程序设计

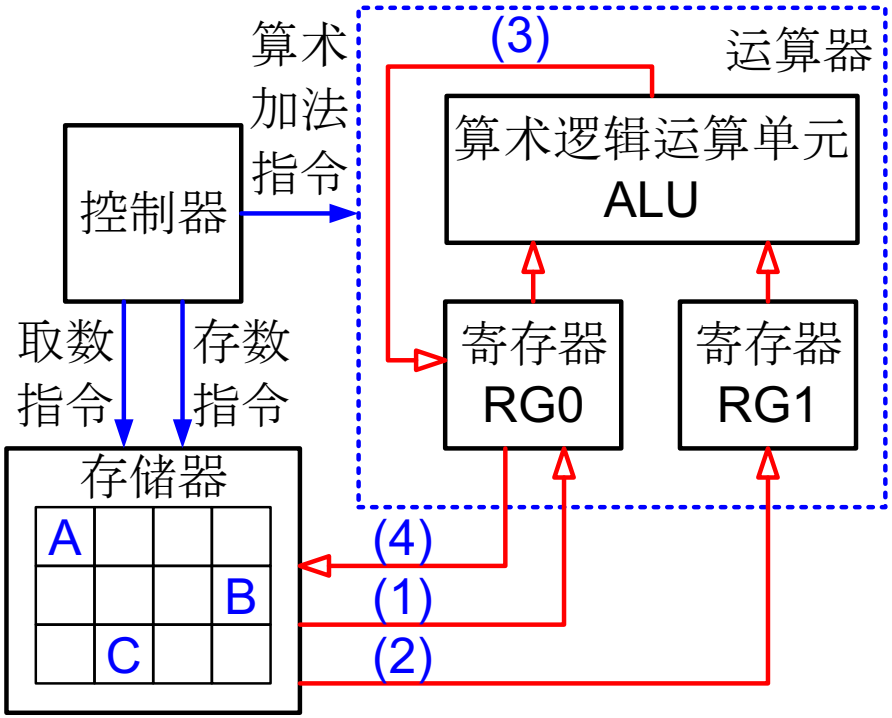
复习课

第一章 引论

1. 程序执行过程
2. 语言及编译
3. 数制及相互转换

程序的执行过程

(1) LD RG0, A　　取数指令：Load Data from Mem A to RG0
(2) LD RG1, B　　取数指令：Load Data from Mem B to RG1
(3) ADD RG0, RG1　　加法指令：Add Data RG0 and RG1 to RG0
(4) ST RG0, C　　存数指令：Store Data from RG0 to Mem C



程序设计语言的层次

- 机器语言
  - 以二进制代码表示的指令集合, CPU 直接能识别和执行的语言
- 汇编语言
  - 用助记符代替操作码，用地址符号或标号代替地址码
- 高级语言
  - 高层次的抽象，接近人类语言
  - 计算的抽象：加减乘除
  - 存储的抽象：变量
  - 控制的抽象：分支、循环

不同语言之间的转换

- 自然语言：
  - 翻译：词汇、语法、语义分析->翻译
  - 自然语言词汇语义的二义性，使得自然语言的自动翻译非常困难
- 高级语言到机器语言的转换：
  - 编译：词汇、语法、语义分析->翻译
  - 严密的词汇、语法、语义定义使得翻译成为可能
  - 翻译的程序称之为编译器

二、八、十六进制转化为十进制

只要写出展开式，计算相应的值便可转换成十进制数。

- 二进制数转换成十进制数示例

  $(10110111)_B = 1*2^7 + 0*2^6 + 1*2^5 + 1*2^4 + 0*2^3 + 1*2^2 + 1*2^1 + 1*2^0$

  $= 128 + 32 + 16 + 4 + 2 + 1 = 183$

  结果为: $(10110111)_B = (183)_D = 183$

- 八进制数转换成十进制数示例

  $(261)_O = 2 * 8^2 + 6 * 8^1 + 1 * 8^0 = 2 * 64 + 6 * 8 + 1$

  $= 128 + 48 + 1 = 177$

  结果为: $(261)_O = (173)_D$ 或 $261_O = 177$

- 十六进制数转换成十进制数示例

  $(F261)_H = 15*16^3 + 2*16^2 + 6*16^1 + 1*16^0$

  $= 15*4096 + 2*256 + 96 + 1 = 62049$

  结果为: $(F261)_H = (62049)_D$ 或 $F261_H = 62049$

八进制和十六进制数向二进制数转换

八进制数转换成二进制数

以小数点为界，按照每一位分别转换为一个三位的二进制数，最后将整数部分的最高位0 和小数部分的最低位0 删去。

例如：$(23.46)_O = (\mathbf{0}10\ 011\ .\ 100\ 11\mathbf{0})_B = (10011.10011)_B$

十六进制数转换成二进制数

以小数点为界，按照每一位分别转换为一个四位的二进制数，最后将整数部分的最高位0 和小数部分的最低位0删去。

例如：$(13.98)_H = (\mathbf{000}1\ 0011\ .\ 1001\ 1\mathbf{000})_B = (10011.10011)_B$

八进制数与十六进制数之间的转换

通过与二进制转换进行过渡

十进制数向二进制、八进制和十六进制数转换

- 十进制数向二进制数转换

  转换法则：

  整数部分：除基取余

  小数部分：乘基取整

  示例： $37.8125_D = (\ ?\ )_B$

2 | 37	取余：1	低位
2 | 18	取余：0	
2 | 9	取余：1	
2 | 4	取余：0	
2 | 2	取余：0	
1	取余：1	高位

0.8125		
X 2		
1.625	取整：1	高位
X 2		
1.25	取整：1	
X 2		
0.5	取整：0	
X 2		
1.0	取整：1	低位

转换结果为: $37.8125_D = (100101.1101)_B$

- 十进制数向八进制和十六进制数转换

  通过与二进制转换进行过渡

第二章 基本数据类型及运算

1. 变量命名
2. 基本数据类型及常量表示
3. 输入输出方法
4. 基本数据类型的运算

C程序设计中的变量

- C程序中的变量表示程序中需要存储的数据对象，对应内存中的存储空间
- 变量用变量名来标识
  - 变量名以下划线或字母开头，后接0个或任意多个字母、数字和下划线，C程序中的关键字不能用来命名变量(关键字见教材14页)
    - 有效变量名：a, _b, c1_2
    - 无效变量名：2a, &c, $d1
  - C程序中变量在使用前需要声明其数据类型
    - C程序中提供丰富的数据类型，使得使用者可以选择最优的存储方案来存储数据
  - C程序中变量的声明方法：变量类型 变量名1, 变量名2;
    - int a;
    - long b;
    - float c;
    - double d;
    - char dd;

