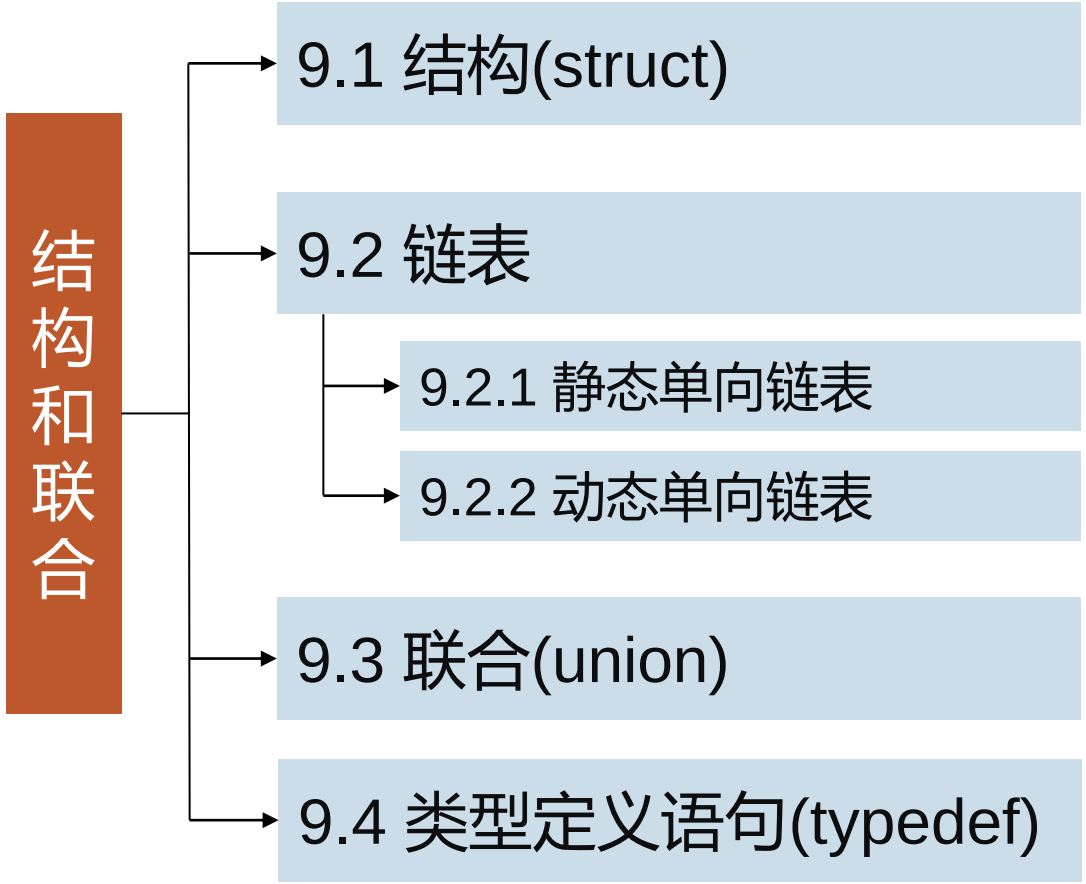
第 9 讲

结构和联合

Struct and Union

第9讲 结构和联合



学习要点

1. 结构类型的定义
2. 结构成员的访问
3. 包含结构的结构
4. 单向链表的定义、操作与应用
5. 联合类型的定义与成员的访问
6. 类型定义(typedef)语句

【复习】

	定义	赋值	引用

变量	int a=0,b;
	double d=2.3;
	char c='s';
		a=20;
		d=a+43.2;
		c='A'+3;
			b=a-2;
			printf("%f",d);
			scanf("%c",&c);

数组	int arr_a[10],arr_b[200]={0};
	double arr_d[5]={0.0};
	char arr_c[1024]={'\0'};
		arr_a[3]=11;
		for(b=0;b<5;b++) arr_d[b]=b+2;
		arr_c[b]='0';
			for(b=0;b<5;b++)
			arr_d[b]=arr_a[9-b]+2;
			printf("%s",arr_c);

指针	int* p_a,*p_b=&b;
	double* p_d=arr_d+3;
	char* p_c=&arr_c[2];
		p_a=p_b;
		p_d=&arr_d[4]-3;
		p_c=&arr_c[3];
			*p_b=33;
			p_d++;
			if(*p_c!='\0') p_c++;

基本表达形式，表示最简单的个体

为什么需要结构类型和联合类型

为什么需要结构类型和联合类型

程序设计中常需要处理一些相互关联的复杂数据：

- 为表示三维空间中的一个点，需要记录x、y、z轴分量。
- 学生管理系统中，学生信息包括姓名、班级、学号、家庭住址、电话号码、电子邮箱等。
- OJ排行榜：姓名、学号、得分、罚时、每道题过题时间

为什么需要结构类型和联合类型

相互关联的复杂数据：

- OJ排行榜：姓名、学号、得分、罚时、每道题过题时间（如OJ排名表，如下）

2018级 航空航天类期中自测与期末模拟考试综合练习

比赛排名 更新于 2019-04-23 15:05:48

« ‹ 1 2 3 4 5 6 7 8 9 › »

简介
题目
排名
提交
回复&&公告
导出成绩

服务器当前时间
2019-04-23 15:05:48

比赛结束时间
2019-04-27 20:30:00

比赛剩余时间
101:24:11

排名	用户	账号	学号	得分	罚时	A 848/861	B 592/761
1	董翰元		18375354	800	3:26:01	0:01:51	0:06:35(+1)
2	赵锴		18376195	800	6:03:51	0:02:09	0:11:08(+2)
3	李国琛		18375221	800	7:01:05	0:04:40	0:14:22
4	周钰皓		18376166	800	7:13:33	0:05:08	0:20:47
5	商瑞辰		18376233	800	7:58:03	0:06:58	0:23:22
6	刘子航		18374159	800	8:08:53	0:19:46	0:34:08
7	崔子昂		18374412	800	8:23:36	0:03:52	0:16:46(+3)
8	邢静怡		18376407	800	8:37:21	0:05:41	0:22:30(+2)
9	张亦弛		18375251	800	8:50:24	1:07:11	1:13:16(+1)
10	周家乐		18374434	800	8:58:00	0:03:56	1:13:24(+1)

```
struct  Student_Info
{
    char  ID[9], name[20];
    unsigned int  grade;
    char cost_time[15];
    char problems_time[20][15];
};
struct  Student_Info  student[2000];
```

这些计算对象具有多种属性，不同属性的数据类型可能相同、也可能不同，为把这些数据组织在一起、构成由一个或多个类型不一定相同的变量组成的集合，C语言提供了可由程序员自行定义的构造类型：结构(struct)和联合(union)

为什么需要结构类型和联合类型

不使用结构体

```
…
int main()
{
    char  ID[2000][9], name[2000] [20];
    unsigned int  grade[2000];
    char cost_time[2000] [15];
    char problems_time[2000] [20][15];
  …
    print_info( ID, name, grade, … );
  …
}
```

