

C语言程序设计

The C Programming Language



第9章 结构和联合

武汉光电国家研究中心

李春花



主要内容

■ 结构类型

- 结构类型的声明和结构变量的定义
- 结构成员的引用（. 和->运算符）
- 结构数组和结构指针
- 结构做函数参数与返回值

■ 字段结构

■ 联合类型

■ 结构指针的应用：链表

- ✓ 能对链表进行增、删、改、查等基本操作
（如，编写简单的同学通讯录程序）

为什么需要结构(体)类型

编程实现以下三个基本功能：

假如N个学生信息：学号，姓名，3门科成绩，要求：

1. 输出平均分最高的学生信息
2. 按照指定要求排序，并按序输出所有学生信息
3. 修改指定学生的成绩

为什么需要结构体类型

独立变量: `int a, b, c;` //彼此独立, 相互无关联

关联变量: 变量间有内在联系, 逻辑上是一个整体

如, 生日信息: 年月日;

`int year, month, day;` //用三个独立变量描述 使用不方便; 易误操作

`int date[3];` //用数组描述年、月、日可方便实现, 但难以反映内在联系

又如, 学生信息: 学号、姓名、性别、年龄、专业、成绩等;

`int num; char name[20]; char sex; int age; float score;`

| num | name | sex | age | score | addr |
|--------|------|-----|-----|-------|---------|
| 100101 | Fun | M | 18 | 87.5 | Beijing |

涉及多个有某种联系的特征、不同的数据类型

用数组还能描述吗?

使用结构类型处理组合数据 ——用户自定义的数据类型

人们希望把这些具有某种内在联系的数据组成一个组合数据
方便实现；易于理解、增强可读性；大大简化了编程。

学生信息：

```
int num;  
char name[20];  
char sex;  
int age;  
float score;
```



```
struct Student {  
    int num;  
    char name[20];  
    char sex;  
    int age;  
    float score;  
};
```

如，定义一个名为stu1的结构体变量，该变量中就包含了该学生的学号、姓名、性别等诸多结构体成员信息。

如：坐标点结构、通讯地址等

1. 结构体的定义

■ 一般形式:

struct 结构名

```
{  
    成员变量1;  
    成员变量2;  
    ...  
    成员变量n;  
};
```

↑
以分号结束，C语言中把结构的定义看作是一条语句

结构名命名原则
与变量名相同

关键字**struct**和它后面的**结构名**一起组成一个新的数据类型名，如，struct Student

通过定义后，**struct Student**就是一个在程序中可以使用的合法类型名，它和系统提供的标准类型（如int,char,float等）一样，都可用来定义变量的类型，只不过int等类型是系统默认的，而**结构体类型**由用户根据需要在程序中定义的

```
struct Student {  
    int num;  
    char name[20];  
    char sex;  
    int age;  
    float score;  
};
```

```
struct Student stu1,stu2;  
Student stu3;
```

stu1, stu2, stu3被定义为结构体变量

如何访问结构体成员？

结构变量名.成员 或 结构指针名->成员

定义结构体变量stu1
定义结构体指针 p

```
struct Student {  
    int num;  
    char name[20];  
    char sex;  
    int age;  
    float score;  
};
```

1. 定义结构体变量、引用或指针
2. 使用成员运算符“.”或“->”访问结构体成员

```
struct Student stu1;  
struct Student *p=&stu1;  
  
//通过结构体变量访问成员  
stu1.num=100; //设置学号  
strcpy(stu1.name, "Li"); //设置姓名  
stu1.sex='F'; //设置性别  
p->age=18; //通过结构体指针访问成员  
printf("num=%d,name=%s,sex=%c  
\n", stu1.num,p->name,p->sex);  
  
stu1.name="Li"; ?? //输出学生信息
```

结构变量的定义与初始化

(1) 先声明再定义(单独定义)

如: struct Student stu1, stu2;

Student stu1,stu2; //C++中struct可省略

```
struct Student {  
    int num;  
    char name[20];  
    char sex;  
    int age;  
    float score;  
} stu1,stu2; (混合定义)
```

局部变量，需要时定义，用完后系统自动收回分配的空间——推荐方式

(2) 在声明类型的同时定义变量(混合定义)

全局变量，浪费空间——尽量避免

(3) 在声明类型时省略结构名，直接定义变量 (无类型名定义)

```
struct {  
    int num;  
    char name[20];  
    char sex;  
    int age;  
    float score;  
} stu1; (无类型名定义)
```

适用范围小，很少使用——只能在定义该结构的局部作用域内使用

结构变量所占空间

```
struct Student {  
    int num;  
    char name[20];  
    char sex;  
    int age;  
    float score;  
};
```

struct Student stu1, stu2;

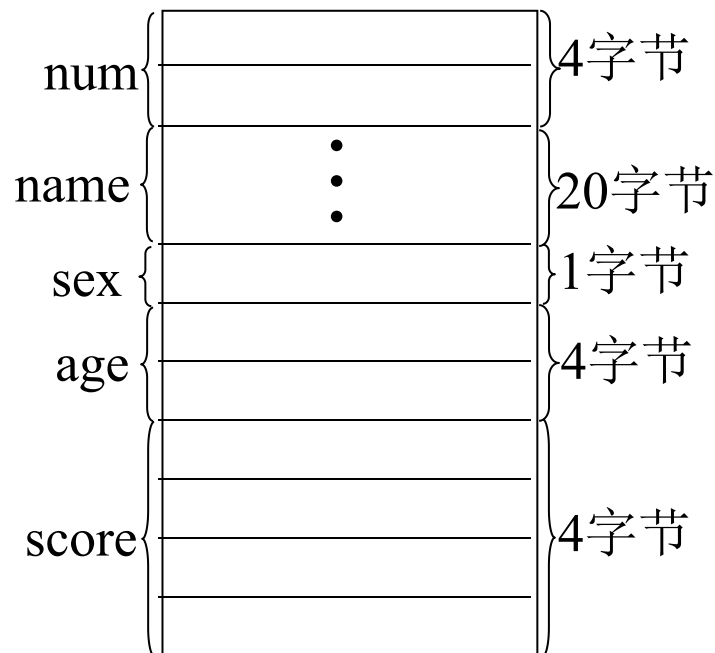
注意：

结构类型如同int一样，是个抽象的概念

编译时，**是对结构变量分配空间**，而不是结构类型！！

(4+20+1+4+4=33) 紧凑方式

为Student结构体的所有成员，按自上向下顺序，分配连续的33个字节空间



但，一般编译器实际分配的空间是
`sizeof(stu1)=sizeof(stu2)=36` //对齐方式
`printf("stusize=%d", sizeof(stu1));`



结构变量的初始化

与数组的初始化类似

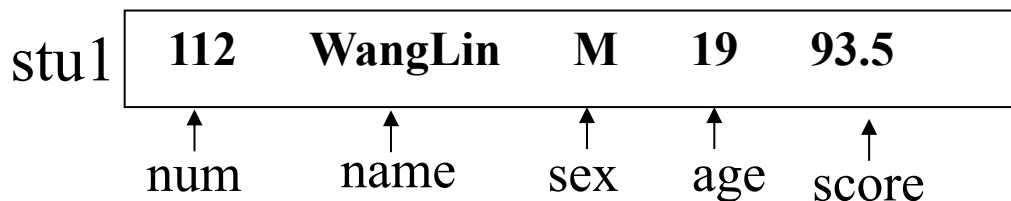
```
int a[3]={1,2,3};
```

(struct) 结构体名 结构体变量={初始数据列表};

```
struct Student stu1={112,“Wang Lin”,‘M’,19, 93.5};
```

```
struct Student {  
    int num;  
    char name[20];  
    char sex;  
    int age;  
    float score;  
};
```

- 在{ }中依次为每个成员赋值, 中间用逗号隔开
- 每个初始数据必须符合对应的成员项的数据类型。



```
printf(“%d, %s, %d\n”, stu1.num, stu1.name, stu1.age);
```

例1: 输出平均分最高的学生信息

- 假设学生的基本信息包括学号、姓名、三门课程成绩以及个人平均成绩。要求在main函数中输入 n 个学生的成绩信息， 计算并输出平均分最高的学生信息。

运行结果

```
Input n: 3
Input the student's number, name and course scores
No. 1: 101 Zhang 78 87 85
No. 2: 102 Wang 91 88 90
No. 3: 103 Li 75 90 84
num: 102, name: Wang, average: 89.67
```

例1 程序解析

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

```
#include<iomanip.h>
```

```
struct Student{    /* 学生信息结构定义 */
```

```
    int num;          /* 学号 */
```

```
    char name[20];    /* 姓名 */
```

```
    int computer, english, math; /* 三门课程成绩 */
```

```
    double average;   /* 个人平均成绩 */
```

```
};
```

```

int main(void)
{
    int i, n;

    struct Student s1,max;    /* 定义结构变量 */
    printf("Input n:");    scanf("%d", &n);
    printf("Input the student's number, name and course scores\n");
    for(i = 1; i <= n; i++){
        fflush(stdin); //清空输入缓冲区 <iomanip.h>
        printf("No. %d:", i);
        scanf("%d %s %d %d %d", \
            &s1.num,s1.name, &s1.computer,&s1.english, &s1.math);
        s1.average=(s1.computer + s1.english + s1.math) / 3.0;
        if(i == 1){ max = s1; continue; }    /* 结构变量操作 */
        if( max.average < s1.average ) max = s1;
    }
    printf("num:%d, name: %s, average: %.2lf", max.num,max.name,max.average);

    /*为变量设置输出宽度、输出精度*/

    return 0;
}

```

```

struct Student{    /* 学生
    int num;        /* 学号
    char name[20];    /* 姓名
    int computer, english, math;
    double average;    /* 个人
};

```

特点：可以传递多个数据 且 参数形式简单

结构变量的用途---作为函数参数

例2：假设学生的基本信息包括学号、姓名和三门课成绩。要求在main函数中为各成员赋值，并在另一函数print中将他们的值输出。

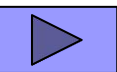
```
#include<stdio.h>
#include<string.h>
typedef struct {
    //声明结构体类型
    int num;          // 学号
    char name[20];    // 姓名
    float score[3];   // 3门课成绩
} Student;
```

```
void print( Student stu)
{
```

```
    printf("num:%d, name: %s, score: %.2lf, %.2lf, %.2lf", \
        stu.num,stu.name,stu.score[0],stu.score[1],stu.score[2]);
```

```
}
```

```
void print( Student ); //必须先前向声明
int main( )
{
    Student s1;
    s1.num=123;
    strcpy(s1.name, "Li");
    s1.score[0]=90;
    s1.score[1]=92;
    s1.score[2]=95.5;
    print(s1); //Student stu=s1;
    return 0;
}
```



结构变量作为形参时，函数调用过程分析

```
print(s1); //void print( Student stu){...};
```

Student stu=s1; //函数调用时，1) 编译器首先给形参**stu**分配空间;
2) 实参**s1**给形参**stu**赋值

// 函数调用结束，形参变量**stu**空间被回收

// 按照结构中定义成员的顺序依次赋值

```
stu.num=s1.num;
```

```
stu.name=s1.name;
```

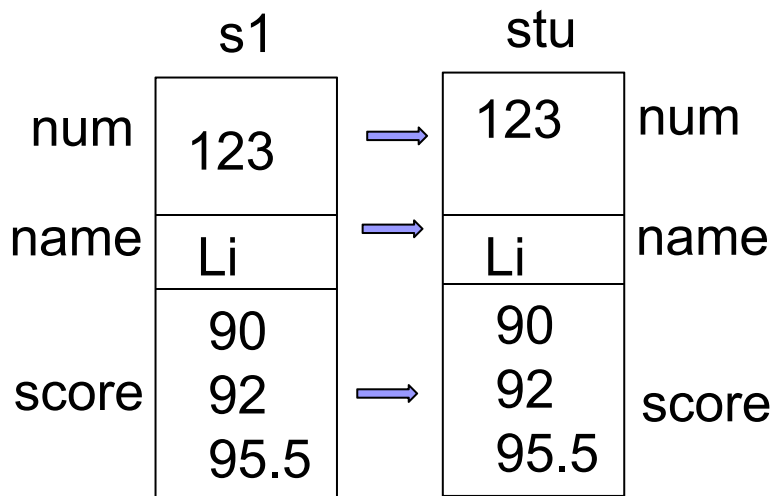
```
stu.score=s1.score;
```

位模式拷贝

结构变量作为参数缺点：（值传递）

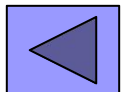
1) 占用内存较多

2) 效率较低：（数据复制效率低；
分配空间、回收空间消耗时间）



形参变量**stu**的作用域，
仅限于**print**函数体

一般采用结构指针或引用作为参数（修改例2）



使用结构变量注意事项

- 1) 只有同类型的结构变量之间可以相互赋值，如`stu=s1;`
- 2) 既可以引用结构变量成员的地址，也可以引用结构变量的地址

如，`&stu1.num;` // 引用`stu1`的成员`num`的地址

`&stu1;` // 引用`stu1`的首地址

- 3) 不能将一个结构体变量作为一个整体进行输入和输出
- 4) 对结构变量成员，如同普通变量，可以进行相关运算

`stu.age++;` `++stu.age;`

- 5) 如果结构成员本身是另一个结构类型，则要用若干个“.”一级一级地找到最低的一级的成员。（逐级引用）

如，`stu1.add.city;`

即：只能对最低级的成员进行赋值、存取、运算



结构变量的赋值操作

- **同类型**的结构变量可以相互赋值。例如：

```
struct point {  
    int x;  
    int y;  
} a={1,2}, b;
```

则

b=a; /* 合法，对应的各个成员赋值 */

b={3,5} ; /* 错 */

b=(struct point){ 3,5}; /* 对，C99复合文字：结构常量 */



结构的嵌套定义

- 在实际生活中，一个较大的实体可能由多个成员构成，而这些成员中有些又有可能是由一些更小的成员构成。
- 如，在学生信息中包含：“生日”（年、月、日），或“通信地址”，它又可以再划分为：城市、街道、门牌号、邮政编码

| num | name | 通信地址 | | | | sex | age | score |
|-----|------|------|----|-----|----|-----|-----|-------|
| | | 城市 | 街道 | 门牌号 | 邮编 | | | |

由此，我们可以定义嵌套的结构：

先定义成员的结构类型，再定义主结构类型。

```
typedef struct
{
    int num;
    char name[20];
    struct Address add;
    char sex;
    int age;
    float score;
} Student;
```

定义结构变量stu1并初始化：

```
Student stu1={112,“Wang Lin”,
               {“Wuhan”, “Guanshan street”,168, 430074},
               ‘M’,19, 93.5};
```

```
struct Address
{
    char city[15];
    char street[15];
    int houseNo;
    int zipcode;
};
```

| num | name | 通信地址 | | | | sex | age | score |
|-----|------|------|----|-----|----|-----|-----|-------|
| | | 城市 | 街道 | 门牌号 | 邮编 | | | |

访问嵌套成员时，需要逐级引用，如

printf(“%s”,stu1.add.city); //输出城市

即 结构变量名.结构成员名.成员名

2. 结构数组【9.3】

- 当需要表示多个相同类型的实体，就需要使用结构数组
- 定义：与普通数组类似

`struct Student students[50];`

结构数组students，它有50个数组元素，从students[0]到students[49]，每个数组元素都是一个结构类型struct Student的变量

| | | | | | | |
|--------------|-----|-------|-----|-----|-----|--|
| students[0] | 101 | Zhang | 76 | 85 | 78 | |
| students[1] | 102 | Wang | 83 | 92 | 86 | |
| ... | ... | ... | ... | ... | ... | |
| students[49] | | | | | | |

■ 结构数组的初始化

```
struct Student students[50] = {  
    { 101,"zhang", 76, 85, 78 },  
    { 102, "wang", 83, 92, 86 } //用{}为结构元素初始化  
};
```

```
struct Student{           /* 学生  
    int num;              /* 学号  
    char name[20];        /* 姓名  
    int computer, english, math;  
    double average;       /* 个人  
};
```

| | | | | | | |
|--------------|-----|-------|-----|-----|-----|--|
| students[0] | 101 | Zhang | 76 | 85 | 78 | |
| students[1] | 102 | Wang | 83 | 92 | 86 | |
| ... | ... | ... | ... | ... | ... | |
| students[49] | | | | | | |

■ 结构数组的使用方法与结构变量完全相同

```
students[i].num = 101;  
strcpy(students[i].name, "zhang");  
students[i] = students[k];
```

设有数组a，则表达式：

sizeof a / sizeof a[0]

计算数组a中元素的个数。

例3 结构数组的使用--显示学生信息

```
#include <stdio.h>
#define STUNUM 3
struct Data {           //定义一个结构
    char name[20];       // 姓名
    short age;           // 年龄
    char adr[30];        // 地址
    long tel;            // 电话号码
};
int main( )
{
    //定义一个结构数组并初始化
    struct Data stu[STUNUM] = {
        {"王伟", 20, "东八舍416室", 87543641},
        {"李玲", 21, "南三舍219室", 87543945},
        {"张利", 19, "东八舍419室", 87543645}
    };
    int i;
```

