



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

计算机程序设计

第五章 系统级编程初探

白雪飞

中国科学技术大学微电子学院

目录

- 引言
- 指针的基本概念与用法
- 函数中的指针
- 指针用于内存操作
- 小结





引言



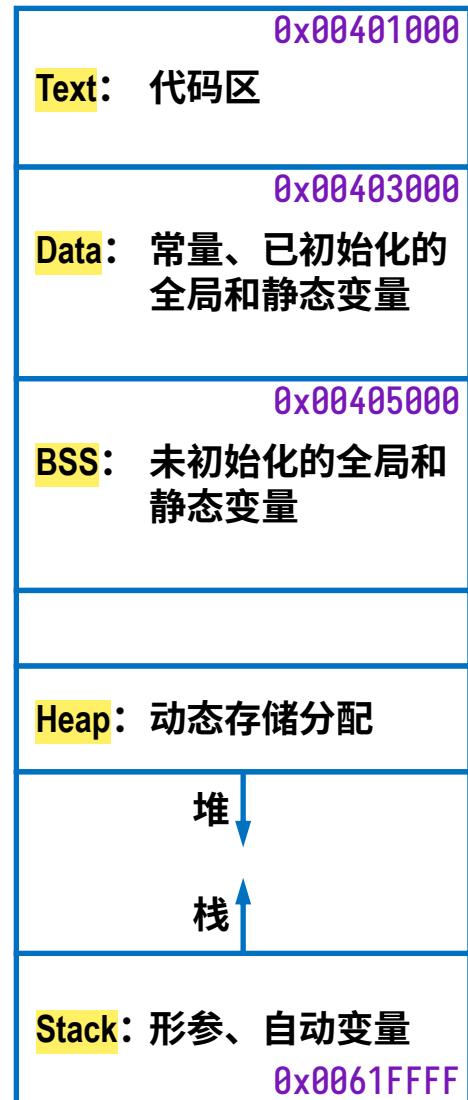
程序空间

■ 程序空间

- 操作系统为程序分配内存空间用于加载指令和数据
- 每条指令和每个数据都占据一定的内存单元
- 指针使用地址访问内存中存储的数据对象

■ 程序空间分配模式举例

- 代码段（文本段）
 - 编译后的二进制程序机器指令
- 堆栈段
 - 堆区：程序运行过程中动态分配的内存空间
 - 栈区：函数参数列表、非静态局部变量、返回值
- 数据段
 - Data区：已初始化的全局变量和静态局部变量、常量
 - BSS区：未初始化的全局变量和静态局部变量





指针的基本概念与用法



- 指针的基本概念
- 一维数组与指针
- 二维数组与指针
- 指针数组与指向指针的指针



指针的基本概念

■ 指针 (Pointer)

- 内存对象的**地址**
- 指针常常是表达某个运算的唯一途径
- 使用指针通常可以生成更高效、更紧凑的代码

■ 指针类型

- 表示**内存地址**的数据类型
- 由其他数据类型派生而成，必须依托**基类型**进行定义
- 指针的**值**表示内存对象在存储空间的**起始地址**
- 指针的**基类型**表示内存对象**数据类型**
 - 数据长度、编码格式、能够进行的运算等



指针的基本语法

■ 指针变量定义的一般形式

类型名 *指针变量名；

- **类型名**即指针的基类型，指针类型可表示为**类型名 ***
- 变量名前带有*****表示该变量为指针类型变量

■ 取地址运算符

优先级	运算符	描述	举例	操作数	结合性
2	&	取内存对象的地址	&a	一元	右结合

- 取操作数在内存中的起始地址，生成指向操作数的指针

■ 间接访问运算符

优先级	运算符	描述	举例	操作数	结合性
2	*	间接访问内存对象	*p	一元	右结合

- 按照基类型从操作数表示的内存地址取值

指针的基本语法举例

■ 例5.1-1：指针基本语法示例。

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

```
int main()
{
```

```
    float x=3.14159;
```

```
    float *p;
```

```
    p = &x;
```

```
    printf("%f\n", *p);
```

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

```
    return 0;
}
```

程序解析：

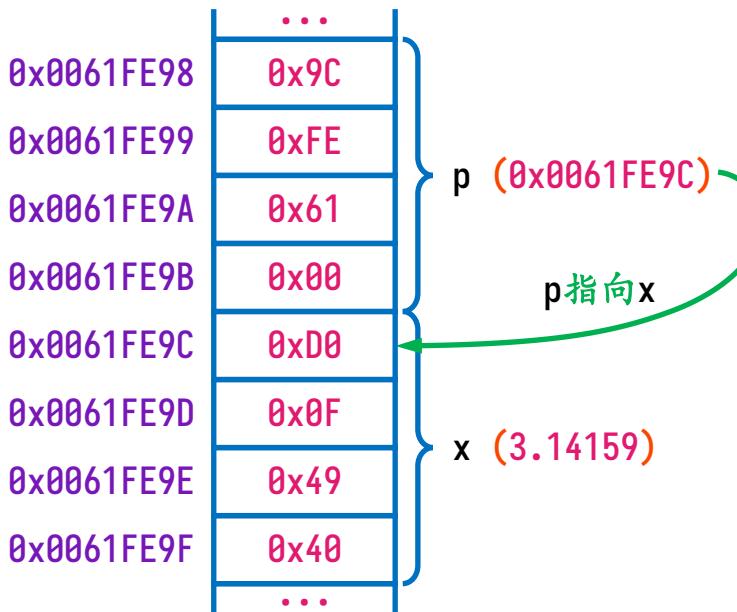
1 变量定义中的*表示数据类型，不表示运算；

// 定义指针变量p

// 用p保存变量x的地址
// 用p从地址中取数据

2 表达式中的*则表示间接访问运算。

变量监测：(32位编译环境)



运行结果：(64位编译环境)

3.141590

000000000062FE1C

000000000062FE1C

000000000062FE10

运行结果：(32位编译环境)

3.141590

0061FE9C

0061FE9C

0061FE98



- 指针的基本概念
- 一维数组与指针
- 二维数组与指针
- 指针数组与指向指针的指针



一维数组与指针

■ 数组名

- 数组名通常被解释为指向该数组起始元素的指针，以下情况除外
 - 数组名作为求字节数运算符**sizeof**的操作数，得到数组整体的字节数
 - 数组名作为取地址运算符**&**的操作数，得到指向数组整体的、基类型为数组的指针
- 数组名不是变量，不能被修改
 - 数组名不能作为赋值运算符、自增/自减运算符的操作数

■ 指向数组元素的指针

- 使用数组元素的地址对指针变量赋值或初始化，指针变量指向该数组元素

```
double a[5], *pa=a; // 使用数组名a初始化指针变量pa, pa指向a[0]
double b[5], *pb;
pb = &b[2];           // 使用数组元素b[2]的地址对指针变量pb赋值, pb指向b[2]
```

- 指针加/减1，表示其地址值增加/减少**sizeof(基类型)**，即**基类型的字节数**
- 指向数组元素的指针加/减n，表示指针后移/前移n个元素
- 注意避免指针在移动过程中越过数组边界

指向数组元素的指针举例

■ 例5.1-2：利用指针输出数组中所有元素的值和地址。

```
#include <stdio.h>

int main()
{
    int i;
    double a[4]={1.1,2.2,3.3,4.4};
    double *p;

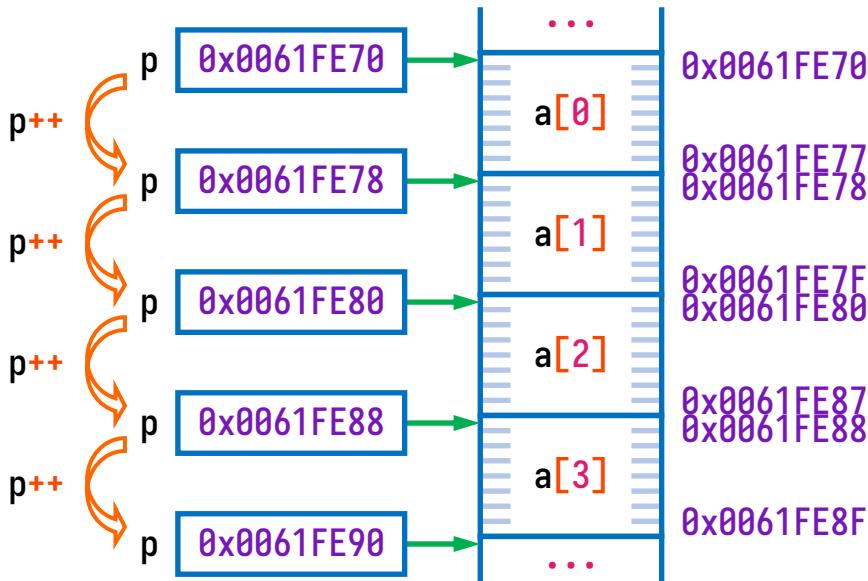
    p = a;          // 指针p指向a[0]
    for(i=0; i<4; i++) {
        printf("&a[%d]=%p, ", i, &a[i]);
        printf("p=%p, ", p);
        printf("a[%d]=%.1f, ", i, a[i]);
        printf("*p=%.1f\n", *p);
        p++;           // 指针p指向下一个元素
    }
    return 0;
}

2 指向数组元素的指针变量自增，从而指向数组的下一个元素。
```

程序解析：

1 使用数组名对指针变量赋值，则指针指向数组起始元素；

变量监测：(32位编译环境)



运行结果：(32位编译环境)

```
&a[0]=0061FE70, p=0061FE70, a[0]=1.1, *p=1.1
&a[1]=0061FE78, p=0061FE78, a[1]=2.2, *p=2.2
&a[2]=0061FE80, p=0061FE80, a[2]=3.3, *p=3.3
&a[3]=0061FE88, p=0061FE88, a[3]=4.4, *p=4.4
```



指针的运算

■ 指针的运算类型

- 赋值运算：基本赋值、加赋值、减赋值
- 算术运算：加法、减法、自增、自减
- 关系运算：所有关系运算
- 间接访问运算
- 下标运算
- 指针变量的取地址运算
- 结构体类型指针的指向成员运算

■ 前提条件

- 指针的部分运算需要满足一定的前提条件
 - 赋值运算、算术运算、关系运算、下标运算等
- 不满足前提条件但符合语法的运算不会引起语法错误，但可能造成危险操作



指针的赋值运算

■ 同类型指针赋值

- 指针类型数据可以赋值给同类型指针变量

```
int a, *pa=&a;      // 定义整型变量和整型指针变量，并对整型指针变量进行初始化  
int b, *pb;          // 定义整型变量和整型指针变量，但未做初始化操作  
pb = pa;             // 整型指针变量之间直接赋值  
pa = &b;              // 将整型变量的地址赋值给整型指针变量
```

■ 非同类型指针赋值

- 将**非指针类型数据或其他类型指针数据**赋值给指针变量，将引起编译警告

```
int a=5, *pa=&a;          // 定义整型变量和整型指针变量  
float *pb;                // 定义浮点型指针变量  
pb = pa;                  // 编译警告：不兼容的指针类型赋值  
printf("%d %e", *pa, *pb); // 输出结果：5 7.006492e-045
```

- 如有必要，可将其他类型指针或整型数据经**强制类型转换**后赋值给指针变量
- 任何类型的指针都可以直接赋值给**空类型指针**变量，反之亦然
- 整数**0**可以直接赋值给指针变量，得到**空指针**



空类型指针和空指针

■ 空类型指针

```
void *p;
```

- 没有指定其指向的数据类型，无法引用其指向的数据，仅存储地址值
- 一般用于函数的形参类型和返回值类型
- 任何类型的指针都可以**隐式转换**为空类型指针，反之亦然
 - 任何类型指针和空类型指针变量之间，都可以**不经强制类型转换**而直接相互赋值

■ 空指针

- 值为**0**的指针称**为空指针**
 - **0**是唯一允许直接赋值给指针变量的整数值
 - 为更清楚地表示其含义，头文件中将**0**定义为符号常量**NULL**
- **不能引用空指针指向的内存空间**
 - 将指针变量初始化或赋值为空指针，可以防止未经赋值的指针指向未知区域

```
#define NULL ((void *)0)  
int *p=NULL;           // 将指针变量初始化为空指针，防止野指针误操作
```

指针的强制类型转换

■ 强制类型转换为指针类型

- 非指针类型数据或**其他类型指针数据**不能直接赋值给指针变量
 - 整数0（空指针）、空类型指针除外
- 其他类型指针数据或整型数据经**强制类型转换**后可以赋值给指针变量
- 需保证指针及其操作的正确性和有效性

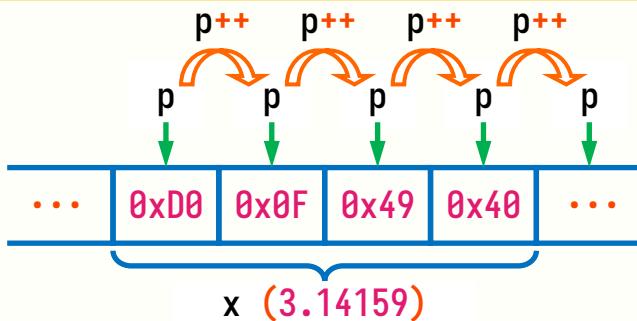
```

int *p;
double d;
p = (int *) &d;           // 将浮点型指针强制转换为整型指针，并赋值给整型指针变量
p = (int *) 0x1234;      // 将整型常量强制转换为整型指针，并赋值给整型指针变量
    
```

■ 应用举例

```

unsigned char *p, i;
float x=3.14159;
p = (unsigned char *) &x;
for (i=0; i<sizeof(float); i++)
    printf("%02X", *p++);
    
```





指针的算术运算

■ 指向数组元素的指针与整数之间的加法、减法运算

- 若 p 指向 $a[i]$, n 为整数, 则 $p+n$ 指向 $a[i+n]$, $p-n$ 指向 $a[i-n]$

■ 指向数组元素的指针变量的自增、自减运算

- 若 p 指向 $a[i]$, 则
- $p++$ 的值为 $a[i]$ 的地址, p 自增后指向 $a[i+1]$
- $++p$ 的值为 $a[i+1]$ 的地址, p 自增后指向 $a[i+1]$
- $p--$ 的值为 $a[i]$ 的地址, p 自减后指向 $a[i-1]$
- $--p$ 的值为 $a[i-1]$ 的地址, p 自减后指向 $a[i-1]$

■ 指向同一数组中元素的指针之间的减法运算

- 若 p 指向 $a[i]$, q 指向 $a[j]$, 则 $p-q$ 的值为 $i-j$

■ 说明

- 当上述算术运算不满足前提条件时, 不会引起语法错误, 但危险且无意义
- 注意避免数组越界

指针的算术运算举例

■ 例5.1-3：指针的算术运算举例。

```
#include<stdio.h>

int main()
{
    int a[8]={1,2,3,4,5,6,7,8};
    int *p=&a[3];           // 指针可指向数组任一元素

    printf("%3d", *p);      // p指向a[3], 输出4
    printf("%3d", *(p+2));  // p+2指向a[5], 输出6
    printf("%3d", *(p-2));  // p-2指向a[1], 输出2
    printf("%3d", *p);      // p指向a[3], 输出4
    printf("%3d", *++p);    // p指向a[4], 输出5
    printf("%3d", *p);      // p指向a[4], 输出5
    printf("%3d", *p--);    // 输出5, p指向a[3]
    printf("%3d", *p);      // p指向a[3], 输出4
    printf("%3d", *--p);    // p指向a[2], 输出3
    printf("%3d", --(*p));  // *p即a[2]自减, 输出2

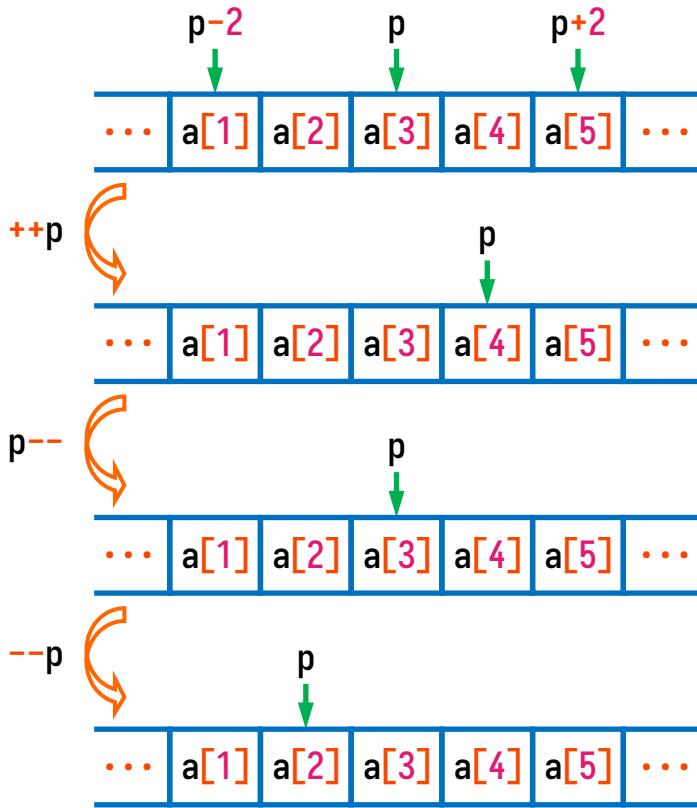
    return 0;
}
```

程序解析：

1 使用 $a[3]$ 的地址对指针变量 p 初始化，而不是对 $*p$ 初始化；

2 $--(*p)$ 也可写作 $--*p$ ，为了使程序易读，仍建议写作 $--(*p)$ 。

变量监测：



运行结果：

4 6 2 4 5 5 5 4 3 2



指针的关系运算

■ 判断指针是否为空指针

- 任何类型指针都可以与`0`或`NULL`进行**相等**或**不等**关系运算
- 用于判断指针是否为**空指针**

```
p==NULL, p==0, !p      // 判断指针是否为空指针  
p!=NULL, p!=0, p      // 判断指针是否不是空指针
```

■ 判断同类型指针是否相等

- 同类型指针之间可以进行**相等**或**不等**关系运算
- 用于判断两个指针是否指向了同一个数据对象

■ 判断同类型指针的大小关系

- 指向同一个数组中元素的指针可以进行**所有**关系运算
- 用于判断两个指针所指元素在数组中的**前后次序关系**



指针的间接访问运算

■ 间接访问运算

- 通过间接访问运算符*可以引用指针指向的数据对象
- 空类型指针、空指针、无效指针除外
 - 空类型指针：间接访问运算引起语法错误
 - 空指针：间接访问运算引起运行错误
 - 无效指针：间接访问运算引起运行错误或未知运算结果