1. 影响程序员进行 C 语言编程的便利性
2. 程序出错的概率大一些
3. 尽管不影响机器执行

采用结构体

```
…
struct  Student_Info
{
    char  ID[9], name[20];
    unsigned int  grade;
    char cost_time[15];
    char problems_time[20][15];
};
int main()
{
      struct  Student_Info  student[2000];
      …
      print_info(student);
      …
}
```

结构、联合是编程语言发展、丰富语义的发明。

9.1 结构

"Hello World" of struct

【例9-1】输入圆的半径，求周长和面积。

circle 是一个构造类型（自己定义的），可以用它来定义变量。

cir_1和cir_2是circle类型的变量，cir_1和cir_2中都各自含有半径、周长、面积。

```c
#include <stdio.h>
#include <string.h>
#define PI 3.1415927
char prnFormat[40] = "r = %-8.3f: P = %-8.3f, A = %-8.3f\n";
#define PRINT(C) printf(prnFormat, C.radius, C.perimeter, C.area);
struct circle
{
    float radius;
    float perimeter;
    float area;
};

int main()
{
    struct circle cir_1, cir_2;
    scanf("%f", &cir_1.radius);    scanf("%f", &cir_2.radius);
    cir_1.perimeter = 2*PI*cir_1.radius;
    cir_1.area = PI*cir_1.radius*cir_1.radius;
    cir_2.perimeter = 2*PI*cir_2.radius;
    cir_2.area = PI*cir_2.radius*cir_2.radius;
    PRINT(cir_1);      PRINT(cir_2);
    return 0;
}
```

什么是结构

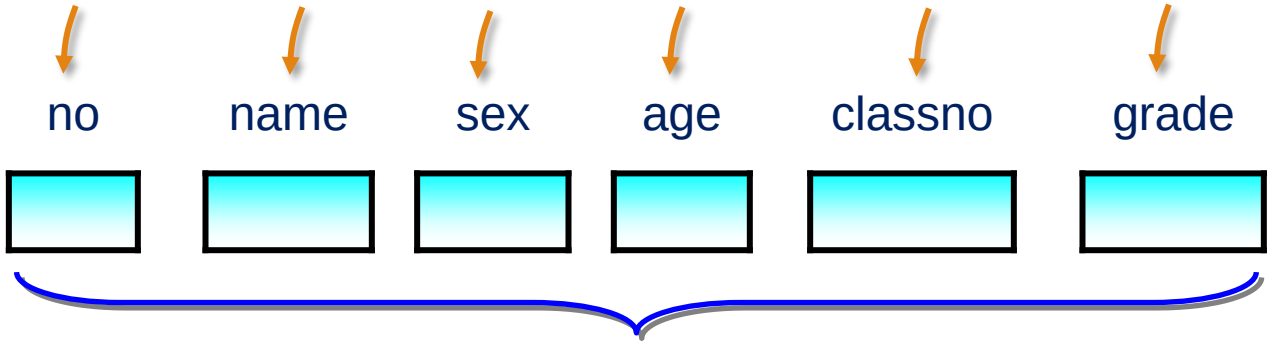
- 结构（结构体）是一种构造数据类型
- 结构类型把一组在运算中关系密切的变量组织在一个名字下，用以描述复杂运算对象的多种属性和成分。
- 引入结构类型的好处：加强数据项之间的联系。

如学生的基本信息，包括学号 、姓名、 性别、 年龄、 班级、 成绩等数据项。

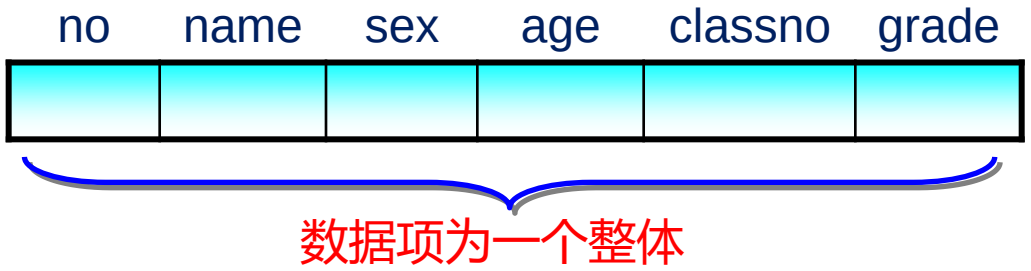
```
unsigned int no;        //学号
char         name[20]; //姓名
char         sex;       //性别
unsigned int age;       //年龄
unsigned int classno;   //班级
float        grade;     //成绩
```

这些数据项描述了一名学生的几个不同属性。

独立的变量表示：



结构类型变量表示：



结构类型的定义

```
struct  [结构类型名]
{
  数据类型名1  成员名1;
  数据类型名2  成员名2;
  …… ……
  数据类型名n  成员名n;
};
```

struct是关键字，不能省略

成员类型可以是基本型或构造型

合法标识符
可省:无名结构体

结构体类型并非只有一种，而是可以设计出许多种结构体类型，各自包含不同的成员。

例:

```
struct  pt_2d
{
  int x, y;  //二维坐标
};
```

描述二维点

例:

```
struct Date
{
  int year;   //年
  int month;  //月
  int day;    //日
};
```

描述日期

例:

```
struct  Student_Info{
 unsigned int  no;        //学号
 char          name[20];  //姓名
 char          sex;       //性别
 unsigned int  age;       //年龄
 unsigned int  classno;   //班级
 float         grade;     //成绩
};
```