// 显示表头提示信息

```
printf("编号\t姓名\t年龄\t地址\t电话\n\n");
```

//输出结构数组的数据

```
for(i = 0; i < STUNUM; i++) //print(stu[i]);
    printf("%-d\t%-s\t%-d\t%-s\t%ld\n", \
        i+1,stu[i].name, stu[i].age, \
        stu[i].adr,stu[i].tel);
```

// 每个输出项都左对齐，编号从1开始

```
printf("\n结构类型Data的数据长度：\n
    %d字节\n", sizeof(struct Data));
return 0;
```

} 运行结果：

| 编号 | 姓名 | 年龄 | 地 址 | 电 话 |
|----|----|----|---------|----------|
| 1 | 王伟 | 20 | 东八舍416室 | 87543641 |
| 2 | 李玲 | 21 | 南三舍219室 | 87543945 |
| 3 | 张利 | 19 | 东八舍419室 | 87543645 |

结构类型Data的数据长度：56字节

例 对候选人得票进行统计：设有4个候选人，N个人参加投票选举，每次输入一个得票的候选人的名字，要求最后输出个人的得票结果。

```
#include <stdio.h>
#include <string.h>
#define N 10
struct person //结构定义
{
    char name[20];
    int count;
};
int main()
{
    //结构数组定义及初始化
    struct person leader[4]={\
        {"wang",0}, {"zhang",0},\
        {"zhou",0}, {"gao",0}
    }; //每人票数初始化为0
    int i, j;
    char name[20];
```

```
//模拟选举过程，循环一次代表一次选举
    for(i=0;i<N;i++)
    {
        gets(name);
        //在候选人中查找匹配的人
        for(j=0; j<4; j++)
            if(strcmp(name,leader[j].name)==0)
            {
                leader[j].count++; //更新得票数
                break;
            }
    }
    printf("\n");
    for(j=0; j<4; j++)
        printf("%s:%d\n", leader[j].name,\
            leader[j].count);
    }
```

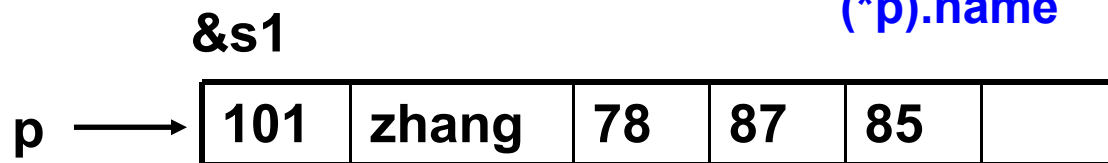
运行结果：
//输入
wang
wang
zhang
zhang
zhang
zhang
zhou
zhou
gao
gao
//输出
wang:2
zhang:4
zhou:2
gao:2

3. 结构指针---指向结构变量的指针【9.4】

```
struct student s1 = {101, "zhang", 78, 87, 85}; //定义结构变量s1
```

```
struct student *p=&s1; //将结构指针p指向结构变量s1
```

从而，通过***p**可以访问**p**所指向的结构变量**s1**的成员内容，如，**(*p).num**
(*p).name



结构指针的使用

(1)用 **->** 访问指针指向的结构成员。如：

```
printf("%d",p->num); strcpy(p->name,"Zhang");
```

(2) 用***p**访问结构成员。如：

```
printf("%d", (*p).num);
```

注意：括号不能少！

***p.num是非法表达式！**

下面三条语句等效：

```
s1.num = 101;
```

```
p->num = 101;
```

```
(*p).num = 101;
```


例 通过结构指针访问结构成员

```
#include <stdio.h>
struct Date //定义一个结构
{
    int month;
    int day;
    int year;
};
```

运行结果：
Today is 4/15/1990

```
int main()
{
    //定义一个结构变量和结构指针变量
    struct Date today, *date_p;
    //将结构指针指向一个结构变量
    date_p = &today;
    //采用结构指针给目标变量赋值
    date_p->month = 4;
    date_p->day = 15;
    date_p->year = 1990;
    //采用结构指针输出目标变量的数据
    printf("Today is %d/%d/%d\n", \
        date_p->month, date_p->day, \
        date_p->year );
    return 0;
}
```

结构指针p的增量运算

若 $p = \text{stu}$; 即指向第一个元素, 则(各语句相互独立)

$p++$;

p 指向下一个结构变量 $\text{stu}[1]$, 即 $p = \&\text{stu}[1]$;

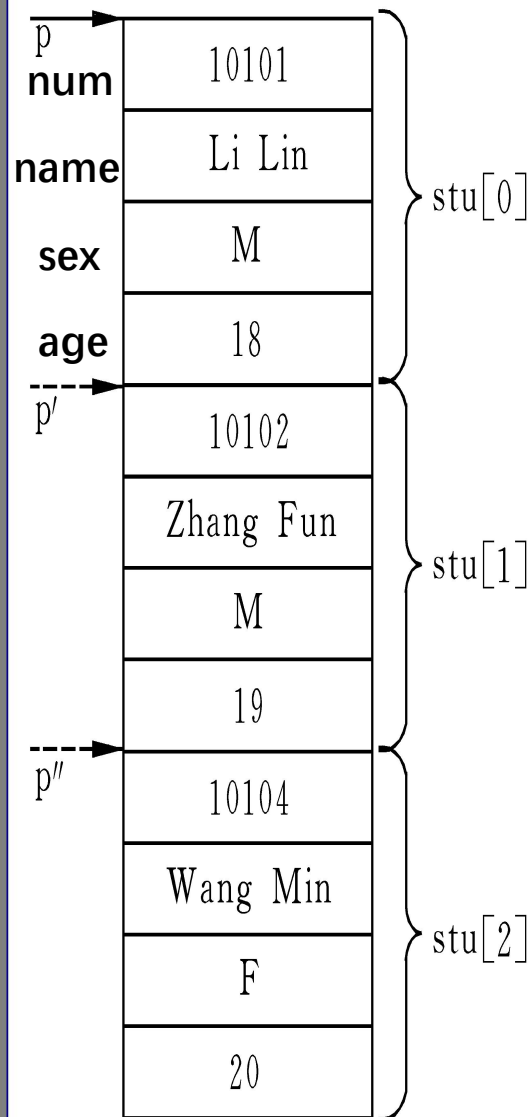
$(++p) \rightarrow \text{num}$;

先使 p 自加 1 (即 $p = \&\text{stu}[1]$;) 然后得到它指向的结构体元素中的 num 成员值 (即 10102)

$(p++) \rightarrow \text{num}$;

先得到 $p \rightarrow \text{num}$ 的值 (即10101), 然后使 p 自加 1, 指向 $\text{stu}[1]$

注意与 $++p \rightarrow \text{num}$ 、 $p \rightarrow \text{num}++$ 的区别



请分析以下几种运算：

$p \rightarrow \text{num}$ //假设 $\text{num}=100$

获取 p 指向的结构变量中的成员 num 的值

$p \rightarrow \text{num}++$

获取 p 指向的结构变量中的成员 num 的值，用完该值后
 num 加 1，即先用后变

$++p \rightarrow \text{num}$

获取 p 指向的结构变量中的成员 num 的值，先使 num 加 1，
然后再使用它，即先变后用

例 指向结构数组的指针

```
#include <stdio.h>
struct student //定义一个结构
{
    int No;
    char name[20];
    char sex;
    int age;
};
int main()
{
    //定义一个结构数组并初始化
    struct student stu[3]={
        {10101,"Li Lin",'M',18},
        {10102,"Zhang fan",'M',19},
        {10104,"Wang Min",'F',20}
    };
```

```
//定义一个结构指针变量
struct student *p;
printf("No. Name Sex Age\n");
for( p=stu; p<stu+3; p++)
    printf("%8d%-12s%6c%4d\n",\
        p->No,p->name,p->sex,p->age);
return 0;
}
```

p++, 指向内存中下一个结构变量
或结构数组中下一个元素

运行结果:

| No. | Name | Sex | Age |
|-------|-----------|-----|-----|
| 10101 | Li Lin | M | 18 |
| 10102 | Zhang fan | M | 19 |
| 10104 | Wang Min | F | 20 |

计算表达式的值

设有：

```
char u[]="abcde";
```

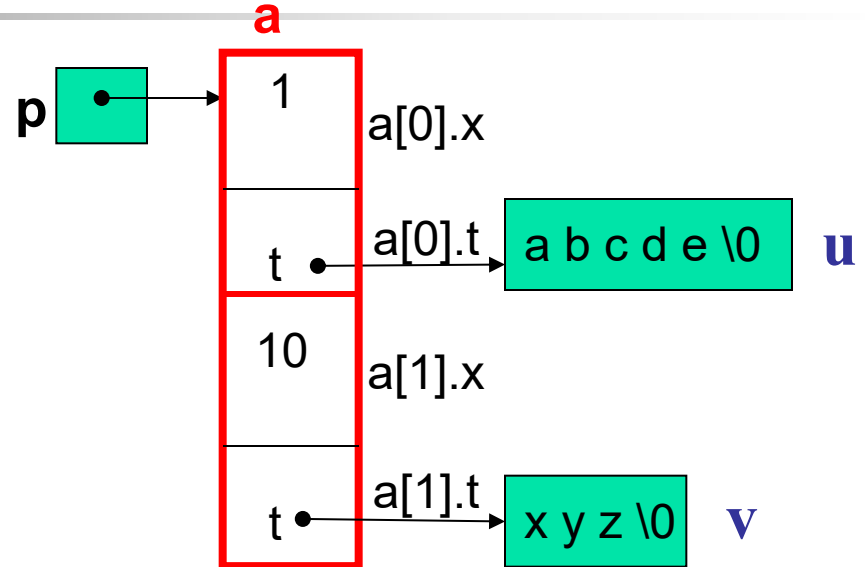
```
char v[]="xyz";
```

```
struct {
```

```
    int x;
```

```
    char *t;
```

```
}a[ ]={{1,u},{10,v}}, *p=a;
```



各表达式相互无关

| 表达式 | 值 | 表达式执行的操作 |
|-----------------------------------|---|----------|
| <code>++p->x</code> | | |
| <code>p->x++</code> | | |
| <code>(++p)->x</code> | | |
| <code>p++->x , *p->t</code> | | |
| <code>*p->t++</code> | | |
| <code>*++p->t</code> | | |
| <code>++*p->t</code> | | |
| <code>*(++p)->t</code> | | |

计算表达式的值【例9.4】

设有：

```
char u[]="abcde";
```

```
char v[]="xyz";
```

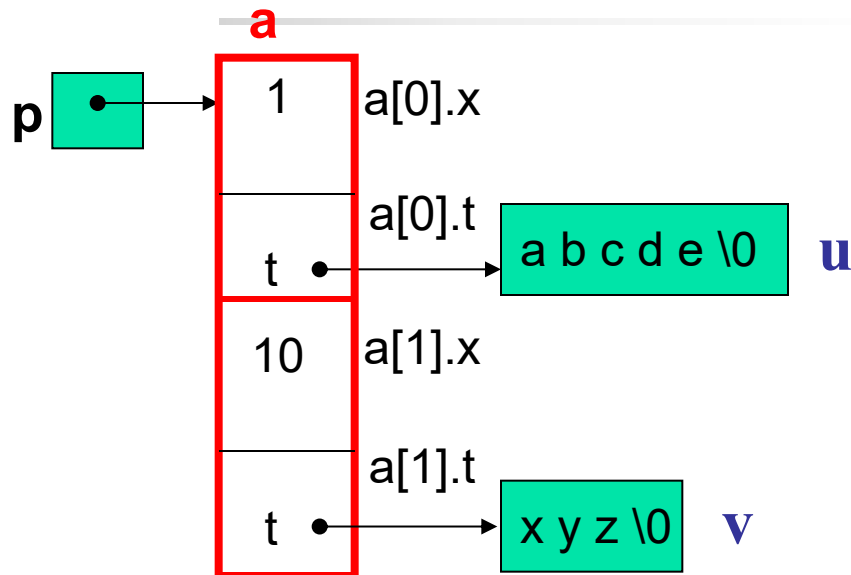
```
struct {
```

```
    int x;
```

```
    char *t;
```

```
}a[ ]={{1,u},{10,v}}, *p=a;
```

各表达式相互无关

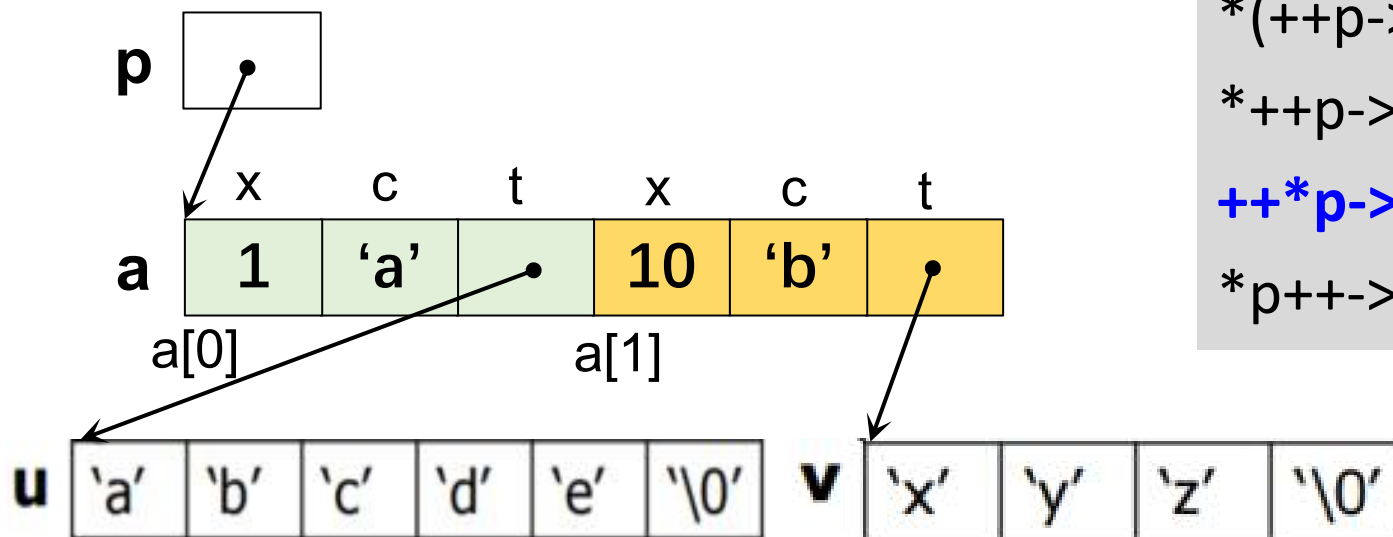


| 表达式 | 值 | 表达式执行的操作 |
|----------------------------------|-----|------------------------------|
| <code>++p->x</code> | 2 | 访问x并使其增1 |
| <code>p->x++</code> | 1 | 访问x,取其原值参与运算,再使x增1 |
| <code>(++p)->x</code> | 10 | p先自增,指向a[1],再访问x |
| <code>p++->x, *p->t</code> | 'x' | 逗号表达式,逗号右侧p指向a[1],访问t所指字符'x' |
| <code>*p->t++</code> | 'a' | 先访问t所指字符'a'后,t增1指向字符'b' |
| <code>*++p->t</code> | 'b' | 先访问t,然后t增1,再访问t所指字符'b' |
| <code>++*p->t</code> | 'b' | 先访问t,取其值'a'后,'a'值+1,即得字符'b' |
| <code>*((++p)->t)</code> | 'x' | p先自增,指向a[1],再访问t所指字符'x' |

`*(p++)->x;` 非法操作, `p->x`值为整数1, `*(p->x)`即取1所指地址的空间内容

```
char u[] = "abcde";
char v[] = "xyz";
struct T {
    int x;
    char c;
    char *t;
};
```

```
struct T a[ ] = {{1, 'a', u}, {10, 'b', v}}, *p=a;
```



判断下列表达式是否正确，然后计算其值
(各语句相互独立)

`(++p)->x;` 10

`++p->x;` 2

`(p++)->x;` 1

`*(p++)->x;` 错误表达式

`*(++p)->t;` x

`*(++p->t);` b

`*++p->t;` b

`++*p->t;` b

`*p++->t;` a

有什么差异？

```
int main (void) {  
    char pm[] = {"abcdef"};  
  
    (*pm)++;  
    ++*pm;  
    printf ("%s\n", pm);  
    return 0;  
}
```

cbcddef

```
int main(void) {  
    char *pm = {"abcdef"};  
  
    (*pm)++;  
    ++*pm;  
    printf ("%s\n", pm);  
    return 0;  
}
```

非法操作!