整型变量

- 整型数据：
  - 基本整数型：int表示，包含符号位
  - 短整型：short int或short表示，包含符号位
  - 长整型：long int或long表示，包含符号位
  - 用补码来表示负数
- 无符号整数：
  - unsigned int: 无符号位的基本整数
  - unsigned short: 无符号位的基本短整数
  - unsigned long: 无符号位的基本长整数

整数常量的表示方法

- 整数常量：无小数点数字串
  - 123，3456
- 长整数常量：整数常量后加l 或L
  - 456L，87699l
- 无符号整数常量：整数后加u或U
  - 35u, 45U
- 无符号长整型常数：加上字母U和L
  - 5567UL, 23545LU
- 八进制数以数字0开始：0123
- 十六进制数以0x(0X)开头：0xDE

浮点型数据

- 科学计数法：
  - 0.000000714566778=7.14566778e-8
- 采用浮点方法移动小数点位置，可以用有限位数存储最高精度的数据
- IEEE754的单精度和双精度浮点数格式，基为2

符号位	指数部分	尾数部分(无符号数)

  - 尾数部分表示小数点后的数字，小数点前固定为1
  - 单精度浮点数(float)：符号位1位，指数位8位，尾数23位
  - 双精度浮点数(double)：符号位1位，指数位11位，尾数52位
- 长双精度浮点数(long double)：16 Byte

浮点型常量

- 正负号+整数部分+小数点+小数部分+指数部分
  - 正号可以省略
  - 整数部分和小数部分可以任选，但不能同时省略
  - 指数部分为E/e加上整数
  - 小数点和指数部分不能同时省略
  - 合法的浮点数：4.5, 5., .456, 1.e-8
  - 不合法的浮点数：e34, .e45, 4.0e, 1e5.4

浮点型常量

- 浮点型常量后可以加上f/F, *l*/L
  - f/F表示单精度浮点数
  - *l*/L表示长双精度浮点数
- 不加任何修饰符表示双精度浮点数

字符

- C语言用一个字节对字符进行编码，可以表示256个字符
  - 0-127个字符定义由美国国家标准局定义的美国标准信息交换代码(**A**merican **S**tandard for **C**oded **I**nformation **I**nterchange)，简称ASCII码
  - 128-255为扩展的ASCII码
- ASCII码对照表，见课本附录B

字符常量的表示方法

- 普通字符常量：
  - 单引号括住：’ a’, ‘%’, ‘&’
- 转义字符常量：
  - ‘\t’：制表符
  - ‘\n’: 换行
  - ‘\\’: 反斜杠
  - ‘\’’: 单引号
  - ‘\”’: 双引号
  - ‘\0’:字符串结束
  - ‘\ooo’: ooo表示三位八进制数，表示该八进制数对应ASCII码
  - ‘\xhh’: hh表示两位十六进制数，表示该十六进制数对应的ASCII码

格式化输出函数

- 将二进制存储的变量嵌入字符串中按照指定格式输出
- C标准函数库格式化输出函数调用格式：
  - printf(“输出格式控制字符串”, 输出项);
  - 输出格式控制方式：%[修饰符][控制符]

格式化输出控制符

- 输出格式控制符
  - %d, %i: 十进制输出整型数
  - %o,%O：八进制输出整数
  - %x, %X: 十六进制输出整数
  - %u, %U：无符号十进制
  - %f, %F：小数形式输出单精度、双精度浮点数
  - %e, %E：以指数形式输出单、双精度浮点数
  - %c: 输出单个字符
  - %s: 输出字符串
  - %%: 输出%
- 注意：不要用浮点数格式输出整数，不要用整数格式输出浮点数

格式化输出修饰符

- %-d: 左对齐，默认是右对齐
- %+d: 正数也输出符号
- %#o: %#x: 输出八进制前面加0，十六进制前面加0x
- %#f: 浮点数输出小数点
- %4d：指定输出整数宽度，不足空格补齐
- %6.5f: 整体宽度为6，小数部分宽度为5
- %Hd：短整型输出
- %ld：输出长整型数
- %lf： 输出长双精度浮点数

格式化输入函数

- 将字符串格式化为整数/浮点数
- C标准函数库格式化输入函数调用格式
  - scanf(“输入格式控制字符串”, 输入存储地址项)
  - 输入格式控制方式：%[输入格式修饰符][输入格式控制符]

输入格式控制符

- %i, %d: 十进制整数
- %o,%O：八进制整数
- %x, %X: 十六进制整数
- %u, %U：无符号十进制
- %f, %F：小数形式单精度、双精度浮点数
- %e, %E：以指数形式单、双精度浮点数
- %c: 单个字符
- %s: 字符串

输入格式修饰符

- %*d：输入数据但不传送到变量
- %5d: 输入数据数字字符数为5
- %Hd: 短整型输入
- %ld: 输入长整型
- %lf：输入长双精度浮点数

表达式

- 算术表达式：+, - , *, /, %
- 关系表达式：>, <, >=, <=, ==, !=
- 逻辑表达式：&&, ||， !
- 赋值表达式：x = y+1; x += 5;
- 条件表达式：max = x > y ? x : y
- 逗号表达式：a = 1, b = 2

- 类型转换及强制类型转换

第三章 结构化程序设计

1. 基本语句及复合语句
2. 分支：if/else　switch/case
3. 循环：while/do while/for

基本语句

- 表达式语句
- 空语句
- break语句
- continue语句
- return语句
- goto语句

复合语句

复合语句：

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

- 复合语句通常是其他控制结构的组成部分

两路选择结构

- 两路选择结构

  if(表达式)

  　　语句1/复合语句1；

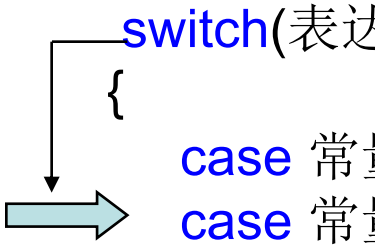
  else

  　　语句2/复合语句2；

- 条件表达式：
  - 非0为真
  - 0为假
- else分支可以省略
- 分支语句可以是复合语句
- if/else支持嵌套

多路选择结构

- 多路选择结构



```
switch(表达式)
{
    case 常量表达式1: 语句序列1;
    case 常量表达式2: 语句序列2;
    …
    case 常量表达式n: 语句序列n; break;
    default: 语句序列n+1; break;
}
```

- switch后表达式的值只能为整型、字符或枚举型
- 通过计算switch后表达式的值，寻找相应的case作为执行入口往下执行，加入break语句可以中断执行
- case后值为常量，且互不相等
- case后可以接单一语句或语句序列
- default可以省略，但最多只能出现一次

do…while与while的区别

- while循环
  - 首先判断条件，条件满足再执行语句
  - 语句可能一次都不执行

- do…while循环
  - 首先执行语句，再判断条件
  - 语句至少被执行一次

for循环

for(表达式1；表达式2； 表达式3)
    语句/复合语句;

