《计算概论A》课程 程序设计部分

结构体、链表、枚举、共同体

李　戈

北京大学 信息科学技术学院 软件研究所

2010年12月24日

北京大学

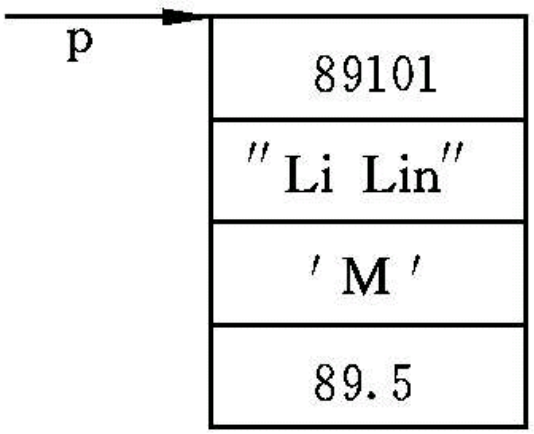
结构体（续）

指向结构体类型数据的指针

- 因为
  - 结构体类型与其他数据类型相同;
  - 一个结构体变量在内存中占用一段连续的区域，有一个起始地址;
- 所以
  - 可以设计一个指针变量，用于存放结构体变量的起始地址;
  - 即，指向结构体类型数据的指针;

指向结构体类型数据的指针

```
main()
{   struct  student
      {    long    num;
           char    name[20];
           char sex;
           float score;
    } stu_1;
       struct    student    * p;
       p = &stu_1;
       stu_1.num=89101;
       strcpy(stu_1.name，  "Li Lin");
       stu_1.sex='M';
       stu_1.score=89.5;
     cout<<stu-1.num<<stu-1.name<<stu_1.sex<<stu-1.score);
     cout<<(*p).num<<(*p).name<<(*p).sex<<(*p).score);
}
```



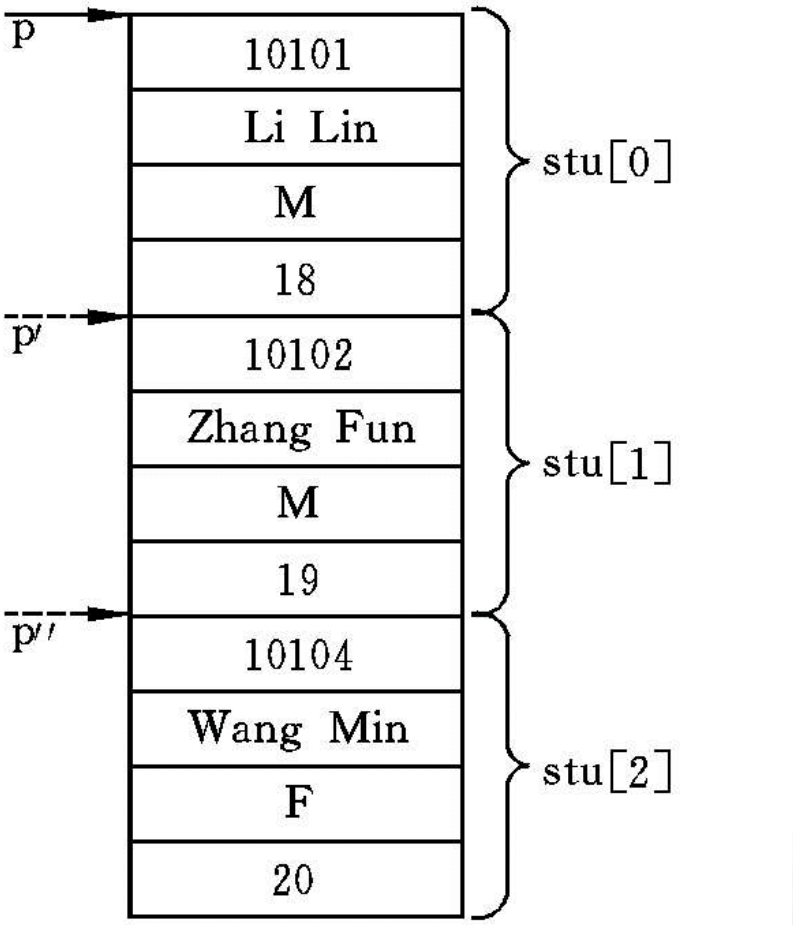
结构与指针

```
void main()
{   student stu_1,*p;
    int *p1;
    char *p2, *p3;
    float *p4;
    p = &stu_1;
    p1 = &stu_1.stuNo;
    p2 = stu_1.name;
    p3 = &stu_1.sex;
    p4 = &stu_1.score;
    stu_1.stuNo = 89101;
    strcpy(stu_1.name, "Li Lin");
    stu_1.sex = 'M';
    stu_1.score = 90;
    cout<<stu_1.stuNo<<" "<<stu_1.name<<" "<<stu_1.sex<<"
    "<<stu_1.score<<endl;
    cout<<(*p).stuNo<<" "<<(*p).name<<" "<<(*p).sex<<"
    "<<(*p).score<<endl;
}
```

```
struct student
{
    int stuNo;
    char name[20];
    char sex;
    float score;
};
```

指向结构体数组的指针

- 解释
  - p用来指向一个struct student型的数据。
  - p加1意味着p所增加的值为结构体数组stu的一个元素所占的字节数;
  - 本例为:
    2+20+1+2=25 字节



指向结构体的指针做参数

例：在主函数中输入结构体各成员的值，在子函数中输出；

（要求：在主函数中，实参是地址，在子函数中用指针接收.）

```
struct student
{   int No;
    char name[20];
    float score[3];
};
void print(student *p){
    cout<<p->No<<p->name<<p->score[0]
        <<p->score[1]<<p->score[2];}
int main(){
    struct student stu;
    cin >>stu.No>>stu.name>>stu.score[0]
        >>stu.score[1]>>stu.score[2];
    print(&stu);
    return 0;
}
```

结构体数组做参数

```
void main()
{ student allone[4]=
    {
        {1001,“jone”,60,60,80},
        {1002,“david”,70,70,90},
        {1003,“marit”,80,80,60},
        {1004,“yoke”,90,90,70}
    };
    print(allone, 4);
}
```

```
void print(student *p, int n)
{
    for (int i=0; i<n; i++, p++)
        cout<< p->No
            << p-> name
            << p-> score[0]
            << p-> score[1]
            << p-> score[2]
            << endl;
}
```

北京大学

指向结构体数组的指针

```
struct test *p,  stu[4];
p = stu;
cout << p++->num << endl;
cout << ++p->num << endl;
cout << p->num++ << endl;
cout << p->num << endl;
cout << (++p)->num++<<endl;
cout << p->num << endl;
```

num	name
10	‘A’
20	‘B’
30	‘C’
40	‘D’

- 注意：->运算的优先级非常高，仅次于（ ）[ ]

指向结构体数组的指针

```
struct test *p,  stu[4];
p = stu;
cout << p++->num << endl;
cout << ++p->num << endl;
cout << p->num++ << endl;
cout << p->num << endl;
cout << (++p)->num++<<endl;
cout << p->num << endl;
```

```
10
21
21
22
30
31
Press
```

num	name
10	‘A’
20	‘B’
30	‘C’
40	‘D’

- 注意：->运算的优先级非常高，仅次于（ ）[ ]

结构体直接做函数参数

```
struct stru
{   int x;
    char c;
};
void func(stru b)
{   b.x=20;     b.c='y';}
int main()
{   stru a = {10,'x'};
    func(a);
    cout<<a.x<<" "<<a.c;
    return 0;
}
```

结构体直接做函数参数

```
struct stru
{   int x;
    char c;
};
void func(stru b)
{   b.x=20;     b.c='y';}
int main()
{   stru a = {10,'x'};
    func(a);
    cout<<a.x<<" "<<a.c;
    return 0;
}
```

- 结构体做参数时采用值传递的方式;
- 系统会构造一个结构体的副本给函数使用;

```
10 x
Press any key
```

结构体做函数返回值

```
student GetStudent()
{  student t;
   cout <<“请输入学号”;
   cin >> t.No;
   cout<<“请输入学生姓名：”;
   cin.getline(t.name,20);
   cout<<“请输入数学、英语、C++成绩：”;
   cin>>t.score[0]>> t.score[1]>> t.score[2];
   return t;
}
int main( )
{  student stu[4];
   for(int i=0;i<4;i++)
   {
     stu[i] = GetStudent();
     print(stu[i]);
   }
}
```