描述学生的基本信息

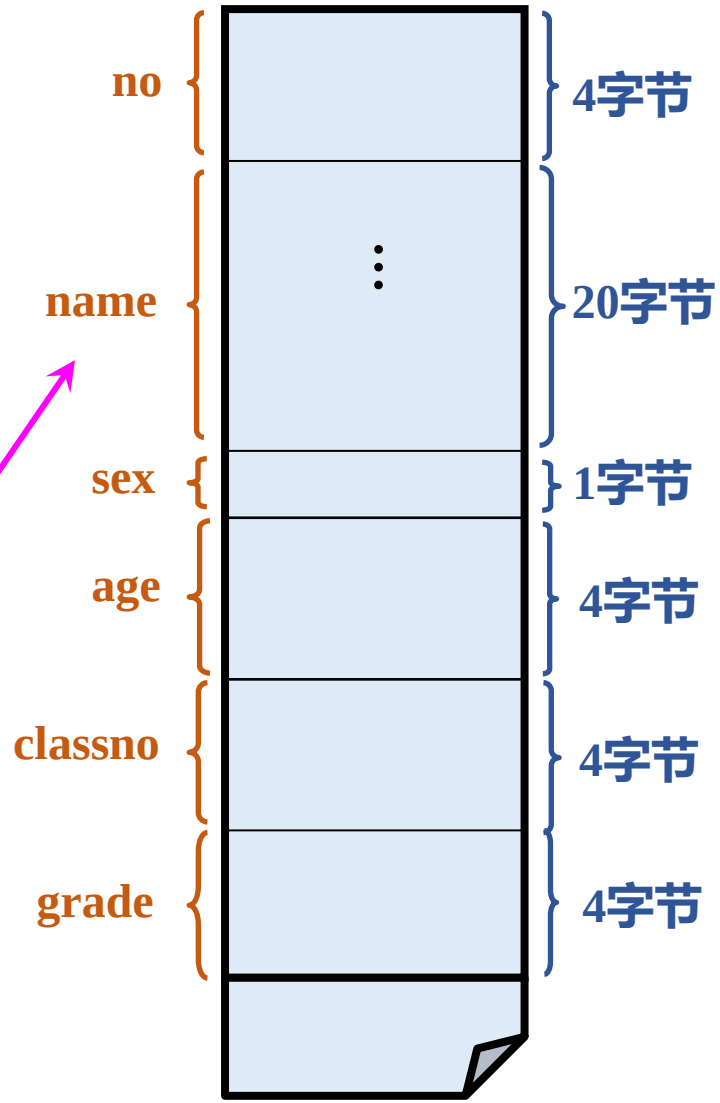
结构类型变量的定义

- **间接定义法：先定义结构类型，再定义结构变量**

```
struct 结构类型名
{
  数据类型名1  成员名1;
  …… ……
  数据类型名n  成员名n;
};
struct 结构类型名 变量名列表;
```

```
struct  Student_Info
{
  unsigned int  no;
  char name[20], sex;
  unsigned int  age, classno;
  float  grade;
};
struct  Student_Info  student;
```

内存映像



注意

结构类型只是用户自定义的一种数据类型，用来定义描述结构的组织形式不分配内存，只有用它定义某个结构变量时，才会为该变量分配结构类型所需要大小的内存单元。所占内存的大小是它所包含的成员所占内存大小之和。

结构类型变量的定义

- **直接定义法：定义结构类型的同时定义结构变量**

```
struct [结构体类型名]
{
  数据类型名1   成员名1;
  …… 
  数据类型名n   成员名n;
} 变量名列表;
```

```
struct  Student_Info
{
  unsigned int   no;        //学号
  char           name[20];  //姓名
  char           sex;       //性别
  unsigned int   age;       //年龄
  unsigned int   classno;   //班级
  float          grade;     //成绩
} student1, student2;
```

或

```
struct
{
  unsigned int   no;        //学号
  char           name[20];  //姓名
  char           sex;       //性别
  unsigned int   age;       //年龄
  unsigned int   classno;   //班级
  float          grade;     //成绩
} student1, student2;
```

无名结构体定义，定义变量只能一次

注意

结构体类型与结构体变量是不同的概念，不要混淆。只能对变量赋值或运算，不能对类型赋值、存取或运算。

结构变量的初始化

```
struct 结构类型名
{ …… };
struct 结构体类型名 变量名 = {成员1的值， …， 成员n的值};
```

初值表

```
struct Student_Info
{
  unsigned int  no;       //学号
  char          name[20];//姓名
  char          sex;      //性别
  unsigned int  age;      //年龄
  unsigned int  classno; //班级
  float         grade;    //成绩
};
```

```
struct Student_Info stu = { 20020306 , "ZhangMing",  'M',  18,      1,      90};
```

√

no　name　sex　age　classno　grade

```
struct  Student_Info  stu = {18, "ZhangMing", 'M',  20020306 , 1, 90};
```

×

注意

- ✓ 赋初值时，{ }中间的数据顺序必须与结构成员的定义顺序一致，否则就会出现混乱。
- ✓ 对结构体变量初始化，不是对结构体类型初始化

结构成员的访问

- **结构变量不能整体引用，只能引用变量成员**

结构成员访问方式：

结构变量名.成员名 //非指针型结构成员的访问

结构指针->成员名 或 (*结构指针).成员名 //指针型结构成员的访问

```
struct  Student_Info
{
     unsigned int no;
     char  name[20];
     char sex;
     unsigned int age;
     unsigned int classno;
     float grade;
} stu;
```

执行语句？

printf("%u, %s, %c, %u, %u, %f\n", stu); (×)

执行语句？

stu={20020306, "Wan Lin",'M',19,200203,87.5}; (×)

结构成员的访问

- **结构变量不能整体引用，只能引用变量成员**

  结构成员访问方式：

结构变量名.成员名 //非指针型结构成员的访问

结构指针->成员名 或 (*结构指针).成员名 //指针型结构成员的访问

```
struct  Student_Info
{
    unsigned int no;
    char  name[20];
    char sex;
    unsigned int age;
    unsigned int classno;
    float grade;
} stu;
```

strcpy (stu.name, "zhangMing");

stu.grade = 80;

stu.grade *= 1.2;

printf ("%s %f", stu.name, stu.grade);

scanf ("%f", &stu.grade);

执行这些语句？

对结构体变量的成员可以像普通变量一样进行各种运算（根据其类型决定可以进行的运算）。

结构成员的访问

【例9-2】输入两个学生的学号、姓名和成绩，输出成绩较高的学生的学号、姓名和成绩。

注意

可以引用结构体变量成员的地址。但不能用以下语句整体读入结构体变量。

scanf("%d%s%f\n", &student1); ✗

```
#include <stdio.h>
int main()
{   struct Student
    {   int num;
        char name[20];
        float score;
    }student1,student2;
    scanf("%d%s%f",&student1.num,student1.name,&student1.score);
    scanf("%d%s%f",&student2.num,student2.name,&student2.score);
    printf("The higher score is:\n");
    if(student1.score>student2.score)
        printf("%d  %s  %6.2f\n",student1.num,student1.name,student1.score);
    else if(student1.score<student2.score)
        printf("%d  %s  %6.2f\n",student2.num,student2.name,student2.score);
    else
    {   printf("%d  %s  %6.2f\n",student1.num,student1.name,student1.score);
        printf("%d  %s  %6.2f\n",student2.num,student2.name,student2.score);
    }
    return 0;
}
```

声明结构体类型

定义两个结构体变量

输入学生1的数据

输入学生2的数据

运行结果：

```
202001 Wang 70
202002 Ling 90
The higher score is:
202002  Ling   90.00
```

结构成员的访问

- **结构变量不能整体引用，只能引用变量成员**

  结构成员访问方式：

结构变量名.成员名 //非指针型结构成员的访问

结构指针->成员名 或 (*结构指针).成员名 //指针型结构成员的访问

```
struct  Student_Info
{
      unsigned int no;
      char  name[20];
      char sex;
      unsigned int age;
      unsigned int classno;
      float grade;
} stu, *pstu = &stu;
```

strcpy (pstu->name, "zhangMing");

(*pstu).grade = 80;

pstu->grade *= 1.2;

printf ("%s %f", pstu->name, (*pstu).grade);

scanf ("%f", &((*pstu).grade));

执行这些语句？

结构体指针是指向结构体变量的指针，一个结构体变量的起始地址就是这个结构体变量的指针。如果把一个结构体变量的起始地址存放在一个指针变量中，那么，这个指针变量就指向该结构体变量。