1.计算表达式1
2.计算并测试表达式2，为真，转步骤3，否则结束循环
3. 执行循环体语句
4. 计算表达式3
5. 转步骤2

三种循环的比较

- while循环
  - 条件成立时执行，直到条件不成立时结束
- do…while循环
  - 重复执行某个计算，直至条件不成立时结束
- for循环
  - 某个变量从初值开始，顺序变化，当条件成立时，重复执行某个计算，直至条件不成立为止

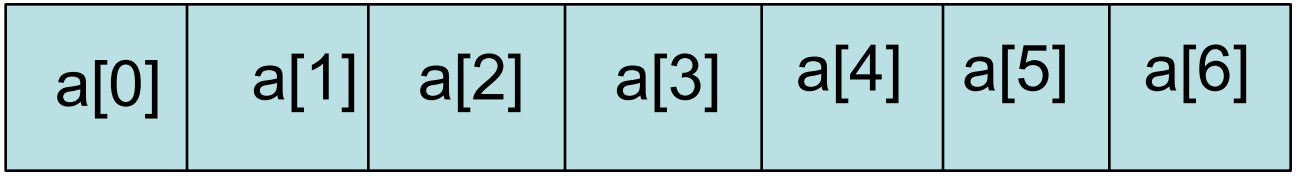
第四章 数组

1. 一维数组
2. 多维数组
3. 字符串

数组的基本概念

- 数组：
  - 逻辑上，表示同类数据组成的数据表
  - 物理上，对应一组连续的存储空间
  - 通过下标来引用数组，C语言数组下标从0开始
  - 数组的单个元素与相同类型的独立变量一样使用
  - 数组可分为：一维数组，二维数组，多维数组

int a[7];



一维数组

- 一维数组定义方式：

  类型说明　变量标识符[常量/常量表达式]

  - 数组通过类型说明指明数组元素的类型
  - 数组也是变量，变量标识符命名规则与一般变量命名规则一致
  - []是数组的标志，方括号中的常量/常量表达式表示数组元素的个数
  - 数组元素下标从0开始到数组元素个数减1
  - 数组的下标必须是整数表达式

数组的初始化

- 数组定义后，数组中存储的值是随机的，要引用数组元素，必须进行初始化
- 数组的初始化方法
  - 数组定义时，顺序列出全部元素初值

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

  - 给前面部分元素设置初值，其他元素初值为0

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

  - 对全部元素进行初始化，可以不指定数组长度

```
int g[]={10, 11, 2, 5, 9}; //g 有5个元素
```

  - 通过输入设置数组元素初值

```
for(k = 0; k < 4; ++k)
    scanf(“%d”, &a[k]);
```

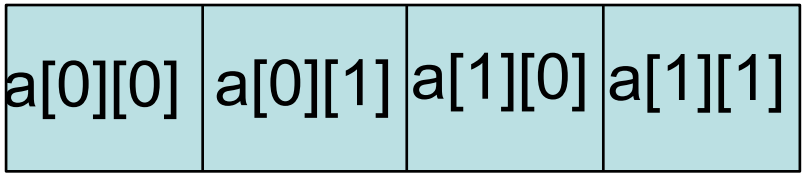
多维数组

- 计算机的物理内存是一维的
- 多维数组在C语言中是逻辑上的一种抽象
- 多维数组：

int a[2][2]; 定义一个2*2的数组

包括以下元素：a[0][0], a[0][1], a[1][0], a[1][1]

- 多维数组物理上的存储方式：



- 所以多维数组a[2][2]可以认为是由a[0]，a[1]两个一维数组组成

多维数组

- 多维数组的定义方法：

类型说明 标识符 [常量/常量表达式] [常量/常量表达式]

- 多维数组的初始化，以二维数组为例：

方法一：按行给全部元素赋初值

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

方法二：按元素存储顺序给所有元素赋初值

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

方法三：按行给部分元素赋初值，其他元素初始化为0

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

方法四：按元素存储顺序给部分元素赋初值，其余元素初始化为0

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

方法五：按元素存储顺序给部分元素赋初值，省略第一维长度

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

方法六：对各行部分元素初始化，省略第一维长度

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

字符串的概念

- 字符串是一种特殊的字符数组：

char a[16]=“china”;

a[0]=‘c’, a[1]=‘h’, a[2]=‘i’, a[3]=‘n’, a[4]=‘a’, a[5]=‘\0’，a[6]=…a[15]=‘\0’

- 字符串约定以’\0’结束，程序在输出字符串数组时，碰到’\0’认为字符串结束，不再输出
- 存储字符串的字符数组的大小一定要大于有效字符+1，需要多余存储一个’\0’来表示字符串结束
- 如果字符数组不包含’\0’，那么它不再是字符串而仅仅是字符数组

字符串的初始化

- 方法一：以数组方式初始化

char a[10] = {‘C’, ‘h’, ‘i’, ‘n’, ‘a’};

a[5]-a[9]不指定，默认为’\0’

- 方法二：以字符串方式初始化

char a[10] = “China”;

char a[10] = “Chi” “na”;

a[5]-a[9]不指定，默认为’\0’

思考，下列方法是否正确？

char a[10];

a[0] = “China”;

C标准库中的字符串处理函数

包含头文件: #include <string.h>

- 取字符串长度：strlen

字符串长度不包含'\0'字符

- 字符串复制函数strcpy(str1, str2)

将str2复制到str1中，str1必须能够容纳str2

- 字符串部分复制函数strncpy(str1, str2, n)

将str2中的前n个字符复制到str1中

- 字符串连接函数strcat(str1,str2)

将str2中内容连接到str1后，str1必须能容纳str1中的内容

C标准库中的字符串处理函数

- 字符串比较函数：strcmp()