- 一个函数可以返回一个结构体！

结构体应用示例-生日相同问题

- **Description**
  - 在一个有100人的大班级中，存在两个人生日相同的概率非常大，现给出每个学生的学号，出生月日。试找出所有生日相同的学生。
- **Input**
  - 第一行为整数n，表示有n个学生，n<100。
  - 此后每行包含一个字符串和两个整数，分别表示学生的学号（字符串长度小于10）和出生月(1<=m<=12)日(1<=d<=31)。
  - 学号、月、日之间用一个空格分隔。
- **Output**
  - 对每组生日相同的学生，输出一行，
  - 其中前两个数字表示月和日，后面跟着所有在当天出生的学生的学号，数字、学号之间都用一个空格分隔。
  - 对所有的输出，要求按日期从前到后的顺序输出。
  - 对生日相同的学号，按输入的顺序输出。

```
void main ()
{
    int i, j, k, n, flag, count[100]={0};
    cout << "how many students ?";
    cin>>n;
    for(int i=0; i<n; i++)
        cin>>stu[i].ID >> stu[i].month>> stu[i].day;
    for(int m=1; m<=12; m++)
        for(int d=1; d<=31; d++)
        {
                flag=0; j=0;
                for(int i=0; i<n; i++)
                        if (stu[i].month == m && stu[i].day == d)
                                {count[++j] = i; flag++; }
                if(flag>1)
                {       cout<<m<<" "<<d<<" ";
                        for(k=1; k<=j;k ++)
                                cout << stu[count[k]].ID << " "<< endl;
                }
        }
}
```

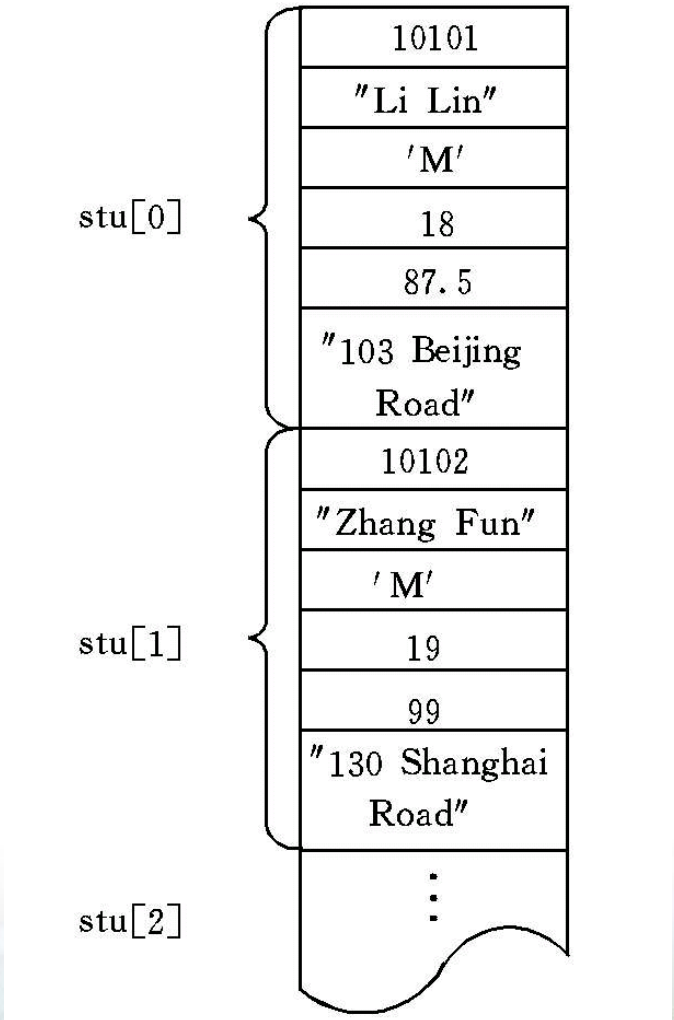
```
struct student
    {
        char ID[10];
        int month;
        int day;
    }stu[100];
```

思考一个问题

- 问题:
  - 已知在教务系统中，需要经常根据学生的学号顺序地读取学生的相关信息;
- 要求:
  - 定义一个结构，按照学生的学号，顺序存放班级学生的信息;
  - 学生信息包括：学号、姓名、性别、年龄、出生日期、地址 等;

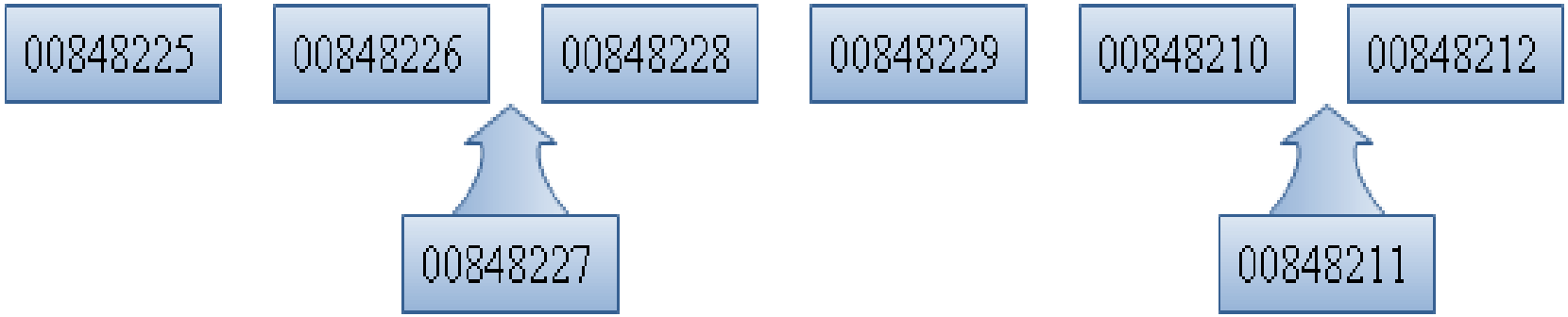
结构体数组

```
struct   date
{ int   month;
   int   day;
   int   year; };
struct   student
{ int   num;
   char   name[20];
   char   sex;
   int   age;
   struct   date birthday;
   char   addr[30];
}students[1000];
```