例 结构指针运算【原例标黄处错误】

```
#include <stdio.h>
struct Key
{
    char *keyword;
    int keyno;
};
void main()
{
    struct Key kd[ ]={{“are”, 123},\
        {“my”, 456}, {"you", 789} };
    struct Key *p=kd;
    int a;
    char chr;
    a=p->keyno;
    printf("p=%d,a=%d\n", p, a);
    a=++p->keyno; //a=++(p->keyno)
    printf("p=%d, a=%d\n", p, a);
    a=(++p)->keyno; //变化后指向目标的成员
    printf("p=%d,a=%d\n", p, a);
```

```
    a=(p++)->keyno; //变化前指向目标的成员
    printf("p=%d,a=%d\n", p, a);
    p=kd; //重新定位p
    chr=*p->keyword;
    printf("p=%d, chr=%c(adr=%d)\n",\
        p, chr, p->keyword);
    chr=*p->keyword++;
    printf("p=%d,chr=%c(adr=%d)\n",\
        p, chr, p->keyword);
    chr=(*p->keyword)++; 不能修改*p->keyword
    printf("p=%d,chr=%c(adr=%d)\n",\
        p, chr, p->keyword);
    chr=*p++->keyword; //(*p->keyword), p++
    printf("p=%d,chr=%c(adr=%d)\n",\
        p, chr, p->keyword);
    chr=*++p->keyword;
    printf("p=%d,chr=%c(adr=%d)\n",\
        p, chr, p->keyword);
}
```

例8.5 结构指针运算【纠正版】：使成员指针指向变量地址

```
#include <stdio.h>
struct Key
{
    char *keyword;
    int keyno;
};
void main()
{ char s[ ][4]={"are","my","you"};
  struct Key kd[ ]={{s[0], 123},\
                    {s[1], 456}, {s[2], 789} };
  struct Key *p=kd;
  int a;
  char chr;
  a=p->keyno;
  printf("p=%d,a=%d\n", p, a);
  a=++p->keyno; //a=++(p->keyno)
  printf("p=%d, a=%d\n", p, a);
  a=(++p)->keyno; //变化后指向目标的成员
  printf("p=%d,a=%d\n", p, a);
```

```
    a=(p++)->keyno; //变化前指向目标的成员
    printf("p=%d,a=%d\n", p, a);
    p=kd; //重新定位p
    chr=*p->keyword;
    printf("p=%d, chr=%c(adr=%d)\n",\
           p, chr, p->keyword);
    chr=*p->keyword++;
    printf("p=%d,chr=%c(adr=%d)\n",\
           p, chr, p->keyword);
    chr=(*p->keyword)++;
    printf("p=%d,chr=%c(adr=%d)\n",\
           p, chr, p->keyword);
    chr=*p++->keyword; //(*p->keyword), p++
    printf("p=%d,chr=%c(adr=%d)\n",\
           p, chr, p->keyword);
    chr=*++p->keyword;
    printf("p=%d,chr=%c(adr=%d)\n",\
           p, chr, p->keyword);
}
```

例 结构指针运算【纠正版】：使成员指针指向新分配的地址

```
#include <stdio.h>
struct Key
{
    char *keyword;
    int keyno;
};
void main()
{
    //初始化时成员指针置空
    struct Key kd[ ]={{NULL,123},\
        {NULL, 456}, {NULL,789} };
    struct Key *p=kd;
    int i=0; char chr;
    char s[ ][4]={"are","my","you"};
    for(i=0;i<3;i++)
    {
        //依次为每个成员指针申请所需要的空间
        kd[i].keyword=(char *) malloc(sizeof(s[i]));
        strcpy(kd[i].keyword, s[i]);
    }
}
```

```
p=kd;
chr=*p->keyword;
printf("p=%d, chr=%c(adr=%d)\n",\
    p, chr, p->keyword);
chr=*p->keyword++;
printf("p=%d,chr=%c(adr=%d)\n",\
    p, chr, p->keyword);
chr=(*p->keyword)++;
printf("p=%d,chr=%c(adr=%d)\n",\
    p, chr, p->keyword);
chr=*p++->keyword;
printf("p=%d,chr=%c(adr=%d)\n",\
    p, chr, p->keyword);
chr=*++p->keyword;
printf("p=%d,chr=%c(adr=%d)\n",\
    p, chr, p->keyword);
for(i=0;i<3;i++)
    free(kd[i].keyword);
} //用完后要free空间
```

4. 结构类型作为函数的参数

将一个结构变量中的数据传递给另一个函数，有3种方法：

(1) 用结构变量的成员作参数 值传递—数据的单向传递

如，`query(stu1.num);` //根据学号查询学生信息

(2) 用结构变量作参数 值传递—形参结构不影响实参结构的值

如，`void outputs(Student stu);` //输出学生信息

将结构变量所占的内存单元的内容全部顺序传递给形参（数据复制）

占用内存多，传参、分配空间/回收空间耗时长，适用于较小的结构。

(3) 用结构引用/指针作参数 地址传递—数据的双向传递

如，`void inputs(Student& stu);`

`void sort(Student* stu);`

效率高，最常用方式

占用内存少，传参时间短，适用于较大的结构

传地址，适合于要修改实参指针所指的结构变量值的情况

引用类型--被引用变量的别名

```
int i=0;
```

```
int &fi=i; //定义fi为i的引用变量，引用变量一旦定义不能修改，  
          因此，引用变量在声明的同时必须进行初始化
```

- 引用变量是不需要分配空间的，它总是与被引用变量共享同一个存储空间，因此，对引用变量的修改，也即对被引用变量的修改。

如，`fi++;` // 即 `i=1`

引用变量除声明方式外，其用法与普通变量基本相同，但**无引用数组**

- 引用变量是通过地址与被引用变量产生联系，达到与指针相同的效率。因此，引用变量主要作为函数参数，通过地址关联实现数据的双向传递。

如，`void modify(struct Student &stu);` 不能对结构引用变量进行++等算术运算

函数调用时，`modify(stu1);` //用实参stu1给形参引用变量stu初始化，
即，`struct Student &stu=stu1;` 从而实现stu1与stu共享同一空间

用结构指针作为函数参数

```
#include <stdio.h>
#define MAX 30
struct Address //定义一个通信录的结构
{
    char name[20];
    char addr[50];
    char tel[15];
};
int input(struct Address *pt); //录入信息
void display(struct Address *pt, int n); //显示通讯录信息

int main()
{
    struct Address man[MAX]; //定义结构数组
    int i, con=0;
    for(i=0;i<MAX;i++) //建立通信录
    {
        con=input(&man[i]); //传结构变量地址
        if(con==0) break;
    }
    display(man, i); //显示整个通信录, 传数组名
    return 0;
}
```

例 通信录的建立和显示

```
// 以人机对话方式输入数据, 形参为结构指针
int input(struct Address *pt)
{
    printf("Name?");    gets(pt->name);
    if(pt->name[0]=='0') return 0; //结束输入
    printf("Address?");  gets(pt->addr);
    printf("Telephone?"); gets(pt->tel);
    return 1;    //正常返回1
}

// 显示通讯录信息, 形参为结构指针, 指向数组首址
void display(struct Address *pt, int n)
{
    int i;
    printf("name      address      tel\n");
    printf("-----\n");
    for(i=0 ; i<n ; i++, pt++)
        printf("%-15s%-30s%s\n", pt->name, \
            pt->addr, pt->tel);
}
```

用结构引用作为函数参数

```
#include <stdio.h>    //*.cpp
#define MAX 30
struct Address //定义一个通信录的结构
{
    char name[20];
    char addr[50];
    char tel[15];
};
int input(struct Address &pt); //录入信息
void display(struct Address *pt, int n);
                                //显示通讯录信息
int main()
{
    struct Address man[MAX]; //定义结构数组
    int i, con=0;
    for(i=0;i<MAX;i++)        //建立通信录
    { con=input(man[i]);      //传结构变量名
      if(con==0) break;
    }
    display(man, i); //显示整个通信录, 传数组名
    return 0;
}
```

不能对结构引用变量进行++等算术运算

例 通信录的建立和显示

```
// 以人机对话方式输入数据, 形参为结构引用变量
int input(struct Address &pt)
{
    printf("Name?");    gets(pt.name);
    if(pt->name[0]=='0') return 0; //结束输入
    printf("Address?");  gets(pt.addr);
    printf("Telephone?"); gets(pt.tel);
    return 1;    //正常返回1
}

// 显示通讯录信息, 形参为结构指针, 指向数组首址
void display(struct Address *pt, int n)
{
    //无引用数组, 此处若改为引用形参, 将出错
    int i;
    printf("name      address      tel\n");
    printf("-----\n");
    for(i=0 ; i<n ; i++, pt++)
        printf("%-15s%-30s%s\n", pt->name, \
            pt->addr, pt->tel);
}

引用变量的++等操作, 是对变量值++, 不像指针那样, 以所指类型长度为单位移动
```

//输入
该程序运行结果为:
Name?王伟
Address?东八舍416室
Telephone? 87543641
Name?李玲
Address?南三舍219室
Telephone? 87543945
Name?张利
Address?东八舍419室
Telephone?87543645
0

//输出

| name | address | tel |
|------|---------|----------|
| 王伟 | 东八舍416室 | 87543641 |
| 李玲 | 南三舍219室 | 87543945 |
| 张利 | 东八舍419室 | 87543645 |

结构类型作为函数的返回值

返回结构变量的函数

返回结构指针的函数


实现把函数中处理的若干结构的数据返回给调用它的函数中

用结构变量作为函数返回值

例 求两点之间的距离

```
#include <stdio.h>
#include <math.h>
struct point //点结构类型声明
{
    int x;
    int y;
};
struct point Point (int x, int y); //初始化一个点
double distance(struct point *, struct point *);
//求两点之间的距离

int main()
{
    double d;
    struct point a, b; //声明点结构变量
    a= Point(0, 0);
    b= Point(2, 3);
    d= distance(&a, &b); //计算距离
    printf("the distance is %f\n",d); //输出
    return 0;
}
```



d= distance(a, b);

// 返回结构变量，以初始化点的成员信息

```
struct point Point (int x, int y)
```

```
{
```

```
    struct point temp;
```

```
    temp.x = x;
```

```
    temp.y = y;
```

```
    return temp;
```

```
}
```

return一个结构变量
适用于较小的结构

// 求两点之间的距离, 形参为**点结构指针**

```
double distance(struct point *p1,
```

```
                struct point *p2)
```

```
{
```

```
    double dx, dy, d;
```

```
    dx= (*p2).x - (*p1).x;
```

```
    dy=p2->y - p1->y;
```

```
    d=sqrt(dx*dx+dy*dy); /*计算距离*/
```

```
    return d;
```

```
}
```

```
double distance(const struct point &p1,
                const struct point &p2);
```

用结构指针作为函数的返回值

```
#include <stdio.h>
struct Sample    //定义一个结构
{
    int num;
    char color;
    char type;
};
struct Sample *find(struct Sample *pd,int n);
void display(const struct Sample &car)
int main()        //采用只读引用参数car
{
    int num;
    struct Sample *result;
    struct Sample car[]={101,'G','c'},
                        {105,'R','l'},{222,'B','s'},
                        {308,'P','b'},{0,0,0};

    printf("Enter the number:");
    scanf("%d",&num); //输入查找样品代号
    result=find(car,num);
    display( *result ); //显示找到的结构信息
    return 0;
}
```

例 查找符合要求的结构元素

/*查找符合要求的结构元素，找到，返回该元素，否则返回NULL*/

```
struct Sample *find(struct Sample *pd,int n)
{ //这里pd若改为引用参数，则不能传数组名
    int i;
    for(i=0 ; pd[i].num!=0 ;i++)
        if(pd[i].num==n) return pd+i; // &pd[i]
        //返回查找结果
    return NULL;
}
```

当要返回一个较大的结构时，使用结构指针

```
void display(const struct Sample &car){
    if(car.num!=0)
    {
        printf("number :%d\n", car.num);
        printf("color :%c\n", car.color);
        printf("type :%c\n", car.type);
    }
    else //未找到，给出提示信息
        printf("not found!");
}
```

结构数组做函数参数

例9.5：输入商品信息（包括商品编码、名称、价格），分别按照价格和名称排序，并输出所有商品信息。

| 商品编码 | 名称 | 价格（元） |
|------|----|-------|
| 1 | 笔 | 2.0 |
| 2 | 毛巾 | 10.5 |
| | | |
| | | |

结构类型声明和相关函数原型

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

```
#define N 10
```

```
typedef struct {
```

```
    long code;          /* 货物编码 */
```

```
    char name[20];      /* 名称 */
```

```
    float price;        /* 价格 */
```

```
} GOODS;
```

```
void input(GOODS *,int);    /*输入n件物品的信息 */
```

```
void display(GOODS *,int); /*显示n件物品的信息 */
```

```
void sort(GOODS *, int , int (*)(const void *,const void *)) /*排序 */
```

```
int cmpbyName(const void *,const void *); /* 按名称比较*/
```

```
int cmpbyPrice(const void *,const void *) ; /* 按单价比较*/
```

```
int main()
{
    GOODS g[N];
    input(g,N);
    display(g,N);
    sort(g,N, cmpbyName);
    display(g,N);
    sort(g,N, cmpbyPrice);
    display(g,N);
    return 0;
}
```



input函数

/* 输入n件物品的信息 */

void input(GOODS *p,int n)

{

for(int i=0;i<n;i++){

scanf("%ld",&p[i].code);

scanf("%s",(p+i)->name);

scanf("%f",&(p+i)->price) ;

}

}



display函数

/*显示n件物品的信息 */

void display(GOODS *p,int n)

{

for(int i=0;i<n;i++){

printf("%ld\t",(*p+i).code);

printf("%s\t",(p+i)->name);

printf("%f\n",p[i].price);

}

}

sort函数

/* 按函数指针fp指明的规则对n件物品排序 */

void sort(GOODS *p, int n,int (*fp)(const void *,const void *))

{

GOODS t;

for(int i=0;i<n-1;i++) { // 冒泡法

for(int j=i+1;j<n;j++) {

if(fp(p+i,p+j)) { //不符合排序规则，交换

t=*(p+i); // *(p+i)等价于p[i]

***(p+i)=*(p+j); // *(p+j)等价于p[j]**

***(p+j)=t;**

}

}

}

}

回调函数： cmpbyPrice

/* 按单价比较*/

```
int cmpbyPrice(const void *s,const void *t)  
{  
    if(s->price < t->price) // 按价格降序  
        return 1;  
    else return 0;  
}
```



©2007 by the author. All rights reserved.



回调函数：cmpbyPrice

/* 按单价比较*/

```
int cmpbyPrice(const void *s,const void *t)  
{  
    const GOODS *p1,*p2;  
    p1=(const GOODS *)s  
    p2=(const GOODS *)t;  
    if(p1->price < p2->price) // 按价格降序  
        return 1;  
    else return 0;  
}
```



回调函数： cmpbyName

/* 按名称比较 */

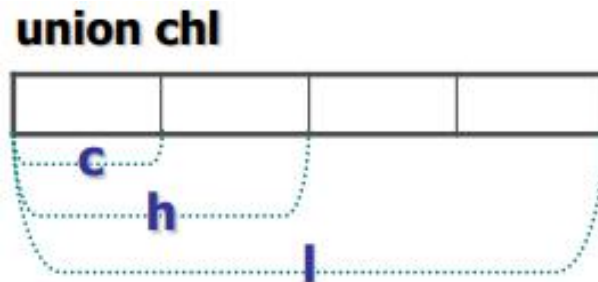
```
int cmpbyName(const void *s,const void *t)  
{  
    const GOODS *p1,*p2;  
    p1=(const GOODS *)s;  
    p2=(const GOODS *)t;  
    return(strcmp(p1->name,p2->name)>0);  
}    // 按名称升序
```

9.6 联合

- 与结构类似，联合类型也是一种构造类型。
- 结构变量的各成员变量占据各自不同空间。
- 联合变量的各成员变量占用同一内存空间。
- 联合特点：各成员共享存储

```
struct chl {  
    char c;  
    short h;  
    long l;  
};
```

```
union chl {  
    char c;  
    short h;  
    long l;  
};
```



联合存储区域的大小由各个成员中所占字节数最大的成员决定

一个联合类型中的所有成员共享同一个存储区域

联合变量的声明、初始化 和成员引用

- 除了用关键字**union**取代**struct**之外，联合类型的定义、联合变量的声明、以及联合成员的引用在语法上与结构完全相同。即可采用3种方式之一：

1 联合类型和其变量分别声明

```
union chl {  
    char c;  
    short h;  
    long l;  
};  
union chl v = {.l=0x9876};
```

v

| | | | |
|----|----|----|----|
| 76 | 98 | 00 | 00 |
|----|----|----|----|

v.c

v.h

v.l

C99标准：
可以对联合变量的任意成员初始化

printf("%c",v.c); ??