■ 无效指针

- 未指向有效数据对象或函数的非空指针
 - 野指针：没有经过初始化或赋值的指针变量
 - 悬空指针（悬挂指针、迷途指针）：原本所指向的内存空间已经被释放的指针
 - 指向数组元素的指针移动到了数组范围之外
 - 将非0整数值强制转换成指针，可能产生无效指针
- 引用无效指针指向的数据，通常不会引起语法错误，但会造成未知的后果



指针的下标运算

■ 指向数组元素的指针的下标运算

- 若 p 指向 $a[0]$, 则 $p+i$ 指向 $a[i]$, $a[i]$ 可以表示为 $*(p+i)$
- 又由于 $p=a$, 则 $a[i]$ 也可以表示为 $p[i]$ 或 $*(a+i)$
- 若 q 指向 $a[j]$, 则 $q+i$ 指向 $a[i+j]$, $a[i+j]$ 可以表示为 $*(q+i)$ 或 $q[i]$ 或 $*(a+i+j)$
- 注意避免数组越界

■ 指针的 θ 下标运算

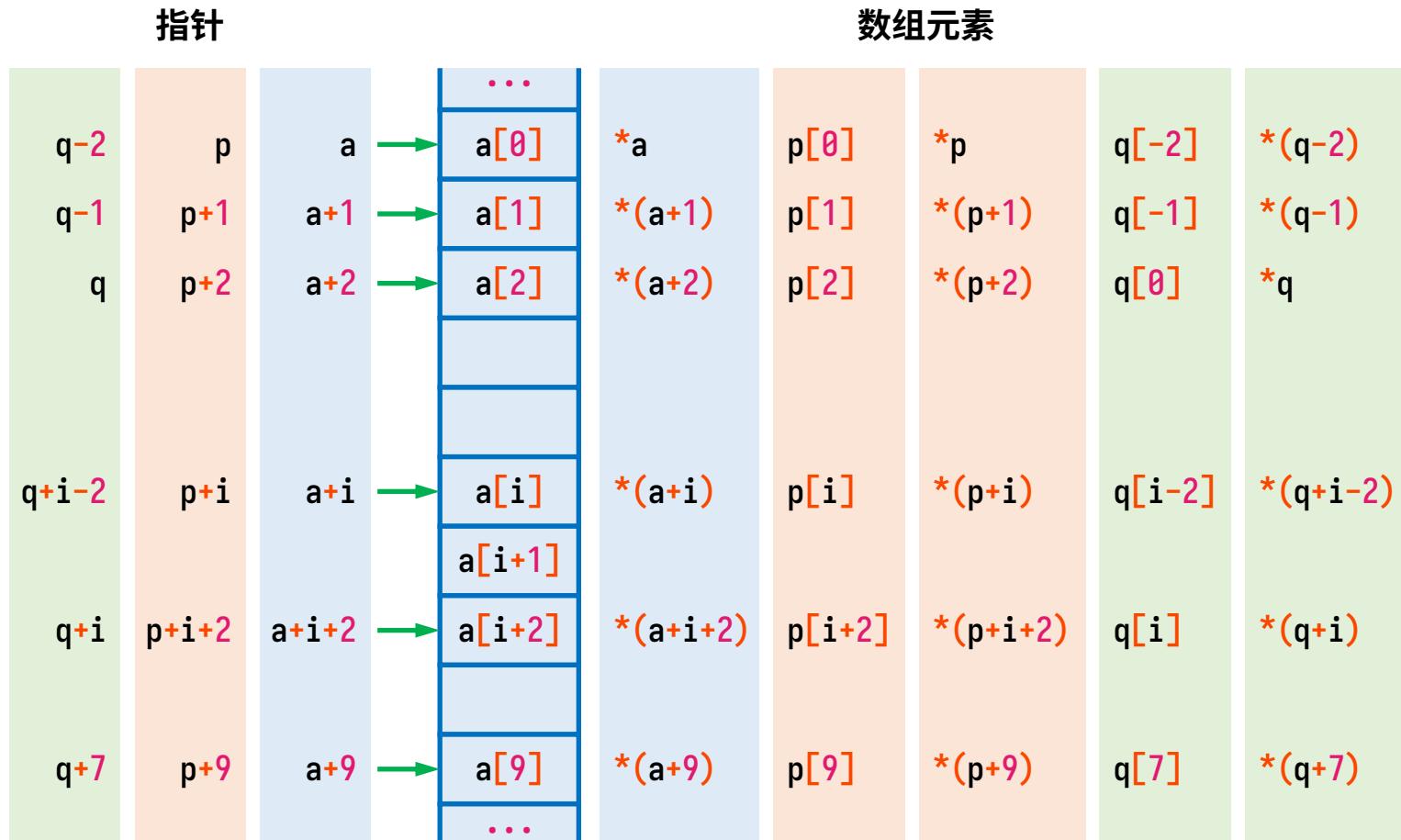
- 无论是否指向数组元素, 指针 p 都可以进行 $p[\theta]$ 运算, 表示 $*p$
- 空类型指针、空指针、无效指针除外

■ 下标运算与指针运算

- 下标运算 $a[i]$ 本质上是加法和间接访问运算 $*(a+i)$
- 以下表达式等价: $a[i] \Leftrightarrow *(a+i) \Leftrightarrow *(i+a) \Leftrightarrow i[a]$

通过指针访问数组元素

```
int a[10], *p=a, *q=&a[2];
```



数组与指向数组元素的指针



字符数组、字符串与字符指针

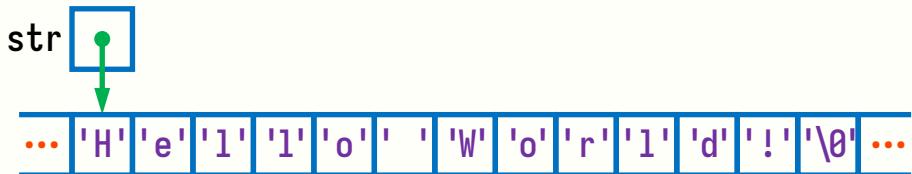
■ 字符数组与字符指针

- 字符数组存储若干独立的字符型元素
 - 字符指针与字符数组的关系，即指针与数组的关系
- 字符数组存储字符串
 - 包括字符串结束标志在内，不得超过字符数组长度
 - 字符指针对字符串的访问，至字符串结束标志为止

■ 字符串常量与字符指针

- 字符串常量为无名字符数组，以其字面形式代替数组名
- 字符串常量通常被解释为**指向该字符串常量起始字符的指针**
 - 作为求字节数运算符**sizeof**和取地址运算符**&**的操作数时除外
- 字符串常量可以对字符指针变量赋值或初始化，字符指针变量指向该字符串

```
char *str="Hello World!";
char *str;
str = "Hello World!";
```





字符数组与字符指针举例——字符数组加密

■ 例5.1-4：字符数组加密：将字符数组的每个元素加上其下标和一个偏移量。

```
#include <stdio.h>

int main()
{
    char s[4]={'C','O','D','E'};
    char scpy[4];
    int i, offset=5;
    char *p=s;

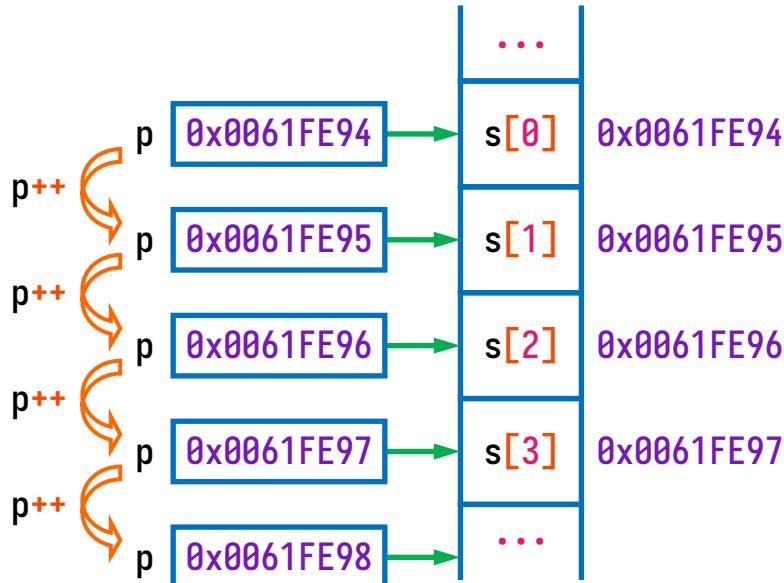
    printf("&s[0]=%p, s=%p\n", &s[0], s);
    for(i=0; i<4; i++) {
        printf("p=%p, *p=%c, ", p, *p);
        scpy[i] = *p + i + offset;
        printf("加密: %c->%c\n", s[i], scpy[i]);
        p++;
    }
    return 0;
}
```

程序解析：

1 使用数组名s对指针变量p初始化，而不是对*p初始化，也不能使用s[4]对p初始化。

2 数组s包含4个字符元素，但并未包含字符串结束标志，所以未组成字符串；

变量监测：(32位编译环境)



运行结果：(32位编译环境)

```
&s[0]=0061FE94, s=0061FE94
p=0061FE94, *p=C, 加密: C->H
p=0061FE95, *p=O, 加密: O->U
p=0061FE96, *p=D, 加密: D->K
p=0061FE97, *p=E, 加密: E->M
```



指向字符串的指针举例

■ 例5.1-5：利用指向字符串的指针统计每个字母出现的次数。

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

```
int main()
```

```
{  
    char *str="The new Science of Complex Systems is providing radical new ways of understanding the \  
    physical, biological, ecological, and social universe. The economic regions that lead \  
    this science and its engineering will dominate the twenty-first century by their wealth and \  
    influence. In all domains, complex systems are studied through increasingly large quantities \  
    of data, stimulating revolutionary scientific breakthroughs. Also, many new and fundamental \  
    theoretical questions occur across the domains of physical and human science, making it \  
    essential to develop the new Science of Complex Systems in an interdisciplinary way.";
```

```
    char c;  
    int anum[26]={0};
```

```
    while (c = *str++) {  
        if (c>='a'&&c<='z') anum[c-'a']++;  
        if (c>='A'&&c<='Z') anum[c-'A']++;  
    }
```

```
    printf("字母a和A有%d个\n", anum[0]);  
    printf("字母b和B有%d个\n", anum[1]);
```

```
    return 0;  
}
```

运行结果：

字母a和A有42个

字母b和B有 3个

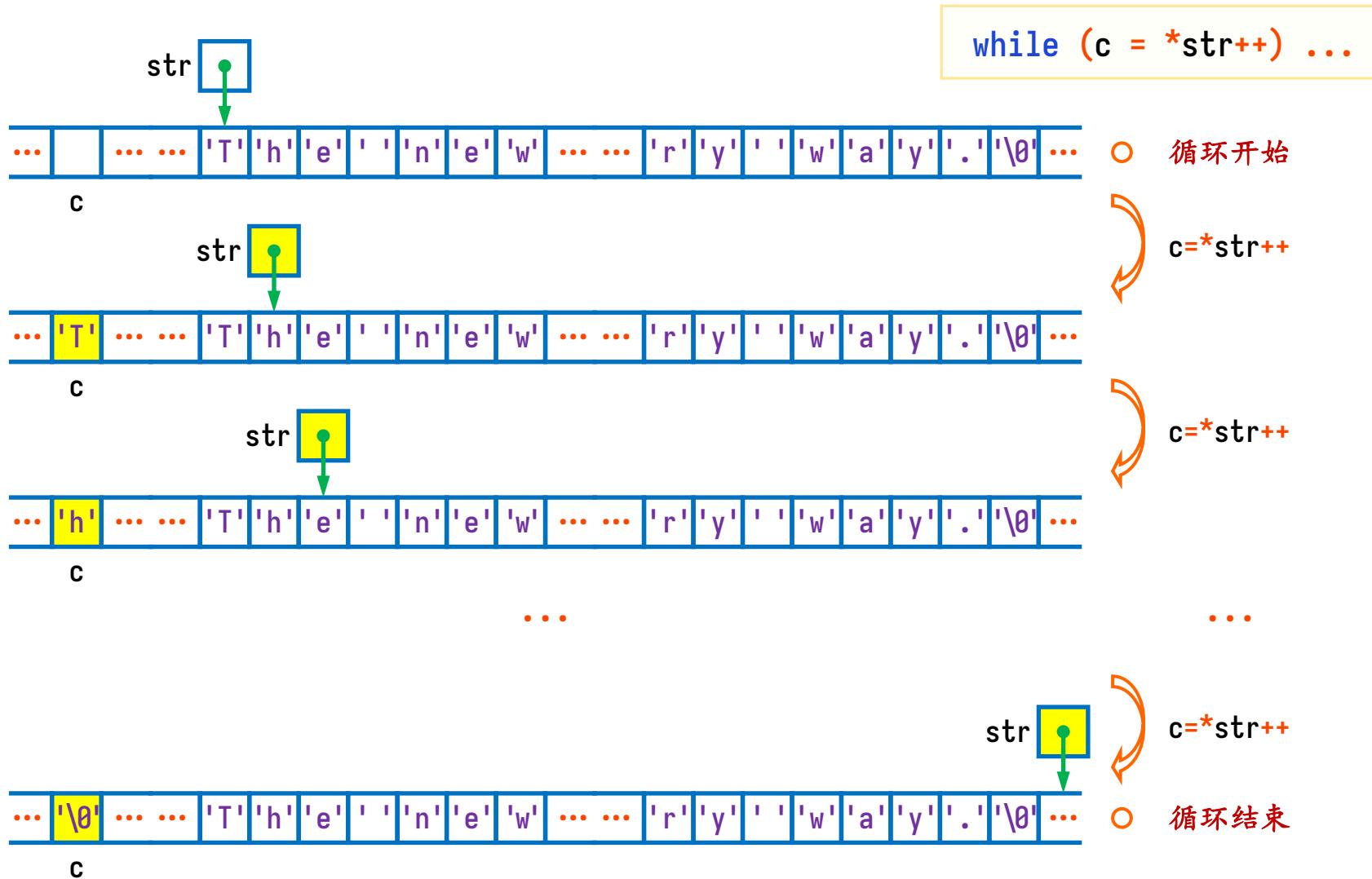
程序解析：

1 使用字符串常量对字符指针变量 str 进行初始化，str 指向字符串常量的起始字符 'T'；

2 循环条件使用表达式 `c = *str++`，获取 str 当前指向的字符赋值给变量 c，并以所赋之值作为判断条件，若其为字符串结束标志 '\0'，则循环结束；同时，str 自增 1，指向下一个字符。

指向字符串的指针举例

■ 例5.1-5：利用指向字符串的指针统计每个字母出现的次数。





- 指针的基本概念
- 一维数组与指针
- 二维数组与指针
- 指针数组与指向指针的指针

指向二维数组行的指针

■ 二维数组

- 二维数组的行是一维数组
- 二维数组可以看作行的一维数组
- 二维数组名通常表示指向该数组起始行的指针

`char (*p)[4];`

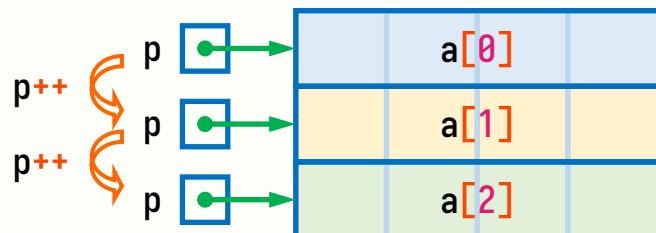
- ① 按照基类型数组定义形式书写；
- ② 添加`*`表示标识符为指针；
- ③ 添加`()`保证`*`比`[]`先起作用。

■ 指向二维数组行的指针

行指针的定义形式

- 也称为行指针
- 基类型是二维数组的行，即长度为二维数组列数的一维数组
 - 行指针定义中的基类型数组长度必须指定，二维数组定义中的列数也必须指定
- 可以使用对应列数的二维数组名赋值或初始化
- 行指针加/减n时，指针后移/前移n行

```
char a[3][4];      // 定义3行4列的二维数组
char (*p)[4];      // 定义行指针变量
p = a;              // p指向a[0]
p++;                // p指向a[1]
```



指向二维数组元素的指针

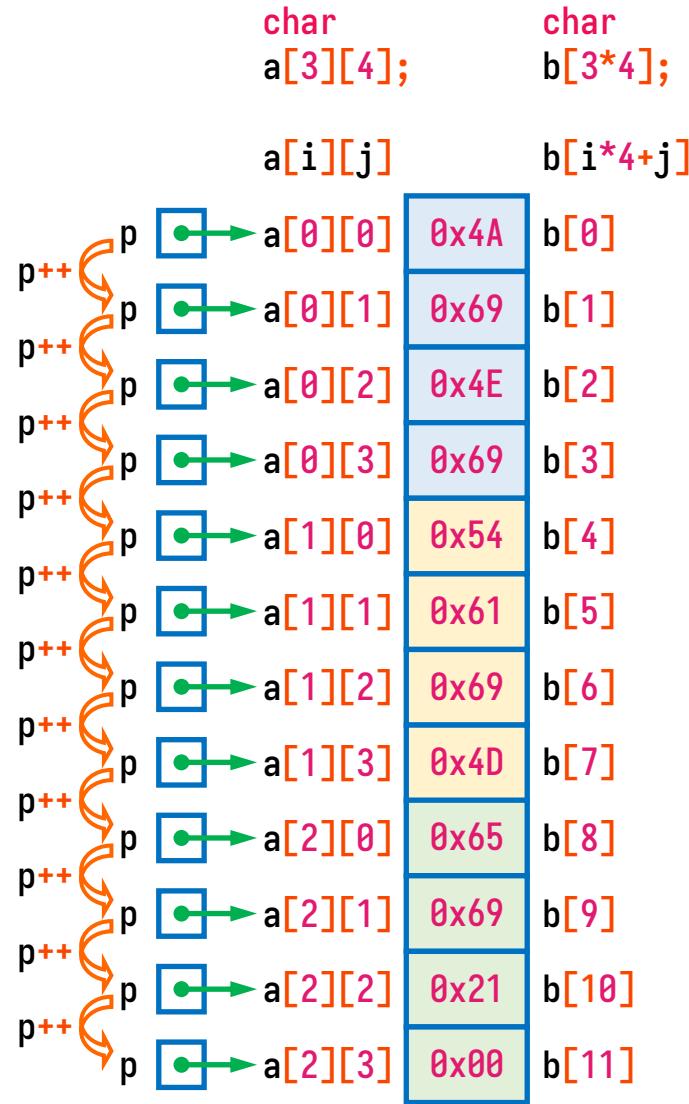
■ 二维数组的存储形式

- 二维数组元素在内存中按照一维形式连续存储
 - 与元素个数相同的一维数组存储形式相同
- 二维数组元素按照指定行列数组织成二维形式
 - 反映数据的逻辑关系，便于数据的处理