结构变量在程序中赋值

- 如果在定义结构变量时并未对其赋初始值，那么在程序中要对它赋值的话，就只能一个一个地对其成员逐一赋值，或者用已赋值的同类型的结构变量对它赋值 。

```
struct  Student_Info   stu;

stu.no = 20020306;
strcpy (stu.name, "ZhangMing");
stu.sex = 'M';
stu.age = 18;
stu.classno = 1;
stu.grade = 90;
```

逐一赋值

stu1 += stu; (×)

```
stu1.no = stu.no;
strcpy (stu1.name, stu.name);
stu1.sex = stu.sex;
stu1.age = stu.age;
stu1.classno = stu.classno;
stu1.grade = stu.grade;
```

利用已赋值的结构变量赋值

```
struct  Student_Info  stu1;
stu1 = stu;
```

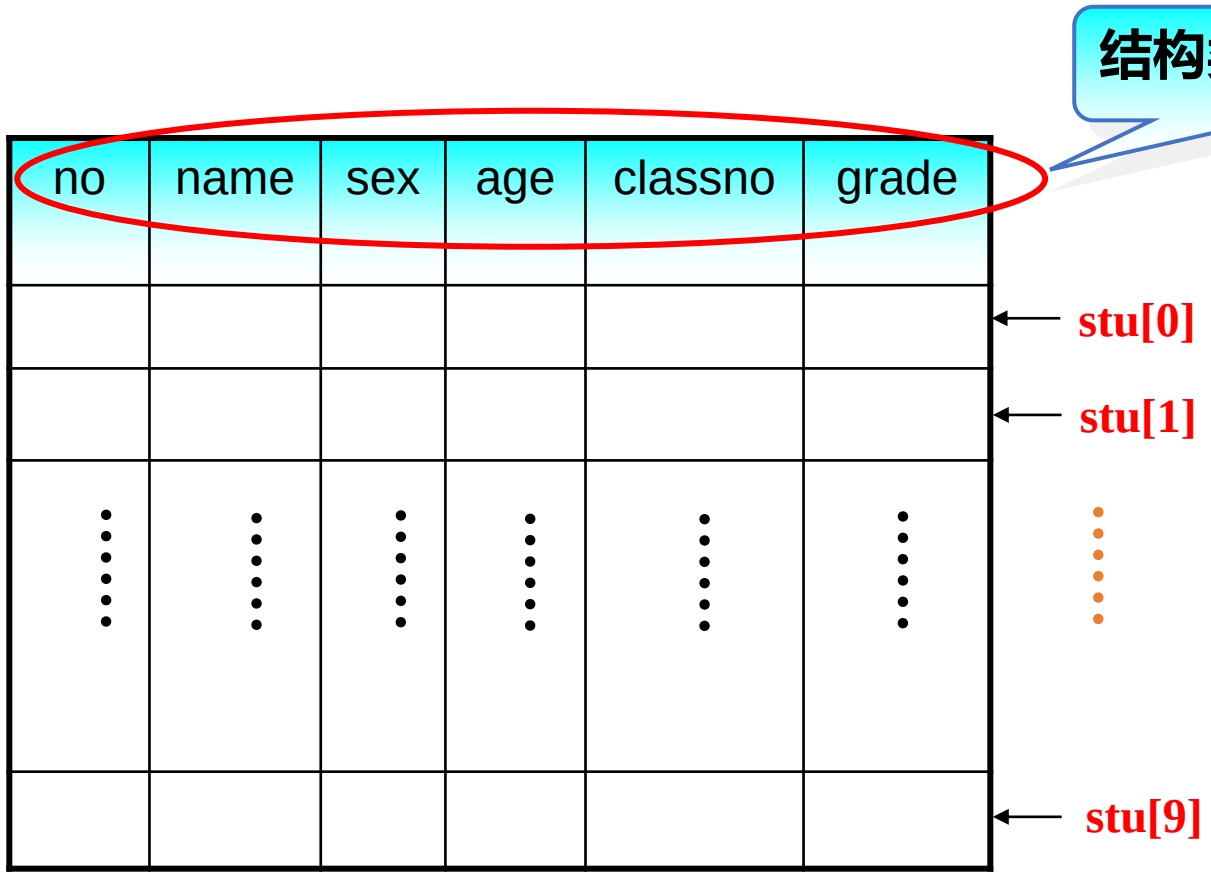
利用memcpy赋值

```
memcpy (&stu1, &stu, sizeof(struct Student_Info));
```

memcpy是c使用的内存拷贝函数，其功能是从源src所指的内存地址的起始位置开始拷贝n个字节到目标dest所指的内存地址的起始位置中。

结构数组与二维表的对应关系

- 结构数组就相当于一张二维表，一个表的框架对应的就是某种结构类型，表中的每一列对应该结构的成员，表中每一行信息对应该结构数组某元素各成员的具体值，表中的行数对应结构数组的大小。




```
struct  Student_Info
{
    unsigned int no;
    char name[20];
    char sex;
    unsigned int  age;
    unsigned int  classno;
    float  grade;
} stu[10];
```

结构数组

【例9-3】有n个学生的信息(包括学号、姓名、成绩)，要求按照成绩的高低顺序输出各学生的信息。

```
#include <stdio.h>
struct Student //声明结构体类型struct Student
{   int num;
    char name[20];
    float score;
};
int main()
{   struct Student stu[5]={{202001,"Zhang",67.4},{202002,"Wang",97.5},
     {202003,"Li",92.3}, {202004,"Ling",84.2},{202005,"Sun",75}};
    struct Student temp;  //定义结构体变量temp，用作交换时的临时变量
    const int n=5;   //定义常变量n
    int i,j,k;

    for(i=0;i<n-1;i++)
    {   k=i;
        for(j=i+1;j<n;j++)
            if(stu[j].score>stu[k].score)  //进行成绩的比较
                k=j;
        temp=stu[k]; stu[k]=stu[i]; stu[i]=temp;//stu[k]和stu[i]元素互换
    }
    for(i=0;i<n;i++)
        printf("%6d %7s %6.1f\n",stu[i].num,stu[i].name,stu[i].score);
    printf("\n");
    return 0;
}
```

定义结构体数组并初始化

运行结果：

```
202002    Wang   97.5
202003      Li   92.3
202004    Ling   84.2
202005     Sun   75.0
202001   Zhang   67.4
```

