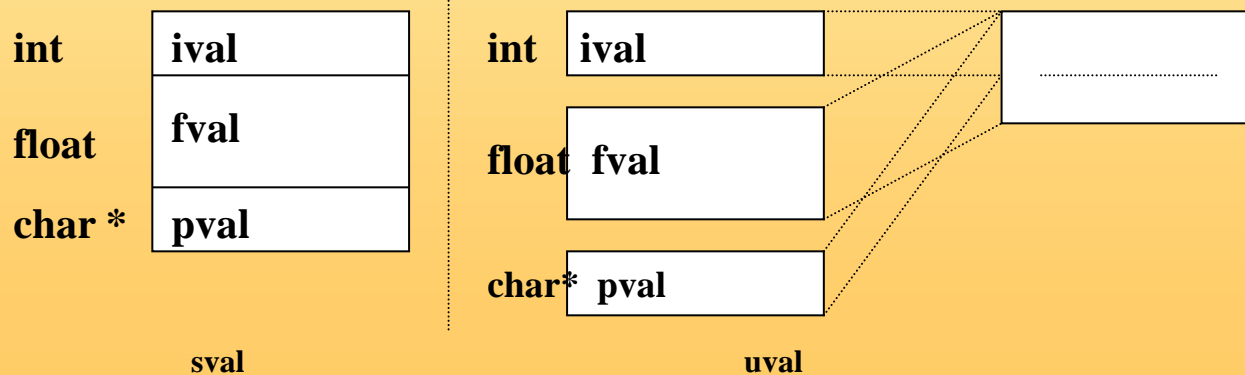
■ 结构与联合的异同:

假设我们定义如下具有相同内容的结构和联合:

```
struct v_tag {
    int ival;
    float fval;
    char *pval;
} sval;
```

```
union v_tag {
    int ival;
    float fval;
    char *pval;
} uval;
```

则struct v_tag和union v_tag的物理存贮形式可参见下图:





联合（union）（续）*

■ 结构与联合的异同：

- 在定义或说明形式上，union和struct很类似，但在任何时刻联合只允许联合中说明的某一成员留在联合里。
- 结构由多个成员（分量）所组成，而联合只有一个成员，只不过该成员的名字和类型可以在规定的几个里选定一个。
- 因此，联合可以看作是一个特殊的结构，其所有成员在结构中的位移量都是0，当对联合变量分配存贮空间时，应保证它能容纳最大的一个成员的大小，而存贮空间的边界应能适于联合中的所有可选成员的类型。（见上图）
- 对联合的初始化只能对应于它的第一个可选成员。
- 对联合成员的引用方式完全和结构的情况一样。

本讲结束！