■ 指向二维数组元素的指针

- 指向元素的指针可以访问二维数组的所有元素
 - 以访问一维数组元素的方式访问二维数组元素
 - 二维数组下标可以转换为对应的一维数组下标
- 二维数组的行可以看作一维数组名使用
 - 可以通过二维数组名或行指针及相应运算得到

```
char a[3][4]; // 定义3行4列的二维数组
char *p=a[0]; // 定义指针变量p并指向a[0][0]
p++; // p指向a[0][1]
```





二维字符数组

■ 二维字符数组

- 每行都是一维字符数组，可以存储一个字符串
- 二维字符数组可以用作**字符串数组**
 - 常用来表示多行文本信息
- 每行长度必须相同，列数必须大于等于最长字符串所需空间
 - 字符串通常长度不一
 - 二维字符数组表示字符串数组会有一部分元素空置

■ 二维字符数组与指针

- 二维字符数组存放独立的字符数据
 - 与其他数据类型类似，可以使用指向行的指针、指向元素的指针访问
- 二维字符数组用作字符串数组
 - 使用指向行的指针整体访问每行存储的字符串



字符串数组举例——月份名称

■ 例5.1-8：利用字符串数组，根据月份数字查找并输出月份名称。

```
#include<stdio.h>

int main()
{
    // 二维字符数组name的每行分别使用一个字符串进行初始化
    char name[][20]={"Illegal Month", "January", "February", "March",
                      "April", "May", "June", "July", "August",
                      "September", "October", "November", "December"};
    int n;

    while (1) {
        printf("请输入月份数字: ");
        scanf("%d", &n);
        printf("%s\n", (n<1||n>12)?name[0]:name[n]); // name[n]是保存有字符串的一维字符数组
        if (n < 0)
            return -1;
    }

    return 0;
}
```

运行结果：

请输入月份数字: 11↙

November

请输入月份数字: 13↙

Illegal Month

请输入月份数字: 9↙

September

请输入月份数字: 5↙

May

请输入月份数字: -1↙

Illegal Month



- 指针的基本概念
- 一维数组与指针
- 二维数组与指针
- 指针数组与指向指针的指针



指针数组与指向指针的指针

■ 指针数组

- 元素是指针类型的数组

类型名 *指针数组名[数组长度];

- 指针类型的数组元素可以分别指向一维数组
 - 数组元素所指的一维数组长度不必相等，也不必连续存储
 - 具有类似于二维数组元素访问的便利性，同时提高了数据组织的灵活性
- 字符型指针数组适用于对多行文本、多个字符串的处理
- 注意与二维数组的行指针类型加以区分

■ 指向指针的指针

- 基类型是指针类型的指针

类型名 **指针变量名;

- 指针数组和指向指针的指针的关系即数组和指针的关系



指针数组访问字符串举例——月份名称

■ 例5.1-10：利用指针数组，根据月份数字查找并输出月份名称。

```
#include <stdio.h>

int main()
{
    // 指针数组pname中的元素分别指向13个字符串常量
    char *pname[13]={ "Illegal Month", "January", "February", "March",
                      "April", "May", "June", "July", "August",
                      "September", "October", "November", "December" };
    int n;

    while (1) {
        printf("请输入月份数字: ");
        scanf("%d", &n);
        printf("%s\n", (n<1||n>12)?pname[0]:pname[n]); // pname[n]是指向字符串的指针
        if (n < 0)
            return -1;
    }

    return 0;
}
```

运行结果：

请输入月份数字: 11↙

November

请输入月份数字: 13↙

Illegal Month

请输入月份数字: 9↙

September

请输入月份数字: 5↙

May

请输入月份数字: -1↙

Illegal Month



指向指针的指针举例——月份名称

■ 例5.1-11：利用指向指针的指针输出月份名称。

```
#include <stdio.h>

int main()
{
    // 指针数组pname中的元素分别指向13个字符串常量
    char *pname[13]={"Illegal Month", "January", "February", "March", "April",
                     "May", "June", "July", "August", "September", "October",
                     "November", "December"};
    char **p;
    int i;

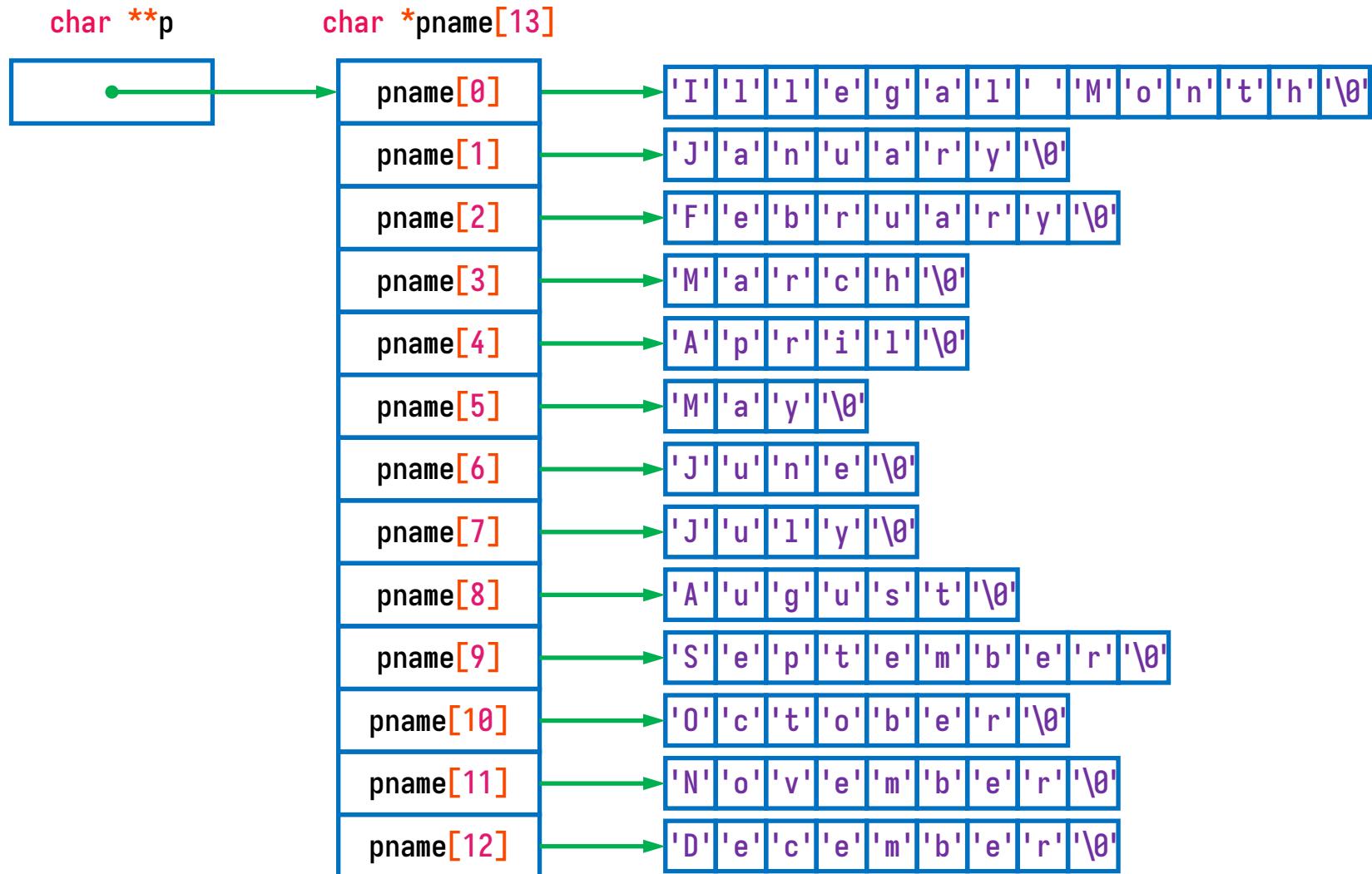
    p = pname;           // 指向指针的指针变量p指向指针数组pname的首地址
    for (i=1; i<13; i++) {
        p++;            // p指向pname的下一个指针类型元素
        printf("%s\n", *p); // *p是p当前指向的pname元素，该元素指向一个字符串
    }

    return(0);
}
```

运行结果：

January
February
March
April
May
June
July
August
September
October
November
December

字符指针数组与字符串



指向字符指针的指针、字符指针数组与字符串



函数中的指针



- 指针用作函数参数
- 指针用作函数返回值
- 用函数处理字符串

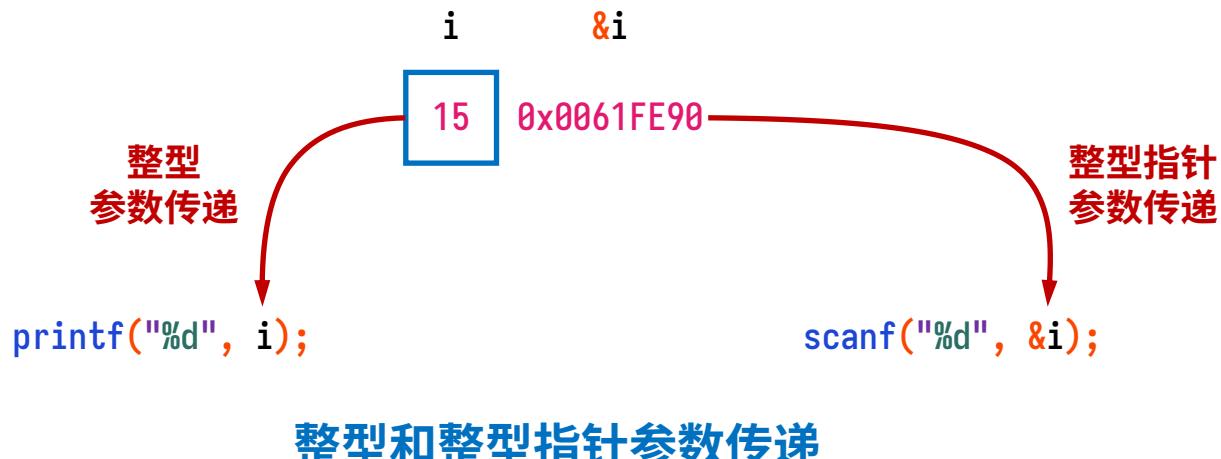
指针类型参数传递

■ 参数传递

- 实参到形参的**单向值传递**
- 被调函数对形参所做的修改**不会回传给实参**

■ 指针类型参数传递

- 指针类型实参和形参指向**相同**的数据对象
- 对形参所指数据的修改，就是对实参所指数据的修改
- 利用指针类型参数和间接访问，可以通过函数调用实现**对多个数据的修改**



指针类型参数举例——交换变量的值

■ 例5.2-1a：编写函数，通过函数调用交换两个整型变量的值。

```
#include <stdio.h>

void swap1(int x, int y)
{
    int t;
    t = x, x = y, y = t;      // 交换x和y
}

int main()
{
    int a=3, b=5;

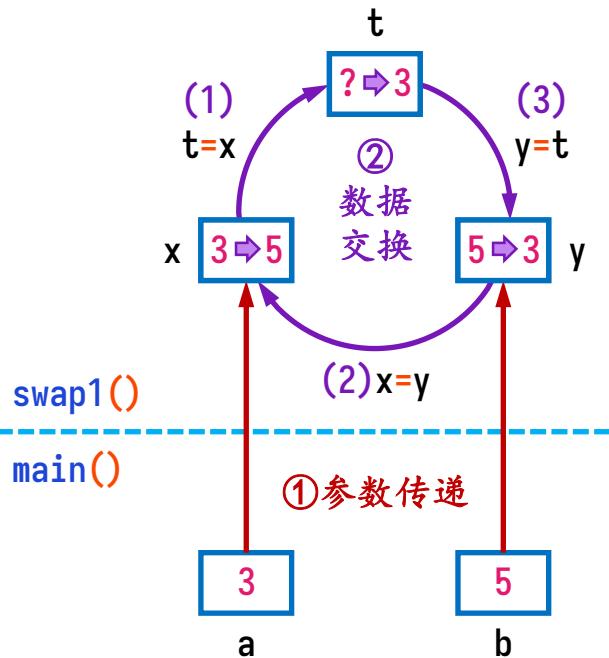
    printf("Before: a=%d, b=%d\n", a, b);
    swap1(a, b);
    printf("After : a=%d, b=%d\n", a, b);

    return 0;
}
```

运行结果：

Before: a=3, b=5
After : a=3, b=5

变量监测：



指针类型参数举例——交换变量的值

■ 例5.2-1b：编写函数，通过函数调用交换两个整型变量的值。

```
#include <stdio.h>

void swap2(int *px, int *py)
{
    int t;
    t=*px, *px=*py, *py=t; // 交换px和py所指数据
}

int main()
{
    int a=3, b=5;
    int *pa=&a, *pb=&b;

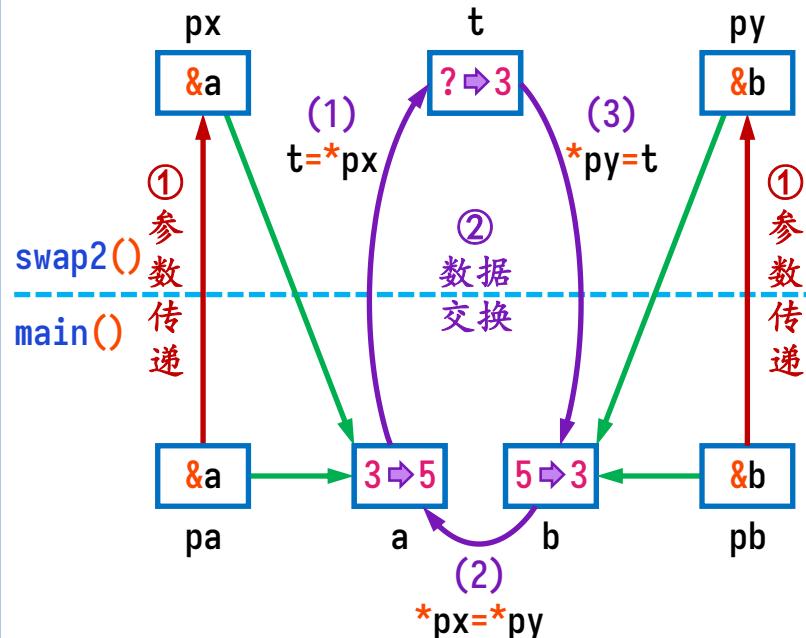
    printf("Before: a=%d, b=%d\n", a, b);
    swap2(pa, pb);
    printf("After : a=%d, b=%d\n", a, b);

    return 0;
}
```

运行结果：

Before: a=3, b=5
After : a=5, b=3

变量监测：



指针类型参数举例——整数部分和小数部分

■ 例5.2-2：编写函数，求浮点数的整数部分和小数部分。

```
#include <stdio.h>

void decompose (double x, long *int_part, double *frac_part)
{
    *int_part = (long)x;
    *frac_part = x - *int_part;
}

int main()
{
    long i;
    double d;

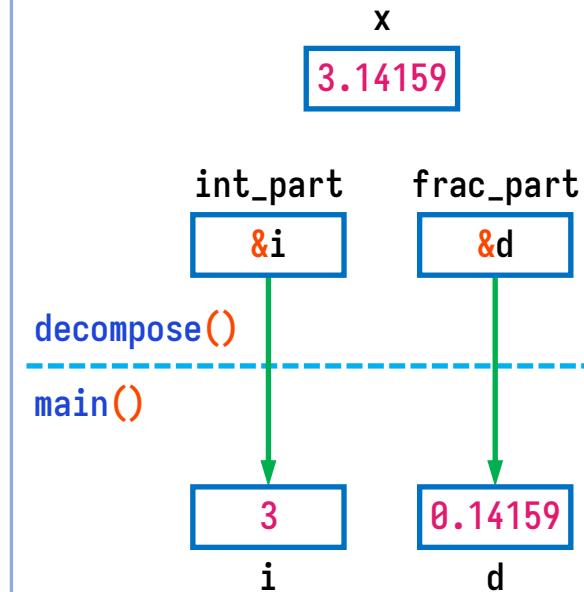
    decompose(3.14159, &i, &d);
    printf("int_part=%d\nfrac_part=%f\n", i, d);

    return 0;
}
```

运行结果：

int_part=3
frac_part=0.141590

变量监测：





指针类型参数举例——输出格式

■ 例5.2-3：设置浮点数输出的小数位数并输出。

```
#include <stdio.h>

int main()
{
    char    format[] = "%10.2f\n";
    double f = 3.1415926535;
    int     i;

    printf("Please input decimal place number (0-9): ");
    scanf("%d", &i);
    format[4] = i + '0';
    printf(format, f);

    return 0;
}
```

运行结果：

```
Please input decimal place number (0-9): 5
3.14159
```



数组用作函数参数

■ 数组用作函数参数

- 数组类型形参实质上是**指针类型**
- 数组类型形参也可以写作指针类型形参
- 实参数组将起始地址传递给形参数组（指针）
- 形参数组（指针）通过间接访问处理实参数组的元素
- 形参数组（指针）在函数中可以被修改，如赋值、自增、自减等

■ 函数处理二维数组

- 形参为二维数组行指针类型，实参为二维数组
 - 行指针类型中包含二维数组列数信息
 - 只能处理任意行数、特定列数的二维数组
- 形参为二维数组元素指针类型，实参为二维数组起始行
 - 元素指针类型中不包含列数信息
 - 可以处理任意行数、列数的二维数组



数组用作函数参数举例——数组逆序

■ 例5.2-4：编写函数，实现一维数组逆序操作。

```
#include <stdio.h>
#define N 10

void reverse(int *a, int n)
{
    int i, t;

    for (i=0; i<n/2; i++)
        t=a[i], a[i]=a[n-1-i], a[n-1-i]=t;
}

int main()
{
    int i, a[N]={1,2,3,4,5,6,7,8,9,10};

    reverse(a, N);
    for (i=0; i<N; i++)
        printf("%4d", a[i]);

    return 0;
}
```