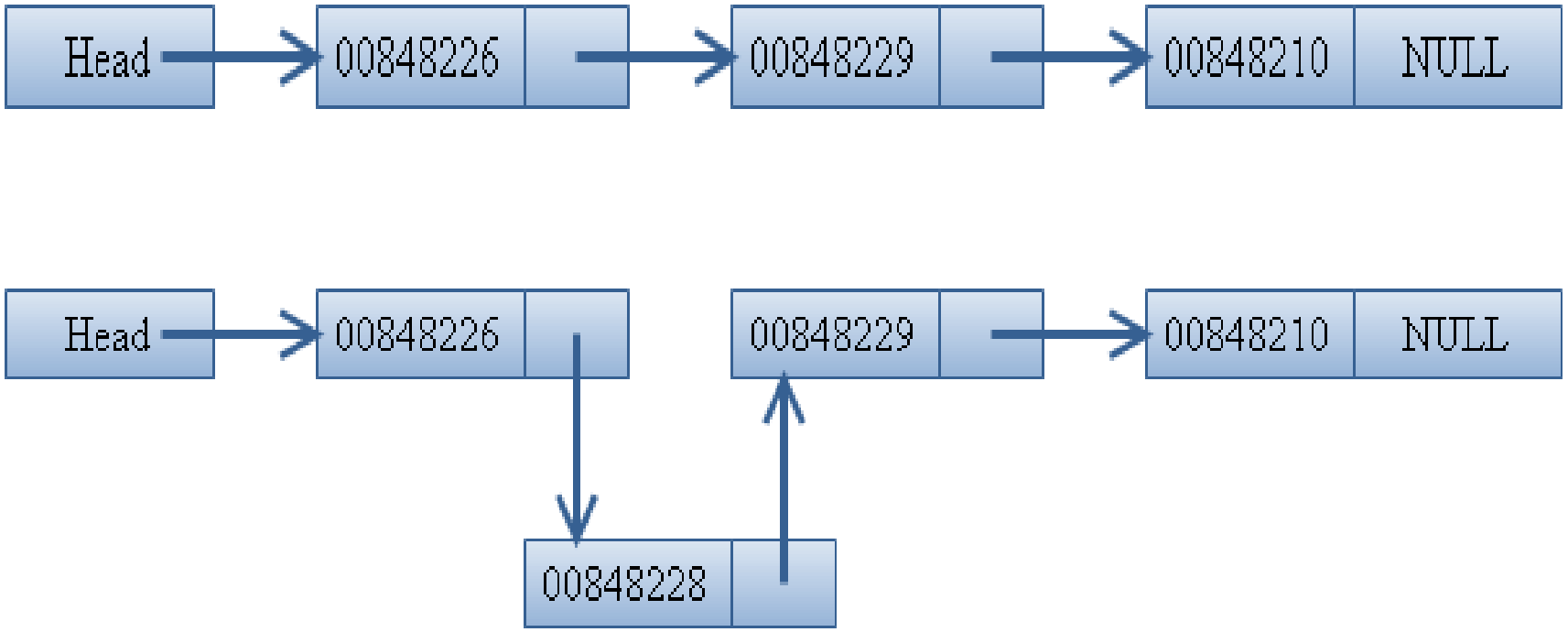
使用数组的缺点

- 插入元素
  - 每次插入元素时，需要将后面元素统统后移！



可以采取的办法

- 用指针把结构体链起来！



链 表

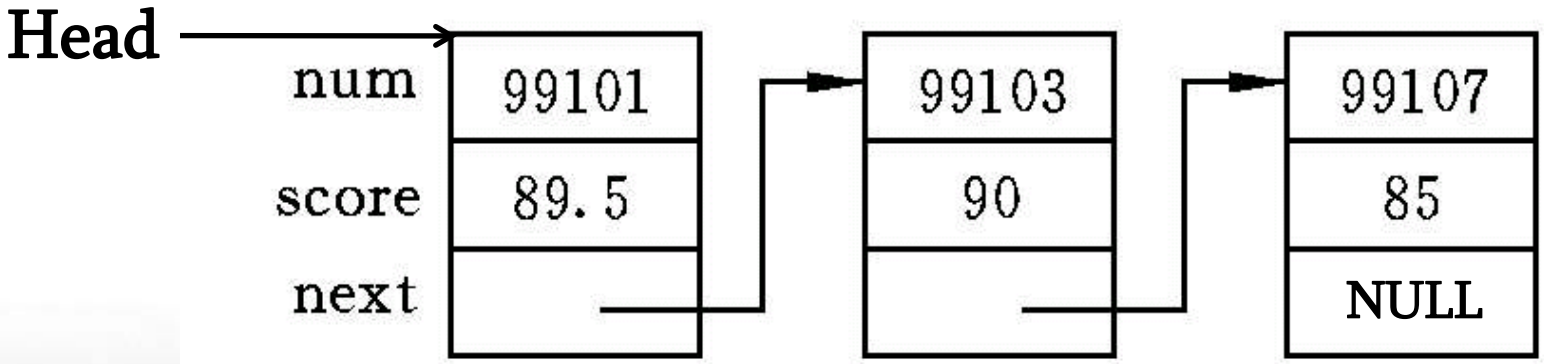
链表

- 链表是一种常用的重要的数据结构
  - 链表头：指向第一个链表结点的指针；
  - 链表结点：链表中的每一个元素，包括：
    - 当前节点的数据
    - 下一个结点的地址
  - 链表尾：不再指向其他结点的结点，其地址部分放一个NULL，表示链表到此结束。

链表的定义

```
struct student
{
    long num;
    float score;
    struct student *next;
};
```

```
student a,b,c,*head,*p;
```



北京大学

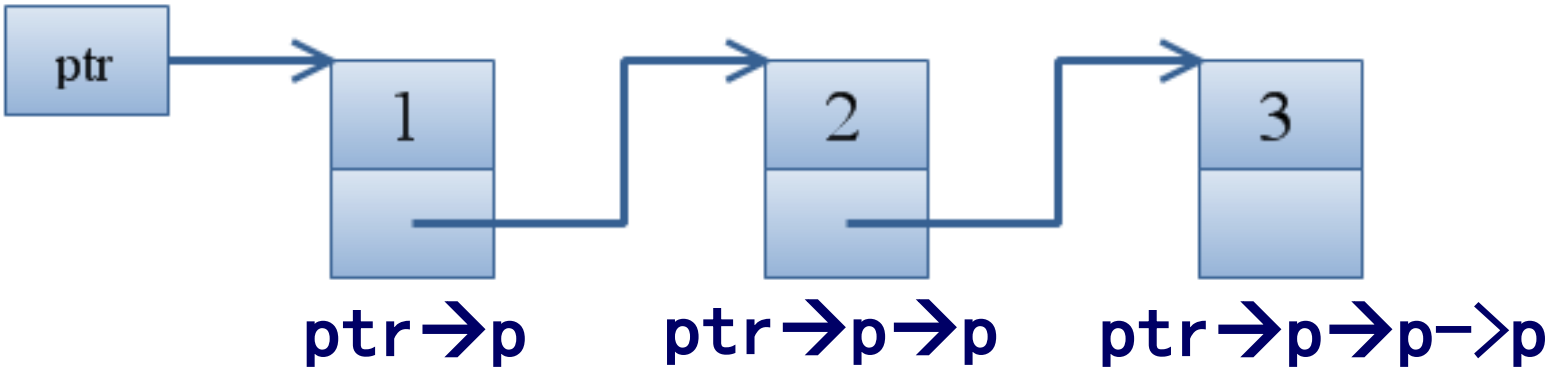
建立一个简单链表并打印出每个节点的信息

```
#include<iostream>
using namespace std;
int main() {
    student a,b,c,*head,*p;
    a.num = 99101;
    a.score = 89.5;
    b.num = 99103;
    b.score = 90;
    c.num = 99107;
    c.score = 85;
    head = &a;          /*将结点a的起始地址赋给头指针head*/
    a.next = &b;        /*将结点b的起始地址赋给a结点的next成员*/
    b.next = &c;        /*将结点c的起始地址赋给b结点的next成员*/
    c.next = NULL;      /*c结点的next成员不存放其他结点地址*/
    p = head;                   /*使p指针指向a结点*/
    do {
            cout<<p->num<<" "<<p->score; /*输出p指向的结点的数据*/
            p=p->next;          /*使p指向下一结点*/
    } while (p!=NULL);          /*输出完c结点后p的值为NULL*/
    return 0;
}
```

访问链表中的元素

```
ptr = new linker ;
ptr->p = new linker ;
ptr->p->p = new linker ;
```

```
struct student{
    long id;
    struct student *next;
};
```



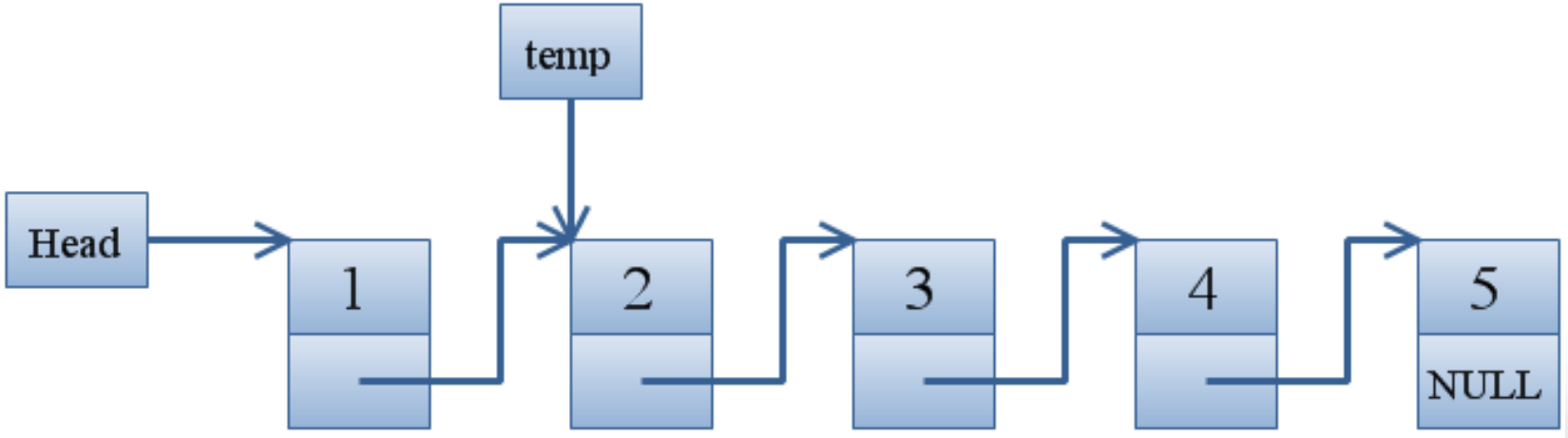
- 要想访问第4个元素，必须表示成:

  head->p->p->p->num

链表元素的遍历

- 利用temp指针遍历链表

```
temp = head;
do {        cout<<temp→num;
            temp = temp→p;
   } while (temp != null);
```



建立一个简单链表
并打印出每个节点的信息

```
#include<iostream>
using namespace std;
int main() {
    student a,b,c,*head,*p;
    a.num = 99101;
    a.score = 89.5;
    b.num = 99103;
    b.score = 90;
    c.num = 99107;
    c.score = 85;
    head = &a;                  /*将结点a的起始地址赋给头指针head*/
    a.next = &b;                /*将结点b的起始地址赋给a结点的next成员 */
    b.next = &c;                /*将结点c的起始地址赋给b结点的next成员 */
    c.next = NULL;  /*c结点的next成员 不存放其他结点地址*/
    p = head;                   /*使p指针指向a结点*/
    do {
        cout<<p->num<<" "<<p->score; /*输出p指向的结点的数据*/
        p=p->next;              /*使p指向下一结点*/
    } while (p!=NULL);          /*输出完c结点后p的值为NULL*/
    return 0;
}
```

链表可以 动态地 创建

- 动态地 建立数组
  - int *p = new int[20];
- 动态地 建立链表节点

```
struct student
{
        int id;
        student *next;
};
```

```
student *head;
head = new student;
```