2 联合类型和其变量同时声明

```
union chl {  
    char c;  
    short h;  
    long l;  
} v = {.l=0x9876};
```

3 typedef定义之后声明变量

```
typedef union chl {  
    char c;  
    short h;  
    long l;  
} CHL;  
CHL v = {.l=0x9876};
```

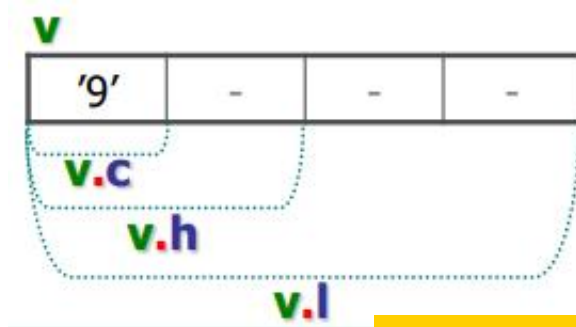
联合变量的声明、初始化 和成员引用

■ 可采用3种方式之一：

- ρ 联合类型和其变量分别声明
- ρ 联合类型和其变量同时声明
- ρ typedef 定义之后声明变量

早期C标准规定：只能对联合的第1个成员进行初始化。

```
union chl {  
    char c;  
    short h;  
    long l;  
} v = {'9'};
```



```
v.h=100;  
v.l=0x9876;  
printf("%c",v.c);  
printf("%c",v.h);
```

//将v中当前的long值作为char和short来解释

联合所有成员都是从同一存储空间的边界（低地址）开始存放，联合各成员的地址和联合变量的地址是相同的，但类型不同

| &v | &v.c | &v.h | &v.l | 值相同 |
|-------------|--------|---------|--------|-----|
| | | | | |
| union chl * | char * | short * | long * | |

例 设有若干人员的数据，其中有学生和教师。

学生的数据中包括：姓名、号码、性别、职业、班级；

教师的数据包括有：姓名、号码、性别、职业、职务。

现要求把它们放在同一表格中，要求先输入人员的数据，然后再输出。

| name | num | sex | job | Class/position |
|------|------|-----|-----|----------------|
| li | 1011 | f | s | 501 |
| wang | 2058 | m | t | professor |

【分析】可以看出：学生和老师包含的信息大部分相同，可以将极少数不同的数据项用union结构组织。若job项为s（学生），则第五项为class；如果job项是t（教师），则第五项为position。

```
union Career //定义一个联合
{
    int myclass;
    char position[10];
};
```

```
struct PersonInfo //定义一个结构
{
    int num;
    char name[10];
    char sex;
    char job;
    union Carrer category;
}
```

联合

```
struct Audio {  
    part A;  
    part B;  
}  
struct Vedio {  
    part A;  
    part C;  
}
```

```
struct AVdio {  
    part A;  
    union {  
        part B;  
        part C;  
    }  
}
```

```
struct AVdio {  
    part A;  
    part B;  
    part C;  
}
```

通过联合数据结构，
既能避免重复代码，
又能避免耦合，
同时还能节约资源



利用union分离short数据的高低字节

```
union {  
    short n;    //存放要进行分离的数据  
    char a[2];  //n与数组a占的字节数相同  
} test;  
  
test.n=0x1234;  
low =test.a[0]; //低字节数据, low=0x34  
high=test.a[1]; //高字节数据, high =0x12
```

利用union判断CPU大小端

如何检查处理器是big-endian还是little-endian?

联合体union的存放顺序是：所有成员都从低地址开始存放，利用该特性就可以轻松地获得了CPU对内存采用Little-endian还是Big-endian模式读写。

```
/*return 1 : little-endian, return 0:big-endian*/
```

```
int checkCPUendian( )
```

```
{
```

```
    union {
```

```
        unsigned int a;
```

```
        unsigned char b;
```

```
    } c;
```

```
    c.a = 1;
```

```
    return (c.b == 1);
```

```
}
```

9.7 字段结构

压缩：把表示21世纪日期的日、月和年3个整数压缩成1个16位的整数。

分析：因为日有31个值，月有12个值，年有100个值，所以可以在一个整数中**用5b表示日**，**用4b表示月**，**用7b表示年**。



由**字中**多个相邻的二进制位组成的存储区域，称为**字段**(bit field)组成字段的二进制位的数目称为**字段宽度**。以字段为成员的结构称为**字段结构**。

字段的类型必须是整型，通常说明为无符号整型，且是在**结构或者联合中进行说明**。

字段结构在操作系统、编译程序、计算机接口的C语言编程方面使用较多

字段结构的声明

```
struct date {  
    unsigned short year : 7 ;    /* 年 */  
    unsigned short month : 4 ;   /* 月 */  
    unsigned short day : 5 ;     /* 日 */  
};
```

同一字段结构中的
所有字段类型相同

↑ ↑
字段名 字段宽度

字段宽度：组成字段的二进制位的数目，是一个非负的整型常量表达式。

字段结构(或联合)类型的定义、变量的声明和成员的引用，与一般结构(或联合)类型是完全相同的。

字段结构变量

```
struct date {
```

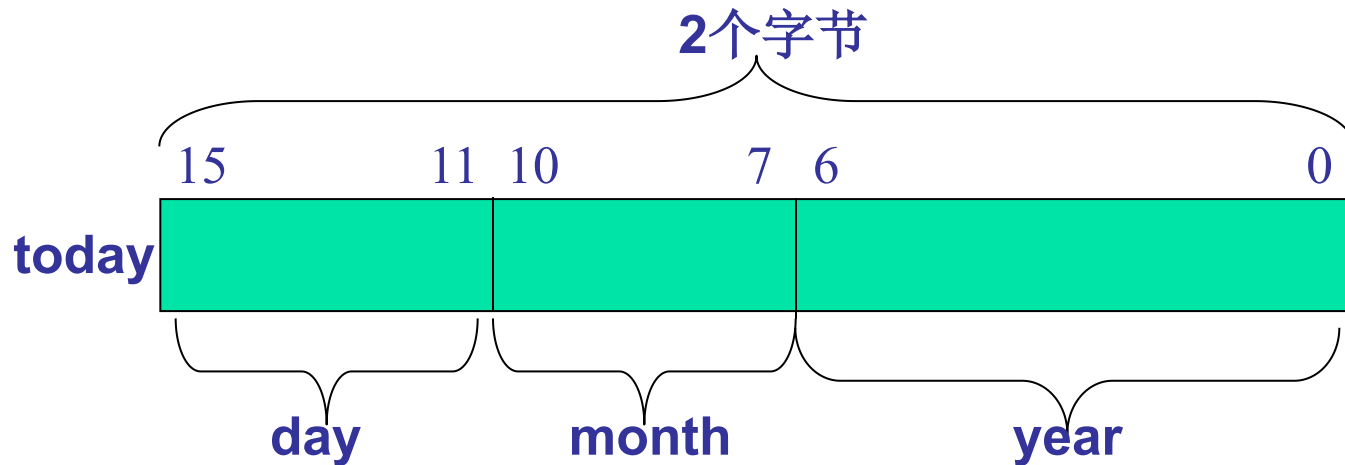
```
    unsigned short year : 7 ;    /* 年 */
```

```
    unsigned short month : 4 ;    /* 月 */
```

```
    unsigned short day : 5 ;    /* 日 */
```

```
};
```

```
struct date today; /* today是date 字段结构变量 */
```



引用字段

- ◆ 与结构体完全相同：. 或 ->
- ◆ 字段就是一个小整数，它可以出现在其它整数可以出现的任何地方。字段在参与运算时，被自动转换为int或unsigned int类型的整数。

today.year=13; (2013)

today.month=5;

today.day=9;

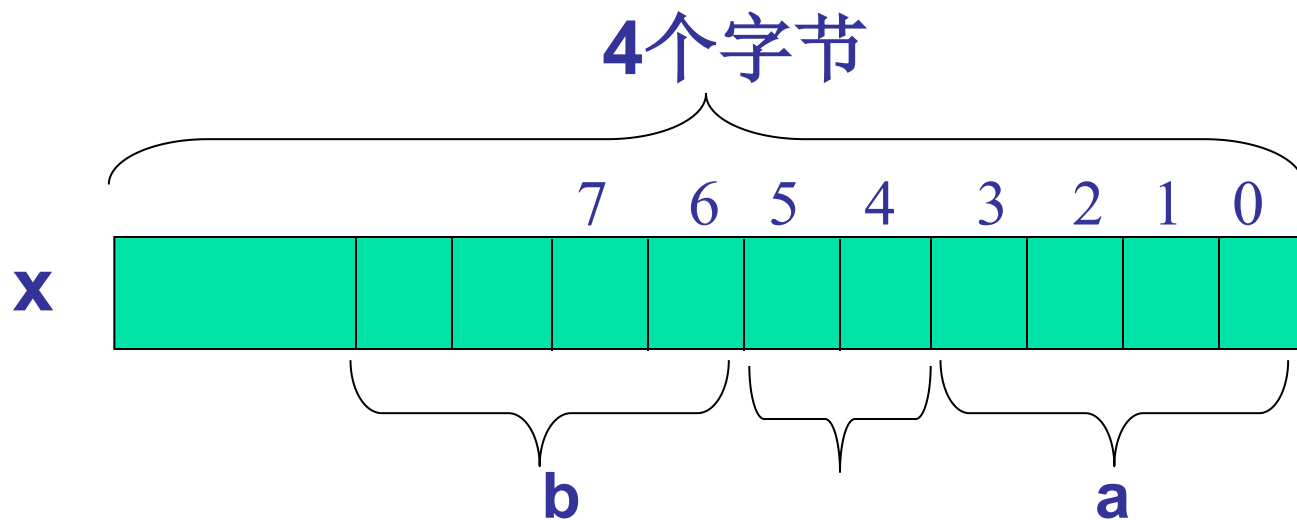
字段使用注意问题

P305

```
struct {  
    unsigned int is_keyword : 1 ;  
    unsigned int is_external : 1 ;  
    unsigned int is_static   : 1 ;  
} flags;  
flags.is_keyword=1; //打开标志位  
flags.is_static=0; //关闭标志位
```

无名字段（匿名）

```
struct {  
    unsigned a: 4;  
            : 2;  
    unsigned b: 4;  
} x;
```



字段结构的应用

考虑十字路口的交通灯控制应用问题。

- ◆ 交通灯分东、南、西、北4组，每组灯都有灯开启时的红灯、黄灯和绿灯以及关闭4种状态。

----每组灯状态可以用2个二进制位描述

- ◆ 每组灯开通时间，每次不超过15分钟。

-----每组灯开通时间可以用4个二进制位描述

- ◆ 4组交通灯的字段结构类型的声明如下：

```
struct traffic_light{
    unsigned short int east:2;      /* 东组灯状态字段 */
    unsigned short int south:2;     /* 南组灯状态字段 */
    unsigned short int west:2;      /* 西组灯状态字段 */
    unsigned short int north:2;     /* 北组灯状态字段 */
    unsigned short int reserve:8;    /* 保留字段 */
    unsigned short int east_on:4;    /* 东组灯开通时间字段 */
    unsigned short int south_on:4;   /* 南组灯开通时间字段 */
    unsigned short int west_on:4;    /* 西组灯开通时间字段 */
    unsigned short int north_on:4;   /* 北组灯开通时间字段 */
};
```


字段结构的应用

```
#include "stdio.h"
enum color{ OFF=0,RED=1,YELLOW=2,GREEN=3};
struct traffic_light{
    unsigned short int east:2;    /* 东组灯状态字段 */
    unsigned short int south:2;   /* 南组灯状态字段 */
    unsigned short int west:2;    /* 西组灯状态字段 */
    unsigned short int north:2;   /* 北组灯状态字段 */
    unsigned short int reserve:8; /* 保留字段 */
    unsigned short int east_on:4;  /* 东组灯开通时间字段 */
    unsigned short int south_on:4; /* 南组灯开通时间字段 */
    unsigned short int west_on:4;  /* 西组灯开通时间字段 */
    unsigned short int north_on:4; /* 北组灯开通时间字段 */
};

int main(void)
{
    struct traffic_light Jiedaokou={0,0,0,0,0,0,0,0,0,0};
    Jiedaokou.west=GREEN;Jiedaokou.west_on=5;
    printf("Jiedaokou.west=%u\t",Jiedaokou.west);
    printf("Jiedaokou.west_on=%u\n",Jiedaokou.west_on);
    printf("Jiedaokou.east=%u\t",Jiedaokou.east);
    printf("Jiedaokou.east_on=%u\n",Jiedaokou.east_on);
    return 0;
};
```

```
Jiedaokou.west=3 Jiedaokou.west_on=5
Jiedaokou.east=0 Jiedaokou.east_on=0
```



字段结构与联合的应用【例9.12】

如何访问**16位字**中的高低字节和各二进制位？

- ◆ 定义**8位宽**的字段**byte0**、**byte1**
 - 表示一个**16位字**中的高/低字节
- ◆ 定义**1位宽**的字段**b0~b15**
 - 表示一个**16位字**中的**bit**
- ◆ 定义联合类型
 - 使一个**short**变量、**2个byte**字段与**16个bit**字段共享存储。

字段结构与联合的应用

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

```
struct w16_bytes{ //定义8位宽的字段byte0、 byte1
```

```
    unsigned short byte0:8; /* byte0: 低字节*/
```

```
    unsigned short byte1:8; /* byte1: 高字节 */
```

```
};
```

```
struct w16_bits{ //定义1位宽的字段b0~b15
```

```
    unsigned short b0:1,b1:1,b2:1,b3:1,b4:1,b5:1,b6:1,b7:1,
```

```
    b8:1,b9:1,b10:1,b11:1,b12:1,b13:1,b14:1,b15:1;
```

```
};
```

```
union w16{ // 定义联合类型， i、 byte、 bit共享存储
```

```
    short i;
```

```
    struct w16_bytes byte;
```

```
    struct w16_bits bit;
```

```
};
```

字段结构与联合的应用

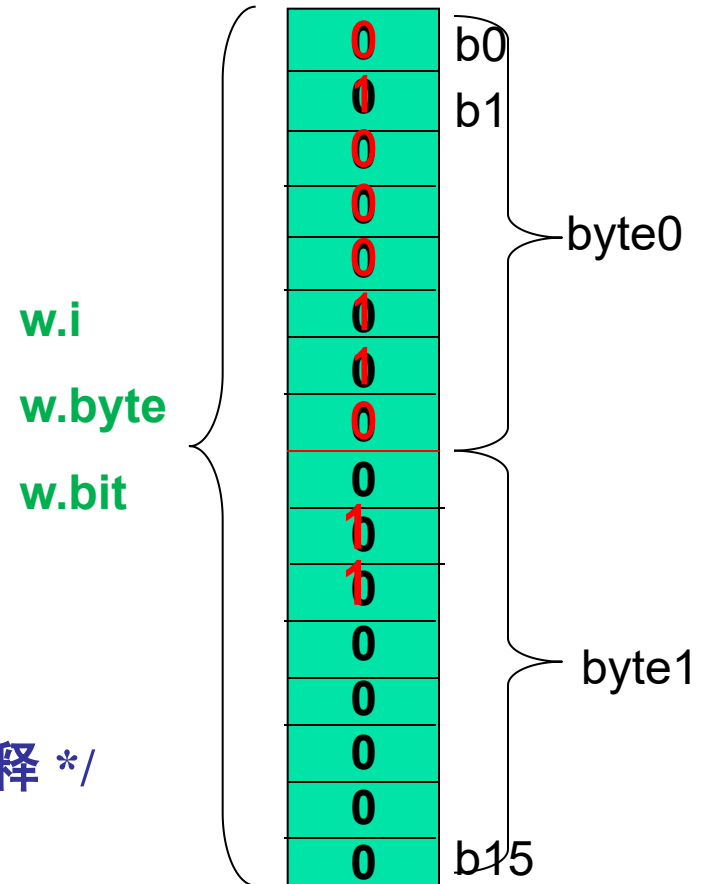
```
int main(void)
{
    union w16 w={0}; /* w.i 为0 */

    w.bit.b9=1;      /* 相当于byte1为2 */
    w.bit.b10=1;     /* 相当于byte1为6 */

    w.byte.byte0=0x62;

    printf("w.i=0x%x\n",w.i); /* 按整型解释 */
    return 0;
}
```

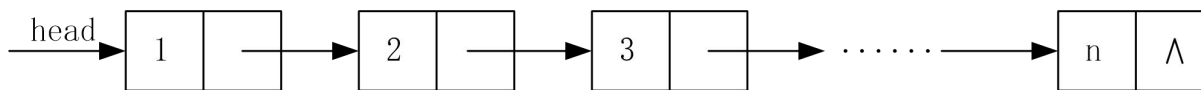
运行结果: **w.i=0x662**



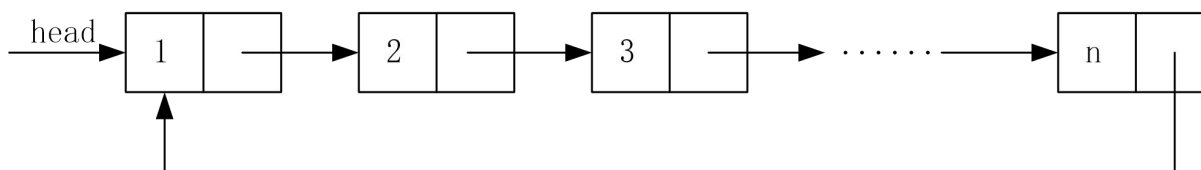
9.8 结构指针的应用—链表

- 数组虽然结构简单、访问元素方便、执行速度快，但
 - 数组是静态数据结构，即必须要预先确定它的大小
 - 只能用**顺序存储**的存储结构，且要求其**存储单元是连续的**
- 链表是一种动态地进行存储分配的数据结构。非常适合处理数据个数预先无法确定、且数据记录频繁变化的场合，**使用动态存储技术和递归结构建立链表**，可通过指针增加或删除结点（Node）。