比较两个字符串，如果两个字符串相同，返回0，否则返回非0

注意：字符串不能通过==和!=进行比较

- 字符串大小写互相转换：strlwr/strupr

strlwr(str)/strupr(str)将str转化为小/大写并存储在str中

- 字符串输入/输出函数：gets(str)/puts(str)

gets(str)输入字符串到str中，puts(str)输出字符串str

第五章 函数

1. 函数的基本概念
2. 库函数的使用方法
3. 函数定义
4. 函数说明
5. 函数调用
6. 递归函数
7. 存储类别、作用域、宏定义

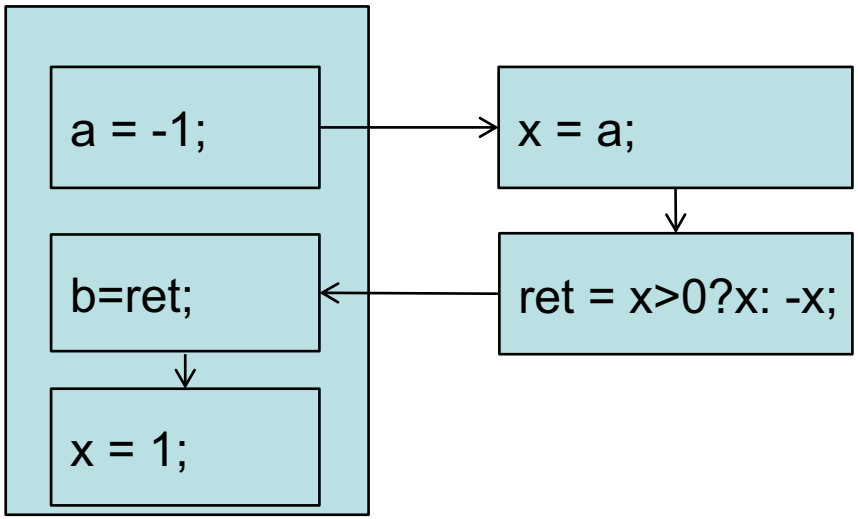
函数的基本概念

- 函数的作用
  - 功能分解
  - 程序复用

```
int abs(int x)  {
    return x > 0 ? x : -x;
}

int main()   {
    int a, b, x;
    a = -1;
    b = abs(a);
    x = 1;
}
```

- 函数的调用过程



函数的声明与定义

- 函数的声明
  - 函数的声明指定了函数的接口
  - 只要声明了函数，在程序中就可以调用这个函数

```
double min(double x, double y);
```

- 函数的定义
  - 函数的定义是函数的真正实现

```
double min(double x, double y)
{
    return x < y ? x : y;
}
```

函数的定义

- 函数定义的语法：

返回类型符 函数标识符(形式参数说明表)

{

说明和定义

语句序列

}

- 函数的三要素：
  - 函数头
  - 函数体
  - 返回值

函数的定义

- 函数头：返回类型符 函数标识符(形式参数说明表)
  - 返回类型：指定函数执行完毕后返回值的类型
  - 函数标识符：函数的“名字”
  - 形式参数说明表：说明函数的形式参数类型和名称，通过逗号隔开。形参

  类型符1 形参1，类型符2 形参2， …

函数的定义

- 函数体和返回值：实现函数功能的语句序列
  - 变量声明：函数中的变量是局部变量，不同函数中的变量名可以同名，互不干扰
  - 语句序列：实现函数功能
  - return语句：
    - 函数返回，结束函数执行，同时返回函数计算的结果
    - 返回值的类型必须与函数声明时定义的返回类型相同。如不相同，会进行隐式类型转换。若定义了返回值的函数，不通过return语句返回值，则会出现错误
    - 空返回值(void)函数，return语句仅仅结束函数，不返回值。空返回值函数，也可以不包含return语句，此时函数执行到函数体末尾结束

函数说明

- 一个函数要调用另一个函数，编译程序必须知道被调用函数的信息：
  - 函数名
  - 函数返回值
  - 函数的参数类型
- 被调用函数与调用函数在程序正文中的顺序：
  - 被调用函数在调用函数之前：无需对函数进行说明，编译程序已经知道了函数的信息
  - 被调用函数在调用函数之后：需要使用函数说明语句，告诉编译程序被调用函数的信息
  - 被调用函数定义在其他文件中：可以使用函数说明语句，也可以通过#include语句包含函数说明语句的头文件
- 函数说明方法：
  - 返回类型　函数名(形参列表)

函数调用

- 在需用使用函数功能的地方，调用函数
- 函数调用的一般形式：
  - 有参数形式：函数标识符(实参列表)
  - 无参数形式：函数标识符()
- 函数调用的应用形式：
  - 函数作为独立的语句：

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

  - 函数出现在表达式中：

```
a= min(x, y);
```

函数的执行过程

- 函数的执行过程如下：
  - 为函数形参分配空间
  - 计算实参表达式值，将实参表达式的值赋给形参
  - 为函数的局部变量分配空间
  - 执行函数体内的语句序列
  - 函数执行完，或遇到return语句，释放为函数分配的所有空间，包括形参及局部变量

递归函数

- 一个函数通过直接或间接的方式调用它本身，该函数称为递归函数
  - 直接递归：函数直接调用其本身
  - 间接递归：A函数调用B函数，B函数再调用A函数
- 递归函数的意义：
  - 通过递归将复杂问题分解为求解小规模的同样问题，并一直分解下去，直到最小规模的问题可以直接求解为止

存储类别

- 存储类别auto:

  默认类别，自动类别，变量只在声明该变量的函数或复合语句时间里，函数返回或控制退出复合语句后撤销。未初始化时，初始值未知

- 存储类别register:

  寄存器类型。如有可能，该变量存放在CPU的寄存器中，存取速度最快。但由于寄存器有限，编译器在资源不够时，可能不会将变量存入寄存器中。未初始化时，初始值未知

- 存储类别static:

  静态变量，存储空间在程序运行前已经分配，到程序执行结束该变量一直存在。未初始化时，默认初始化为0

- 存储类别extern:

  全局变量，该变量可以被其他文件看到。未初始化时，默认初始化为0

作用域

- auto, register, static类型，仅在变量定义时所在的函数体或复合语句内有效。auto和register类型离开函数体或复合语句时变量被撤销，但static类型变量仍存在，再次进入相应函数体或复合语句时，仍然可以访问这一变量。
- 将变量定义在所有函数之外，该变量为全局变量，在该文件内该变量有效。在文件外如需引用该变量，需使用extern关键字，表示在其他文件中定义了这个全局变量。

不带参数宏定义注意点

- 宏名一般用大写字母表示，与变量名区别，当然也可用小写字母表示
- 宏定义不是语句，后面不用加分号
- 宏定义从定义处开始，到程序文件结束，可以用“#undef 标识符”来终止宏的作用
- 宏定义时可以使用前面已经使用过的宏

带参数宏定义

- 带参数的宏定义： #define 标识符(参数列表) 字符序列

```
#define MAX(A, B)  ((A) > (B) ? (A) : (B))
int main()
{
    int x, y, z;
    scanf("%d%d", &x, &y);
    z = MAX(x,y);
}
```

宏展开


```
int main()
{
    int x, y, z;
    scanf("%d%d", &x, &y);
    z = ((x) > (y) ? (x) : (y));
}
```