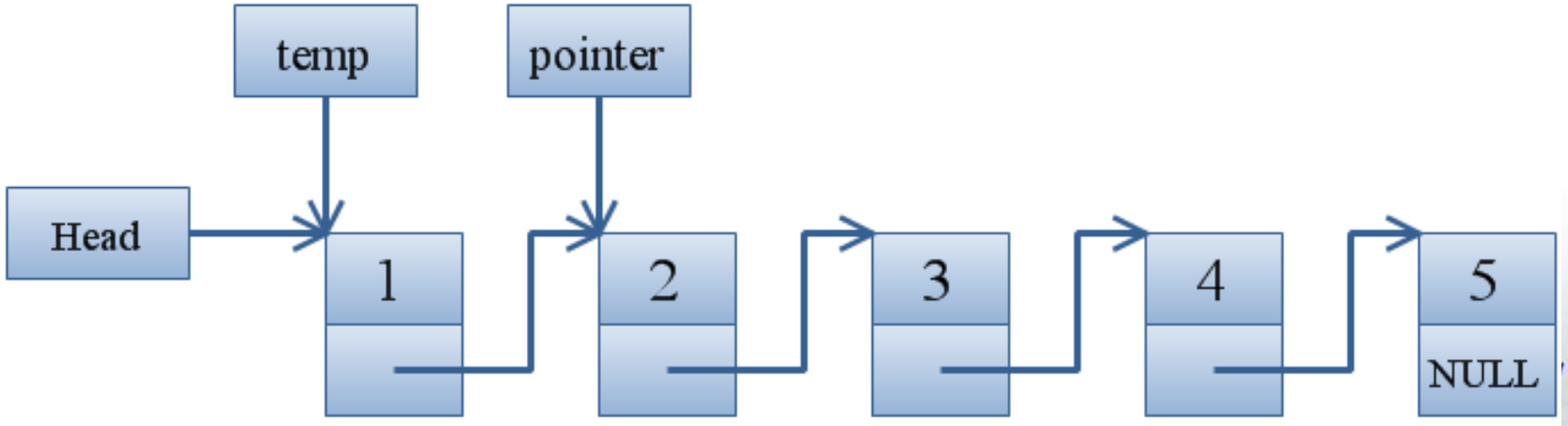
链表结构——逐步建立链表

- 定义一个临时指针: student *temp = head;
- 申请一个新的链表节点:

  pointer = new student

  temp->next = pointer;
- 指针后移一个链表节点:

  temp = temp->next



链表结构——逐步建立链表

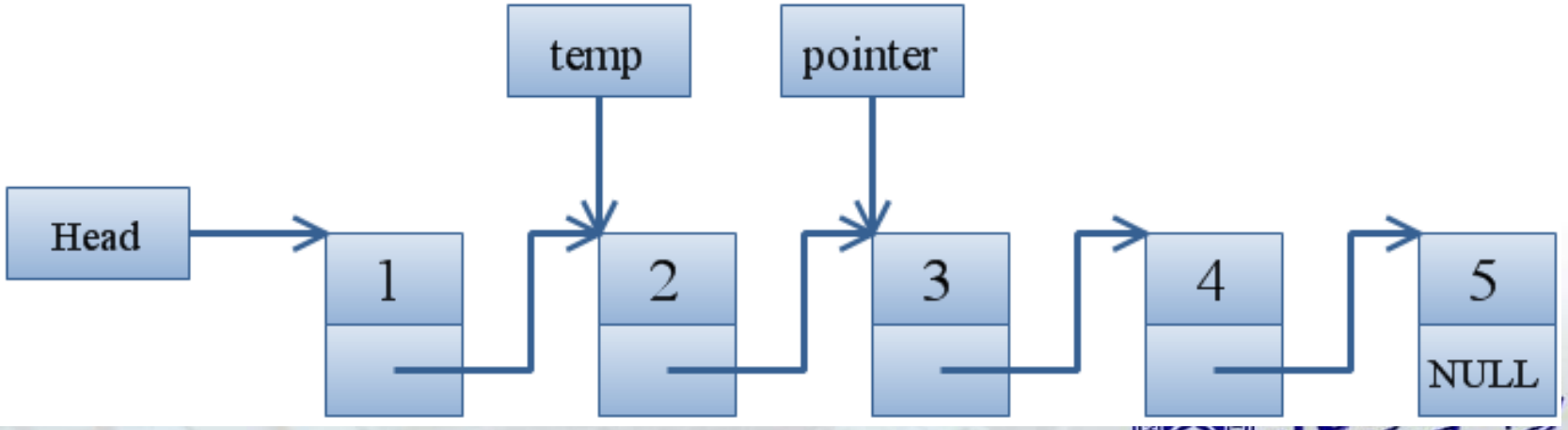
- 再申请一个链表节点：

  pointer = new student

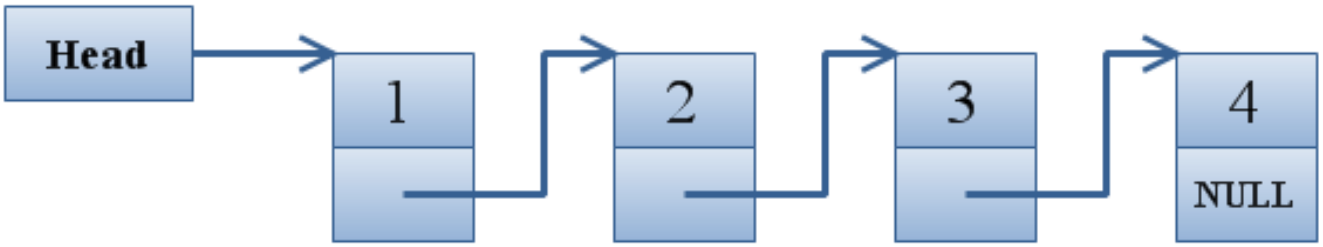
  temp->next = pointer;

- 指针再后移一个链表节点：

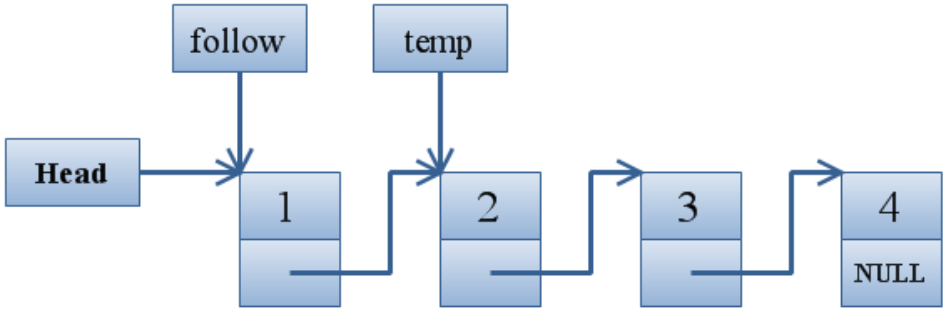
  temp = temp->next

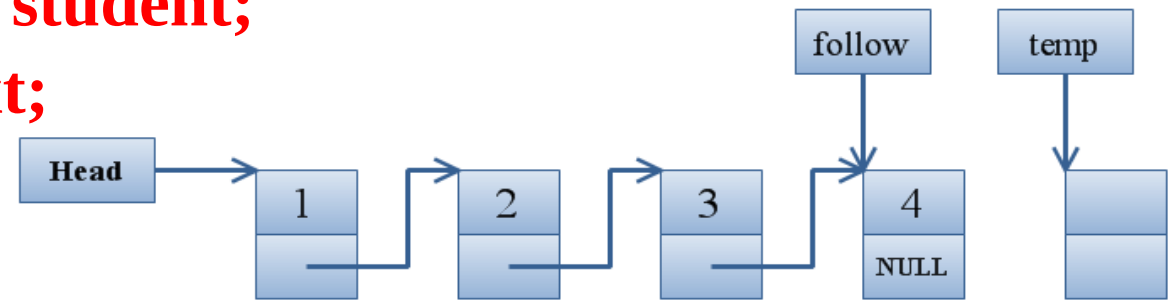


```
struct student
{
   int num;
   struct student *next;
};
student *create();      //返回指针的函数
void showList(student*);
void main( )
{
    student *head;
   head = create();      showList(head);
}
```

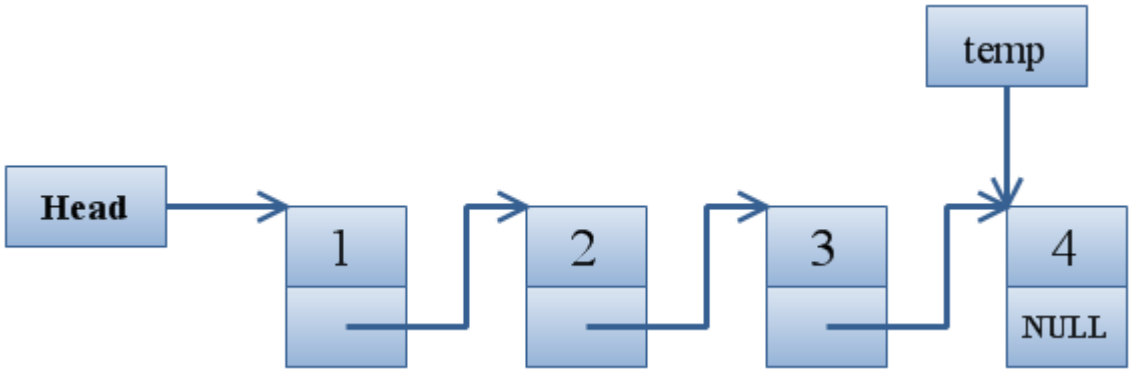


```
student *create()
{   student *follow,*temp,*head; int num, n;
    head = new student;
    temp = head;       n=0;
    cin >> num;
    while(num != -1)
    {      n = n + 1;
           temp->num = num
           follow = temp;
           temp->next = new student;
           temp = temp->next;
           cin>>num;
    }
    if (n==0) head=NULL; else follow->next = NULL;
    delete temp;  return(head);
}
```





```
void showList(student *head)
{   student *temp;
   if (head==NULL) cout<<“This is a null linker\n”;
   else
   {   temp=head;
       do{ cout<<temp->num;
            temp=temp->next;        //指针temp向后走，
          } while(temp!=NULL);    //直到遇到结束标志
   }
   cout<<endl;
}
```