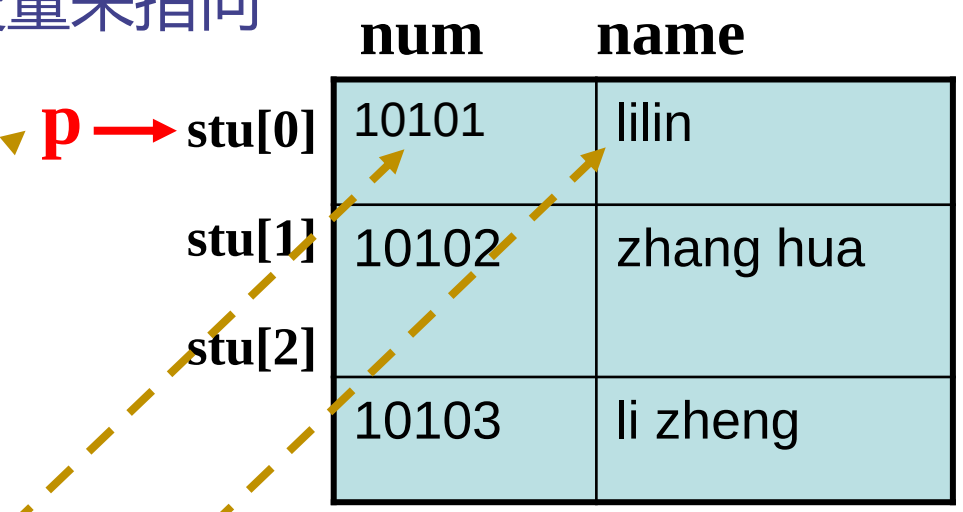
结构指针与结构数组的配合使用

- 对结构数组及其元素也可以用指针变量来指向

```
……
struct student
{   int num;
    char name[20];
};

struct student stu[3] = {
   {10101, "lilin"},
   {10102, "zhang hua"},
   {10103, "li zheng"}};  //学生信息结构数组

int main( )
{
    struct student  *p;  //结构指针
    print("No.      Name\n");
    for (p=stu; p<stu+3; p++)
        print("%5d  %-20s\n", p->num, p->name);
    ...
}
```



若p初值为stu,指向第一个元素，则p+1后指向下一个元素起始地址

如果p指向一个结构体变量或数组元素stu[0]，以下3种用法等价:

① stu[0].成员名

② (*p).成员名

③ p->成员名

结构体和函数参数

```
struct  Student_Info   stu;
strcpy (stu.name, "ZhangMing");
stu.grade = 90;
```

(1) 结构体变量的成员作函数实参

用法和用普通变量或数组作实参是一样的。

应当注意实参与形参的类型保持一致。

```
void print_info(char* name, float grade)
{   printf ("%s %f",name,grade);  }
```

```
print_info( stu.name, stu.grade);
```

(2) 结构体变量作函数实参

将结构体变量所占的内存单元的内容全部按顺序传递给形参，形参也必须是同类型的结构体变量。在函数调用期间形参也要占用内存单元。这种传递方式在空间和时间上开销较大，如果结构体的规模很大时，开销是很可观的。

```
void print_info(struct Student_Info stud)
{printf ("%s %f", stud.name, stud.grade);}
```

```
print_info( stu );
```

(3) 用指向结构体变量的指针作函数实参

将结构体变量的地址传给形参。

```
void print_info(struct Student_Info* stud)
{printf ("%s %f", stud->name, stud->grade);}
```

```
print_info( &stu);
```

结构体和函数参数

【例9-4】有n个结构体变量，内含学生学号、姓名和3门课程的成绩。要求输出平均成绩最高的学生的信息(包括学号、姓名、3门课程成绩和平均成绩)。

```c
#include <stdio.h>
#define N 3          //假设学生数为3
struct Student       //结构体类型
{   int num;         //学号
    char name[20];   //姓名
    float score[3]; //成绩
    float aver;      //平均成绩
};
int main()
{   void input(struct Student stu[]);   //函数声明
    struct Student max(struct Student stu[]);   //函数声明
    void print(struct Student stu); //函数声明
    struct Student stu[N],*p=stu;   //定义结构体数组和指针
    input(p);                       //调用input函数
    print(max(p));  //调用print函数,以max函数的返回值作为实参
    return 0;
}
void input(struct Student stu[])        //定义input函数
{   int i;
    for(i=0;i<N;i++)
    {   scanf("%d %s %f %f %f",&stu[i].num,stu[i].name,
        &stu[i].score[0],&stu[i].score[1],&stu[i].score[2]);
        stu[i].aver=(stu[i].score[0]+stu[i].score[1]+stu[i].score[2])/3.0; //求平均成绩
    }
}
struct Student max(struct Student stu[])    //定义max函数
{   int i,m=0;
    for(i=0;i<N;i++)
    if(stu[i].aver>stu[m].aver) m=i;  //找出平均成绩最高的学生在数组中的序号
    return stu[m];      //返回包含该生信息的结构体元素
}
void print(struct Student stud)         //定义print函数
{
    printf("num:%d\nname:%s\nscores:%5.1f,%5.1f,%5.1f\naver: %6.2f\n",
      stud.num,stud.name,stud.score[0],stud.score[1],stud.score[2],stud.aver);
}
```

运行结果：

```
200201 Fang 66 77 88
200202 Wang 77 88 99
200203 Tang 70 80 90
num:200202
name:Wang
scores: 77.0, 88.0, 99.0
aver:  88.00
```

结构应用举例

【例9-5】输入数据编号问题　从标准输入上读入n(1<n<200000)个实数，将其按数值从小到大连续编号，相同的数值具有相同的编号。在标准输出上按输入顺序以<编号>：<数值>的格式输出这些数据，其中<数值>保持输入时的格式和小数部分的长度。各数据之间以空格符分隔，以换行符结束。

输入样例：5.3　4.7　3.65　12.345　6e2，

输出样例：3:5.3　2:4.7　1:3.65　4:12.345　5:6e2

题目是不是似曾相识…… 回想【例7-13】……

结构应用举例

【例9-5】输入数据编号问题 从标准输入上读入n(1<n<200000)个实数，将其按数值从小到大连续编号，相同的数值具有相同的编号。在标准输出上按输入顺序以<编号>：<数值>的格式输出这些数据，其中<数值>保持输入时的格式和小数部分的长度。各数据之间以空格符分隔，以换行符结束。 输入样例：5.3 4.7 3.65 12.345 6e2 输出样例：3:5.3 2:4.7 1:3.65 4:12.345 5:6e2

回想【例7-13】的解决方案……

使用二维数组作为存储结构
（1）读入数据并记录读入顺序；
（2）对数据按大小排序后编号；
（3）再对数据按输入顺序排序；
（4）按顺序输出编号及数据。

N行3列数组

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

数值　　输入顺序　　数据编号

结构应用举例

【例9-5】输入数据编号问题 从标准输入上读入n(1<n<200000)个实数，将其按数值从小到大连续编号，相同的数值具有相同的编号。在标准输出上按输入顺序以<编号>：<数值>的格式输出这些数据，其中<数值>保持输入时的格式和小数部分的长度。各数据之间以空格符分隔，以换行符结束。 输入样例：
5.3 4.7 3.65 12.345 6e2 输出样例：3:5.3 2:4.7 1:3.65 4:12.345 5:6e2

问题分析

与【例7-13】类似，不同的是输入数据为实数且要求输出数据格式与输入时一致。如果仅仅是输入数据从整数变为实数，仍可采用【例7-13】中n行3列的double或float数组存储，通过两次排序按输入顺序生成带编号的数据。

但是，这样做无法保证输出数据格式与输入格式一致，因为在读入输入数据时每个数据的小数点后位数不一样，很难统一处理，统一处理会丢失有些数据的格式和小数部分的长度等信息。