第六章 指针

1. 指针的基本概念
2. 指向数组元素的指针
3. 指针作为函数形参
4. 数组作为函数形参
5. 指针数组
6. 多级指针
7. 返回指针的函数

指针的基本概念

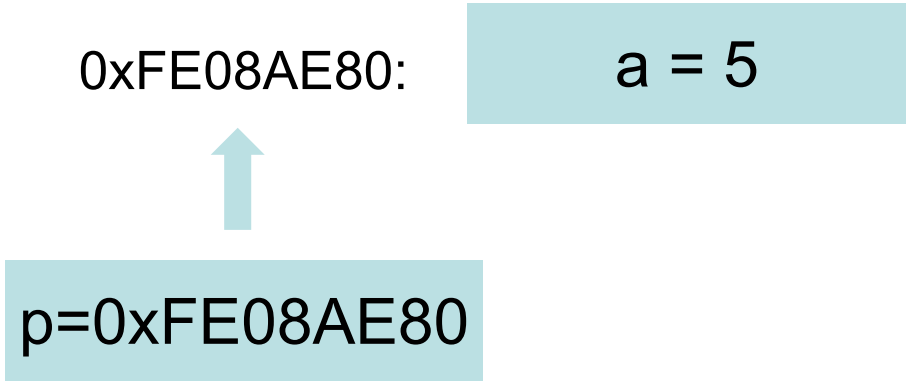
- 变量存储在计算机内存中

int a = 5;

- 指针：存储变量的地址的变量

int a = 5;

int *p = &a;



取地址操作符及引用指针变量所指变量

- 取地址操作符&：通过取地址操作符可以获得变量的地址，并将该地址赋给指针变量

```
int a, *p;
p = &a;  // 取a的地址，并赋给指针变量p
```

- 引用指针变量所指变量操作符*：

```
int a = 5;
int *p = &a;
printf(“%d\n”, *p); // 取指针p指向的变量a
```

指向数组元素的指针

- 指针可以指向数组中的元素

```
int a[100], *ptr;
ptr = &a[50];
```

- 数组名表示该数组的起始地址

```
int a[100], *ptr;
ptr = a; // 等价于ptr = &a[0];
```

指针的运算

- 指向数组的指针可以进行有限的运算
  - 关系运算: <, <=, ==, >=, >, !=
  - 加减运算：指针加上一个整数k后，指向当前指针指向的元素的后k个元素；两个指针相减，表示两个指针所指向元素的下标的差。

访问数组的方式

- 通过[]访问数组，例如a[5]表示数组第6个元素
- 通过数组名来访问数组，例如*(a+5)与a[5]等价
- 利用指向数组的指针访问数组，例如

```
int a[100], *ptr;
ptr = a;
```

*(ptr+5)与a[5]相同

指针作为函数形参

- 在函数调用时，函数体中无法修改实参变量
- 通过指针可以间接的修改调用环境中的变量
  - 指针实参的值无法修改，但是可以修改指针实参所指向的对象的值

```
void incr(int n) {
  n++;
}
void main() {
  int n = 10;
  incr(n);
  // 执行完毕, n=10
}
```

```
void incr(int *ptr) {
  (*ptr)++;
}

void main() {
  int n = 10;
  incr(&n);
  // 执行完毕, n = 11
}
```

数组作为函数形参

- 数组可以作为函数的形参，数组形参对应的实参是数组的地址，因此在形参中无需指定形参中数组的大小，通常需要另外一个整型形参来指定数组的大小
- 数组形参与指针等价

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

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

多维数组作为形参

- 多维数组也可以作为函数形参，作为数组作为函数形参除第一维可以不指定大小外，其他维度均需要明确指定大小

```
void sumAToB(int a[][10], int b[], int n)
{
    int i, j;
    for(i = 0; i < n; i++)
        for(b[i] = 0, j = 0; j < 10; j++)
            b[i] += a[i][j];
}
```

指针数组

- 指针数组的定义方式

  类型说明符 * 标识符[常量表达式]

- 指针数组是一个数组，这个数组存储的是指针
- 指针数组可以管理同类型的指针

多级指针

- 指向指针的指针成为多级指针
- 多级指针的定义方法：类型 ** 标识符
  - 类型表示多级指针最终指向变量的类型

```
int **pp, *ip, i;
ip = &i;
pp = &ip;
```

返回指针值的函数

- 返回变量的指针的函数
  - 函数的返回值是变量的指针

```
int * search(int *p, int size, int key) {
    int i;
    for(i = 0; i < size; i++) {
        if(key == p[i])
            return &p[i];
    }
    return NULL;
}
```

第七章 结构和链表

1. 结构类型和结构变量
2. 结构数组
3. 结构与函数
4. 链表
5. 联合
6. 枚举
7. 类型定义

结构类型

- 结构类型

struct 标识符

{

成员说明表；

};

- 结构类型注意点：
  - 成员说明表：类型 标识符；
  - 成员可以有多个，但不能重名
  - 成员可以是基本类型，也可以是已定义的结构类型

结构变量

- 通过结构类型可以定义结构变量

方式一： struct 结构类型名 标识符列表

方式二：

```
struct   标示符{
    成员说明表;
} 标识符表;
```

- 静态结构变量初始化为0，非静态变量初始值不确定

结构变量的引用

- 结构变量之间可以互相赋值
- 可以定义结构变量指针，通过&运算符取地址，通过*运算符取指针所指向的对象
- 结构成员的引用
  - 对于结构变量，通过”.”来引用成员
  - 对于指向结构变量的指针，通过”->”来引用成员

结构数组

- 结构数组的定义
  - 先定义结构类型，再定义结构数组
  - 在定义结构类型的同时，定义结构数组

```
struct Point {
    int x;
    int y;
};
struct Rectangle {
     struct Point pts[2];
};
```

结构在函数中的应用

- 结构成员作为函数参数
- 结构作为函数参数
- 结构指针作为函数参数
- 函数返回结构类型值

动态分配内存

- 在C函数库中提供了动态分配空间的函数，可以根据用户需要空间的大小来分配存储空间，

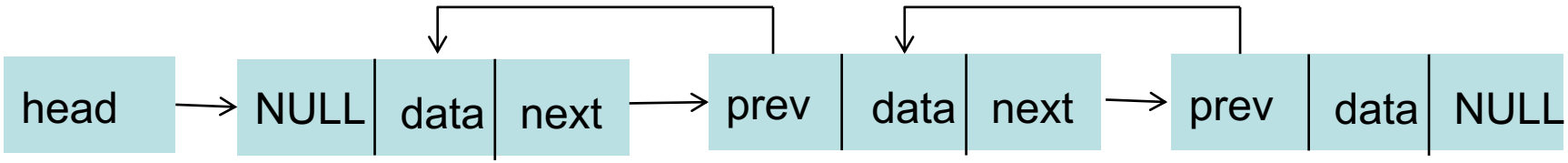
  包含头文件：#include <stdlib.h>
- void * malloc(unsigned size)
  - 分配大小为size个字节的连续空间，函数返回的是分配的空间的指针，如不能分配，函数返回NULL
- void *calloc(unsigned n, unsigned size)
  - 分配n块size个字节的连续空间(n *size字节)，分配成功返回指向连续空间的指针，否则返回NULL
- void free(void* ptr)
  - 释放分配ptr指向的空间
- void *表示无类型的指针，程序需要强制进行类型转换