链表结构——结点检索

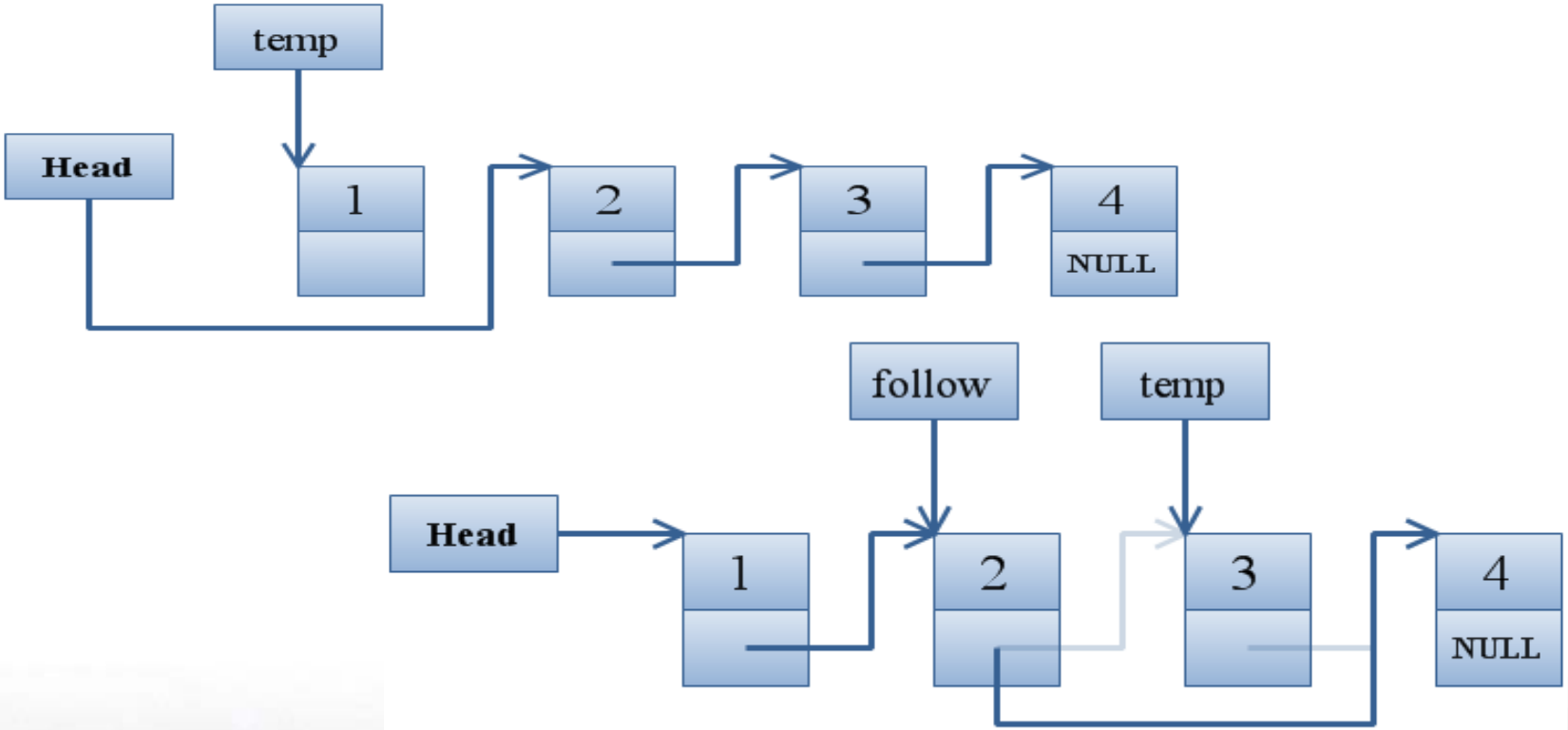
■ 在链表中找出n的位置，并输出。

```
cin >> n;
temp = head;
while (temp != null && temp->num != n)
{  temp = temp->next  };
                        //只要没有和n相同的数 并且
                        //还没有找到链表的末尾，则继续找
if (temp!=null) cout<<temp->num;
else cout<<“not found”;
```

链表结构——删除结点

在链表中将值为n的元素删掉

temp=head; head = head→next; delete temp;



follow→next = temp→next; delete temp;

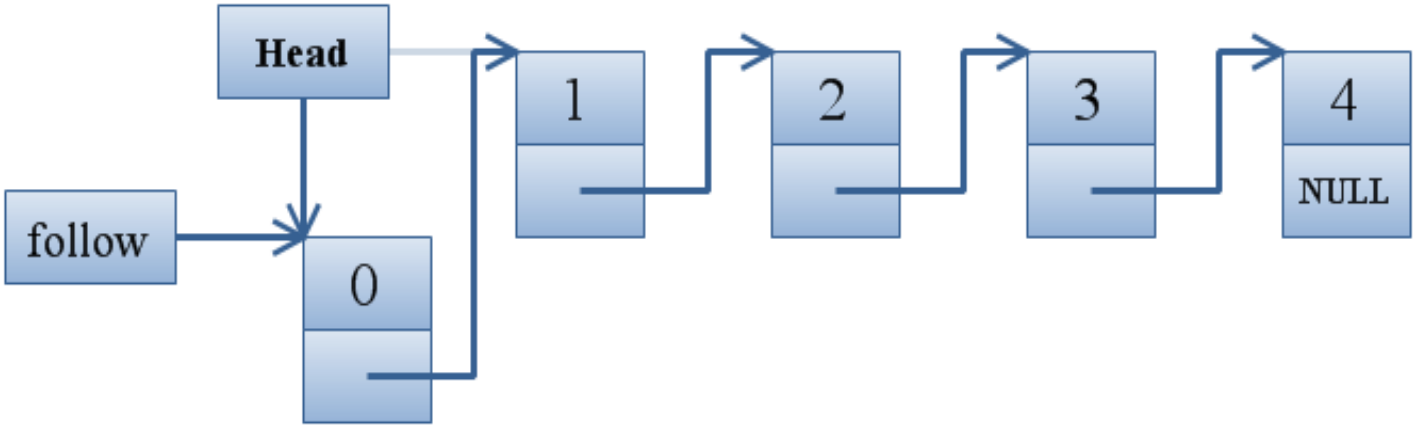
```
linker *dele(student *head; int n)
{   student *temp,* follow;
    temp = head;
    if (head == NULL) {                    //head为空，空表的情况
        return(head);
    }
    if (head->num = = n) {                 //第一个节点是要删除的目标;
        head = head->next;
        delete temp;
        return(head);
    }
    while(temp != NULL && temp->num != n){     //寻找要删除的目标;
        follow= temp;
        temp = temp->next;
    }
    if (temp == NULL) cout<<“not found”;       //没寻到要删除的目标;
    else {
        follow->next =temp->next;              //删除目标节点;
        delete temp;
    }
    return(head);
}
```

链表结构——插入结点

- 将结点unit插入链表:

插在最前面的情况

```
unit->next = head;
head = unit;
```

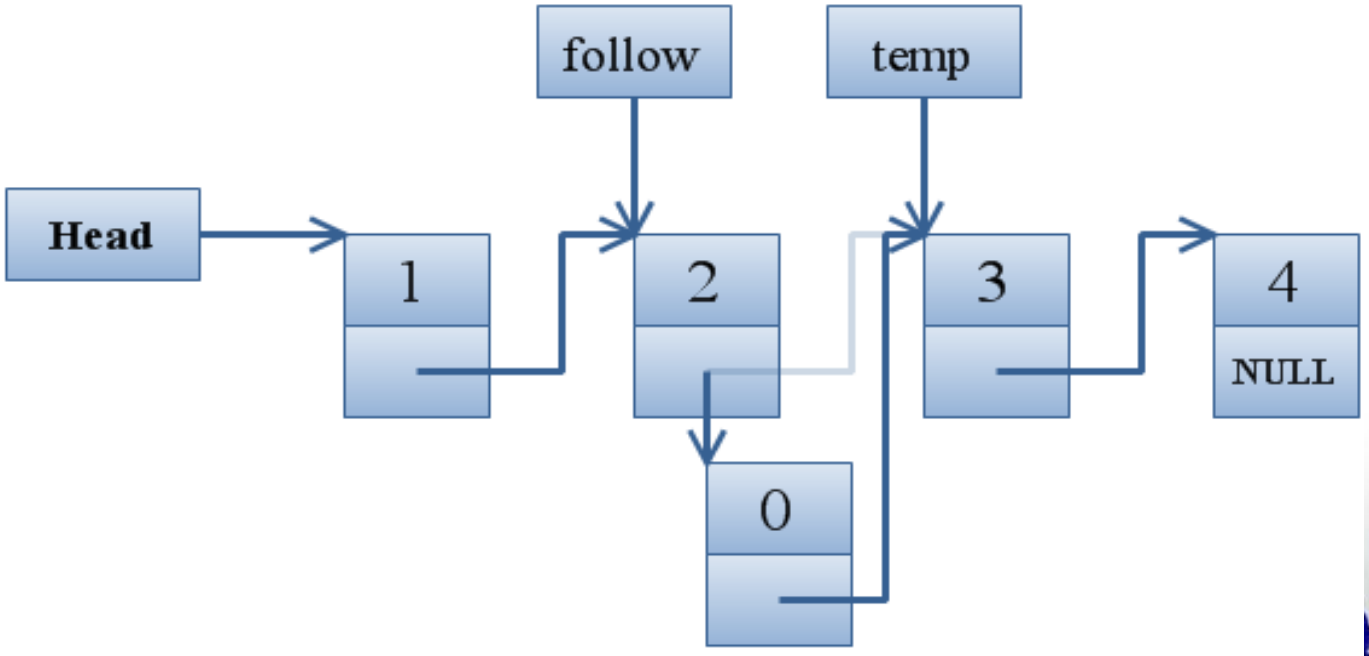


链表结构——插入结点

- 将结点unit插入链表:

插在中间的情况

```
unit->next = temp;
follow->next = unit;
```

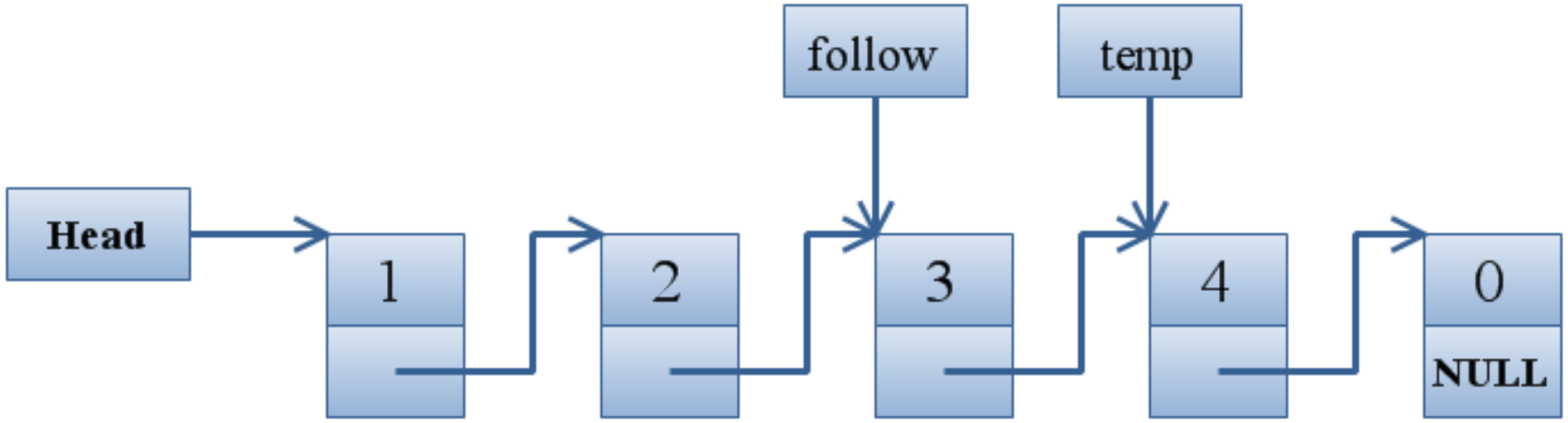


链表结构——插入结点

- 将结点unit插入链表:

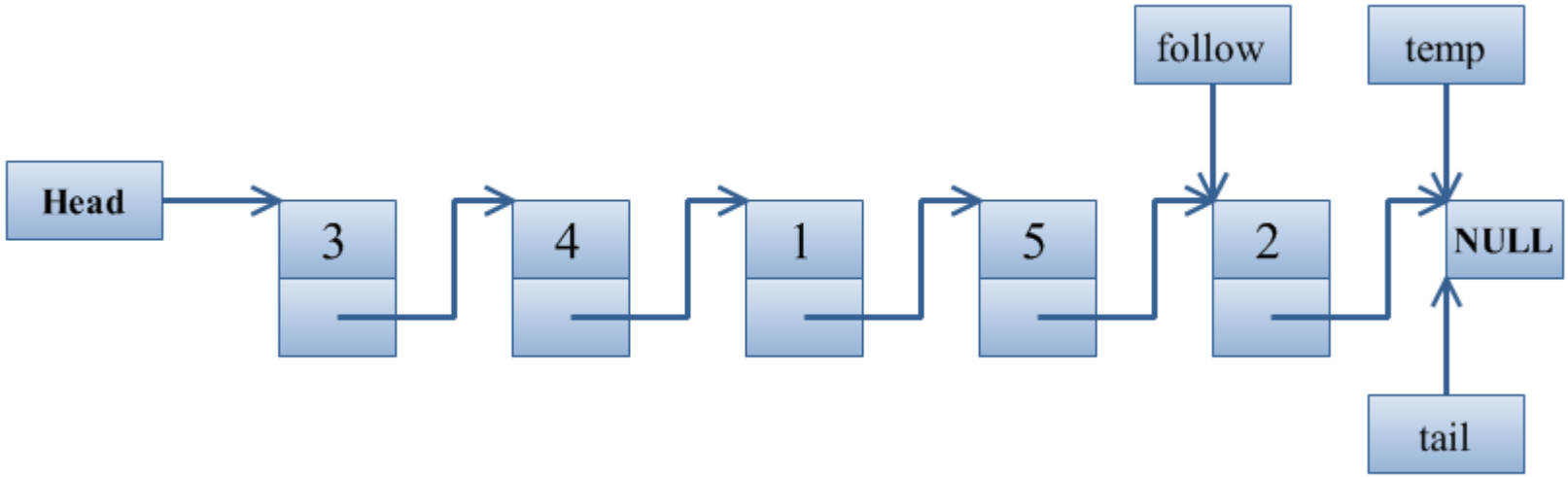
插在最后面的情况

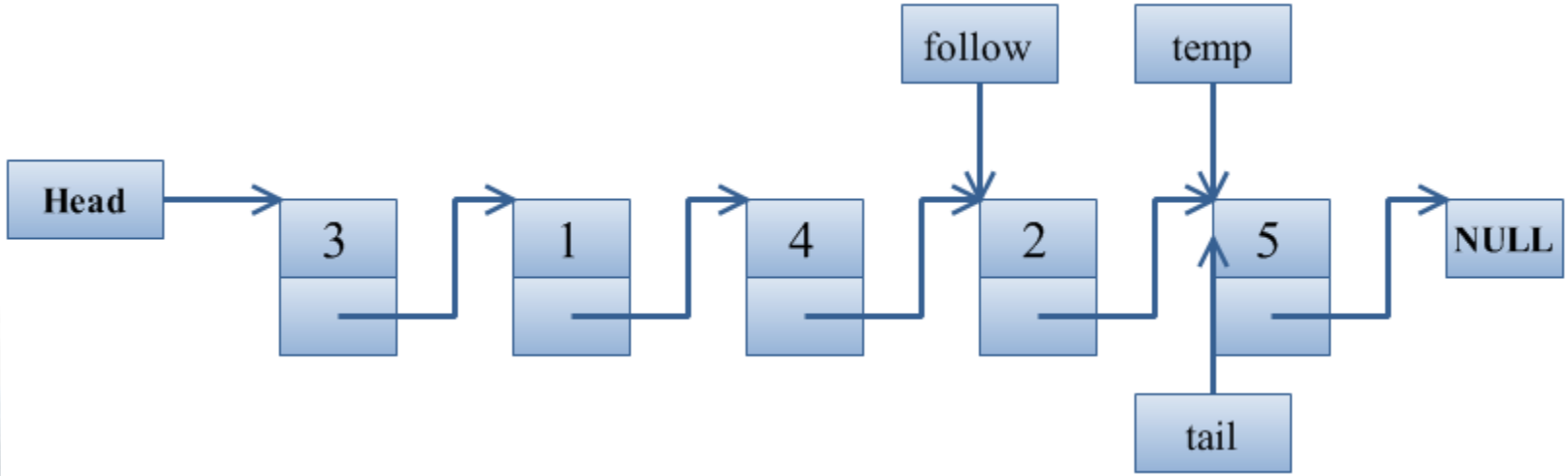
```
temp->next = unit;
unit->next = NULL;
```



```
Student *insert(student *head; int n) {
    student *temp,*unit,*follow;                    //插入结点值为n的结点
    temp = head;   unit = new student;
    unit->num = n;    unit->next = NULL;
    if (head==NULL) {                               //如果链表为空，直接插入
          head=unit;
          return(head);
    }                                   //寻找第一个不小于n或结尾的结点temp
    while((temp→next! = NULL)&&(temp->num < n)){
           follow=temp;  temp=temp->next;
    }
    if (temp==head){                                //如果temp为第一个结点
          unit->next=head; head=unit;
    }
    else{                                           //如果temp为最后一个结点
          if( temp->next == NULL) temp->next = unit;
          else{                                     //如果temp为一个中间结点
                    follow->next=unit; unit->next=temp;
          }
    }
    return(head);
}
```