输入样例：

5.3 4.7 3.65 12.345 6e2

小数点后位数	1位	2位	3位	
记数法	非科学记数法	非科学记数法	非科学记数法	科学记数法

输出： ？？

%.3f 5.300 4.700 3.650 12.345 600.000

%.1f 5.3 4.7 3.7 12.3 600.0 ✗

结构应用举例

【例9-5】输入数据编号问题 从标准输入上读入n(1<n<200000)个实数，将其按数值从小到大连续编号，相同的数值具有相同的编号。在标准输出上按输入顺序以<编号>：<数值>的格式输出这些数据，其中<数值>保持输入时的格式和小数部分的长度。各数据之间以空格符分隔，以换行符结束。 输入样例：5.3 4.7 3.65 12.345 6e2 输出样例：3:5.3 2:4.7 1:3.65 4:12.345 5:6e2

解决思路

为保证输入/输出数据格式一致，最简单的方法是将输入数据以字符串形式与输入数据的值一起保存起来，在数据排序完成后直接输出该字符串。可定义一个包含一个double成员、两个int成员、一个char数组成员的结构数组，保存输入数据的值、数据的输入顺序、数据的排序编号以及输入的字符串。

```
struct data_t {
    double value; //数据的值
    int order;    //数据的输入顺序
    int rank;     //数据的排序编号
    char str[MAX_LEN];
                  //输入的字符串
} list[MAX_N];
```

结构应用举例

【例9-5】输入数据编号问题 从标准输入上读入n(1<n<200000)个实数，将其按数值从小到大连续编号，相同的数值具有相同的编号。在标准输出上按输入顺序以<编号>：<数值>的格式输出这些数据，其中<数值>保持输入时的格式和小数部分的长度。各数据之间以空格符分隔，以换行符结束。 输入样例：5.3 4.7 3.65 12.345 6e2 输出样例：3:5.3 2:4.7 1:3.65 4:12.345 5:6e2

```
struct data_t {
    double value;
    //数据的值
    int order;
    //数据的输入顺序
    int rank;
    //数据的排序编号
    char str[MAX_LEN];
    //输入的字符串
} list[MAX_N];
```

结构数组list[]

value 数值	Order 输入顺序	rank 数据编号	str 字符串
5.3	1		"5.3"
4.7	2		"4.7"
3.65	3		"3.65"
12.345	4		"12.345"
6e2	5		"6e2"

(1) 读入数据并记录数据值value、顺序order和输入的字符串str

结构数组list[]

value 数值	Order 输入顺序	rank 数据编号	str 字符串
3.65	3	**1**	"3.65"
4.7	2	**2**	"4.7"
5.3	1	**3**	"5.3"
12.345	4	**4**	"12.345"
6e2	5	**5**	"6e2"

(2) 按数值value大小排序后编号

结构数组list[]

value 数值	Order 输入顺序	rank 数据编号	str 字符串
5.3	1	3	"5.3"
4.7	2	2	"4.7"
3.65	3	1	"3.65"
12.345	4	4	"12.345"
6e2	5	5	"6e2"

(3) 按输入顺序order排序后输出

结构应用举例

```
struct data_t {
        double value; //数据的值
        int order;  //数据的输入顺序
        int rank;  //数据的排序编号
        char str[MAX_LEN]; //输入的字符串
} list[MAX_N];
```

```
……
int main() {
    int i, n;
    for (n = 0; scanf("%s", list[n].str) == 1; n++){
        list[n].value = atof(list[n].str);
        list[n].order = n + 1;
    }
    qsort(list, n, sizeof(struct data_t), s_value);
    gen_rank(n);
    qsort(list, n, sizeof(struct data_t), s_order);
    for (i = 0; i < n; i++) {
        printf("%d: %s\n", list[i].rank, list[i].str);
    }
    return 0;
}
```

存字符串

将字符串转成double

oj上使用atof, atoi等函数可能会报CE!!
该怎么办?

试试 sscanf, sprintf

```
// list[n].value = atof(list[n].str);
// 转浮点数，oj上可能不支持，就用下一条
  sscanf(list[n].str, "%lf", &list[n].value);
```

第1个qsort按value数据值大小排序

第2个qsort按order输入顺序大小排序

```
void qsort(void *base, size_t num, size_t wid,
     int (*comp)(const void *e1, const void *e2));
```

base: 是指向所要排序的数组的指针（void*指向任意类型的数组）；num: 是数组中元素的个数；wid: 是每个元素所占用的字节数；(*comp)：负责比较两个元素，返回负数、正数和0，分别表示第一个参数先于、后于和等于第二个参数。

结构应用举例

书上的这两个函数写法不完备，可能会报错！为什么？

函数接受的参数类型:无类型指针cont void *

```
int s_value(const void *p1, const void *p2) {
    double value1 = ((struct data_t *)p1)->value;
    double value2 = ((struct data_t *)p2)->value;
    if (value1 > value2) return 1;
    else if (value1 < value2) return -1;
    else return 0;
}

int s_order(const void *p1, const void *p2) {
    int order1 = ((struct data_t *)p1)->order;
    int order2 = ((struct data_t *)p2)->order;
    if (order1 > order2) return 1;
    else if (order1 < order2) return -1;
    else  return 0;
}
```

强制类型转换:
从结构体中取出内容

```
void qsort(void *base, size_t num, size_t wid,
    int (*comp)(const void *e1, const void *e2));
```

base: 是指向所要排序的数组的指针（void*指向任意类型的数组）；num: 是数组中元素的个数；wid: 是每个元素所占用的字节数；(*comp)：负责比较两个元素，返回负数、正数和0，分别表示第一个参数先于、后于和等于第二个参数。

不要害怕函数指针！

模式非常统一！！

用一用就能掌握！！

结构应用举例

按数值value大小排序后需**编号**，
该编号功能在gen_rank函数中实现

```
void gen_rank(int n)
{
    int i;
    list[0].rank = 1;
    for (i = 1; i < n; i++)
        if (list[i].value == list[i-1].value)
            list[i].rank = list[i-1].rank;
        else
            list[i].rank = list[i-1].rank + 1;
}
```

结构数组list[]

value 数值	Order 输入顺序	rank 数据编号	str 字符串
3.65	3	1	"3.65"
4.7	2	2	"4.7"
5.3	1	3	"5.3"
12.345	4	4	"12.345"
6e2	5	5	"6e2"

若相等，连续排；
若不等，则加一。

包含结构的结构

- 在一个结构中也可以包含其他已定义的结构或者结构指针。
- 当一个结构成员本身也是一个结构时，对它的成员的访问方式与一般结构相同
- 对结构成员进行访问的操作符 . 是左结合的，因此对嵌套的结构成员进行访问时不需要使用括号。

结构变量名.成员名.子成员名……最低级子成员名

```
struct  student{
     int  num;      char name[20];

     struct  date{
          int m, d, y;
     } birthday;
 } stu1, stu2, *pstu = &stu1;
```

stu1.birthday.m = 4;

pstu->birthday.y = 2001;

stu2 = stu1;