运行结果：

10	9	8	7	6	5	4	3	2	1
----	---	---	---	---	---	---	---	---	---



函数处理二维数组举例——元素乘积

■ 例5.2-5：编写函数，求二维整型数组所有元素的乘积。

```
#include <stdio.h>
#define M 3
#define N 4

// 使用行指针形参，行数也作为形参
int arrmul_1(int (*p)[N], int m)
{
    int i, j, mul=1;

    for (i=0; i<m; i++)
        for (j=0; j<N; j++)
            mul *= p[i][j];

    return mul;
}
```

```
#include <stdio.h>
#define M 3
#define N 4

// 使用元素指针形参，行数和列数也作为形参
int arrmul_2(int *p, int m, int n)
{
    int i, j, mul=1;

    for (i=0; i<m; i++)
        for (j=0; j<n; j++)
            mul *= p[i*n+j];

    return mul;
}
```

```
int main()
{
    int a[M][N] = {{1,2,3,4},{5,6,7,8},{9,10,11,12}};

    printf("%d, ", arrmul_1(a, M));      // 二维数组名作为实参
    printf("%d\n", arrmul_2(*a, M, N)); // 二维数组起始行（即起始元素地址）作为实参

    return 0;
}
```

运行结果：

479001600, 479001600



- 指针用作函数参数
- 指针用作函数返回值
- 用函数处理字符串



指针用作函数返回值

■ 指针类型函数

- 函数类型被定义为指针类型
- 函数返回一个指针类型的返回值

返回值基类型 *函数名(形参类型 形参名, 形参类型 形参名, ...);

返回值基类型 *函数名(形参类型, 形参类型, ...);

■ 指针类型返回值

- 在主调函数中利用指针类型返回值访问某个内存数据对象
 - 变量、数组、结构体、字符串常量、动态分配内存空间
- 要求被调函数返回的指针在主调函数中仍然有效
 - 所指向的内存空间在函数返回后仍然存在并允许访问



动态分配内存空间地址作为函数返回值举例

■ 例5.2-8：编写函数，生成整型随机数存入动态分配内存空间，并返回其地址。

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

int *randdata(int n)
{
    int i;
    int *pdata = NULL;

    // 动态分配内存空间并将其首地址赋值给变量
    pdata = (int *)malloc(n * sizeof(int));
    if (pdata == NULL) {
        printf("Not enough memory!");
        exit(1);
    }

    srand((unsigned int)time(NULL));
    for (i=0; i<n; i++)
        *(pdata + i) = rand();

    return (pdata); // 返回动态分配空间地址
}
```

```
int main()
{
    int *p=NULL;
    int i;

    // 动态分配空间在函数返回后不会自动释放
    p = randdata(10);

    for (i=0; i<6; i++)
        printf("%5d\n", p[i]);

    return 0;
}
```

运行结果一：

21320
7822
3200
13391
1313
8796

运行结果二：

21441
12297
8814
18997
22504
23013

运行结果三：

21731
18633
25861
30712
22998
12043



悬空指针作为函数返回值的错误示例

■ 例5.2-9：编写函数，返回函数中局部变量的地址。 (错误操作示例)

```
#include <stdio.h>

int *myfunction(int x, int y)
{
    int t;

    if (x > y)
        t = x;
    else
        t = y;

    // 返回自动变量t的地址
    return (&t);
}
```

```
int main()
{
    int a=3, b=4;
    int *p;

    // 自动变量在函数返回后将自动释放,
    // 其存储空间不再可用, p成为悬空指针
    p = myfunction(a, b);
    printf("max is:\n");
    // 此处*p访问结果不确定, 可能导致异常
    printf("%d\n", *p);

    return 0;
}
```

编译信息： (Dev-C++ 5.11, TDM-GCC 4.9.2)

[Warning] function returns address of local variable [-Wreturn-local-addr]



字符串常量地址作为函数返回值举例

■ 例5.2-10：编写函数，返回函数中字符串常量的地址。

```
#include <stdio.h>

char *weekday(int n)
{
    // 使用字符串常量地址对字符指针数组初始化
    char *days[]={"Monday", "Tuesday", "Wednesday", "Thursday",
                  "Friday", "Saturday", "Sunday"};
    if (n<1 || n>7)
        return (NULL);
    return (days[n-1]);           // 返回字符串常量的地址
}

int main()
{
    int num;
    scanf("%d", &num);
    printf("%s\n", weekday(num)); // 字符串常量位于静态存储区，在函数返回后不会自动释放
    return 0;
}
```

运行结果一：

1↙
Monday

运行结果二：

3↙
Wednesday

运行结果三：

8↙



静态数组地址作为函数返回值举例

■ 例5.2-11：编写函数，返回函数中静态数组的首地址。

```
#include<stdio.h>

int *getdata()
{
    static int a[5];           // 静态数组a位于静态存储区
    int      i;

    for (i = 0; i < 5; i++)
        scanf("%d", &a[i]);

    return a;                 // 返回静态数组a首地址
}

int main()
{
    int *p;
    int i;

    p = getdata();           // 指针p指向静态数组a首地址，静态数组a不会在函数返回后释放
    for (i = 0; i < 5; i++)
        printf("%d ", *(p+i)); // 利用指针p输出静态数组a的元素

    return 0;
}
```

运行结果：

```
28 45 31 60 97↙
28 45 31 60 97
```



- 指针用作函数参数
- 指针用作函数返回值
- 用函数处理字符串



用函数处理字符串

■ 字符串的表示

- 起始位置：字符串常量或字符数组范围内某个字符的地址
 - 字符串常量：本质为字符数组，表示该字符串常量的首地址
 - 字符数组名：表示该字符数组的首地址
 - 字符数组元素的地址：表示字符串从该元素开始，而不从数组的起始元素开始
 - 指向字符数组元素的指针：表示字符串从指针指向的字符元素开始
- 结束标志：从起始位置开始，随着地址的递增而出现的**第一个空字符**

■ 用函数处理字符串

- 使用字符指针类型形参和返回值表示字符串
- 字符串常量的内容一般不能修改
- 字符数组的内容可以修改，但存储的字符串不能超过数组范围
- 若使用字符数组存储字符串，字符串范围截止于**空字符**，而非整个数组范围



用函数处理字符串举例——求字符串长度

■ 例5.2-12：编写函数，求字符串长度。

```
#include <stdio.h>

int udf_strlen(char *s) // 形参为指向字符串起始地址的指针
{
    int len=0;

    while (*s++) // while (*s != '\0')
        len++; //     len++, s++;

    return len;
}

int main()
{
    char *str1="C Language"; // 字符指针，使用字符串常量起始地址初始化
    char str2[20]="USTC"; // 字符数组，使用字符串常量初始化

    printf("The length of str1 is %d.\n", udf_strlen(str1));
    printf("The length of str2 is %d.\n", udf_strlen(str2));

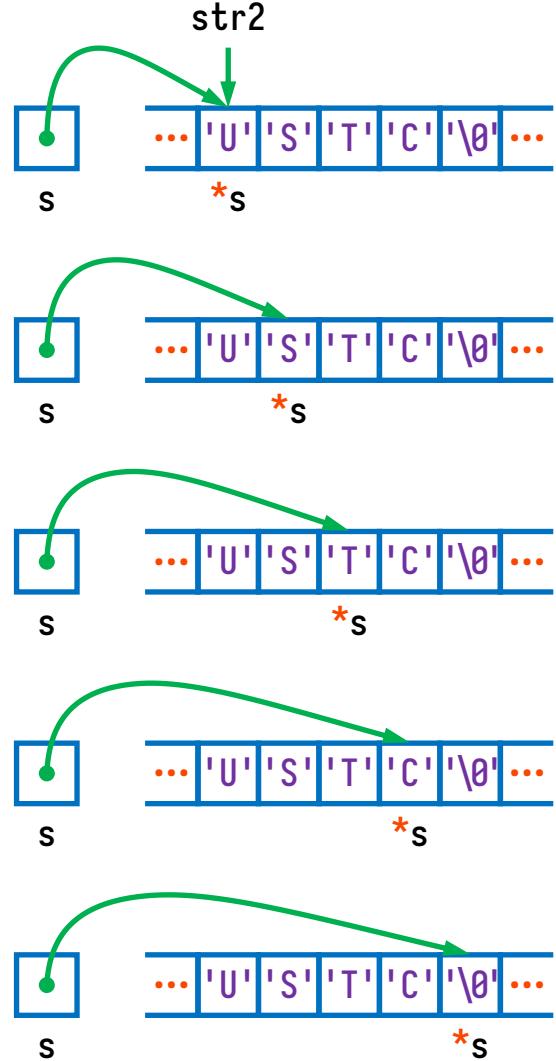
    return 0;
}
```

运行结果：

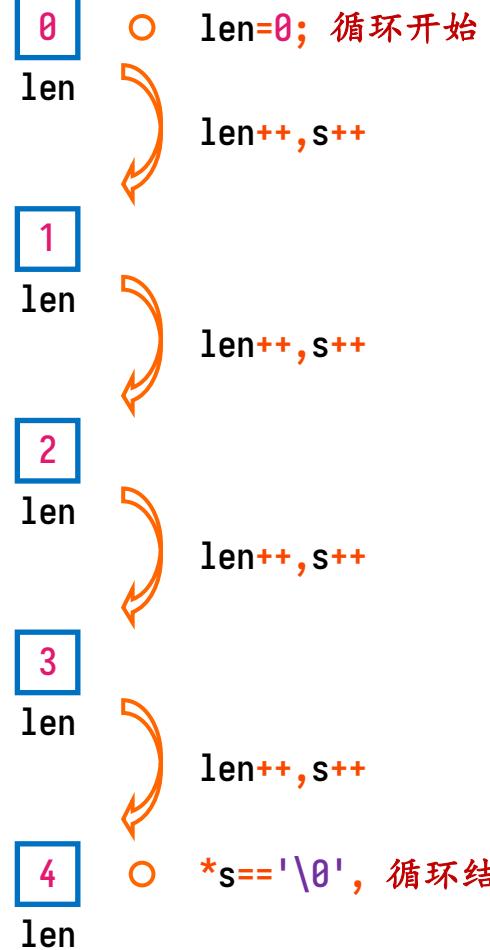
The length of str1 is 10.
The length of str2 is 4.

用函数处理字符串举例——求字符串长度

■ 例5.2-12：编写函数，求字符串长度。



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





用函数处理字符串举例——复制字符串

■ 例5.2-13：编写函数，复制字符串。

```
#include <stdio.h>

char *udf_strcpy(char *d, char *s)      // 形参为指向字符串起始地址的指针
{
    char *t=d;

    while (*d++ = *s++);
    // while (*d = *s)
    //     d++, s++;

    return t;
}

int main()
{
    char *str1="ABC";                  // 字符指针，使用字符串常量起始地址初始化
    char str2[20];                    // 字符数组，目标字符串，应能够容纳源字符串

    udf_strcpy(str2, str1);
    printf("%s\n", str2);

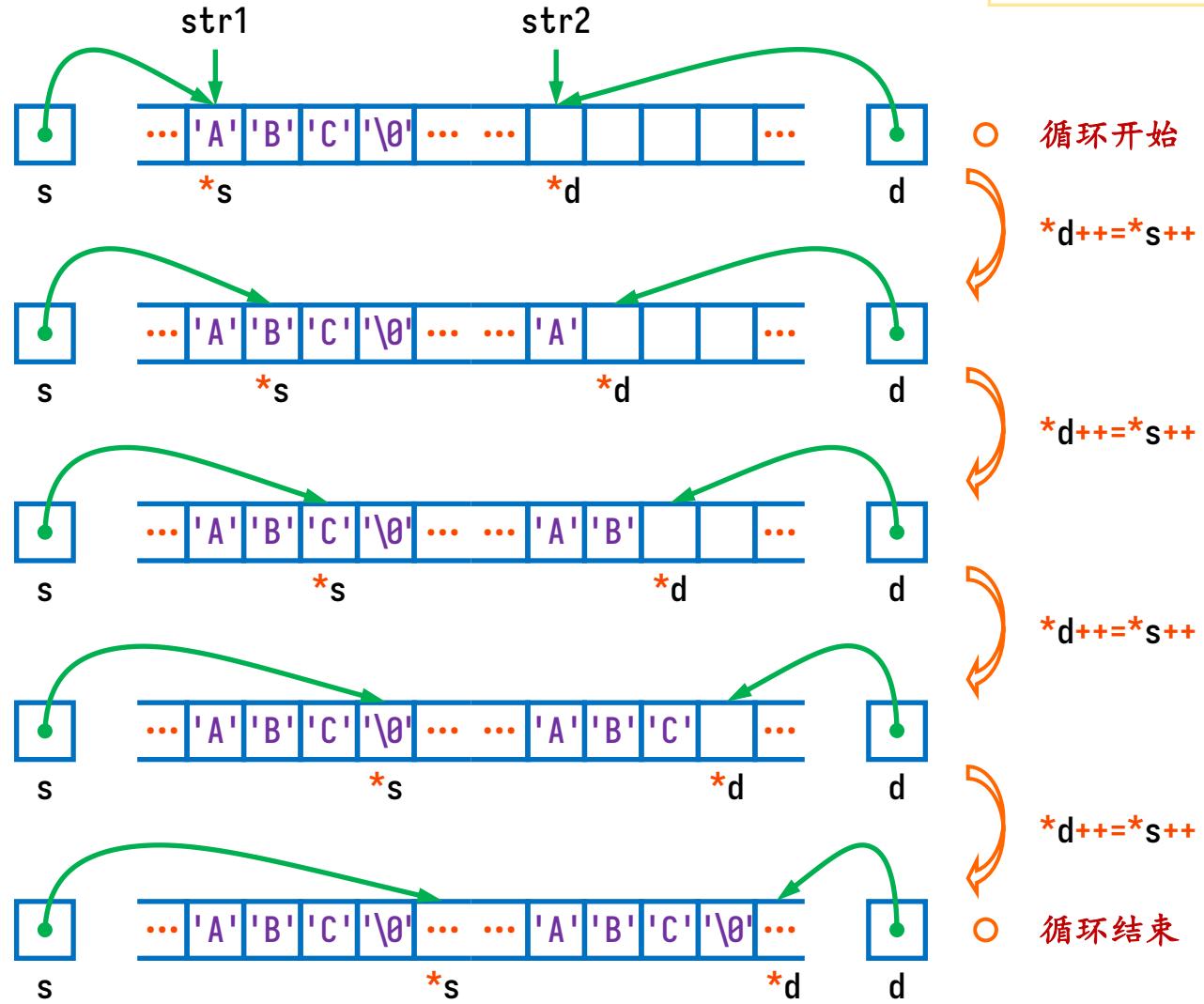
    return 0;
}
```

运行结果：
ABC

用函数处理字符串举例——复制字符串

■ 例5.2-13：编写函数，复制字符串。

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





用函数处理字符串举例——连接字符串

■ 例5.2-14：编写函数，连接字符串。

```
#include <stdio.h>

char *udf_strcat(char *d, char *s)      // 形参为指向字符串起始地址的指针
{
    char *t=d;

    while (*d)                          // 循环1：寻找串尾，与求字符串长度类似，但不计数
        d++;
    while (*d++ = *s++);                // 循环2：复制字符串，将源字符串复制到目标字符串尾部

    return t;
}

int main()
{
    char *str1="ABC";
    char str2[20]="XYZ";                // 字符指针，使用字符串常量起始地址初始化
                                         // 字符数组，目标字符串，应能够容纳连接后的字符串

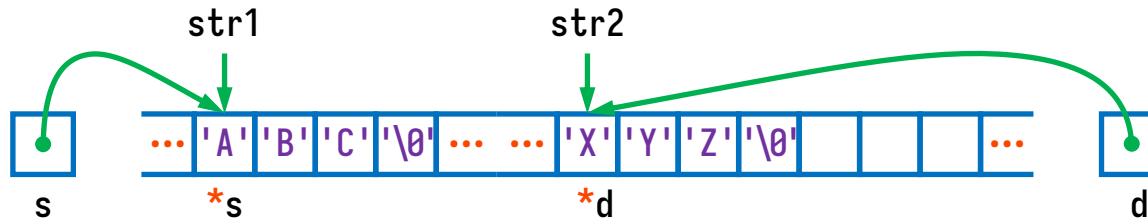
    udf_strcat(str2, str1);
    printf("%s\n", str2);

    return 0;
}
```

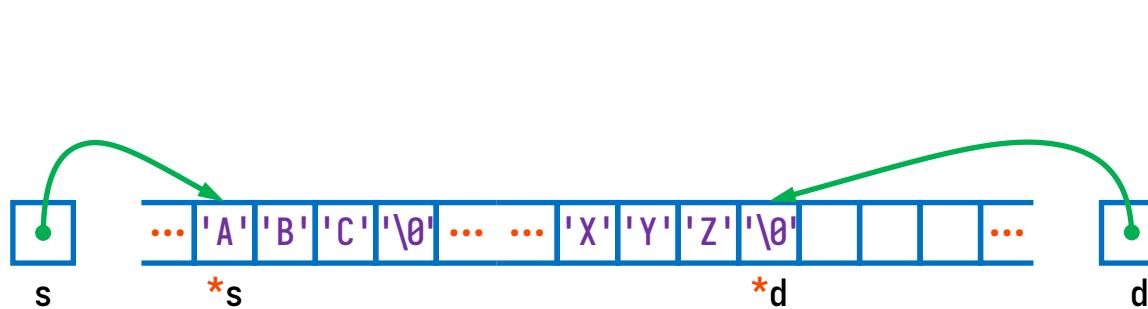
运行结果：
XYZABC

用函数处理字符串举例——连接字符串

■ 例5.2-14：编写函数，连接字符串。



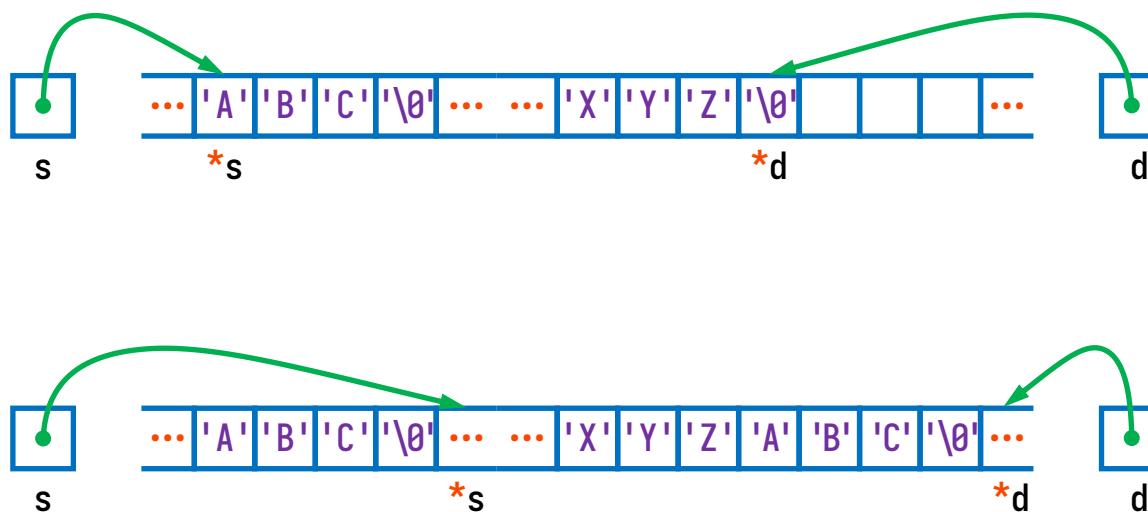
```
while (*d)
    d++;
while (*d++ = *s++);
```



○ 循环1开始

while (*d) d++;

○ 循环1结束
循环2开始



while (*d++= *s++);

○ 循环2结束



用函数处理字符串举例——比较字符串

■ 例5.2-15：编写函数，比较字符串。