动态分配内存的特点

- 与自动存储类型变量和数组相比
  - 动态分配内存可以根据需求，在程序运行过程中动态的申请内存空间，满足动态数据处理的要求
  - 动态分配的内存，在使用完毕后，可以动态的释放，释放后的内存可以再次申请

链表

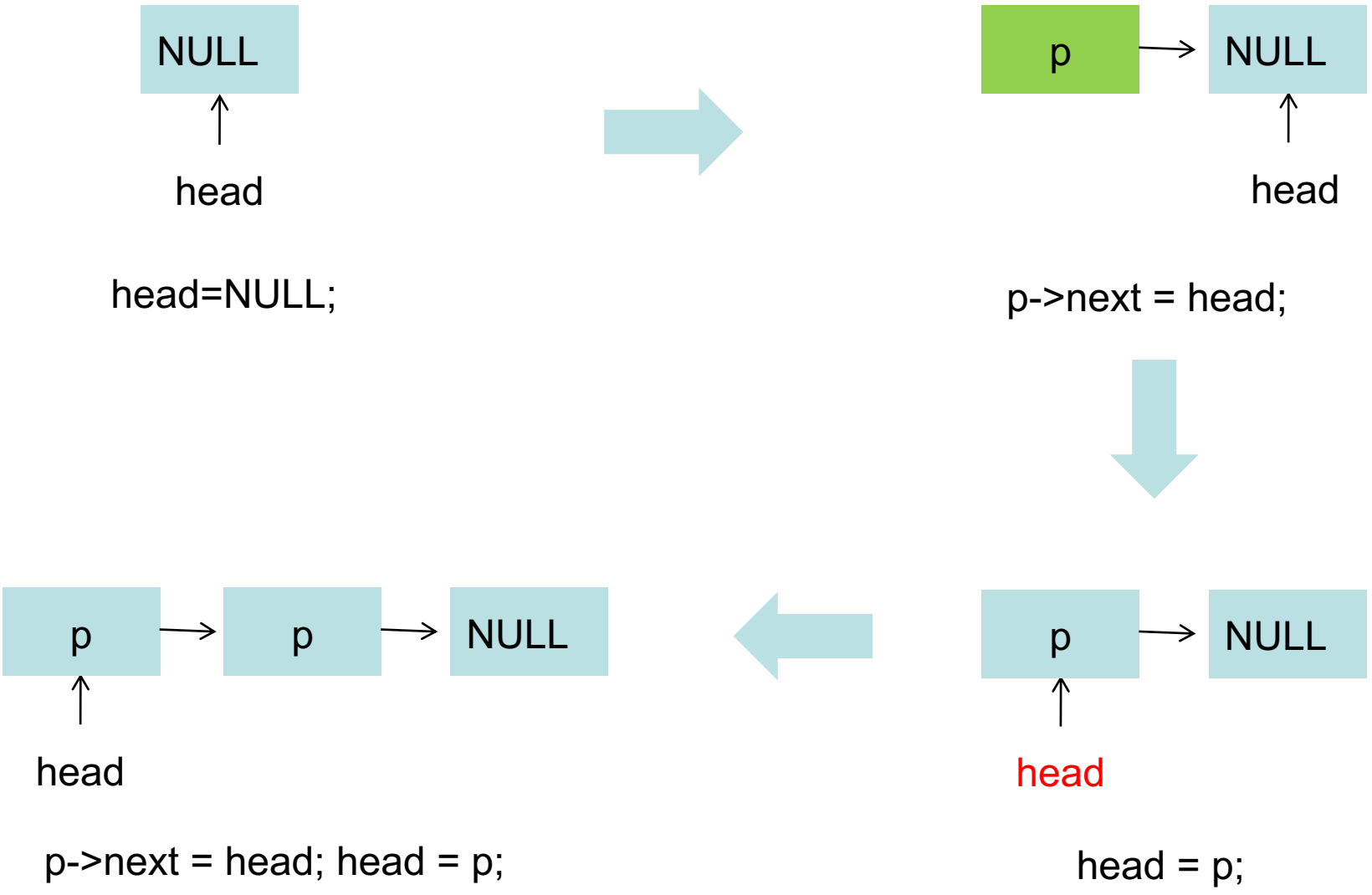
- 利用结构、指针以及动态分配内存的特点，构造了一种可以动态管理的数据链表，可以方便的对数据进行插入、删除、查找等各种操作。
- 链表的基本单元是一个结构体：
  - 基本单元的数据存储单元
  - 指向上一个基本单元的指针(可选)
  - 指向下一个基本单元的指针
- 链表的组织方式：
  - 头指针指向第一个结构体
  - 第一个结构体的指针指向下一个结构体，并一个接一个的将多个结构体链接在一起