注意

如果成员本身又属一个结构体类型，则要用若干个成员运算符，一级一级地找到最低的一级的成员。只能对最低级的成员进行赋值或存取以及运算。

访问嵌套的结构体成员

- 结构体成员运算符 . 是左结合的，因此对嵌套的结构成员进行访问时不需要使用小括号

【例9-6】一般使用一个矩形左上角和右下角的坐标来定义一个矩形。假设我们已定义了结构pt_2d，则可以定义结构类型rect_t 以及相应的变量如下：

```
struct  pt_2d
{
    int x, y;
};
```

```
struct rect_t
{
    struct pt_2d  top_left, bottom_right;
}
rect1= {{5, 6}, {28, 39}},   rect2={{33, 55}};
```

其中变量rect1的两个成员都有指定的初始值，而变量rect2只有top_left有指定的初始值，bottom_right的两个成员则被初始化为0。
右边的语句使**rect2所表示的矩形的左上角位置不变，宽度和高度与rect1相同。**

```
rect2.bottom_right.x = rect2.top_left.x + (rect1
.bottom_right.x - rect1.top_left.x);

rect2.bottom_right.y = rect2.top_left.y + (rect1
.bottom_right.y-rect1.top_left.y);
```

基于结构体指针构造复杂的数据结构

具有结构指针成员的结构经常用于共享数据和构造复杂的数据结构

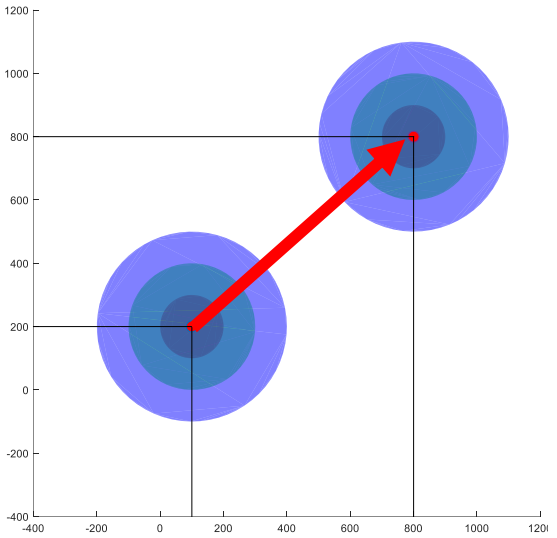
【例9-7】同心圆 定义一组半径分别为100, 200, 300的同心圆

一个圆可以由圆心的位置和圆的半径定义。一组同心圆的圆心是相同的。为各个圆共享同一个圆心位置，我们可以把圆心定义为指向一个二维平面上的点的指针。定义圆的结构如下

```
struct circle_t{
    struct  pt_2d *center;
    int radius;
};

struct pt_2d pt1 = {100, 200};
struct circle_t c1= {&pt1, 100},
    c2= {&pt1, 200},
    c3= {&pt1, 300};

c1.center->x = 800;
cl.center->y = 800;
```

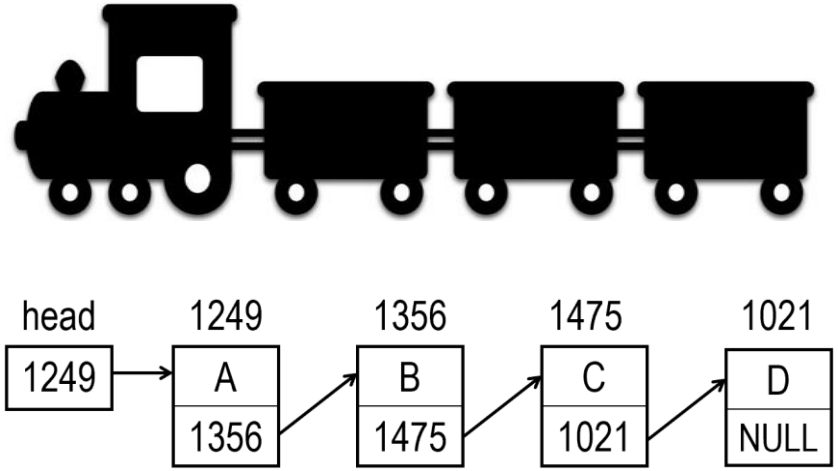


- 在定义了圆心点之后，就可以定义相同的同心圆；
- c1, c2, c3共享**pt1**作为圆心。当我们需要改变同心圆的圆心时，只需要对其中一个圆的圆心进行修改即可；
- 直接修改**pt1**的值也是同样的效果。

9.2 链表

为什么需要链表

- 数组：静态分配存储单元，必须整块内存，容易造成内存浪费。
- 链表：是重要的数据结构，它根据需要动态地分配内存，能有效利用零散内存。
- 链表表示树等数据结构非常有用。



- 链表是一种通过指针将由一系列类型相同的节点（结点）链接在一起的数据结构。
- 结构不能嵌套定义（不可以包含其自身类型的结构），但可以包含指向自身类型结构的指针。这个结构可用来描述递归定义的数据，如链表等。

```
struct  node_t
{
    int value;
    struct node_t *next;
};
```

用指针建立链表

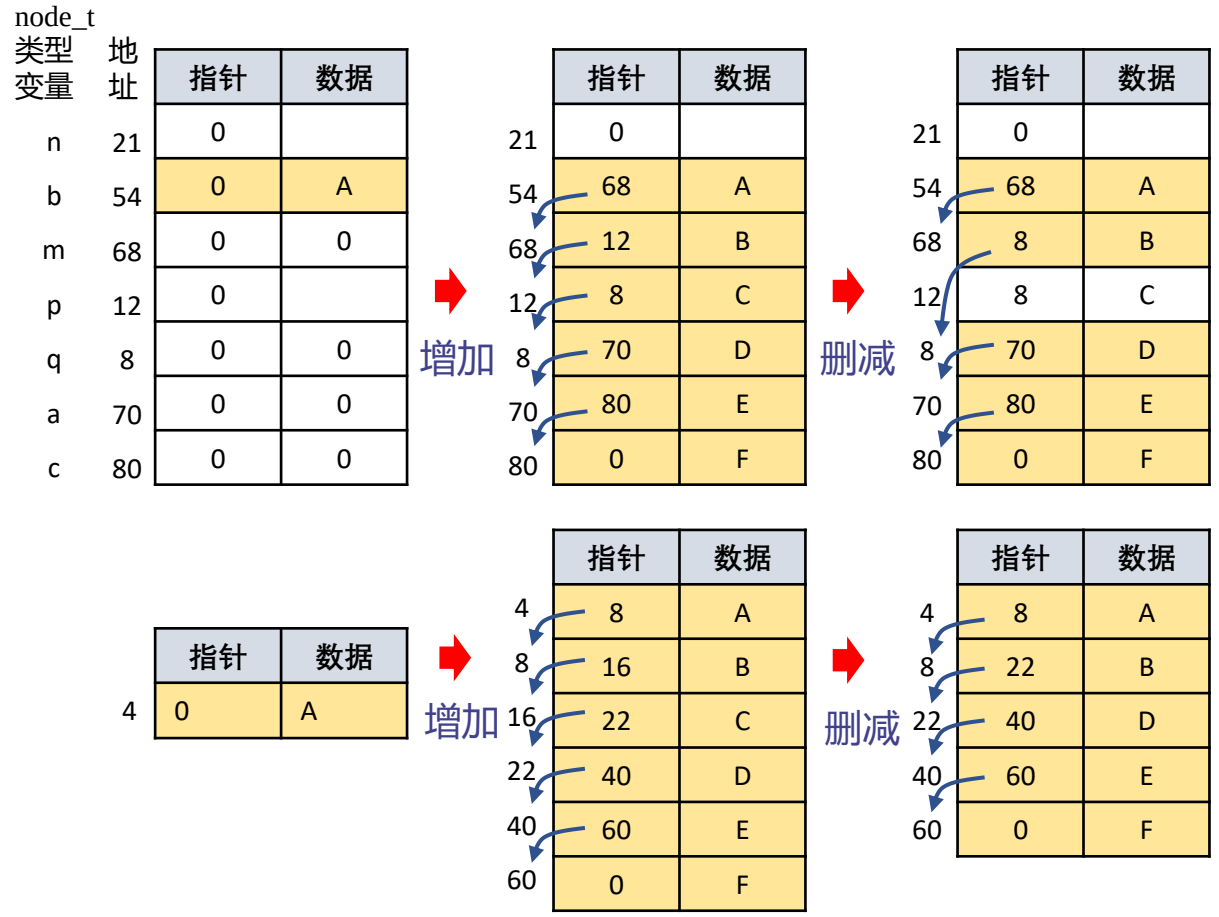
- 链表：数据部分（可有若干项） + 指向跟母结构相同的结构的指针变量（下一节点地址）；