```
#include <stdio.h>

int udf_strcmp(char *s, char *t)
{
    while (*s && *s==*t)           // 若s指向的字符串结束，或s和t指向的字符不等，循环结束
        s++, t++;

    return *s-*t;                  // 返回循环结束后s和t指向的字符ASCII差值
}

int main()
{
    int diff;
    char *str1="ABCD";           // 字符指针，使用字符串常量起始地址初始化
    char str2[20]="ABCE";         // 字符数组，使用字符串常量初始化

    diff = udf_strcmp(str1, str2);
    printf("The difference of str1 and str2 is %d.\n", diff);

    return 0;
}
```

运行结果：

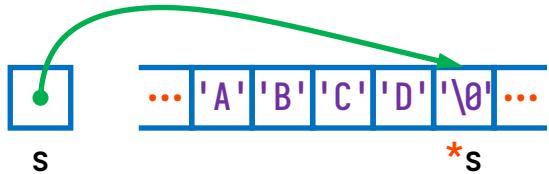
The difference of str1 and str2 is -1.

用函数处理字符串举例——比较字符串

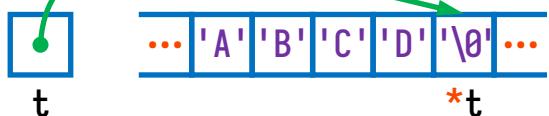


■ 例5.2-15：编写函数，比较字符串。

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

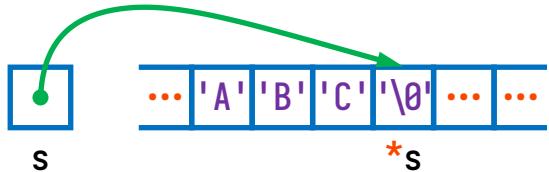


*s=='\0'&&*s==*t, "ABCD"等于"ABCD"

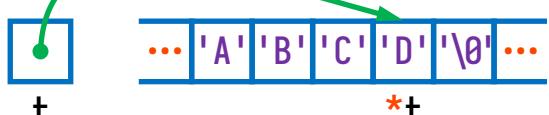


A diagram illustrating a linked list node. The node consists of a blue square containing a green dot, followed by the character 't'. An arrow points from the green dot to the first character 'A' in the string 't'.

*s != '\0' && *s < *t, "ABCD" 小于 "ABED"



*s=='\0' && *s<*t, "ABC"小于"ABCD"



A diagram showing a state transition in a Deterministic Finite Automaton (DFA). A blue square labeled 's' has a green curved arrow pointing to another blue square labeled 's*' (with an asterisk). Between these two states is a horizontal blue line containing several vertical boxes. The first box contains three dots (...), followed by four boxes labeled 'A', 'B', 'C', and 'D' respectively, and then another box containing a backslash character '\'. After the fourth box, there is another set of three dots (...).

A character array represented as a horizontal row of boxes. The first box contains a green dot. A blue arrow points from this box to the second box in the array. The second box contains the character 'A'. The third box contains 'B', the fourth contains 'C', the fifth contains the symbol '\theta', and the sixth contains 't'. Ellipses ('...') are placed before the first box and after the fifth box.

*s != '\0' && *s > *t, "ABCD" 大于 "ABC"



指针用于内存操作



- 结构体指针
- 文件处理
- 内存分配与链表



结构体类型

■ 结构体 (Structure)

- 一种构造数据类型
- 包含若干变量或数组，称为结构体的**成员**(Member)
- 用于描述基本数据类型或数组无法描述的复杂对象
- 结构体类型应“先声明，后使用”

■ 结构体类型的声明

```
struct strStudent {  
    long int lNum;                      // 学号, 长整型  
    char stName[20];                     // 姓名, 字符数组  
    char cGender;                       // 性别, 字符型  
    short int nAge;                     // 年龄, 短整型  
    float fScore;                       // 成绩, 浮点型  
};
```



结构体变量

■ 结构体变量的定义和初始化

```
struct strStudent stu1, stu2;      // 定义结构体变量  
struct strStudent stu3={20006003, "Zhang San", 'M', 18, 98.0}; // 初始化
```

■ 结构体变量的用法

- 相同类型的结构体变量之间可以直接赋值
- 函数的参数和返回值类型可以是结构体类型

```
stu2 = stu3;                      // 结构体变量赋值  
print(stu2);                      // 结构体变量用作函数实参
```

■ 结构体变量成员的引用

- 结构体变量使用**成员访问运算符.**访问其成员
- 结构体变量成员的用法与同类型的变量或数组相同

```
stu1.lNum = 20006001;              // 结构体变量成员赋值  
strcpy(stu1.stName, "Ding Yi"); // 结构体变量成员用作函数实参
```



结构体数组与指针

■ 结构体数组

```
struct strStudent class[40];           // 定义结构体数组  
struct strStudent stu[3]={ {20006001, "Ding Yi", 'F', 18, 98.0},    // 初始化  
                           {20006002, "Wang Wu", 'M', 19, 97.5},  
                           {20006003, "Zhao Si", 'M', 18, 95.0} };
```

■ 结构体指针

- 指向结构体类型数据的指针，表示结构体数据的起始地址
- 结构体指针可以指向相同结构体类型的变量或数组元素
- 函数的参数和返回值可以是结构体指针类型
- 结构体指针可以使用**指向成员运算符->**访问其指向的结构体数据的成员

```
struct strStudent *pstu;                // 定义结构体指针变量  
pstu = class;                          // 结构体数组首地址赋值给结构体指针变量  
pstu->lNum = 20006001;                 // 通过结构体指针变量访问结构体成员
```



结构体成员访问举例

■ 例5.3-1：结构体成员访问举例。

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

int main(void)
{
    struct strStudent {
        long int lNum;
        char stName[20];
        char cGender;
        float fScore;
    };

    struct strStudent stu1;
    struct strStudent *ps;

    stu1.lNum = 20006001; // stu1的成员赋值
    strcpy(stu1.stName, "Ding Yi");
    stu1.cGender = 'F';
    stu1.fScore = 99.0;
}
```

```
ps = &stu1;           // 指针ps指向stu1

printf("%ld, %s, %c, %5.1f\n",
       stu1.lNum, stu1.stName,
       stu1.cGender, stu1.fScore);
printf("%ld, %s, %c, %5.1f\n",
       (*ps).lNum, (*ps).stName,
       (*ps).cGender, (*ps).fScore);
printf("%ld, %s, %c, %5.1f\n",
       ps->lNum, ps->stName,
       ps->cGender, ps->fScore);

return 0;
}
```

运行结果：

```
20006001, Ding Yi, F, 99.0
20006001, Ding Yi, F, 99.0
20006001, Ding Yi, F, 99.0
```



结构体数组与指针举例

■ 例5.3-2：通过指针访问学生结构体数组的元素并输出。

```
#include <stdio.h>

struct strStudent {
    long int lNum;
    char    stName[20];
    char    cGender;
    float   fScore;
} student[4] = {{101, "Ding Yi", 'F', 88.0},
                {102, "Zhao Si", 'M', 78.0},
                {103, "Wang Wu", 'M', 95.5},
                {104, "Ruan Qi", 'F', 87.5}};

int main(void)
{
    struct strStudent *ps;

    printf("No.\tName\tGender\tScore\n");
    for (ps=student; ps<student+4; ps++)
        printf("%d\t%s\t%c\t%.1f\n", ps->lNum, ps->stName, ps->cGender, ps->fScore);

    return 0;
}
```

运行结果：

No.	Name	Gender	Score
101	Ding Yi	F	88.0
102	Zhao Si	M	78.0
103	Wang Wu	M	95.5
104	Ruan Qi	F	87.5



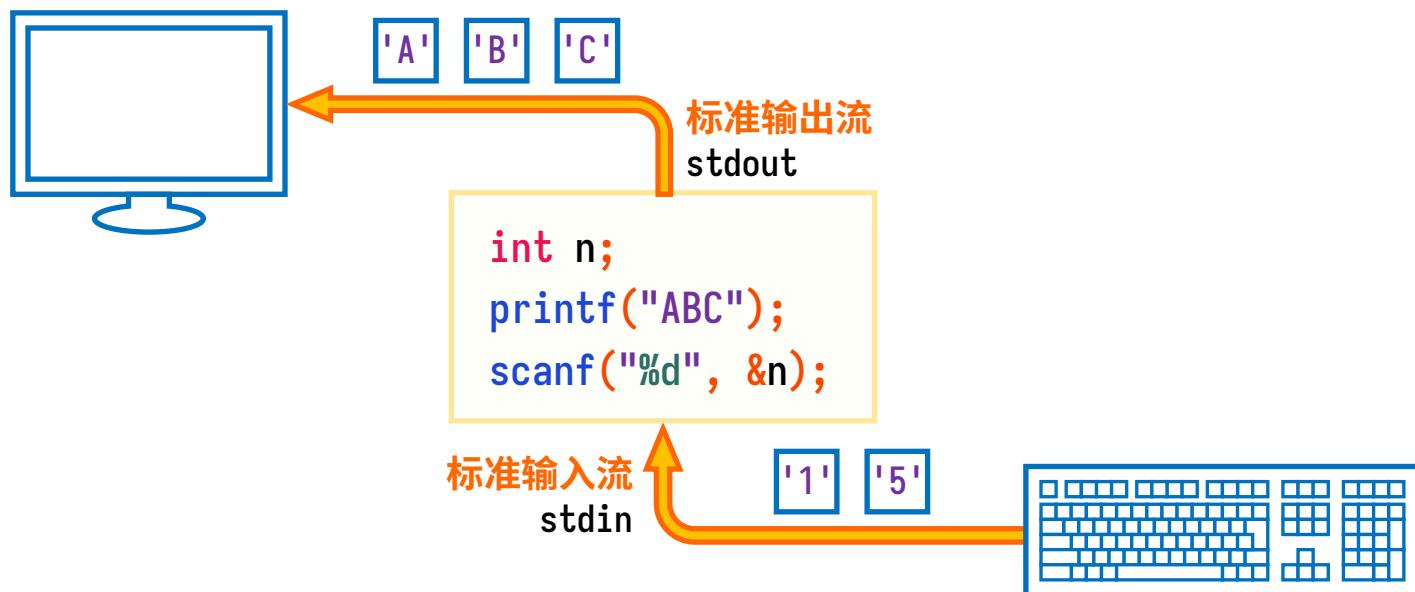
- 结构体指针
- 文件处理
- 内存分配与链表

■ 流 (Stream)

- 对磁盘文件以及外部设备的数据读写操作都通过流进行
- 不同物理性质的存储设备和输入输出设备都映射为流
- 编程时无需考虑不同设备的差异

■ 标准流

- 标准输入输出使用预定义标准流
- 可直接使用，无需打开关闭操作
- **stdin**：标准输入流，默认键盘
- **stdout**：标准输出流，默认屏幕
- **stderr**：标准错误流，默认屏幕





文件的类型

■ 文件的类型

- 磁盘文件、设备文件
- 文本文件、二进制文件
- 输入文件、输出文件
- 顺序存取文件、随机存取文件
- 缓冲文件系统、非缓冲文件系统

■ 文本文件 (Text File)

- 数据以字符序列形式表示
- 每个字节表示普通字符或换行符

■ 二进制文件 (Binary File)

- 数据以内存中的存储形式表示
- 长度取决于数据类型和编译器



数据在文本文件和二进制文件中的存储形式



文件指针

■ 文件类型

- 头文件stdio.h中定义的结构体类型，其类型别名为FILE
- 文件类型的成员包括控制流所需的信息，具体定义形式因编译器而异

```
typedef struct {  
    int          level;           // 缓冲区满空程度  
    unsigned     flags;          // 文件状态标志  
    char         fd;             // 文件描述符  
    unsigned char hold;         // 无缓冲则将字符退回到流  
    int          bsize;          // 缓冲区大小  
    unsigned char *buffer;       // 数据缓冲区指针  
    unsigned char *curp;         // 当前位置指针  
    unsigned      istemp;        // 临时文件指示器  
    short         token;          // 用于有效性检查  
} FILE;
```

■ 文件指针

- 文件指针的值来自文件打开函数的返回值，并用作其他文件处理函数的参数



文件打开函数

■ 函数原型

```
FILE *fopen(const char *filename, const char *mode);
```

■ 功能

- 打开路径为字符串**filename**的文件，并将该文件与流相关联

■ 参数

- **const char *filename** – 需要打开的文件路径
- **const char *mode** – 文件操作模式

■ 返回值

- 若文件打开成功，返回文件指针，用于控制打开的流
- 若文件打开失败，返回空指针**NULL**
- 应判断返回值是否为空指针**NULL**，检查文件打开操作是否成功



文件的操作模式

文本模式	二进制模式	操作方式描述
"r"	"rb"	只读，打开已有文件
"w"	"wb"	只写，打开或创建文件，若文件已存在，则文件长度清零
"a"	"ab"	追加，打开或创建文件，若文件已存在，则从文件末尾开始写入
"r+"	"r+b", "rb+"	读写，打开已有文件
"w+"	"w+b", "wb+"	读写，打开或创建文件，若文件已存在，则文件长度清零
"a+"	"a+b", "ab+"	读写，打开或创建文件，若文件已存在，则从文件末尾开始写入



文件关闭函数

■ 函数原型

```
int fclose(FILE *stream);
```

■ 功能

- 刷新stream指向的流，关闭与该流相关联的文件
- 为防止程序意外中止运行时丢失数据，应及时关闭不再使用的文件

■ 参数

- FILE *stream – 需要关闭的流

■ 返回值

- 若文件关闭成功，返回0
- 若文件关闭失败，返回EOF



文件的打开和关闭举例

■ 例5.3-4：文件的打开和关闭示例。

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

#define FILE_NAME "example.dat"

int main(void)
{
    FILE *fp;

    fp = fopen(FILE_NAME, "r");
    if (fp == NULL) {                                // 若文件打开失败
        printf("无法打开文件: %s\n", FILE_NAME);
        exit(EXIT_FAILURE);                          // 非正常退出程序
    } else
        printf("成功打开文件: %s\n", FILE_NAME);

    // 此处进行文件的其他操作

    fclose(fp);
    return 0;
}
```

运行结果：（文件打开成功）

成功打开文件: example.dat

运行结果：（文件打开失败）

无法打开文件: example.dat

文件的读写



读写操作	库函数	功能描述
格式化读写	<code>fscanf()</code> <code>fprintf()</code>	用于格式化文本文件的读写操作
数据块读写	<code>fread()</code> <code>fwrite()</code>	用于二进制文件的读写操作
字符读写	<code>fgetc()</code> <code>fputc()</code>	用于文本文件和二进制文件的逐个字节读写操作
字符串读写	<code>fgets()</code> <code>fputs()</code>	用于文本文件的字符串读写操作



文件的格式化读写函数

■ 函数原型

```
int fscanf(FILE *stream, const char *format, ...);  
int fprintf(FILE *stream, const char *format, ...);
```

■ 参数

- `FILE *stream` - 文件流指针
- `const char *format` - 指向格式字符串的指针
- `...` - 读参数地址列表/写参数列表

■ 返回值

- 格式化读：返回成功赋值的读参数数量
 - 若在首个读参数赋值前匹配失败，则返回`0`
 - 若在首个读参数赋值前输入失败，则返回`EOF`
- 格式化写：返回写的字符数
 - 若发生写错误，则返回负值



文件的格式化读写举例

■ 例5.3-5：分别计算学生各科总成绩，将数据写入文本文件，再从文件读出数据。

```
#include <stdio.h>

struct student {
    char name[16];
    float english, math, physics, total;
};

int main(void)
{
    int i;
    FILE *fp;
    struct student stu[]={{"Zhangsan",90,92,96,0}, {"Lihua",85,88,95,0}, {"Gaofei",95,86,92,0}};

    if (!(fp = fopen("student.dat", "w"))) { // 文本只写方式打开文件，并判断是否打开成功
        printf("Can't open file.");
        return 1;
    }

    for (i=0; i<3; i++) {
        stu[i].total = stu[i].english + stu[i].math + stu[i].physics;
        fprintf(fp, "%s %.2f %.2f %.2f %.2f\n", stu[i].name, stu[i].english, stu[i].math,
                stu[i].physics, stu[i].total); // 将数据以格式化文本方式写入文件
    }

    fclose(fp); // 关闭文件
}
```



文件的格式化读写举例

■ 例5.3-5：分别计算学生各科总成绩，将数据写入文本文件，再从文件读出数据。

// Continued.