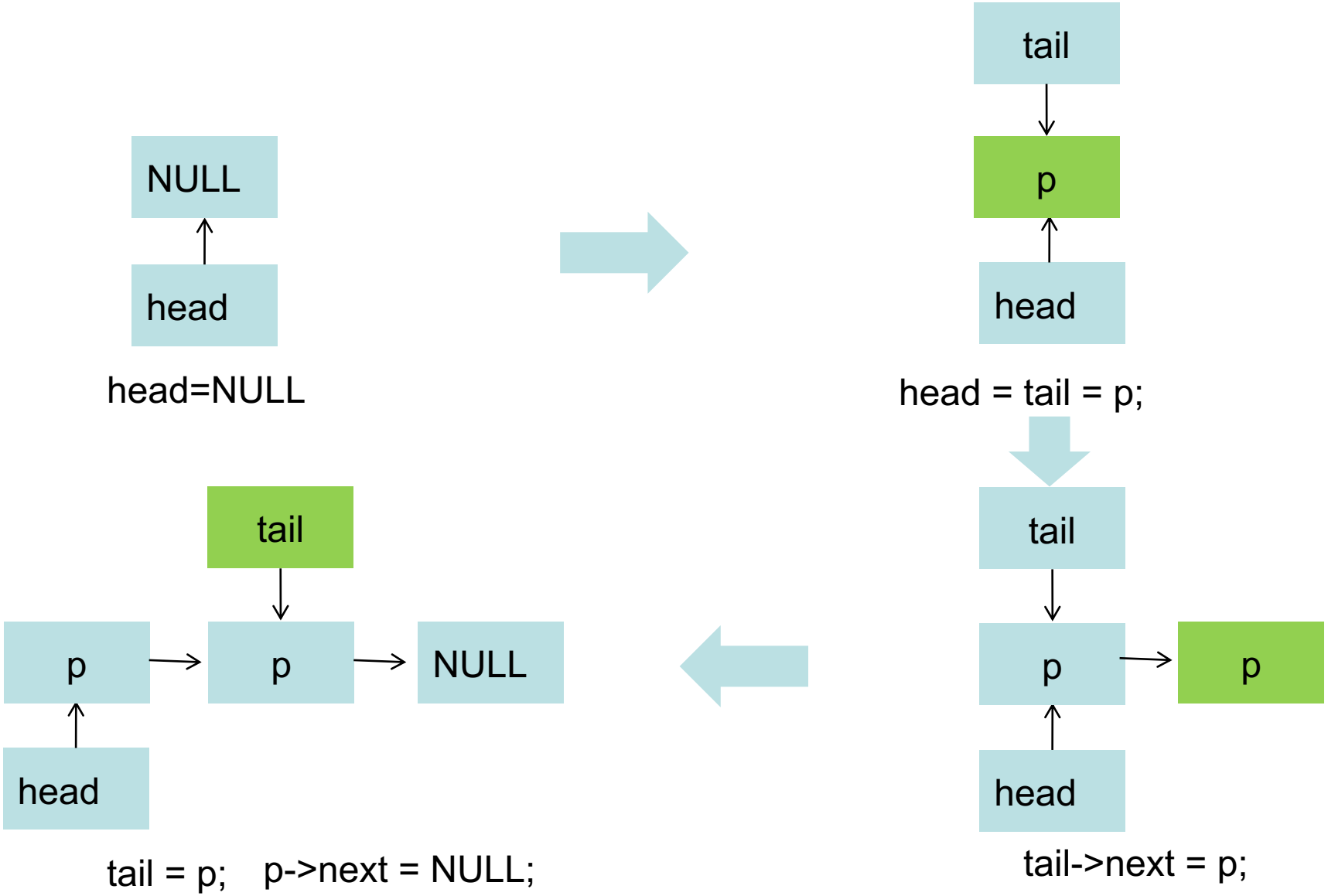
链表的基本操作

- 建立链表
- 遍历链表
- 释放链表
- 在链表中查找表元：有序/无序
- 在链表中插入新元素
- 从链表中删除单元
- 带辅助表元的链表

建立链表—头部插入法


NULL
head
head=NULL;
p
NULL
head
p->next = head;
p
NULL
head
head = p;
p
p
NULL
head
p->next = head; head = p;

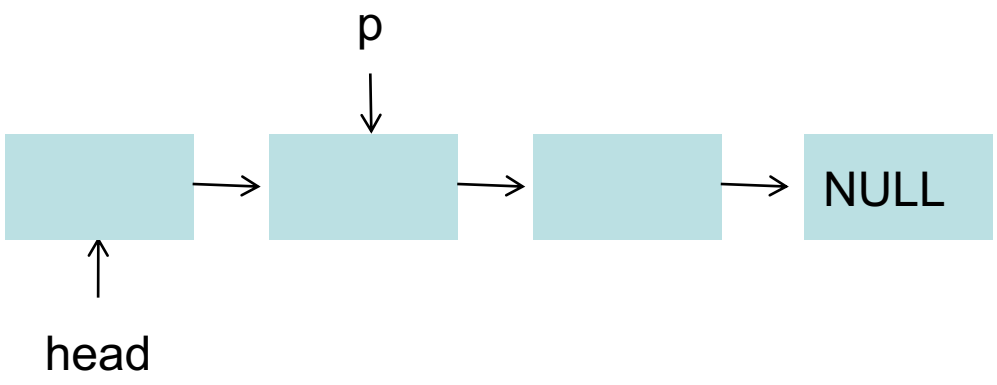
建立链表—尾部插入法


NULL
head
head=NULL
tail
p
head
head = tail = p;
tail
p
p
head
tail->next = p;
tail
p
p
NULL
head
tail = p; p->next = NULL;

遍历链表

```
for(p = head; p; p = p -> next) {
        printf("(%d, %d) \n ", p->x, p->y);
 }

p = head;
while(p != NULL) {
    printf("(%d, %d) \n ", p->x, p->y);
    p = p->next;
}
```