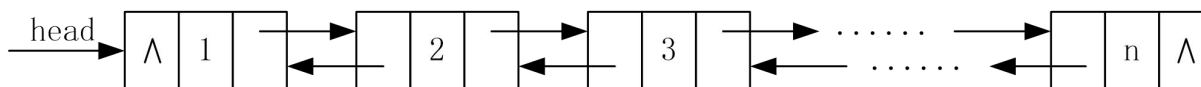
常见链表结构



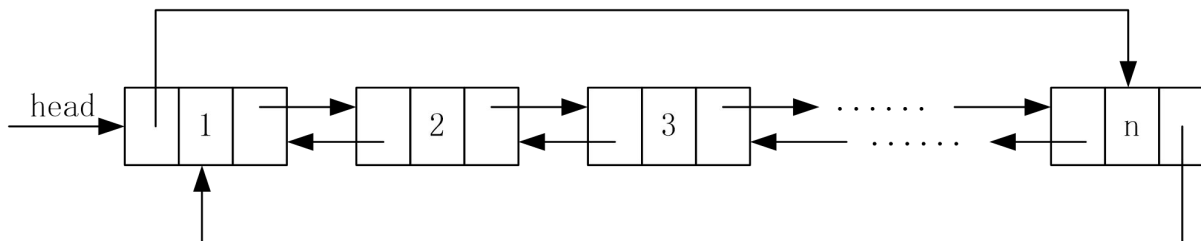
一般单向链表



循环单向链表



一般双向链表



循环双向链表



顺序存储和链式存储

- **顺序存储：数组**，是静态数据结构，其存储是在变量声明时建立的，并且在程序的执行过程中存储大小不能改变。

----- **内存固定，存储单元连续**

- **链式存储：单向链表、双向链表、十字交叉链表**，是动态数据结构，其存储是在程序运行过程中**通过调用系统提供的动态存储分配函数，向系统申请而逐步建立起来的**。在程序运行过程中，所占存储的大小可以根据需要调节，使用完毕时可以通过释放操作将所获得的存储交还给系统供再次分配。

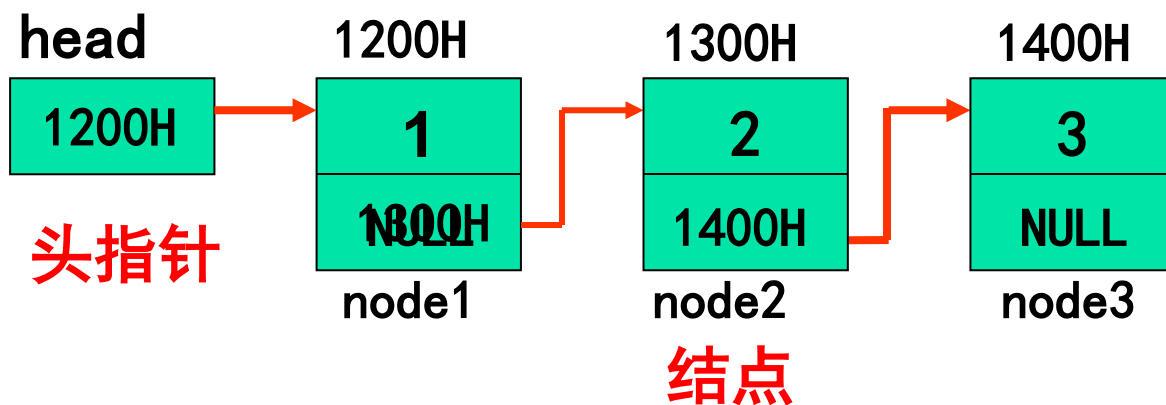
----- **内存可变，存储单元不连续**

自引用结构

如果一个结构类型中包含一个该结构类型自身的指针成员，称该结构类型为**自引用结构类型**，简称**自引用结构**。

```
struct s_list{  
    int data;  
    struct s_list *next;  
} node1={1,NULL} , *head;  
head=&node1;
```

```
struct s_list node2,node3;  
node2.data=2; node3.data=3;  
node1.next=&node2;  
node2.next=&node3;  
node3.next=NULL;
```



静态链表



C 的动态存储分配函数

◆ 动态存储分配函数的原型声明在 **<stdlib.h>** 中

◆ C提供下列与动态存储分配相关的函数

```
void * malloc(size_t size);
```

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

```
void * realloc(void *p_block, size_t size);
```

```
void free(void *p_block);
```

◆ 其中， **size_t** 表示 **unsigned int**。它是在 **<stdio.h>** 中通过 **typedef unsigned size_t;** 定义的。



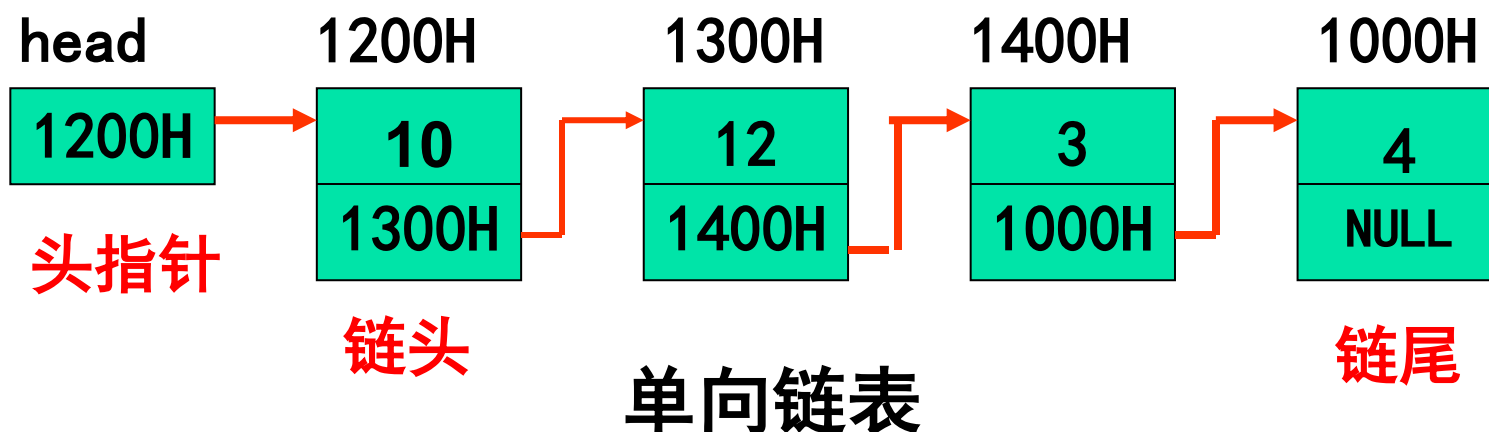
结构指针的应用—链表

要求掌握：

- ◆ 单向链表的建立
- ◆ 单链表的删除
- ◆ 在链表中查找指定的元素
- ◆ 插入一个新元素到链表
- ◆ 删除链表中的一个元素
- ◆ 链表排序

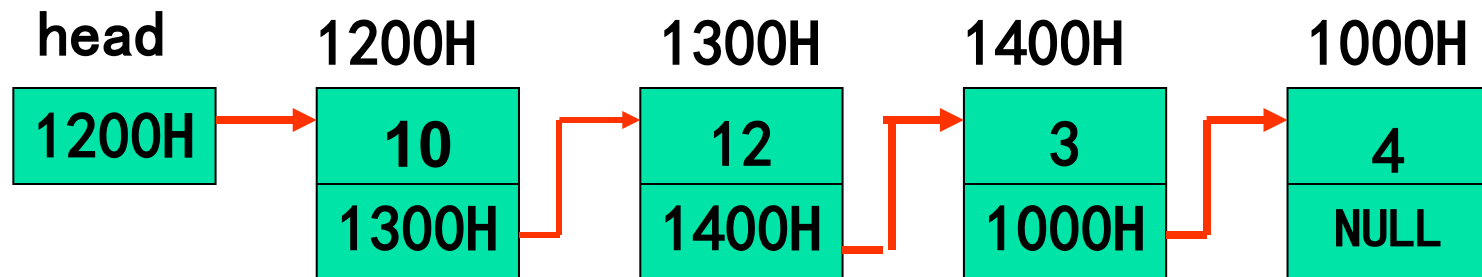
链表

- 链表是一种常用的动态数据结构，它由一系列包含数据域和指针域的结点组成。
- 如果结点的指针域中只包含一个指向后一个结点的指针，这种链表称为单向链表。



当前结点的前面一个结点称为直接前驱结点（简称前驱）
它后面的一个结点称为直接后继结点（简称后继）

单向链表



单向链表结构

head: 头指针, 存放一个地址, 指向第一个元素

结点: 链表中的元素, 包括 **数据域**, 用户数据
指针域, 放下一个结点的地址

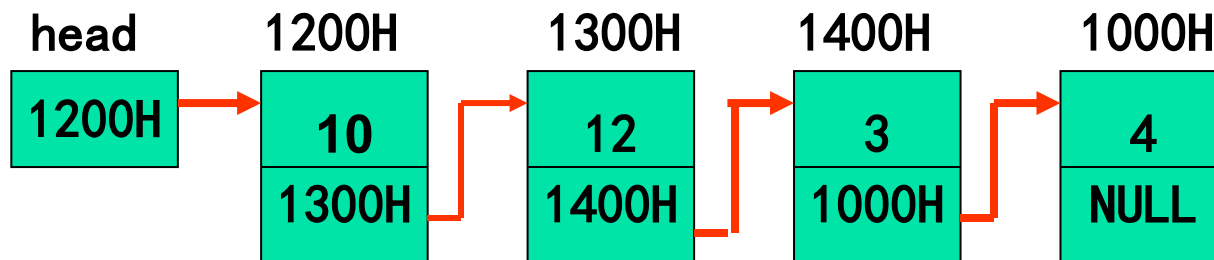
链尾: 最后一个结点, 指针域为NULL

链表结点的结构类型定义

```
struct intNode {  
    int data;  
    struct intNode *next; /*含有一个指向该结构自身的指针*/  
};
```

头指针说明

```
struct intNode *head;
```



动态创建结点

- 可以通过malloc函数来动态创建结点。
- 如：

```
struct intNode *head;
```

```
head=(struct intNode *)malloc(sizeof(struct intNode)) ;
```

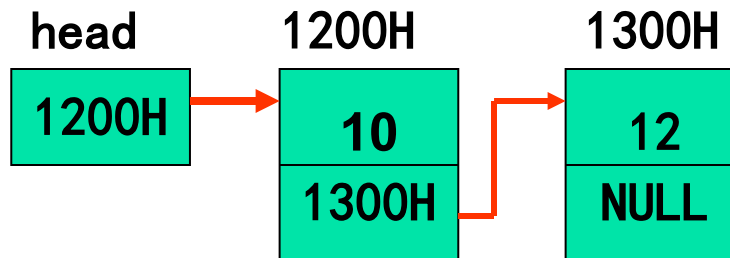
```
head->data=10;
```

```
head->next=(struct intNode *)malloc(sizeof(struct intNode));
```

```
head->next->data=12;
```

```
head->next->next=NULL;
```

```
struct intNode {  
    int data;  
    struct intNode *next;  
};
```





单向链表的建立

先进后出链（用链表实现**栈**）：

结点的排列顺序和数的输入顺序相反

先进先出链（用链表实现**队列**）：

结点的排列顺序和数的输入顺序相同



建立先进先出链表

用循环方式建立先进先出链表的步骤：

(1) 声明头指针，尾指针。

```
struct intNode *head=NULL,*tail;
```

(2) 创建第一个结点。包括：

① 给第一个结点动态分配存储并使头指针指向它。

```
head=(struct intNode *)malloc(sizeof(struct intNode));
```

② 给第一个结点的数据域中成员赋值。

```
head->data=x;
```

③ 使尾指针也指向第一个结点。

```
tail=head;
```




建立先进先出链表

(3) 循环建立后续结点

如果没遇到结束标志，进行下列操作：

- ① 给后继结点动态分配存储并使前驱结点的指针指向它。

```
tail->next=(struct intNode *)malloc(sizeof(struct intNode));
```

- ② 使尾指针指向新建立的后继结点

```
tail=tail->next;
```

- ③ 给后继结点的数据域中成员赋值。

```
tail->data=x;
```

(4) 给尾结点（最后一个结点）的指针赋NULL值。

```
tail->next=NULL;
```

```

struct intNode *createList() /* 用循环方式建立先进先出链表 */
{
    struct intNode *head=NULL,*tail=NULL;
    int x;
    scanf("%d",&x); /* 输入第1个整数 */
    if(x) {
        head=(struct intNode *)malloc(sizeof(struct intNode));
        head->data=x; /* 对数据域赋值 */
        tail=head; /* tail指向第一个结点 */
        scanf("%d",&x); /* 输入第2个整数 */
        while(x){ /* tail所指结点的指针域指向动态创建的结点 */
            tail->next=(struct intNode *)malloc(sizeof(struct intNode));
            tail=tail->next; /* tail指向新创建的结点 */
            tail->data=x; /* 向新创建的结点的数据域赋值 */
            scanf("%d",&x); /* 继续输入整数 */
        }
        tail->next=NULL; /* 对最后一个结点的指针域赋NULL值 */
    }
    return(head);
}

```



调用createList函数建立链表

```
struct intNode *createList(void);
```

```
int main(void)
```

```
{
```

```
    struct intNode* head=NULL;
```

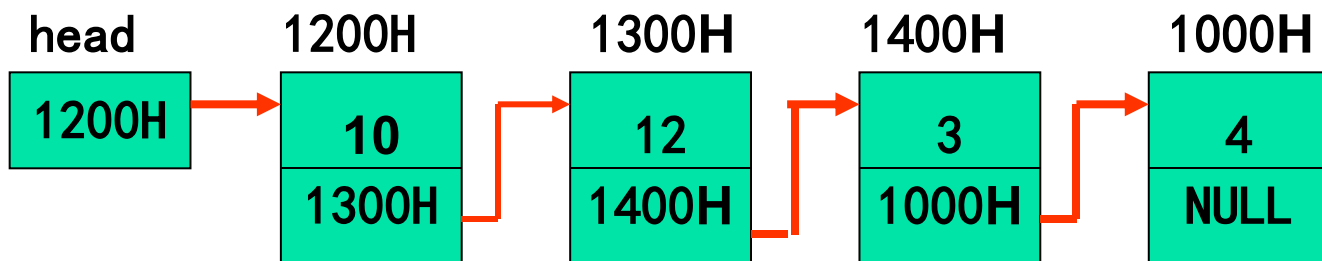
```
    head=createList();    /* 创建链表 */
```

```
    printList(head);    /* 输出链表结点中的数据 */
```

```
    return 0;
```

```
}
```

遍历链表



```
void printList( struct intNode *head)
```

```
{
```

```
    struct intNode *p=head; /*遍历指针p指向链头 */
```

```
    while(p!=NULL){ /* p非空 */
```

```
        printf("%d\t",p->data); //输出结点数据域中成员的值
```

```
        p=p->next; /*遍历指针p指向下一结点 */
```

```
    }
```

```
}
```

递归建立先进先出链表

```
struct intNode *createList()
```

```
{  
    struct intNode *head=NULL;
```

```
    int x;
```

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

```
    if(x==0) /* 遇到结束标记，返回NULL */
```

```
        return NULL;
```

```
    else {
```

```
        head=(struct intNode *)malloc(sizeof(struct intNode));
```

```
        /* head指向新创建的结点 */
```

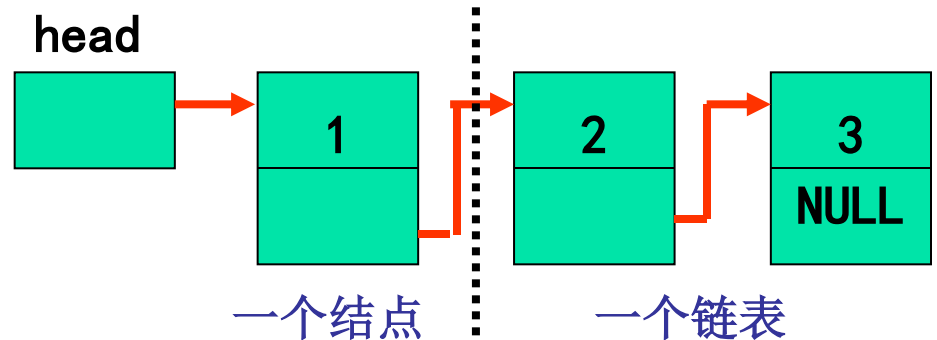
```
        head->data=x;          /* 对新创建结点的数据域赋值 */
```

```
        head->next=createList(); /* 递归创建下一结点 */
```

```
        return head;          /* 返回链头地址 */
```

```
    }
```

```
}
```