```
if (!(fp = fopen("student.dat", "r"))) {      // 文本只读方式打开文件，并判断是否打开成功
    printf("Can't open file.");
    return 2;
}

for (i=0; i<3; i++) {
    fscanf(fp, "%s%f%f%f", stu[i].name, &stu[i].english, &stu[i].math,
           &stu[i].physics, &stu[i].total); // 从文件以格式化文本方式读取数据
    printf("%-16s%6.1f%6.1f%6.1f%7.1f\n", stu[i].name, stu[i].english, stu[i].math,
           stu[i].physics, stu[i].total); // 将数据以格式化文本方式输出到显示器
}

fclose(fp);                                // 关闭文件

return 0;
}
```

运行结果：

Zhangsan	90.0	92.0	96.0	278.0
Lihua	85.0	88.0	95.0	268.0
Gaofei	95.0	86.0	92.0	273.0

文件内容：(student.dat)

Zhangsan	90.00	92.00	96.00	278.00
Lihua	85.00	88.00	95.00	268.00
Gaofei	95.00	86.00	92.00	273.00



文件的数据块读写函数

■ 函数原型

```
int fread(const void *buffer, size_t size, size_t n, FILE *stream);  
int fwrite(const void *buffer, size_t size, size_t n, FILE *stream);
```

■ 参数

- `const void *buffer` – 读写的数据块在内存中的首地址
- `size_t size` – 读写的每个数据项字节数
- `size_t n` – 读写的数据项个数
- `FILE *stream` – 文件流指针

■ 返回值

- 数据块读：返回成功读的数据项个数
 - 若发生读错误或到达文件末尾，则返回值可能小于n
- 数据块写：返回成功写的数据项个数
 - 若发生写错误，则返回值可能小于n



文件的数据块读写举例

■ 例5.3-6：分别计算学生各科总成绩，将数据写入二进制文件，再从文件读出数据。

```
#include <stdio.h>

struct student {
    char name[16];
    float english, math, physics, total;
};

int main(void) {
    int i;
    FILE *fp;
    struct student stu[]={{{"Zhangsan",90,92,96,0}, {"Lihua",85,88,95,0}, {"Gaofei",95,86,92,0}};

    for (i=0; i<3; i++)
        stu[i].total = stu[i].english + stu[i].math + stu[i].physics;

    if (!(fp = fopen("student.dat", "wb"))){ // 二进制只写方式打开文件，并判断是否打开成功
        printf("Can't open file.");
        return 1;
    }

    fwrite(stu, sizeof(struct student), 3, fp); // 将数据以二进制方式写入文件
    fclose(fp); // 关闭文件
}
```



文件的数据块读写举例

■ 例5.3-6：分别计算学生各科总成绩，将数据写入二进制文件，再从文件读出数据。

```
// Continued.
```

```
if (!(fp = fopen("student.dat", "rb"))){ // 二进制只读方式打开文件，并判断是否打开成功
    printf("Can't open file.");
    return 1;
}

fread(stu, sizeof(struct student), 3, fp); // 从文件以二进制方式读取数据
fclose(fp); // 关闭文件

for (i=0; i<3; i++)
    printf("%-16s%6.1f%6.1f%6.1f%7.1f\n", stu[i].name, stu[i].english, stu[i].math,
           stu[i].physics, stu[i].total); // 将数据以格式化文本方式输出到显示器

return 0;
}
```

运行结果：

Zhangsan	90.0	92.0	96.0	278.0
Lihua	85.0	88.0	95.0	268.0
Gaofei	95.0	86.0	92.0	273.0

文件内容：(student.dat)

Zhangsan	張 翱 紫 婉	Lihua
	立 痘 緩 唻	Gaofei
	寧 € 國	緹 瑰



文件的字符读写

■ 字符读函数

```
int fgetc(FILE *stream);
```

- 从stream指向的流中读取一个字符
- 若读取成功，则返回读取的字符
- 若读取错误或到达文件末尾，则返回EOF

■ 字符写函数

```
int fputc(int ch, FILE *stream);
```

- 将ch的值转换为unsigned char类型并写入stream指向的流
- 若写入成功，则返回成功写入的字符
- 若写入错误，则返回EOF



文件的字符读写举例——文件复制

■ 例5.3-7：实现文件复制功能。

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

int main()
{
    FILE *fp1;
    FILE *fp2;
    int c;

    if (!(fp1 = fopen("file1", "rb")))
    {
        puts("File 1 opening failed.");
        return EXIT_FAILURE;
    }
```

```
if (!(fp2 = fopen("file2", "wb")))
{
    puts("File 2 opening failed.");
    return EXIT_FAILURE;
}

while ((c = fgetc(fp1)) != EOF)
    fputc(c, fp2);

fclose(fp1);
fclose(fp2);

return EXIT_SUCCESS;
}
```

输入数据：(文件file1内容)

Test!
C Programming Language.
USTC

运行结果：(文件file2内容)

Test!
C Programming Language.
USTC



文件的字符串读写

■ 字符串读函数

```
char *fgets(char *str, int count, FILE *stream);
```

- 从stream指向的流中读取最多count-1个字符，存储到str指向的字符数组中
 - 字符串末尾将自动添加结束标志'\0'
 - 若读取操作遇到换行符或文件结尾，则提前中止读取
 - 若读取操作遇到换行符，则换行符也被读入字符串
- 若读取成功，则返回str的值
- 若读取失败，则返回空指针NULL

■ 字符串写函数

```
int fputs(const char *str, FILE *stream);
```

- 将str指向的字符串写入stream指向的流，但不写入字符串结束标志'\0'
- 若写入成功，则返回非负值
- 若写入失败，则返回EOF



文件的字符串读写举例

■ 例5.3-8：向文本文件中写入若干行文字，再将这些文字读出并输出。

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

int main(void)
{
    FILE *fp;
    char buf[20];

    if (!(fp = fopen("tmpfile", "w+"))) {
        puts("File opening failed.");
        return EXIT_FAILURE;
    }
```

```
fputs("Alan Turing\n", fp);
fputs("John von Neumann\n", fp);
fputs("Alonzo Church\n", fp);

rewind(fp); // 文件位置指针移至文件开头
while (fgets(buf, sizeof(buf), fp))
    printf("%s", buf);
```

```
fclose(fp);

return 0;
}
```

运行结果：

Alan Turing
John von Neumann
Alonzo Church

文件内容：(tmpfile)

Alan Turing
John von Neumann
Alonzo Church



判断文件结束

■ 检查文件流结束函数

```
int feof(FILE *stream);
```

- 检查是否已到达文件流结尾
- 若已到达文件流结尾，则返回非零值（逻辑“真”）
- 若未到达文件流结尾，则返回0（逻辑“假”）

■ 检查文件流错误函数

```
int ferror(FILE *stream);
```

- 检查文件流是否出现错误
- 若文件流已出现错误，则返回非零值（逻辑“真”）
- 若文件流未出现错误，则返回0（逻辑“假”）



- 结构体指针
- 文件处理
- 内存分配与链表



动态存储分配

■ 动态存储分配

- 在程序运行期间，根据需要分配和释放存储空间
- 动态存储空间的分配和释放通过调用库函数完成
- 动态存储空间不会随着函数返回而自动释放，因此在其他函数中可继续使用
- 动态存储空间不再需要使用时，应及时释放

■ 动态数组

■ 一维动态数组

- 动态存储空间在内存中是连续的，与数组的存储方式相同
- 可以按照指向数组首地址的指针的方式进行数据访问

■ 二维动态数组

- 若干一维动态数组的首地址，分别赋值给一维动态指针数组的各个元素
- 可以按照指向指针数组首地址的指向指针的指针的方式进行数据访问



动态存储分配和释放函数

■ 动态存储分配函数

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

- `malloc()` 函数分配 `size` 字节的动态存储空间
- `calloc()` 函数分配 `n*size` 字节的动态存储空间
- 若分配成功，返回所分配存储空间的首地址
- 若分配失败，返回空指针 `NULL`

■ 动态存储释放函数

```
void free(void *ptr);
```

- 释放 `ptr` 指向的动态分配存储空间
- 实参 `ptr` 仍指向已被释放的存储空间，成为悬空指针

二维动态数组举例

■ 例5.3-9：二维动态数组的创建和使用。

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

void MakeArray(int **v, int m, int n);
void PrintArray(int **v, int m, int n);

int main()
{
    int i, row, col, **pb;

    printf("Input row and column numbers: ");
    scanf("%d%d", &row, &col);

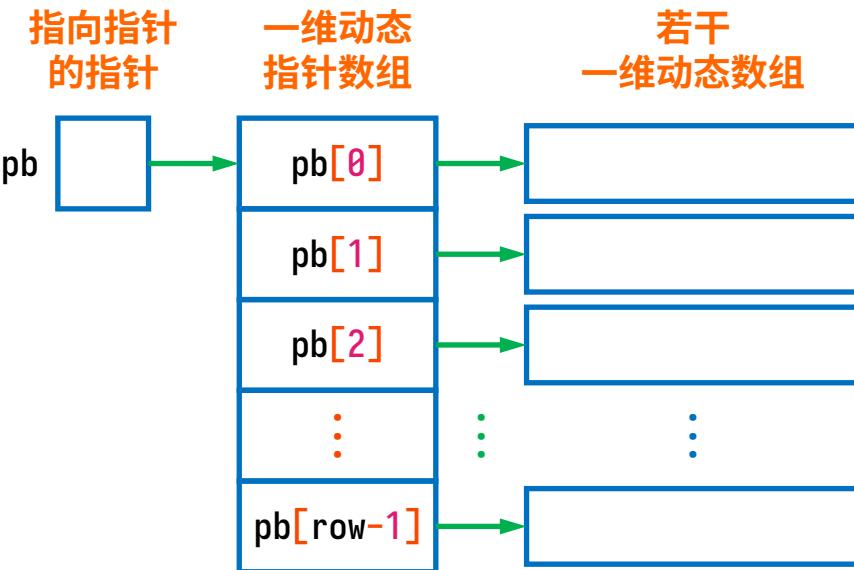
    pb = (int **)malloc(row*sizeof(int *));
    if (pb == NULL) {
        printf("Not enough memory!");
        exit(1);
    }

    for (i=0; i<row; i++) {
        pb[i] = (int *)malloc(col*sizeof(int));
        if (!pb[i]) {
            printf("Not enough memory!");
            exit(1);
        }
    }
}
```

```
MakeArray(pb, row, col);
PrintArray(pb, row, col);

for (i=0; i<row; i++)
    free(pb[i]);
free(pb);

return 0;
}
```





二维动态数组举例

■ 例5.3-9：二维动态数组的创建和使用。

```
// Continued.
```

```
void MakeArray(int **v, int m, int n)
{
    int i, j;

    srand(time(NULL));
    for (i=0; i<m; i++)
        for (j=0; j<n; j++)
            v[i][j] = rand() % 100;
}
```

```
void PrintArray(int **v, int m, int n)
{
    int i, j;

    for (i=0; i<m; i++) {
        for (j=0; j<n; j++)
            printf("%6d", v[i][j]);
        printf("\n");
    }
}
```

运行结果一：

```
Input row and column numbers: 3 4<
    47    84    20    55
    83    30    33    50
    48    87    53    54
```

运行结果二：

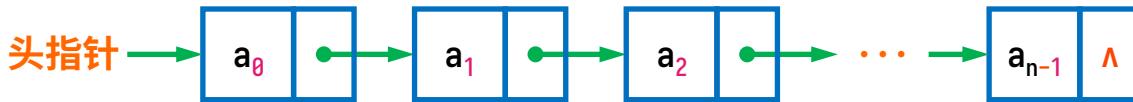
```
Input row and column numbers: 3 5<
    81    85    87    11    96
    86    71     3    95    92
    88    60    61    34    45
```



数据域 指针域

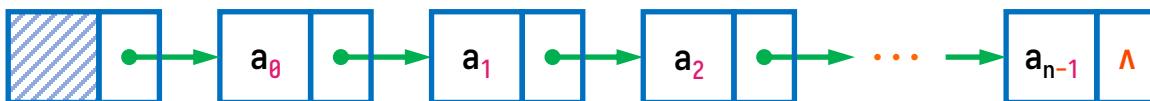
```
struct node {  
    int num;           // 数据域  
    struct node *next; // 指针域  
};
```

单链表结点结构



无头结点单链表

头结点



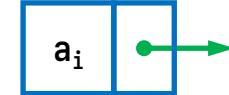
带头结点单链表



链表结点类型

■ 结点结构体类型

```
struct node {  
    int         num;           // 数据域  
    struct node *next;        // 指针域  
};
```



数据域 指针域

■ 链表的初始化

- 链表用头指针表示，初始状态一般为空指针，以避免野指针操作

```
struct node *head = NULL; // 初始化为空指针，表示链表初始状态为空表
```

■ 创建新的结点

```
struct node *p;           // 定义结点指针变量  
p = (struct node *)malloc(sizeof(struct node)); // 动态分配结点存储空间  
scanf("%d", &p->num);          // 输入结点数据域  
p->next = NULL;           // 避免野指针操作
```



链表的基本操作——头插法建立链表

■ 例5.3-10：头插法建立链表。

```
struct node *CreateListF(void)
{
    struct node *head, *p;
    int         num;

    head = NULL;
    printf("Input an Integer: ");
    scanf("%d", &num);

    while (num != 0) {
        p = (struct node *)malloc(sizeof(struct node));
        p->num = num;
        p->next = head;
        head = p;
        printf("Input an Integer: ");
        scanf("%d", &num);
    }

    return head;
}
```

链表的基本操作——头插法建立链表

■ 例5.3-10：头插法建立链表。

```
struct node *CreateListF(void)
{
    struct node *head, *p;
    int num;

    head = NULL;
    printf("Input an Integer: ");
    scanf("%d", &num);

    while (num != 0) {
        p = (struct node *)malloc(sizeof(struct node));
        p->num = num;
        p->next = head;
        head = p;
        printf("Input an Integer: ");
        scanf("%d", &num);
    }

    return head;
}
```



链表的基本操作——头插法建立链表

■ 例5.3-10：头插法建立链表。

```

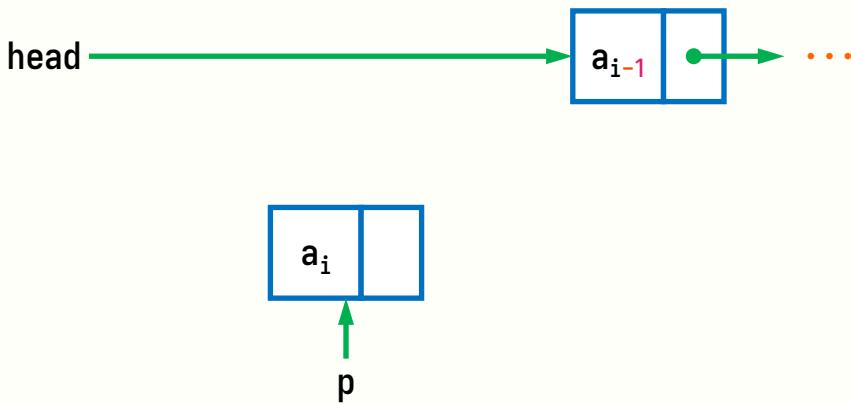
struct node *CreateListF(void)
{
    struct node *head, *p;
    int num;

    head = NULL;
    printf("Input an Integer: ");
    scanf("%d", &num);

    while (num != 0) {
        p = (struct node *)malloc(sizeof(struct node));
        p->num = num;
        p->next = head;
        head = p;
        printf("Input an Integer: ");
        scanf("%d", &num);
    }

    return head;
}

```



链表的基本操作——头插法建立链表

■ 例5.3-10：头插法建立链表。

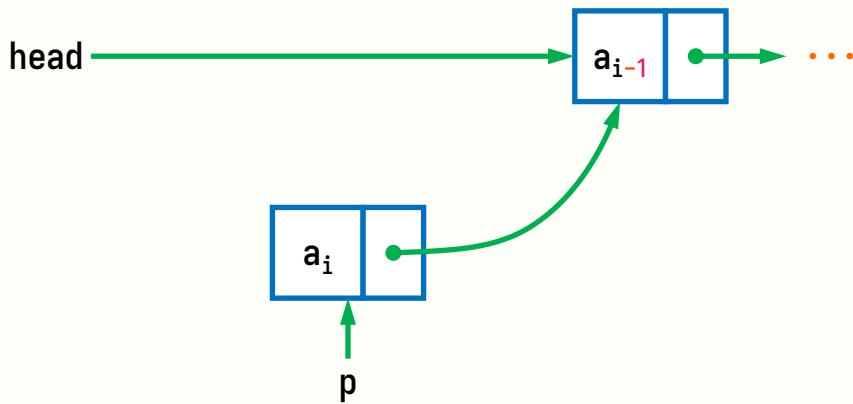
```

struct node *CreateListF(void)
{
    struct node *head, *p;
    int num;

    head = NULL;
    printf("Input an Integer: ");
    scanf("%d", &num);

    while (num != 0) {
        p = (struct node *)malloc(sizeof(struct node));
        p->num = num;
        p->next = head;
        head = p;
        printf("Input an Integer: ");
        scanf("%d", &num);
    }

    return head;
}
  
```



链表的基本操作——头插法建立链表

■ 例5.3-10：头插法建立链表。

```

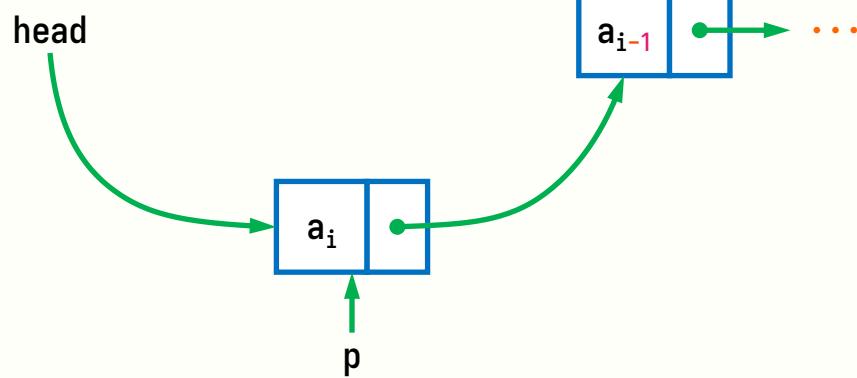
struct node *CreateListF(void)
{
    struct node *head, *p;
    int num;

    head = NULL;
    printf("Input an Integer: ");
    scanf("%d", &num);

    while (num != 0) {
        p = (struct node *)malloc(sizeof(struct node));
        p->num = num;
        p->next = head;
        head = p;
        printf("Input an Integer: ");
        scanf("%d", &num);
    }

    return head;
}

```



链表的基本操作——头插法建立链表

■ 例5.3-10：头插法建立链表。

```

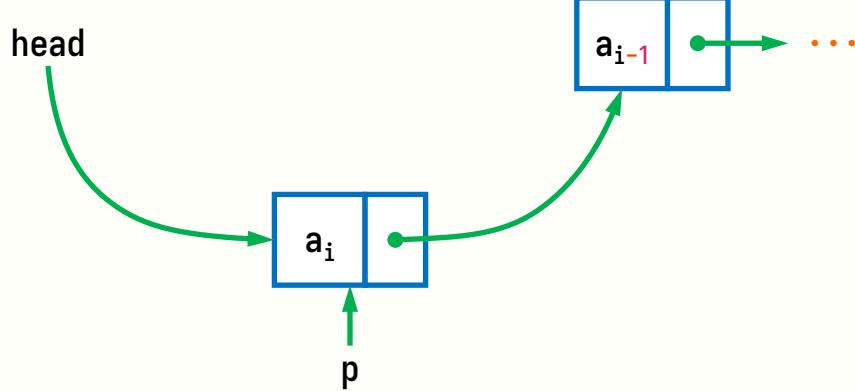
struct node *CreateListF(void)
{
    struct node *head, *p;
    int num;

    head = NULL;
    printf("Input an Integer: ");
    scanf("%d", &num);

    while (num != 0) {
        p = (struct node *)malloc(sizeof(struct node));
        p->num = num;
        p->next = head;
        head = p;
        printf("Input an Integer: ");
        scanf("%d", &num);
    }

    return head;
}