链表结构——结点排序





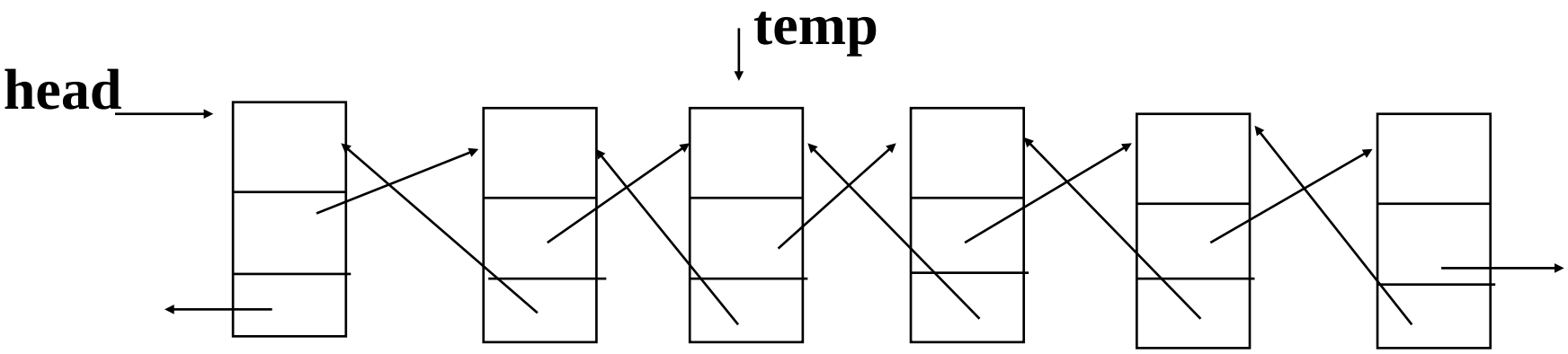
```
void sort (student *head) {  //将链表排序
   student *temp, *follow, *tail = NULL;    int t;
   while(head->next != tail) {
        follow= head;  temp = follow→next;
        while(temp!= tail) {
            if (follow→num > temp→num) {
                t = follow→num;
                follow→num = temp→num;
                temp→num=t;
            };
            follow = temp;
            temp = temp->next;
        }
        tail = follow;
   }
}
```

双向链表

- 单向链表的缺陷
  - 不能通过当前位置找到前面的元素;
- 双向链表
  - 有两个链，一个指向前面的元素，一个指向后面的元素;
  - 双向链表结点的定义

```
struct student
{ int num;
  student *ahead;
  student *next;
}
```

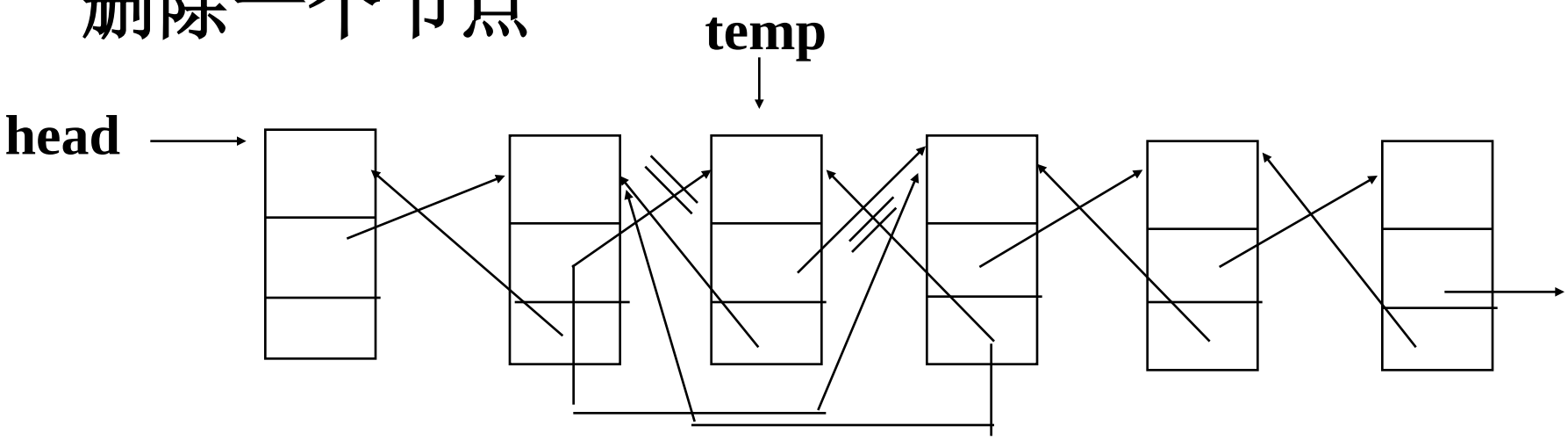
双向链表



- temp->num: 存放数据
- temp->next: 指向下一个
- temp->ahead: 指向前一个

双向链表——删除结点

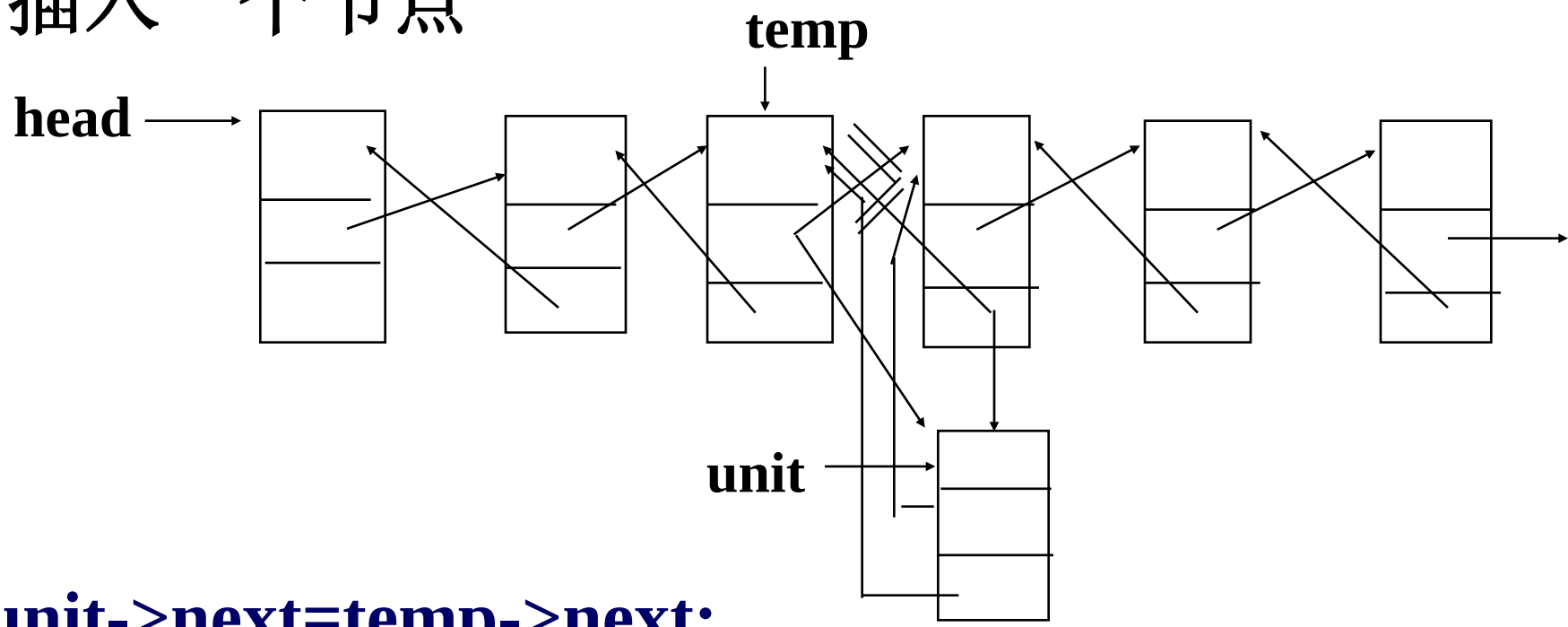
删除一个节点



- temp->ahead->next=temp->next;
- temp->next->ahead=temp->ahead;

双向链表——插入结点

插入一个节点



- unit->next=temp->next;
- unit->ahead=temp;
- temp->next->ahead=unit;
- temp->next=unit;

枚举类型

枚举类型

- 枚举
  - 如果一个变量只有几种可能的取值，则可以将该变量定义为“枚举类型”。
- 定义
  - 声明一个枚举数据类型weekday

    enum weekday {sun, mon, tue, wed, thu, fri, sat};

    花括号中sun,mon,…,sat等称为枚举元素
  - 定义枚举变量:

    enum weekday workday, weekend;

    或　　**weekday workday, weekend;**
  - 枚举变量赋值:

    workday = sun;　　weekend = mon;

枚举类型使用注意事项

1. 枚举元素按常量处理，不能对它们赋值。

   sun = mon; （错误）

2. 枚举类型不能直接输出其值，只能输出其顺序.