统计链表中结点的数目

/* 循环遍历法统计链表head中结点的数目 */

int countNodes(struct intNode *head)

{

struct intNode *p=head;

int num=0;

while (p!=NULL){

num++;

p=p->next;

}

return num;

}

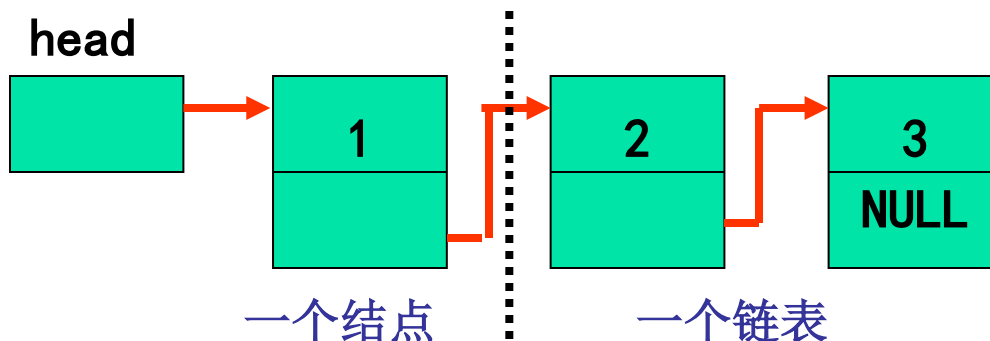
P!=0

P

统计链表中结点的数目

/* 用递归法统计链表head中结点的数目 */
int **countNodes_recursive**(struct intNode *head)

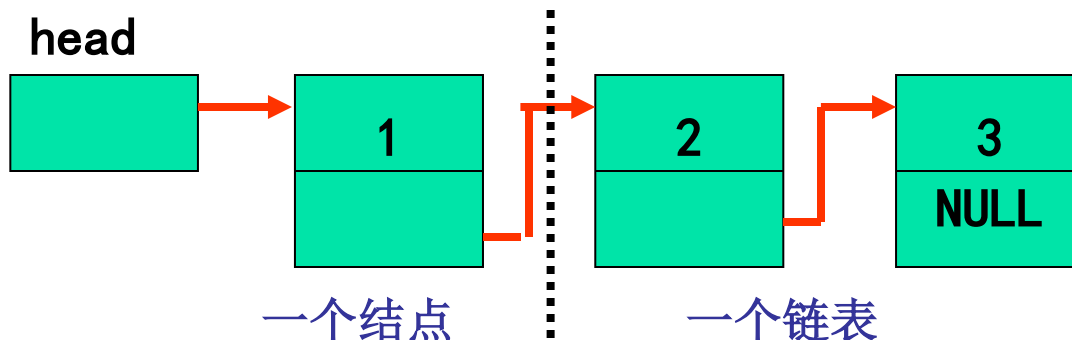
```
{  
    struct intNode *p=head;  
    if(p)  
        return (1+countNodes_recursive(p->next));  
    else  
        return 0;  
}
```



查找结点

/* 用递归法查找**链表**中数据成员值与形参**n**相同的结点。找到，返回该结点的地址，否则，返回NULL。 */

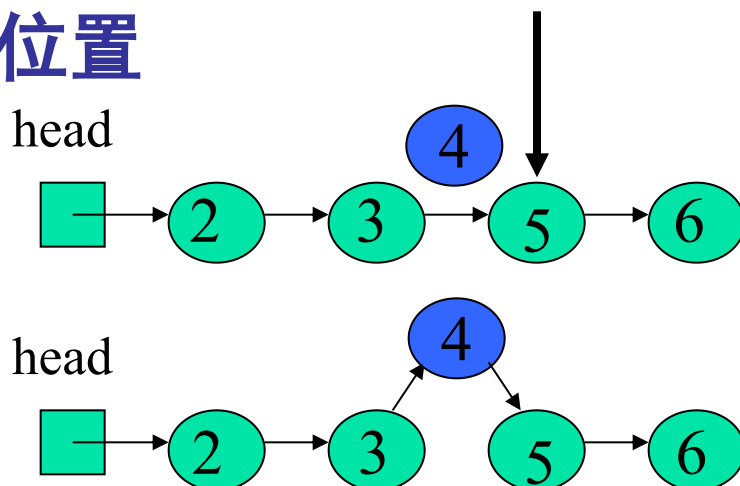
```
struct intNode *findNodes(struct intNode *head, int n)
{
    struct intNode *p=head;
    if(p) { /* 链表非空，查找 */
        if(p->data==n) return p; /* 找到，返回该结点的地址 */
        else
            return (findNodes(p->next,n)); /* 递归查找 */
    }
    else return NULL;
}
```



学习改为循环结构

插入结点

(1) 寻找插入位置



插入点的位置可能是链头、链尾、或者链中。

插入方式有将结点作为插入点的新后继结点和插入点的新前驱结点两种。

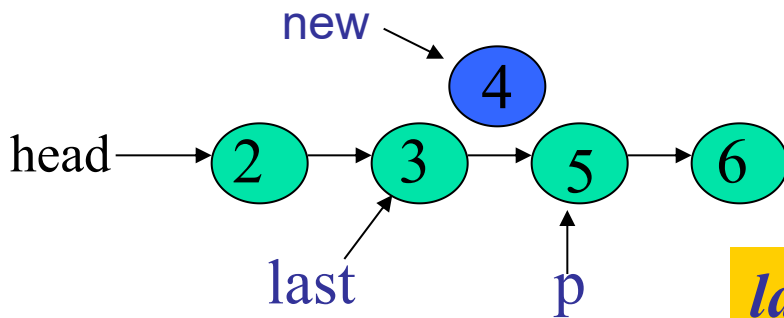
下面介绍将结点作为插入点的新前驱结点。

插入结点

(2) 将新结点插入链表

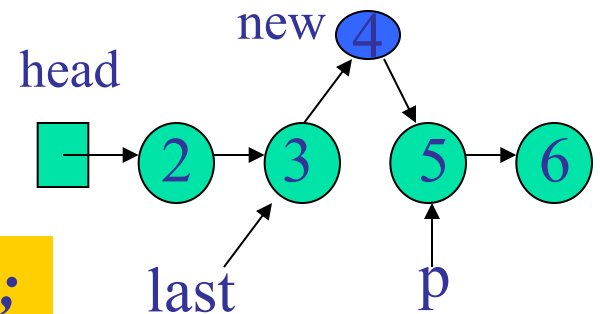
插入点是链中（ p 和 $last$ 之间）

需要两个遍历指针，一个是指向插入点的遍历指针 p ，另一个是指向插入点前驱结点的指针 $last$ 。设新结点由 new 指针所指，它将插入在 p 和 $last$ 所指结点之间。



插入前结点间的指向关系

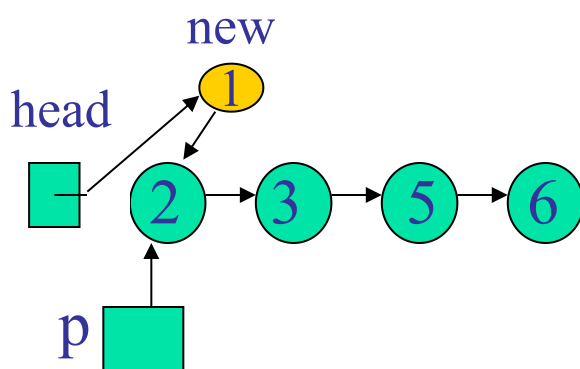
$last \rightarrow next = new;$
 $new \rightarrow next = p;$



插入后结点间的指向关系

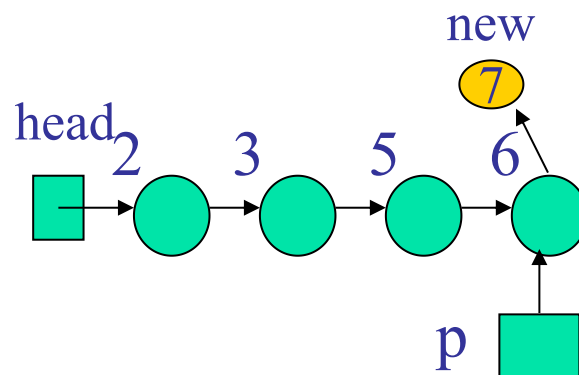
插入结点

插入点是链头



```
head=new;  
new->next=p;
```

插入点是链尾



```
p->next=new;  
new->next=NULL;
```

插入结点

/* 在一个升序链表中插入值为x的结点，返回新结点的地址。 */

```
#define LEN sizeof(struct intNode)
```

```
struct intNode *insertNode ( struct intNode **hp, int x )
```

```
{
```

```
    struct intNode *p, *last, *newnode;
```

```
    newnode = (struct intNode *)malloc(LEN); /* 创建一个新节点 */
```

```
    newnode->data=x;
```

```
    p=*hp;
```

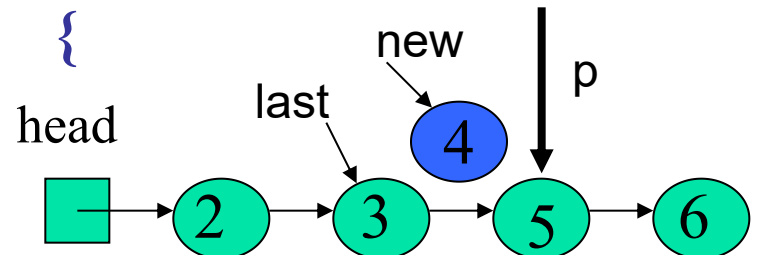
```
    /* 寻找插入位置 */
```

```
    while( p!=NULL && x > p->data ) {
```

```
        last=p;
```

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

```
    }
```



插入结点

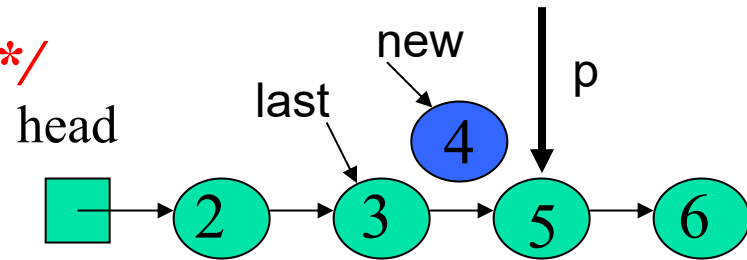
```
if ( p == *hp ) { /* 插入点是链头 */  
    *hp=newnode; /* 新节点为链头*/  
    newnode->next=p;  
}
```

```
}  
else if (p==NULL) { /* 插入点是链尾 */  
    last->next=newnode;  
    newnode->next=NULL; /*新节点为链尾*/  
}
```

```
else { /* 插入点是链中 */  
    newnode->next=p; /*新节点为p的前驱结点*/  
    last->next=newnode;  
}
```

```
return newnode;
```

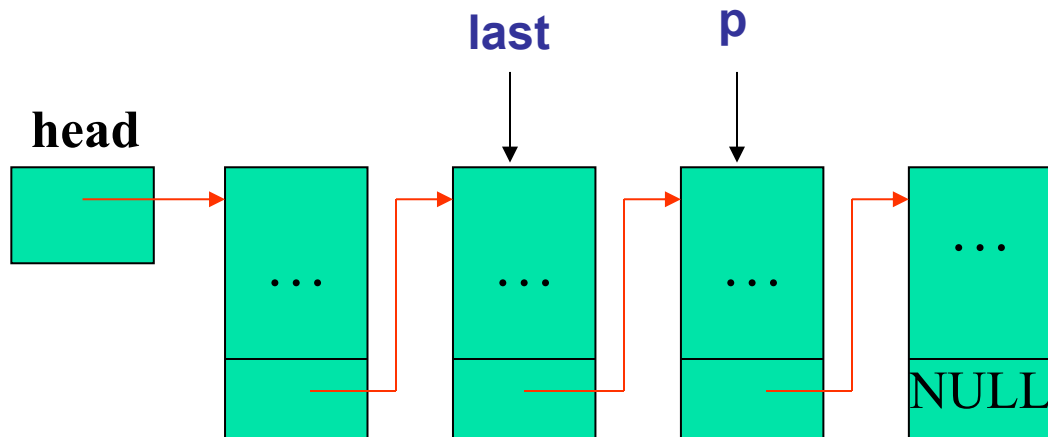
```
}
```



删除结点

(1) 查找被删结点

需设一个指向被删结点的遍历指针 p , 和
指向被删结点的前驱结点的遍历指针 $last$



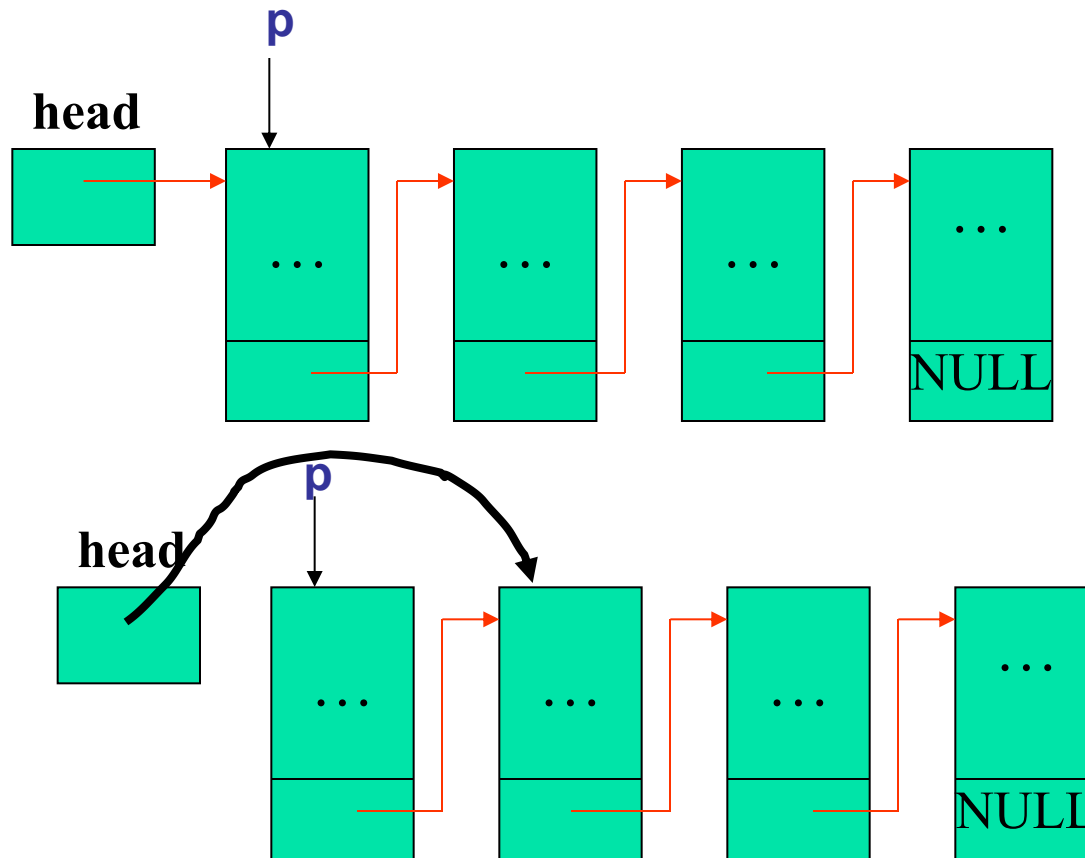
(2) 删除 p 所指结点，即改变连接关系

(3) 释放 p 所指的存储区

删除结点

(2) 删除p所指结点，即改变连接关系

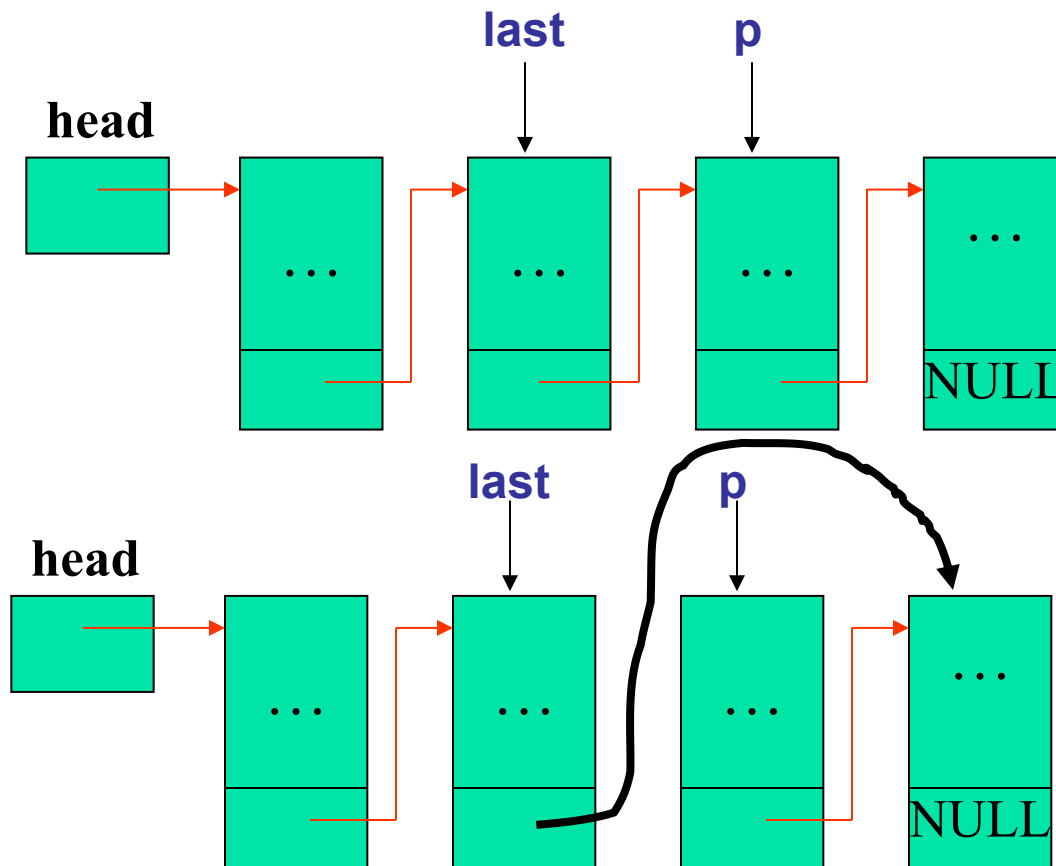
链头： 修改 *head*: $head = p \rightarrow next$