```





链表的基本操作——尾插法建立链表

■ 例5.3-11：尾插法建立链表。

```
struct node *CreateListR(void)
{
    struct node *head, *rear, *p;
    int         num;

    head = NULL;
    printf("Input an Integer: ");
    scanf("%d", &num);

    while (num != 0) {
        p = (struct node *)malloc(sizeof(struct node));
        p->next = NULL;
        p->num = num;
        if (!head)
            head = p;
        else
            rear->next = p;
        rear = p;
        printf("Input an Integer: ");
        scanf("%d", &num);
    }

    return head;
}
```



链表的基本操作——尾插法建立链表

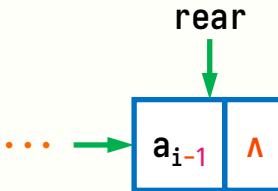
■ 例5.3-11：尾插法建立链表。

```
struct node *CreateListR(void)
{
    struct node *head, *rear, *p;
    int num;

    head = NULL;
    printf("Input an Integer: ");
    scanf("%d", &num);

    while (num != 0) {
        p = (struct node *)malloc(sizeof(struct node));
        p->next = NULL;
        p->num = num;
        if (!head)
            head = p;
        else
            rear->next = p;
        rear = p;
        printf("Input an Integer: ");
        scanf("%d", &num);
    }

    return head;
}
```



链表的基本操作——尾插法建立链表

■ 例5.3-11：尾插法建立链表。

```

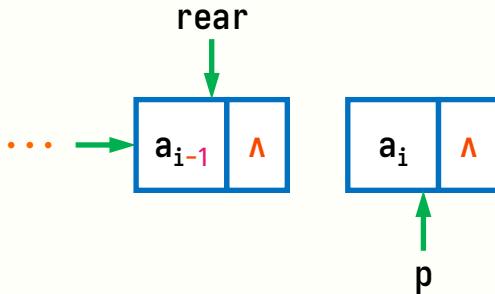
struct node *CreateListR(void)
{
    struct node *head, *rear, *p;
    int num;

    head = NULL;
    printf("Input an Integer: ");
    scanf("%d", &num);

    while (num != 0) {
        p = (struct node *)malloc(sizeof(struct node));
        p->next = NULL;
        p->num = num;
        if (!head)
            head = p;
        else
            rear->next = p;
        rear = p;
        printf("Input an Integer: ");
        scanf("%d", &num);
    }

    return head;
}

```



链表的基本操作——尾插法建立链表

■ 例5.3-11：尾插法建立链表。

```

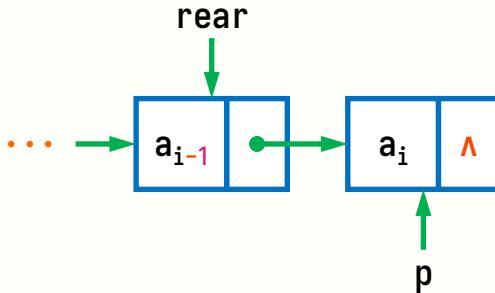
struct node *CreateListR(void)
{
    struct node *head, *rear, *p;
    int num;

    head = NULL;
    printf("Input an Integer: ");
    scanf("%d", &num);

    while (num != 0) {
        p = (struct node *)malloc(sizeof(struct node));
        p->next = NULL;
        p->num = num;
        if (!head)
            head = p;
        else
            rear->next = p;
        rear = p;
        printf("Input an Integer: ");
        scanf("%d", &num);
    }

    return head;
}

```



链表的基本操作——尾插法建立链表

■ 例5.3-11：尾插法建立链表。

```

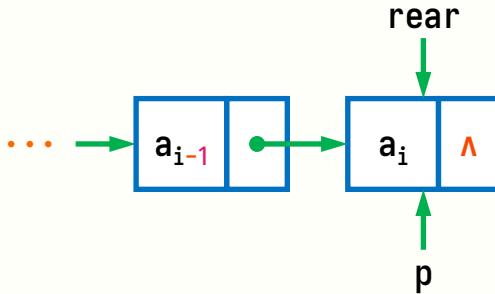
struct node *CreateListR(void)
{
    struct node *head, *rear, *p;
    int num;

    head = NULL;
    printf("Input an Integer: ");
    scanf("%d", &num);

    while (num != 0) {
        p = (struct node *)malloc(sizeof(struct node));
        p->next = NULL;
        p->num = num;
        if (!head)
            head = p;
        else
            rear->next = p;
        rear = p;
        printf("Input an Integer: ");
        scanf("%d", &num);
    }

    return head;
}

```



链表的基本操作——尾插法建立链表

■ 例5.3-11：尾插法建立链表。

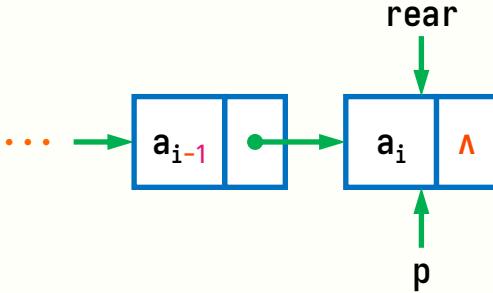
```

struct node *CreateListR(void)
{
    struct node *head, *rear, *p;
    int num;

    head = NULL;
    printf("Input an Integer: ");
    scanf("%d", &num);

    while (num != 0) {
        p = (struct node *)malloc(sizeof(struct node));
        p->next = NULL;
        p->num = num;
        if (!head)
            head = p;
        else
            rear->next = p;
        rear = p;
        printf("Input an Integer: ");
        scanf("%d", &num);
    }

    return head;
}
    
```





链表的基本操作——遍历链表

■ 例5.3-12：遍历链表。

```
void PrintList(struct node *head)
{
    struct node *p=head;

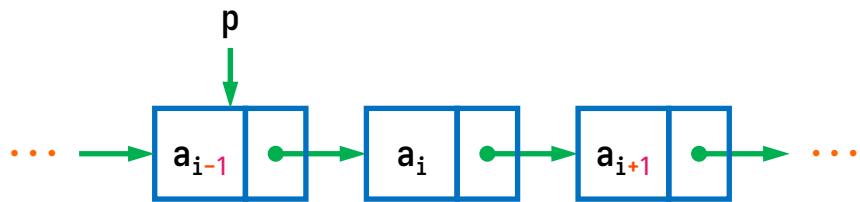
    while (p) {
        printf("%d\n", p->num);
        p = p->next;
    }
}
```

链表的基本操作——遍历链表

■ 例5.3-12：遍历链表。

```
void PrintList(struct node *head)
{
    struct node *p=head;

    while (p) {
        printf("%d\n", p->num);
        p = p->next;
    }
}
```

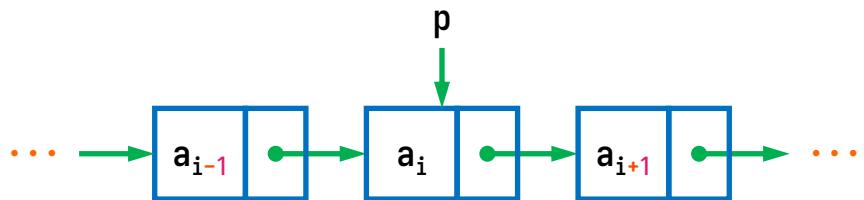


链表的基本操作——遍历链表

■ 例5.3-12：遍历链表。

```
void PrintList(struct node *head)
{
    struct node *p=head;

    while (p) {
        printf("%d\n", p->num);
        p = p->next;
    }
}
```

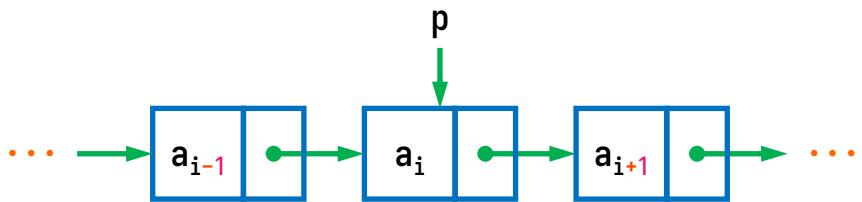


链表的基本操作——遍历链表

■ 例5.3-12：遍历链表。

```
void PrintList(struct node *head)
{
    struct node *p=head;

    while (p) {
        printf("%d\n", p->num);
        p = p->next;
    }
}
```

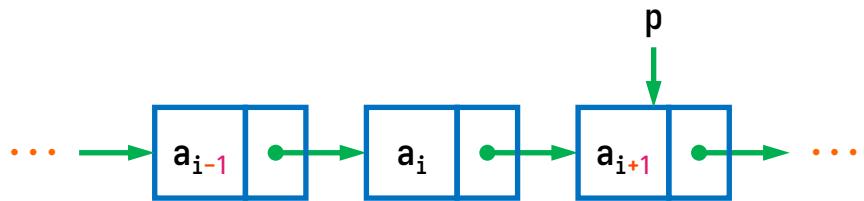


链表的基本操作——遍历链表

■ 例5.3-12：遍历链表。

```
void PrintList(struct node *head)
{
    struct node *p=head;

    while (p) {
        printf("%d\n", p->num);
        p = p->next;
    }
}
```

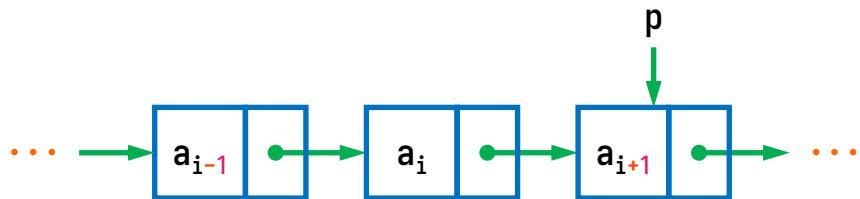


链表的基本操作——遍历链表

■ 例5.3-12：遍历链表。

```
void PrintList(struct node *head)
{
    struct node *p=head;

    while (p) {
        printf("%d\n", p->num);
        p = p->next;
    }
}
```





链表的基本操作——插入结点

■ 例5.3-13：插入链表结点，将新结点插入升序链表中，并保持升序状态。

```
struct node *InsertList(struct node *head, struct node *q)
{
    struct node *p;

    if (!head) {
        q->next = NULL;
        head = q;
        return head;
    }

    if (head->num > q->num) {
        q->next = head;
        head = q;
        return head;
    }

    p = head;
    while (p->next && p->next->num < q->num)
        p = p->next;

    q->next = p->next;
    p->next = q;

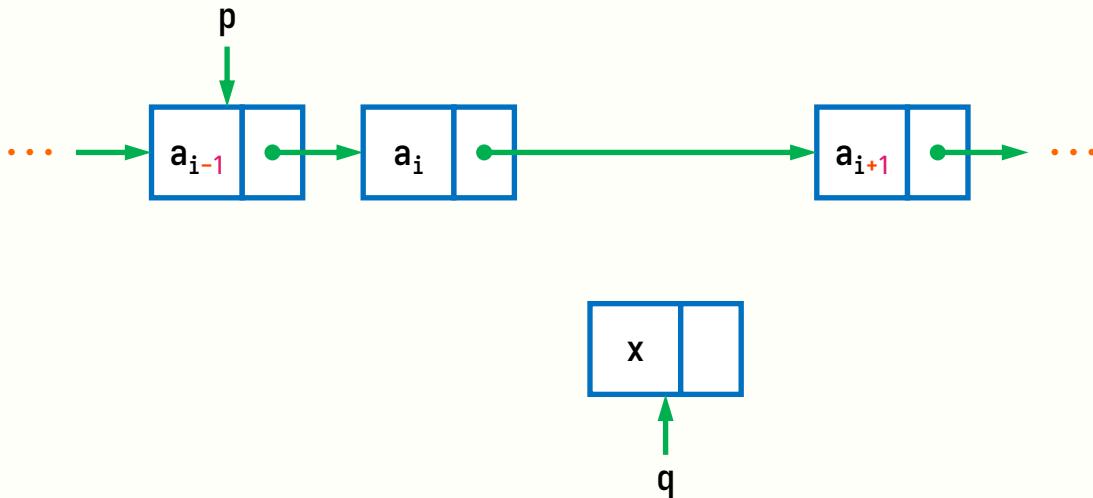
    return head;
}
```

链表的基本操作——插入结点

■ 例5.3-13：插入链表结点，将新结点插入升序链表中，并保持升序状态。

```

struct node *InsertList(struct node *head, struct node *q)
{
    struct node *p;
    if (!head) {
        q->next = NULL;
        head = q;
        return head;
    }
    if (head->num > q->num) {
        q->next = head;
        head = q;
        return head;
    }
    p = head;
    while (p->next && p->next->num < q->num)
        p = p->next;
    q->next = p->next;
    p->next = q;
    return head;
}
    
```



链表的基本操作——插入结点

■ 例5.3-13：插入链表结点，将新结点插入升序链表中，并保持升序状态。

```

struct node *InsertList(struct node *head, struct node *q)
{
    struct node *p;

    if (!head) {
        q->next = NULL;
        head = q;
        return head;
    }

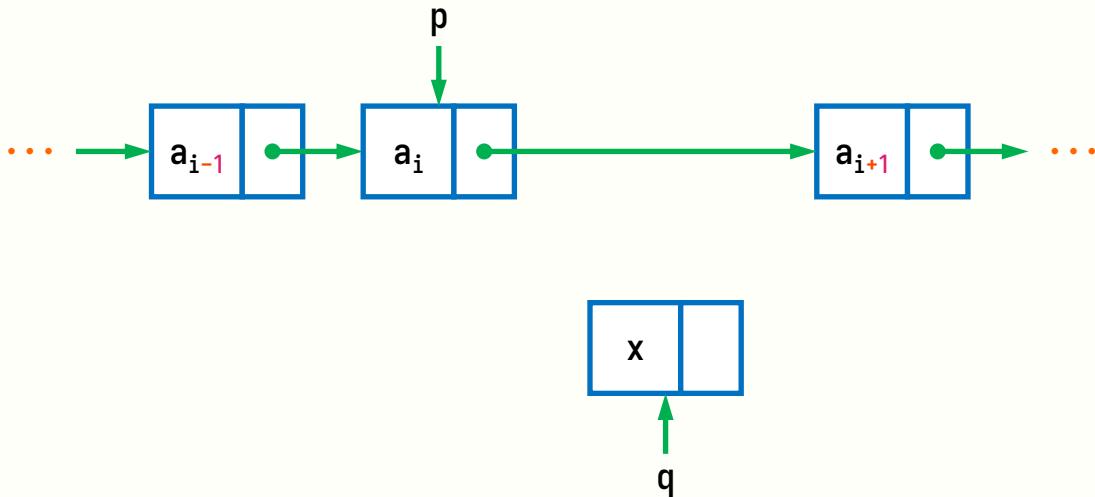
    if (head->num > q->num) {
        q->next = head;
        head = q;
        return head;
    }

    p = head;
    while (p->next && p->next->num < q->num)
        p = p->next;

    q->next = p->next;
    p->next = q;

    return head;
}

```

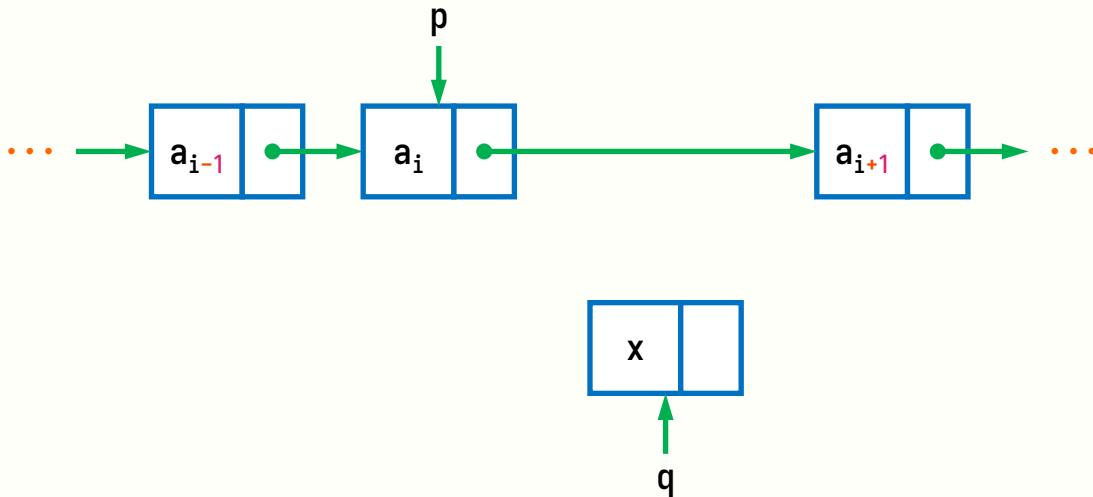


链表的基本操作——插入结点

■ 例5.3-13：插入链表结点，将新结点插入升序链表中，并保持升序状态。

```

struct node *InsertList(struct node *head, struct node *q)
{
    struct node *p;
    if (!head) {
        q->next = NULL;
        head = q;
        return head;
    }
    if (head->num > q->num) {
        q->next = head;
        head = q;
        return head;
    }
    p = head;
    while (p->next && p->next->num < q->num)
        p = p->next;
    q->next = p->next;
    p->next = q;
    return head;
}
    
```



链表的基本操作——插入结点

■ 例5.3-13：插入链表结点，将新结点插入升序链表中，并保持升序状态。

```

struct node *InsertList(struct node *head, struct node *q)
{
    struct node *p;

    if (!head) {
        q->next = NULL;
        head = q;
        return head;
    }

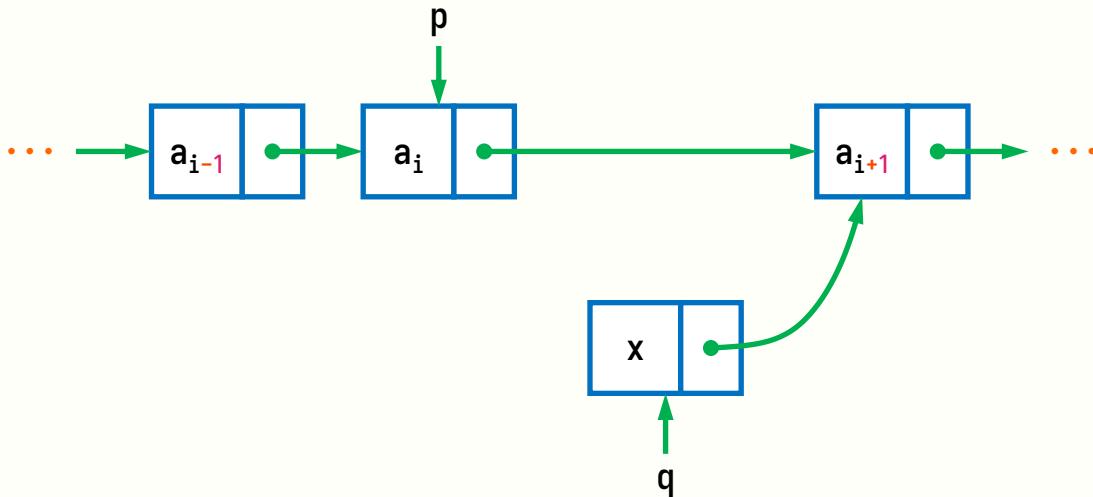
    if (head->num > q->num) {
        q->next = head;
        head = q;
        return head;
    }

    p = head;
    while (p->next && p->next->num < q->num)
        p = p->next;

    q->next = p->next;
    p->next = q;

    return head;
}