   [例如] **enum color {red, green,white,black};**

   color cloth = red;

   cout<<cloth; 结果为0

3. 枚举型可以比较

   if (cloth > white) count++;

4. 一个整型不能直接赋给一个枚举变量

   workday = 2; 错误!

枚举类型

- 枚举元素有值

  enum weekday {sun, mon, tue, wed, thu, fri, sat};

  - 若有**workday = mon;**

    则 **cout<<workday;** 将输出整数**1**

- 整数不能直接赋给枚举变量

  如：**workday = 2;** 错误！

  - 应先进行强制类型转换如:

    workday = (enum weekday) 2;

枚举类型的使用

- 如何输出枚举型变量的内容

```
enum color {red,green,blue,brown,white,black};
enum color choice;
 switch(choice)
{  case red:      cout<<“red\n”;    break;
   case green:    cout<<“green\n”; break;
   case blue:     cout<<“blue\n”;   break;
   case green:    cout<<“brown\n”;break;
   case red:      cout<<“white\n”; break;
   case green:    cout<<“black\n”; break;
}
```

枚举类型应用举例

- 问题
  - 按计时工资方法计算一周的付费。每周有7个工作日，周一至周五，按实际工作小时计算，周六工作时间按实际工作小时的1.5倍计算，周日工作时间按实际工作小时2.0倍计算。
- 用户输入
  - 每小时正常应付工资金额
  - 周一至周日每天的工作时间
- 程序输出
  - 计算出一周应付的工资。要求周一至周日用枚举类型表示;

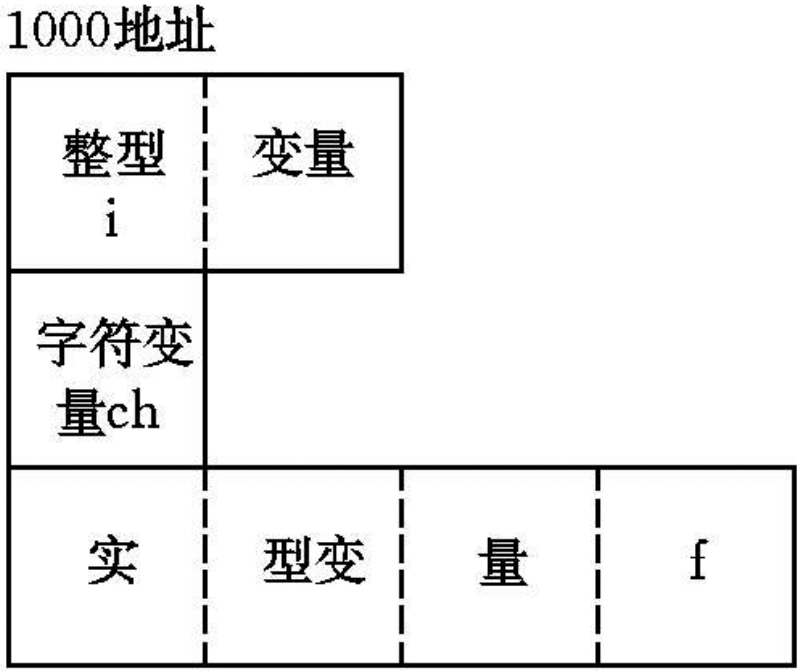
```
int main()
{   enum day{Mon,Tue,Wed,Thu,Fri,Sat,Sun};
    day workDay;
    double times, wages, hourlyRate, hours;
    cout<<"Enter the hourly wages rate.”<<endl;
    cin>>hourlyPay;
    cout<<"Enter hours worked daily\n";
    for (workDay=mon; workDay<=sun; workDay++)
    {     cin>>hours;
          switch(workDay)
          {   case sat:  times=1.5*hours; break;
              case sun:  times=2.0*hours; break;
              default:   times=hours;}
          wages = wages + times*hourlyPay;
    }
    cout<<“The wages for the week are ”<<wages;
    return 0;   }
```

共用体

共用体

- 共用体是什么？
  - 为了节省内存空间，可以将几种不同类型的变量存放到同一段内存单元中，这段内存单元所对应的数据结构称为共用体。

- 例如：
  - 一个整型变量、一个字符型变量、一个实型变量可以放在同一个地址开始的内存单元中。



共用体的定义

- 共用体的定义

```
union 共用体名
{
    成员列表;
} 变量表列;
```

```
union data
{
    int i ;
    char ch ;
    float f ;
} a, b, c;
```

直接定义

```
union data
{
    int i ;
    char ch ;
    float f ;
}
data   a, b, c;
```

分开定义

共用体的引用

- 不能引用共用体变量，只能引用共用体变量中的成员。
  - 例如：

```
union data
{
    int i ;
    char ch ;
    float f ;
} a, b, c;
```

a.i　(引用共用体变量中的类型变量i)

a.ch　(引用共用体变量中的字符变量ch)

a.f　(引用共用体变量中的实型变量f)

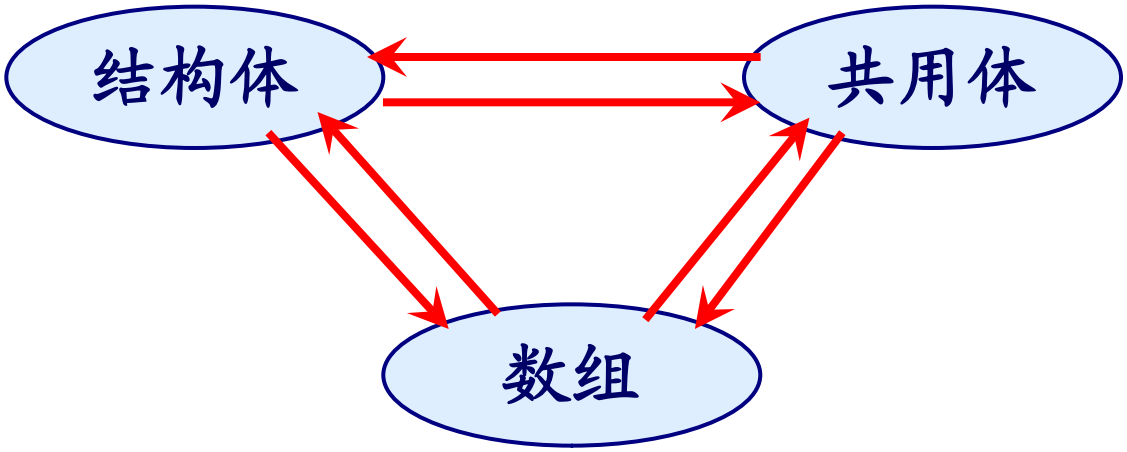
不能只引用共用体变量，例如：

cout<<a;　错误

共用体类型数据的特点

- 同一内存段可以存放几种不同类型的成员，但在同一时刻时只能存放其中一种;
- 共用体变量中起作用的成员是最后一次存放的成员，在存入一个新的成员后原有的成员就失去作用。
- 共用体变量的地址和它的各成员的地址都是同一地址。
  - 例如： **&a, &a.i, &a.c, &a.f** 都是同一地址值

共用体、结构体、数组



- 结构体 中可含 共用体;
- 共用体 中可含 结构体;
- 结构体 和 共用体 中可以包含 数组;
- 可以定义 共用体数组 和 结构体数组;

举 例

例11.12 设有若干个人员的数据，其中有学生和教师。

- 学生数据：姓名、号码、性别、职业、班级。
- 教师数据：姓名、号码、性别、职业、职务。

设计一个数据结构将多个学生和老师的数据用同一个数组存放。

name	num	sex	job	class(班) / position(职务)
Li	1011	f	s	501
wang	2085	m	t	prof

包含 共用体 的 结构体 数组

```
struct
{   int num;
    char name[10];
    char sex;
    char job;
    union
    {
        int class;
        char position[10];
    } category;
} person[2];
```

结构体

共用体

数组

共用体的特点

1. 共用体内的成员是互斥的;
2. 最后一次赋值保留在共用体中;
3. 共用体只有一个首地址;
4. 共用体不能初始化,不能对整个共用体赋值;
5. 在函数中，可以使用共用体的指针，但不能使用名字做函数参数;
6. 共用体的空间是所有成员中最大的一个;

好好想想，有没有问题?

谢　谢　！

北京大学