删除结点

非链头: 修改前驱结点的指针域:

$last \rightarrow next = p \rightarrow next;$





删除结点

(3) 释放 *p* 所指的存储区

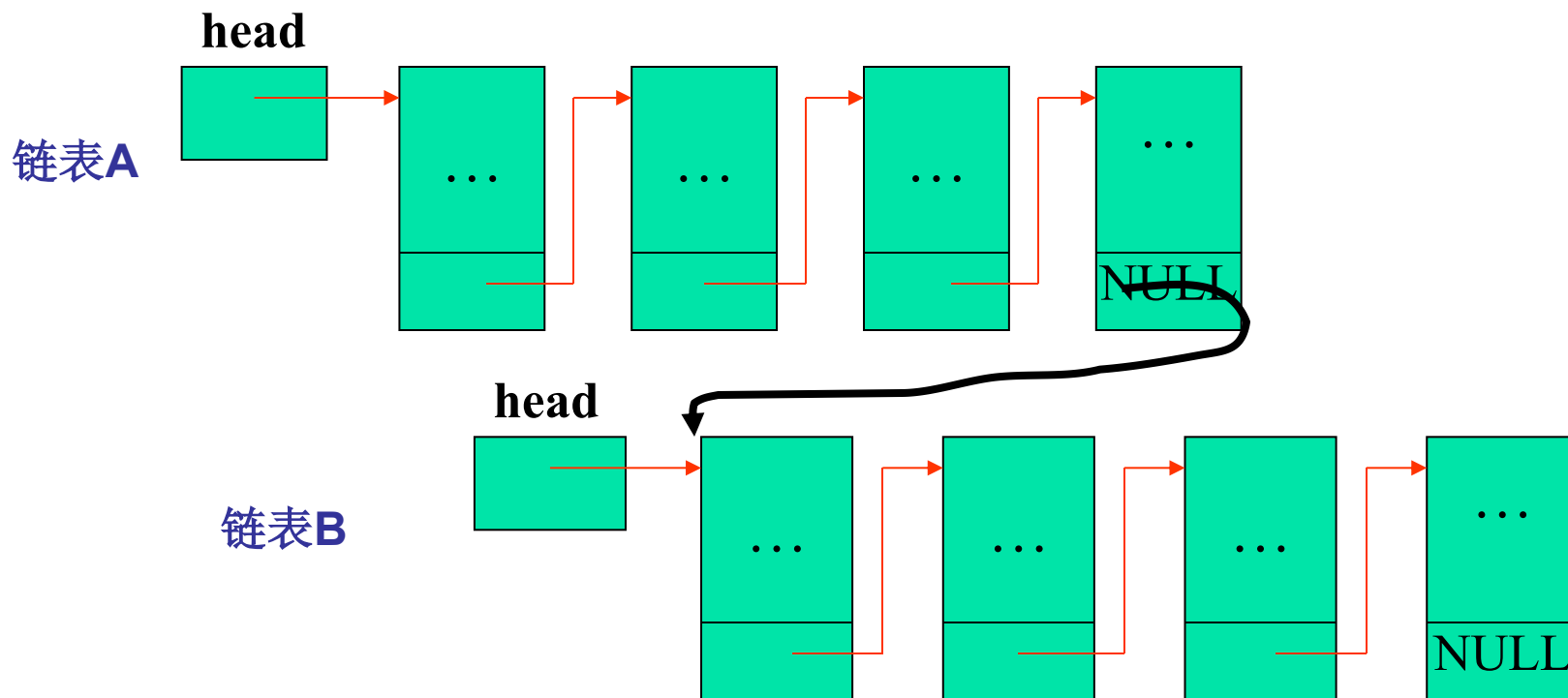
`free(p);`

删除结点

```
/* 删除值为n的结点，成功删除返回1；否则，返回0。 */
int deleteNodes(struct intNode **hp,int n)
{
    struct intNode *p,*last;
    p=*hp;
    while(p !=NULL&& p ->data!=n){ /* 查找成员值与n相等的结点 */
        last = p;          /* last指向当前结点 */
        p = p ->next;      /* p指向下一结点 */
    }
    if(p ==NULL) return 0 ; /* 没有符合条件的结点 */
    if(p ==*hp) /* 被删结点是链头 */
        *hp= p ->next;
    else /* 被删结点不是链头 */
        last->next= p ->next;
    free(p); /* 释放被删结点的存储 */
    return 1;
}
```

归并链表

对于非空链表A、B,将链表B归并到链表A, 指将链表A的链尾指向链表B的链头所形成的一个新的链表。





归并链表

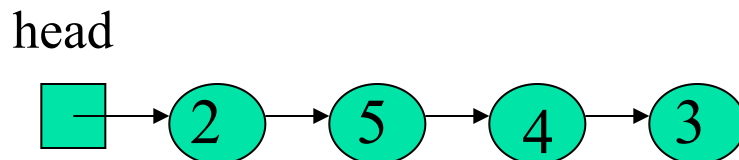
(1) 遍历链表A找到其链尾

(2) 将链表B的头指针值赋给链表A链尾的指针域

```
void conLists ( struct intNode *a, struct intNode *b )  
{  
    struct intNode *p=a;  
    while(p->next != NULL ) {  
        p=p->next;  
    }  
    p->next=b;  
}
```

链表排序

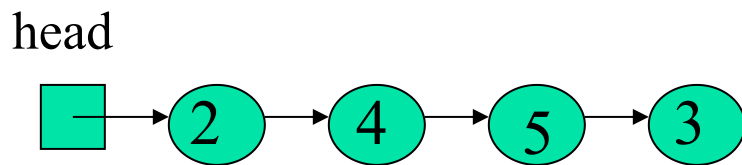
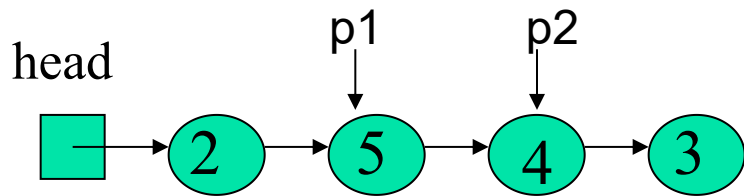
- 链表排序是指将链表的结点按某个数据域从小到大（升序）或从大到小（降序）的顺序连接。



链表排序法1

排序中对链表结点的交换有**两种方法**,

(1) 交换结点的数据域, 不改变结点间的指向关系。



```
if(p1->data > p2->data)
```

```
{
```

```
    int t;
```

```
    t=p1->data; /* 交换数据域 */
```

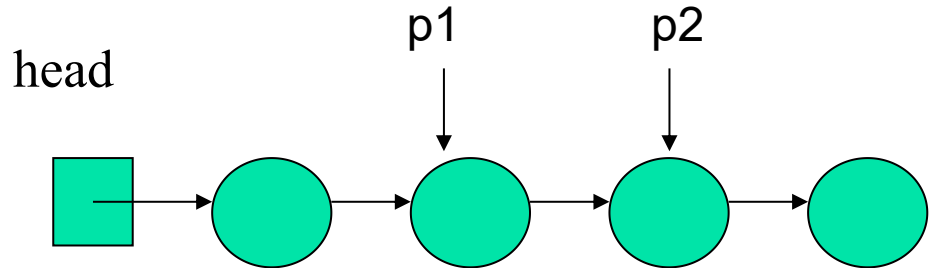
```
    p1->data=p2->data;
```

```
    p2->data=t;
```

```
}
```

链表排序法1

```
struct std {  
    char num[12];  
    char name[9];  
    int score;  
    // struct std *next;  
};  
  
struct list {  
    struct std data;  
    struct list *next;  
};  
  
struct list *head;
```



```
if(p1->data.score > p2->data.score)  
{  
    struct std t;  
    t=p1->data; /* 交换数据域 */  
    p1->data = p2->data;  
    p2->data = t;  
}
```

链表排序函数

```
void sortList(struct intNode *head)
```

```
{
```

```
    struct intNode *p1,*p2;
```

```
    int t;
```

```
    for(p1=head;p1!=NULL;p1=p1->next)
```

```
        for(p2=p1->next;p2!=NULL;p2=p2->next)
```

```
            if(p1->data > p2->data){
```

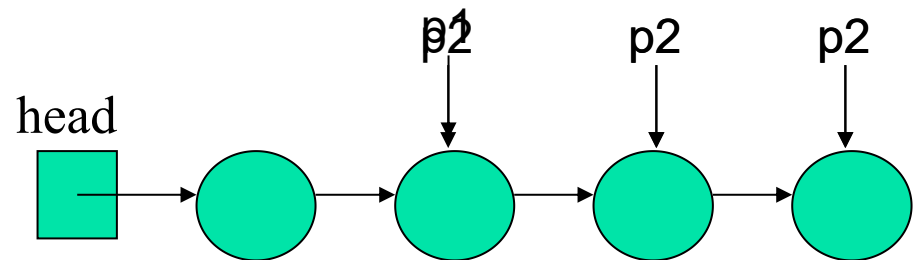
```
                t=p1->data;
```

```
                p1->data=p2->data;
```

```
                p2->data=t;
```

```
            }
```

```
}
```



/* 交换数据域 */



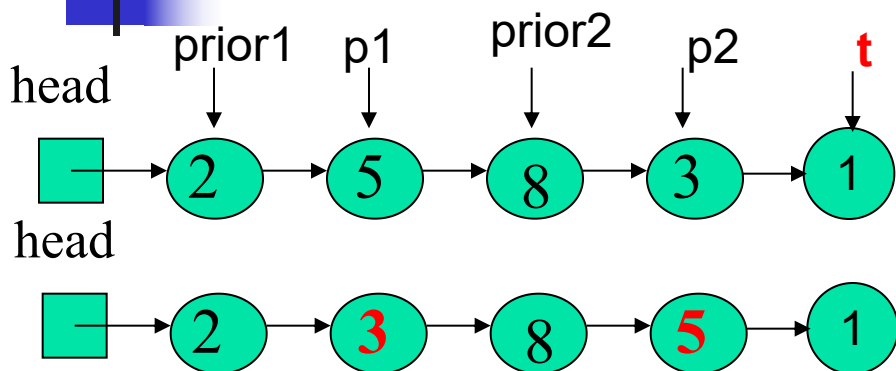
链表排序法2（不做要求）

(2) 改变结点的连接，操作较为复杂。

在数据域较为简单的情况下，多采用交换结点的数据域的方法进行链表排序。

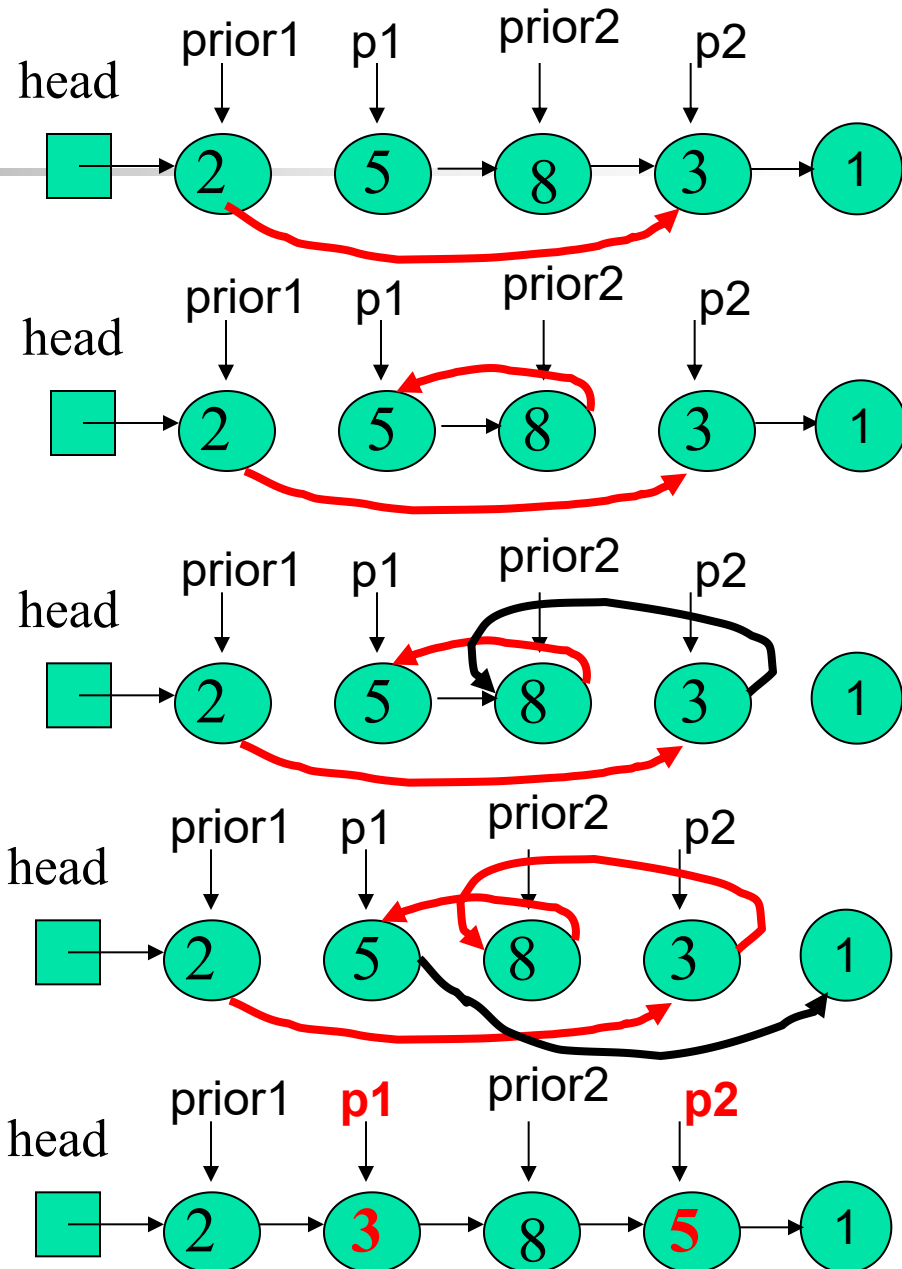
在数据域较为复杂，成员较多，采用改变链表结点之间的连接，实现结点的交换方法则效率较高。

交换p1和p2

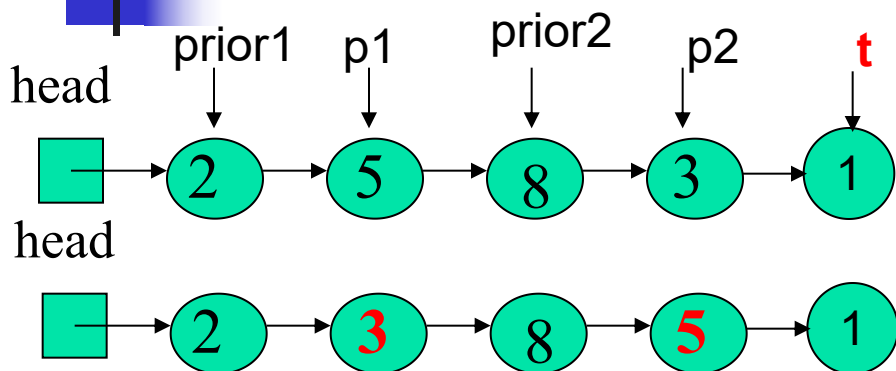


```

if(p1->data > p2->data){
    t=p2->next;
    prior1->next=p2;
    prior2->next=p1;
    p2->next=p1->next;
    p1->next=t;
    p2=p1;
    p1=prior1->next;
}
    
```