在链表中查找表元

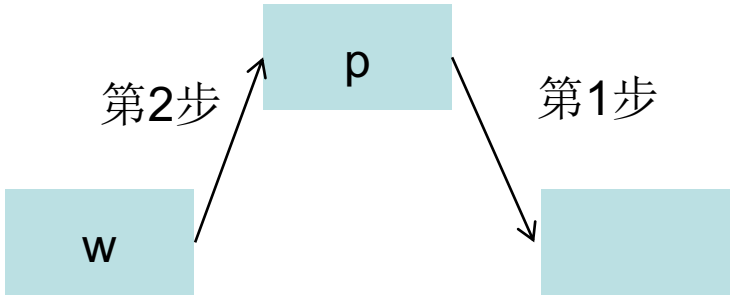
- 无序链表上查找：需要遍历整个链表来查找需要查找的表元

- 有序链表上查找：以从小到大排序的链表为例：如果当前链表表元值已经大于需要查找的值即可结束查找

在链表中插入新表元

- 在指定的表元w之后插入新表元

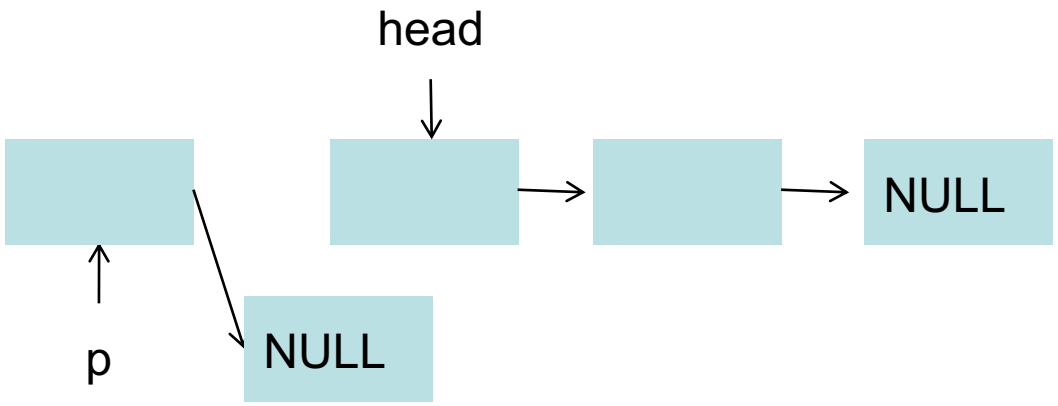
```
p->next = w->next;
w->next = p;
```



从链表中删除表元

- 删除单链表的首表元

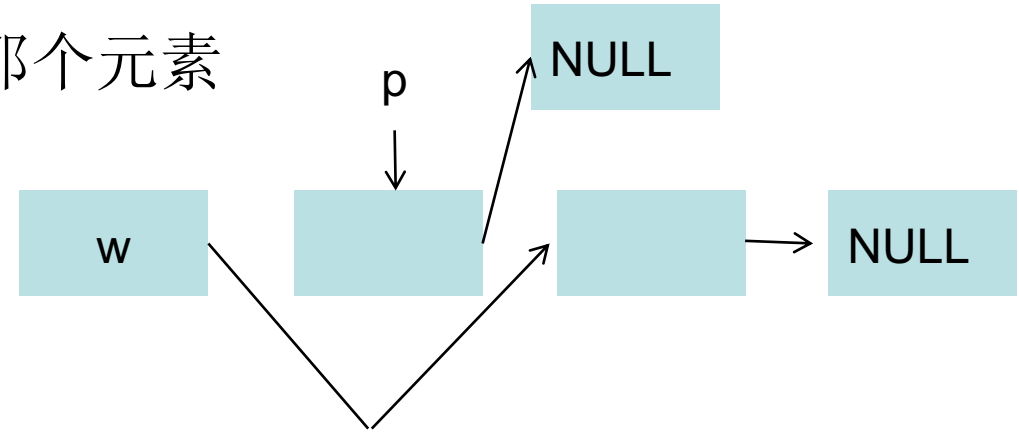
```
p = head;
if(p) {
    head = head->next;
    p->next = NULL;
    free(p);
}
```



- 删除单链表首表元之后的某个表元

w: 要删除表元之前的那个元素

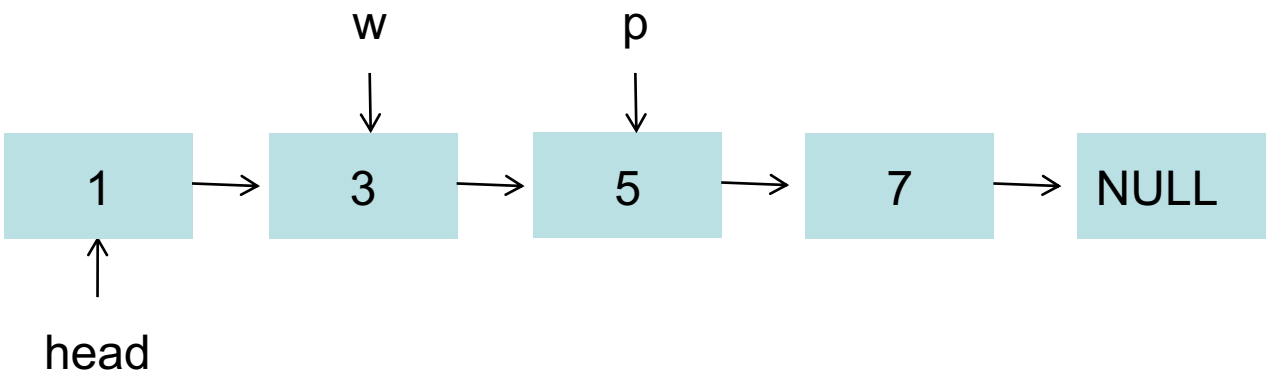
```
p = w->next;
w->next = p->next;
p->next = NULL;
free(p);
```



从链表中删除表元

```
stuS *delStu(stuS *head, int num) {
   stuS *w, *p;
   p = head;
   while(p != NULL && p-> number != num) {
       w = p; p = p->next;  // w指向num对应的上一个表元
   }
   if(p != NULL) {
       if(p == head) head = p -> next; // 若是头指针，头指针链接下一个
       else w->next = p->next; //否则上一个表元链接p的下一个表元
       free(p);// 释放p
    }
    else printf("找不到%d\n", num);
    return head;
}
```

从链表中删除表元



w指针为p的前序表元，必须保留前序表元w，才能删除p表元

释放链表

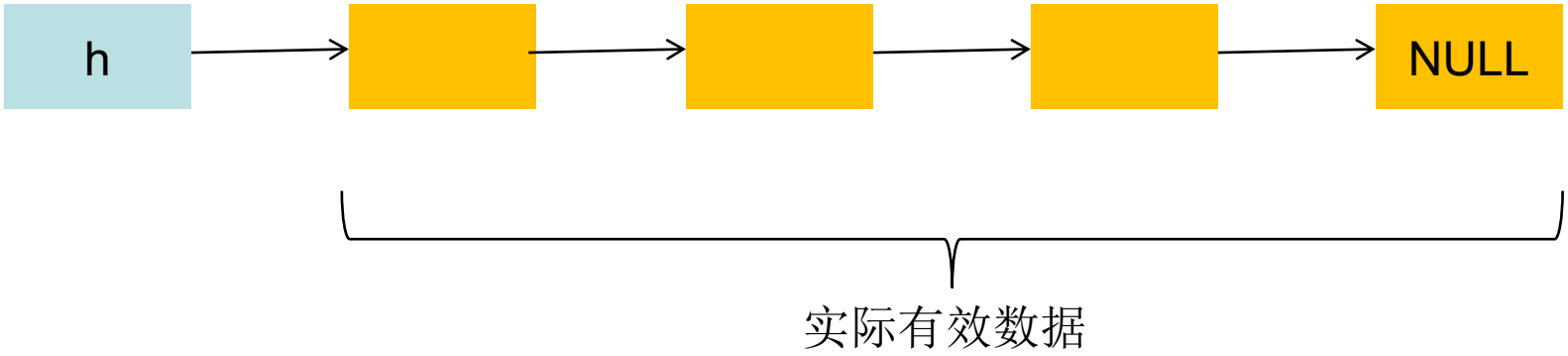
```
while(u) {
    w = u->next;  // w记录链表的下一位置
    free(u); // 释放当前位置的内存
    u = w;   // u再次指向链表下一位置
}
```

带辅助表元的链表

- 以上的方法，在删除首表元，或者在首表元前插入元素时，首表元的指针被修改了
- 在某些系统中，不允许修改首表元的指针，通常通过增加辅助表元来避免修改首表元指针。
- 有辅助表元的情况下，首表元指针永远指向辅助表元，辅助表元的下一元素指针指向第一个有效表元
- 辅助表元还可以简化代码，如插入表元时，无需判断是否首表元

带辅助表元的链表

辅助表元：不存储有意义的数据，仅h->link有意义，存储真正的首表元



- 不管实际链表数据如何变化，辅助表元永远不变
- 对于实际首表元，也存在前序表元(即辅助首表元)，可以简化代码，不用对首表元做特殊处理