```



链表的基本操作——插入结点

■ 例5.3-13：插入链表结点，将新结点插入升序链表中，并保持升序状态。

```

struct node *InsertList(struct node *head, struct node *q)
{
    struct node *p;

    if (!head) {
        q->next = NULL;
        head = q;
        return head;
    }

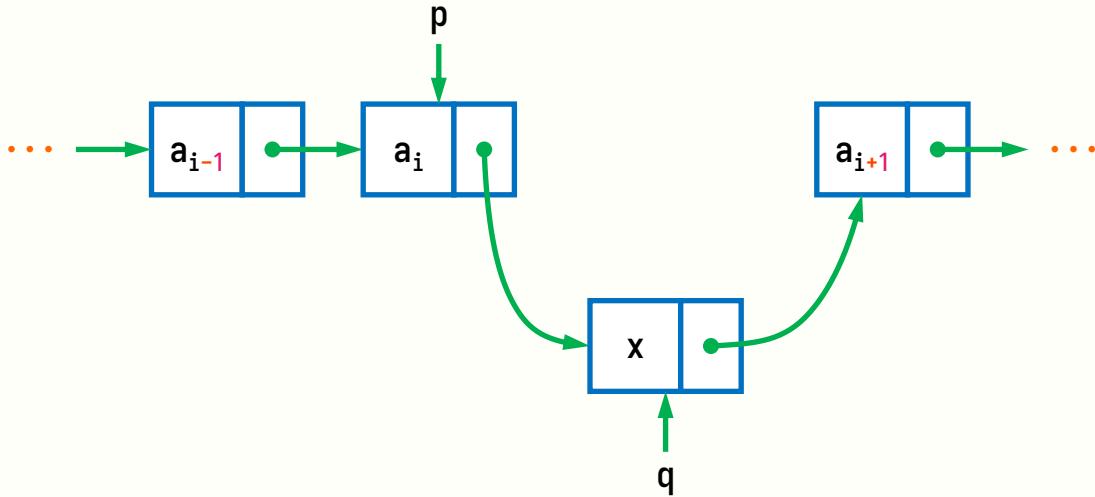
    if (head->num > q->num) {
        q->next = head;
        head = q;
        return head;
    }

    p = head;
    while (p->next && p->next->num < q->num)
        p = p->next;

    q->next = p->next;
    p->next = q;

    return head;
}

```



链表的基本操作——插入结点

■ 例5.3-13：插入链表结点，将新结点插入升序链表中，并保持升序状态。

```

struct node *InsertList(struct node *head, struct node *q)
{
    struct node *p;

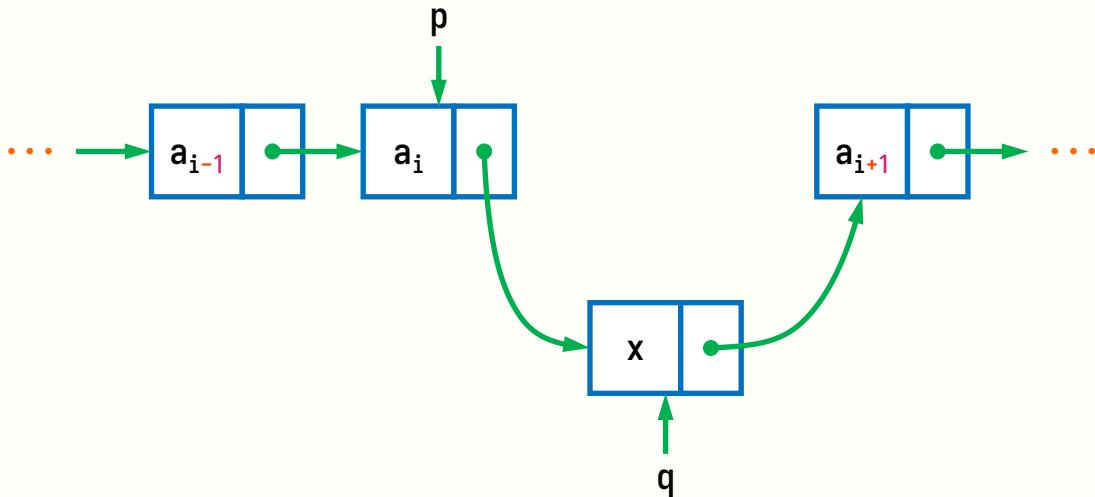
    if (!head) {
        q->next = NULL;
        head = q;
        return head;
    }

    if (head->num > q->num) {
        q->next = head;
        head = q;
        return head;
    }

    p = head;
    while (p->next && p->next->num < q->num)
        p = p->next;

    q->next = p->next;
    p->next = q;

    return head;
}
    
```





链表的基本操作——删除结点

■ 例5.3-14：删除链表结点，若结点数据域无重复值，删除数据域为指定值的结点。

```
struct node *DeleteList(struct node *head, int num)
{
    struct node *p, *q;

    if (head && head->num==num) {
        p = head;
        head = p->next;
        free(p);
        return head;
    }

    q = p = head;
    while (p && p->num!=num) {
        q = p;
        p = p->next;
    }

    if (p) {
        q->next = p->next;
        free(p);
        return head;
    }

    printf("Not Found!\n");
    return head;
}
```

链表的基本操作——删除结点

■ 例5.3-14：删除链表结点，若结点数据域无重复值，删除数据域为指定值的结点。

```

struct node *DeleteList(struct node *head, int num)
{
    struct node *p, *q;

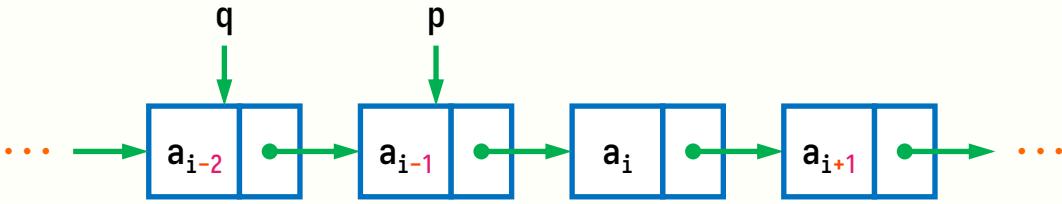
    if (head && head->num==num) {
        p = head;
        head = p->next;
        free(p);
        return head;
    }

    q = p = head;
    while (p && p->num!=num) {
        q = p;
        p = p->next;
    }

    if (p) {
        q->next = p->next;
        free(p);
        return head;
    }

    printf("Not Found!\n");
    return head;
}

```



链表的基本操作——删除结点

■ 例5.3-14：删除链表结点，若结点数据域无重复值，删除数据域为指定值的结点。

```

struct node *DeleteList(struct node *head, int num)
{
    struct node *p, *q;

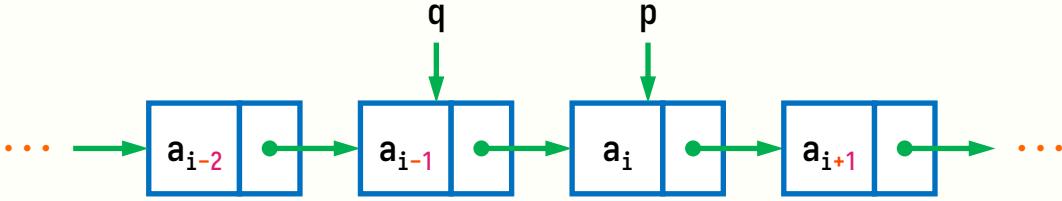
    if (head && head->num==num) {
        p = head;
        head = p->next;
        free(p);
        return head;
    }

    q = p = head;
    while (p && p->num!=num) {
        q = p;
        p = p->next;
    }

    if (p) {
        q->next = p->next;
        free(p);
        return head;
    }

    printf("Not Found!\n");
    return head;
}

```



链表的基本操作——删除结点

■ 例5.3-14：删除链表结点，若结点数据域无重复值，删除数据域为指定值的结点。

```

struct node *DeleteList(struct node *head, int num)
{
    struct node *p, *q;

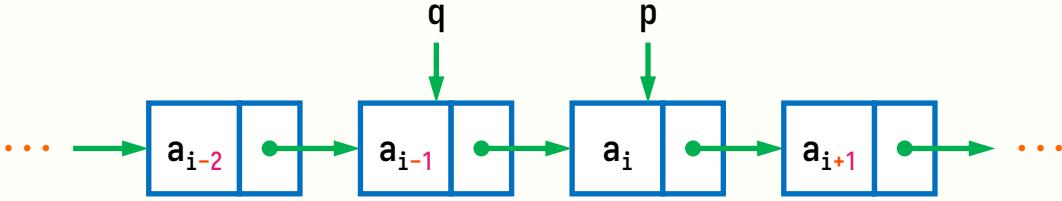
    if (head && head->num==num) {
        p = head;
        head = p->next;
        free(p);
        return head;
    }

    q = p = head;
    while (p && p->num!=num) {
        q = p;
        p = p->next;
    }

    if (p) {
        q->next = p->next;
        free(p);
        return head;
    }

    printf("Not Found!\n");
    return head;
}

```



链表的基本操作——删除结点

■ 例5.3-14：删除链表结点，若结点数据域无重复值，删除数据域为指定值的结点。

```

struct node *DeleteList(struct node *head, int num)
{
    struct node *p, *q;

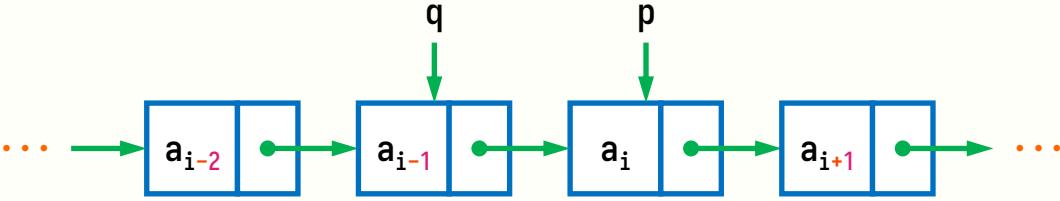
    if (head && head->num==num) {
        p = head;
        head = p->next;
        free(p);
        return head;
    }

    q = p = head;
    while (p && p->num!=num) {
        q = p;
        p = p->next;
    }

    if (p) {
        q->next = p->next;
        free(p);
        return head;
    }

    printf("Not Found!\n");
    return head;
}

```



链表的基本操作——删除结点

■ 例5.3-14：删除链表结点，若结点数据域无重复值，删除数据域为指定值的结点。

```

struct node *DeleteList(struct node *head, int num)
{
    struct node *p, *q;

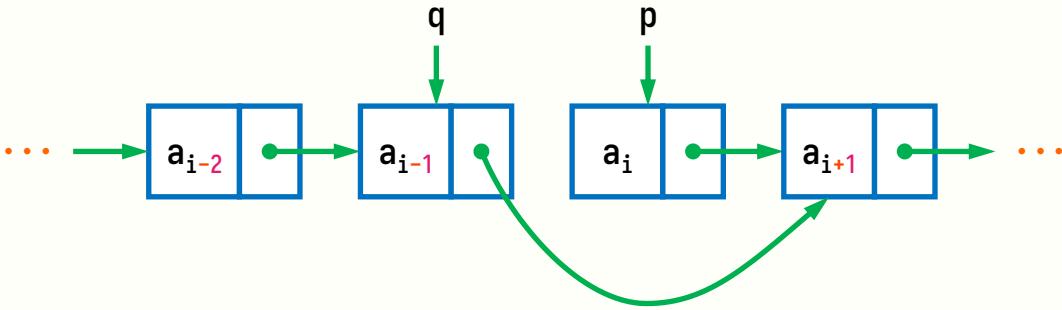
    if (head && head->num==num) {
        p = head;
        head = p->next;
        free(p);
        return head;
    }

    q = p = head;
    while (p && p->num!=num) {
        q = p;
        p = p->next;
    }

    if (p) {
        q->next = p->next;
        free(p);
        return head;
    }

    printf("Not Found!\n");
    return head;
}

```



链表的基本操作——删除结点

■ 例5.3-14：删除链表结点，若结点数据域无重复值，删除数据域为指定值的结点。

```

struct node *DeleteList(struct node *head, int num)
{
    struct node *p, *q;

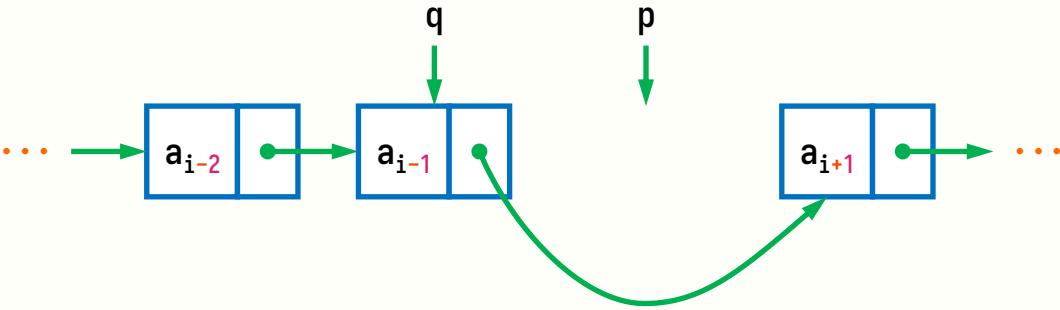
    if (head && head->num==num) {
        p = head;
        head = p->next;
        free(p);
        return head;
    }

    q = p = head;
    while (p && p->num!=num) {
        q = p;
        p = p->next;
    }

    if (p) {
        q->next = p->next;
        free(p);
        return head;
    }

    printf("Not Found!\n");
    return head;
}

```



链表的基本操作——删除结点

■ 例5.3-14：删除链表结点，若结点数据域无重复值，删除数据域为指定值的结点。

```

struct node *DeleteList(struct node *head, int num)
{
    struct node *p, *q;

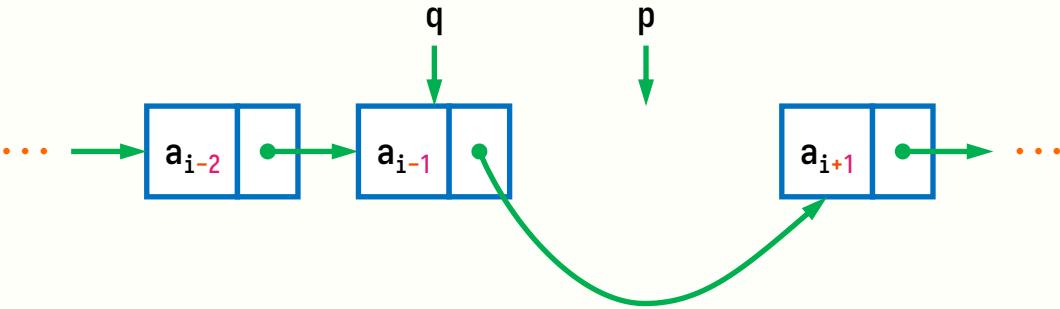
    if (head && head->num==num) {
        p = head;
        head = p->next;
        free(p);
        return head;
    }

    q = p = head;
    while (p && p->num!=num) {
        q = p;
        p = p->next;
    }

    if (p) {
        q->next = p->next;
        free(p);
        return head;
    }

    printf("Not Found!\n");
    return head;
}

```





链表的基本操作——主函数

■ 例5.3-15：链表的基本操作。

```
int main()
{
    struct node *head, *ps;
    int num;

    printf("Create Linked List: \n");
    head = CreateListR();
    printf("Print Linked List: \n");
    PrintList(head);

    ps = (struct node *)malloc(sizeof(struct node));
    printf("Insert an Integer: ");
    scanf("%d", &ps->num);
    head = InsertList(head, ps);
    printf("Print Linked List: \n");
    PrintList(head);

    printf("Delete an Integer: ");
    scanf("%d", &num);
    head = DeleteList(head, num);
    printf("Print Linked List: \n");
    PrintList(head);

    return 0;
}
```

运行结果：

```
Create Linked List:
Input an Integer: 3
Input an Integer: 6
Input an Integer: 9
Input an Integer: 12
Input an Integer: 0
Print Linked List:
3
6
9
12
Insert an Integer: 5
Print Linked List:
3
5
6
9
12
Delete an Integer: 9
Print Linked List:
3
5
6
12
```



链表的基本操作小结

■ 基本编程方法

- 链表以**头指针**表示，尾部结点的**next**指针为**空指针**
- 修改链表结构的函数，其返回值应当为**链表头指针**
 - 建立链表、插入结点、删除结点，链表头指针可能会被修改
- 先处理结点位于链表中部的一般情况，再检查特殊情况是否需要单独处理
 - 空表，头指针为空指针
 - 待处理结点位于链表头部，头指针可能需要修改
 - 带处理结点位于链表尾部，该结点的**next**指针为空
 - 链表只有一个结点，该结点既位于链表头部，也位于链表尾部

■ 注意事项

- 必须保证任何结点都有指针指向，包括指针变量、其他结点的**next**指针
- 已经删除的结点应及时释放，并注意原来指向它的指针变成了悬空指针
- 可以借助图示方法帮助思考操作过程



小结



主要知识点与能力要求

■ 主要知识点

- 指针是操作内存数据的基本工具，可用于间接访问内存数据和指令
 - 指向变量、数组、字符串、结构体、文件类型、动态分配存储空间、函数的指针
 - 指针用作函数参数和返回值
- 结构体和指针结合，可以构建复杂的数据结构，提高存储和计算效率
 - 如，链表
- 文件类型使程序和磁盘文件建立联系，扩展了数据的来源和长期保存方法

■ 能力要求

- 针对具体问题，抽象出待处理数据对象，构建合适的存储结构
- 灵活运用指针、数组、结构体和文件访问，解决复杂实际问题
- 借助指针对数据存储和算法运行效率进行基础优化



本章结束