交换p1和p2



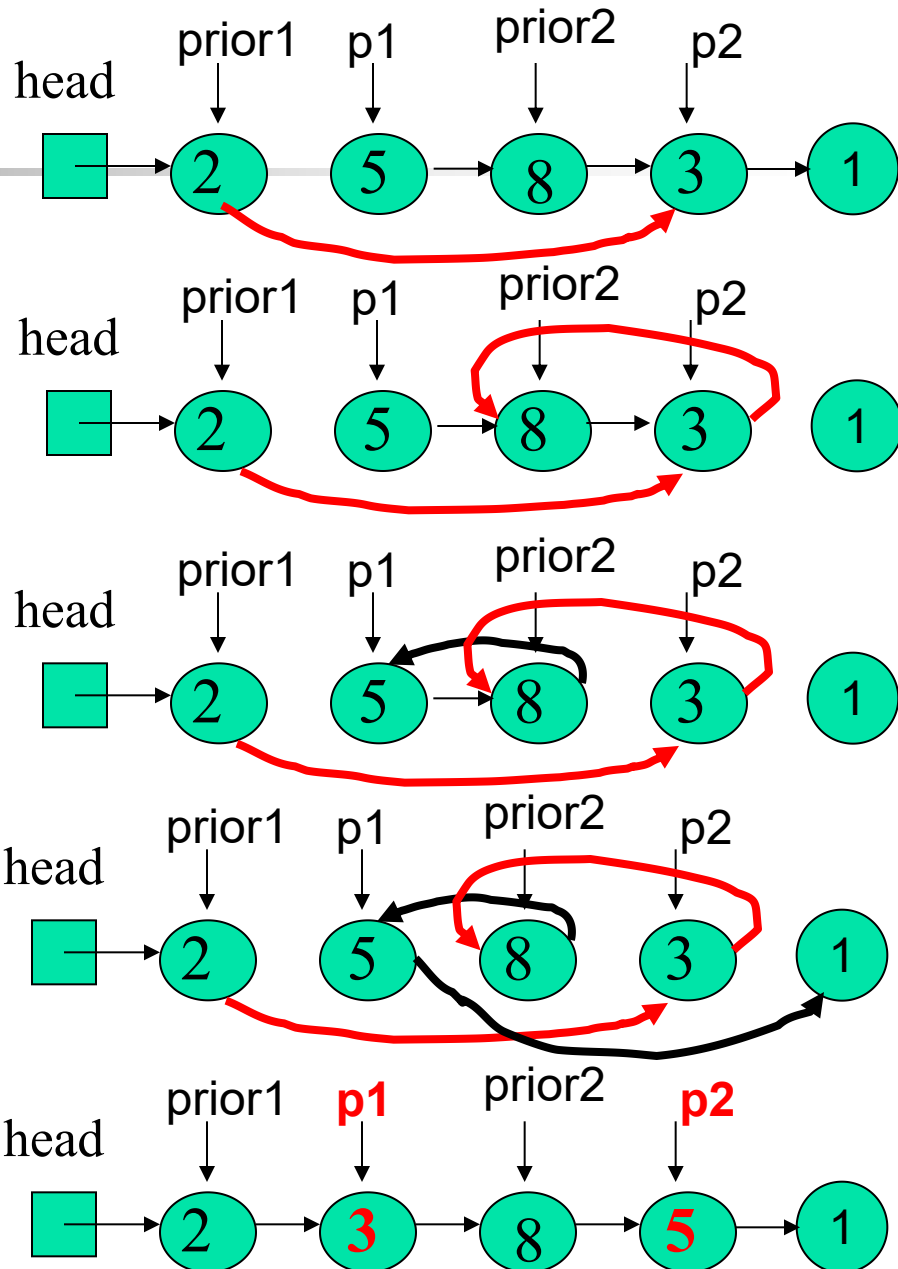
```

if(p1->data > p2->data){
    t=p2->next;

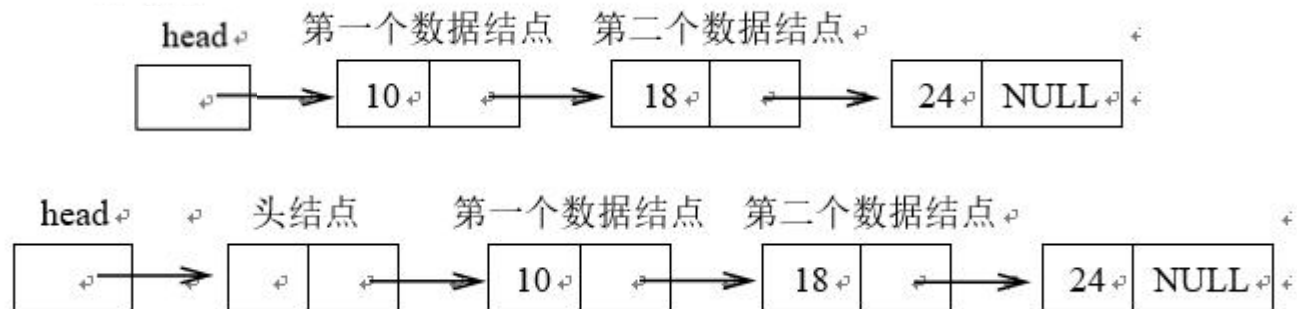
    prior1->next=p2;
    p2->next=p1->next;

    prior2->next=p1;
    p1->next=t;

    p2=p1;
    p1=prior1->next;
}
    
```



新增头结点



// 新增头指针结点的方法

```
void sortList (struct intNode **hp)
```

```
{
```

```
    struct intNode *prior1,*prior2,*p1,*p2,*t;
```

```
    int i=0;
```

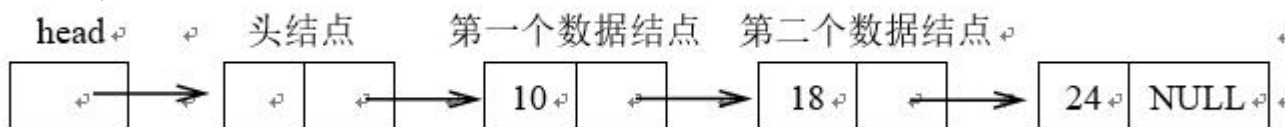
```
    p1=(struct intNode *)malloc(sizeof(struct intNode)); // 新增结点
```

```
    p1->next=*hp;      // 使新结点成为头结点
```

```
    (*hp)=prior1=p1;
```

新增头结点

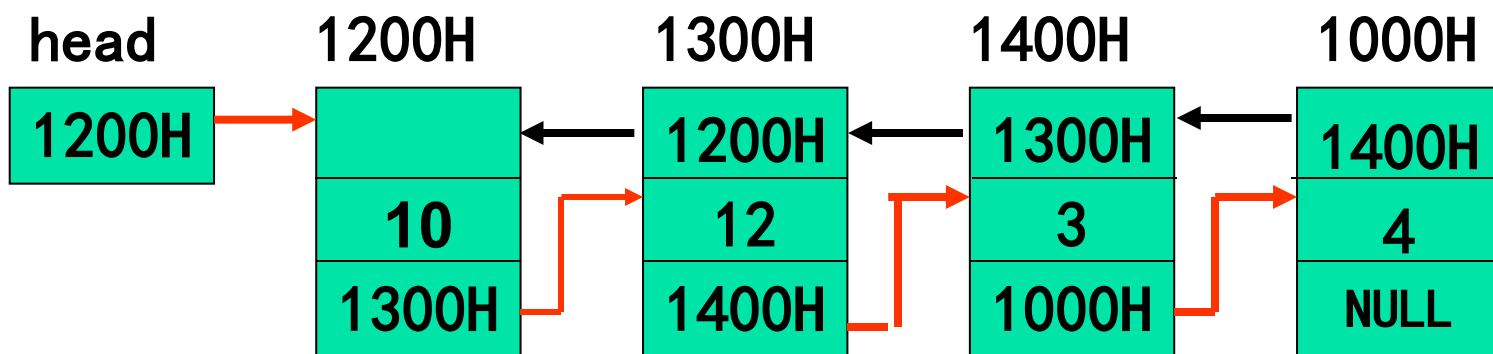
```
for(p1=prior1->next;p1->next!=NULL;prior1=p1,p1=p1->next)
    for(p2=p1->next,prior2=p1;p2!=NULL;prior2=p2,p2=p2->next)
        if(p1->data>p2->data){
            t=p2->next;
            prior1->next=p2;
            prior2->next=p1;
            p2->next=p1->next;
            p1->next=t;
            p2=p1;
            p1=prior1->next;
        }
```



```
p1>(*hp);          /* p1指向新增头指针结点 */
(*hp)=(*hp)->next; /* (*hp)指向排序后链表的链头 */
free(p1);          /* 释放新增头指针结点 */
}
```

双向链表

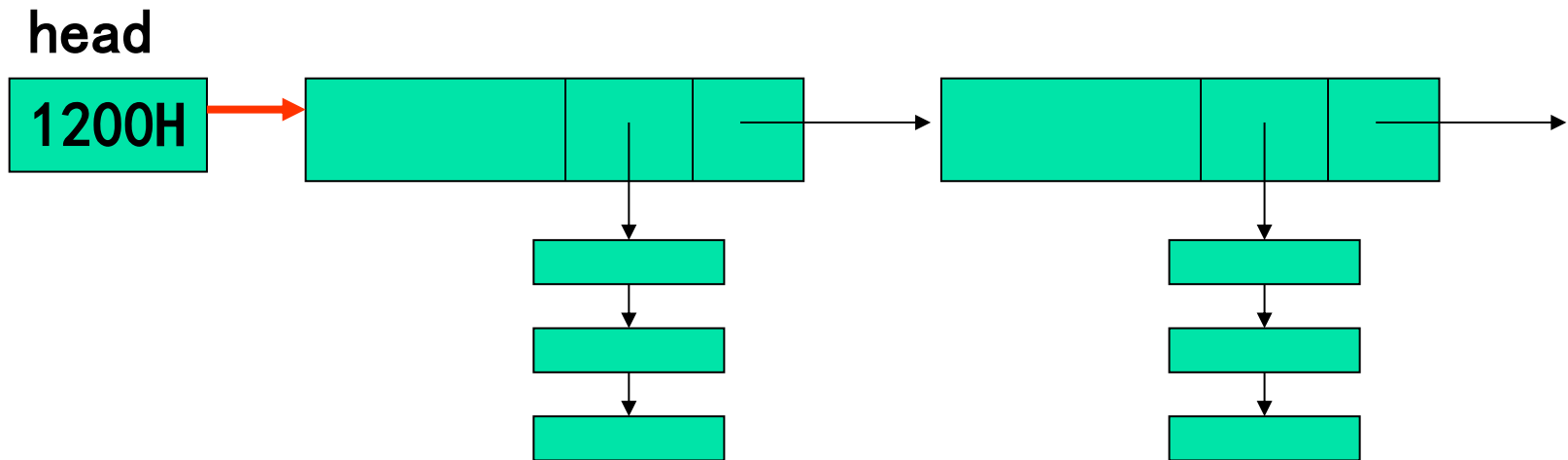
- 如果结点的指针域包含两个指针，且一个指向前一个结点，另一个指向后一个结点，这种链表称为**双向链表**。



双向链表结构

十字交叉链表

- 如果结点的指针域包含两个指针，且一个指向后一个结点，另一个指向另外一个链表，这种链表称为**十字交叉链表**。



十字交叉链表结构



用十字交叉链表保存学生基本信息和成绩

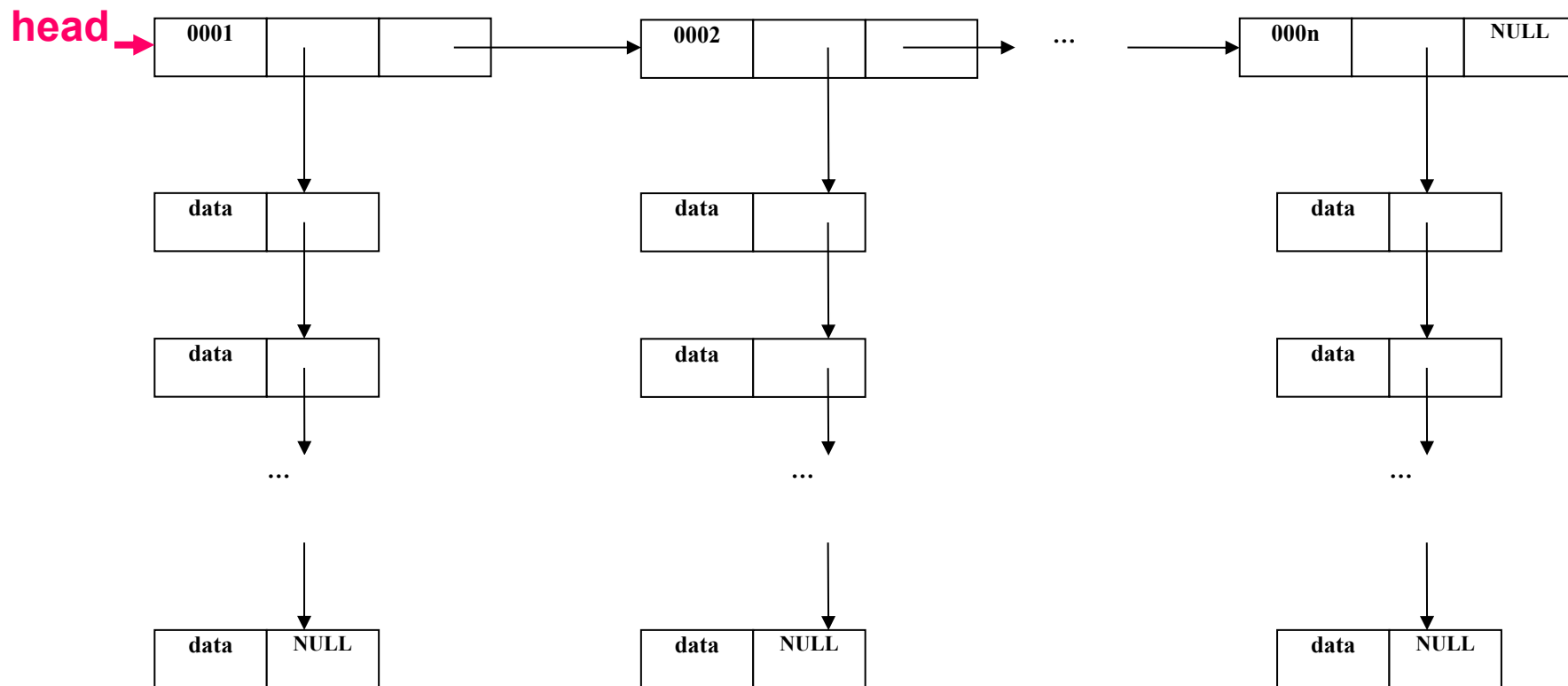
学生基本信息表

| 学号 | 姓名 | 性别 | 年龄 | 家庭住址 | 联系电话 | 备注 |
|------|-----|----|----|-----------------|----------|------------|
| 0001 | aaa | m | 18 | hubei, wuhan | 12345678 | Abc def |
| 0001 | bbb | f | 18 | hunan, changsha | 76545678 | Aaaaaa bbb |
| ... | | | | | | |
| 00xx | zzz | m | 19 | hubei, honghu | 32145678 | Nnn kkk |

学生学习成绩表

| 学号 | 高等数学 | 普通物理 | 电工基础 | 演讲与口才 | 欧洲文化 | 论语初探 |
|------|------|------|------|-------|------|------|
| 0001 | 86 | 85 | 73 | × | 80 | × |
| 0002 | 77 | 83 | 76 | 82 | 87 | 75 |
| 0003 | 87 | 82 | 81 | × | × | 86 |
| ... | | | | | | |
| 00xx | 89 | 87 | 85 | × | × | × |

十字交叉链表



水平方向：学生基本信息链；

垂直方向：各学生课程成绩链



学生课程成绩结点的结构类型定义

该类型的结构变量可以构成纵向单向链表的结点。

```
typedef struct score_tab{  
    char    num[5];        /* 学号 */  
    char    course[20];    /* 课程名称 */  
    int     score;         /* 成绩 */  
    struct score_tab *next; /* 指向下一个课程成绩结点 */  
} courses;
```



学生基本信息结点的结构类型定义

该类型的结构变量可以构成十字交叉链表中横向学生基本信息链的结点。

```
typedef struct student_tab{  
    char    num[5];           /* 学号 */  
    char    name[10];         /* 姓名 */  
    char    sex;              /* 性别 */  
    int     age;              /* 年龄 */  
    char    addr[30];         /* 家庭住址 */  
    char    phone[12];        /* 联系电话 */  
    char*    memo;            /* 备注字段 */  
    courses *head_score;    /* 指向成绩链的头指针 */  
    struct student_tab *next; /* 指向下一个结点 */  
} studs;
```