- 建立静态链表是指所有节点都是在程序中定义的，不是临时开辟的，也不能用完后释放。

- 建立动态链表是指在程序执行中从无到有地建立起一个链表，即一个一个地开辟节点和输入个节点数据，并建立起前后相链的关系。

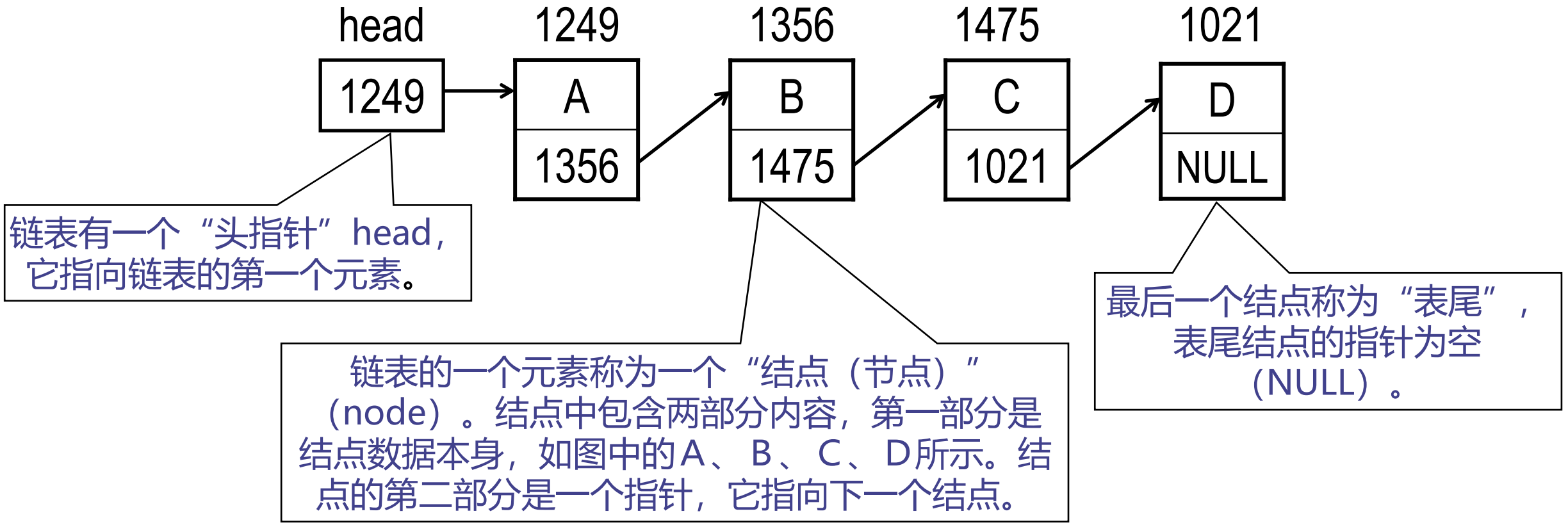
```
struct  node_t
{
    int value;
    struct node_t *next;
};
```

struct node_t
是一个只有一个int型数据域的单向链表中节点的结构



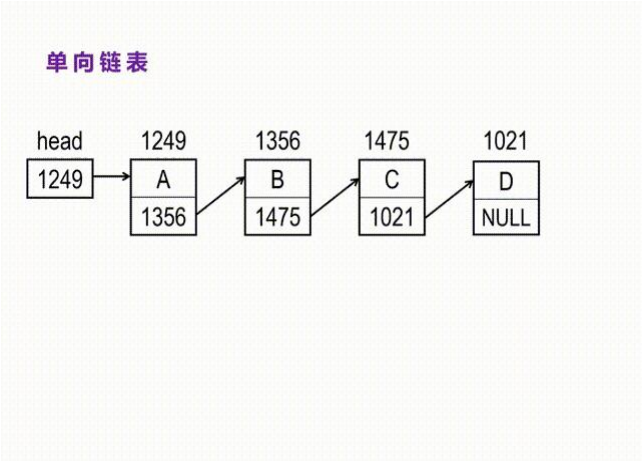
单向链表

head指向第1个元素，第1个元素又指向第2个元素……直到最后一个元素，该元素不再指向其他元素，链表到此结束。

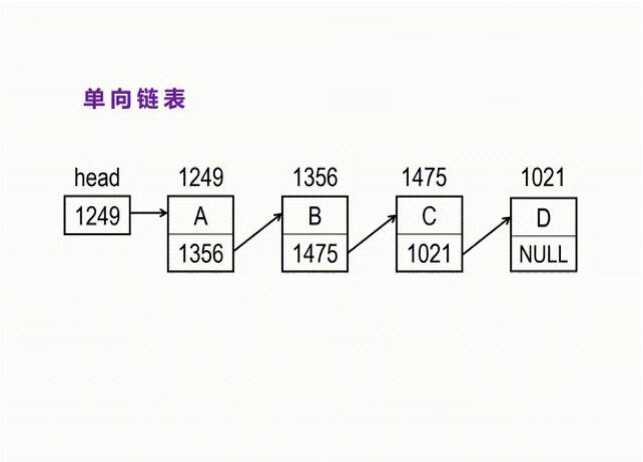


单向链表

- 在链表中插入一个结点，比如，在结点A后插入结点P，只需使P指向B，使A指向P。



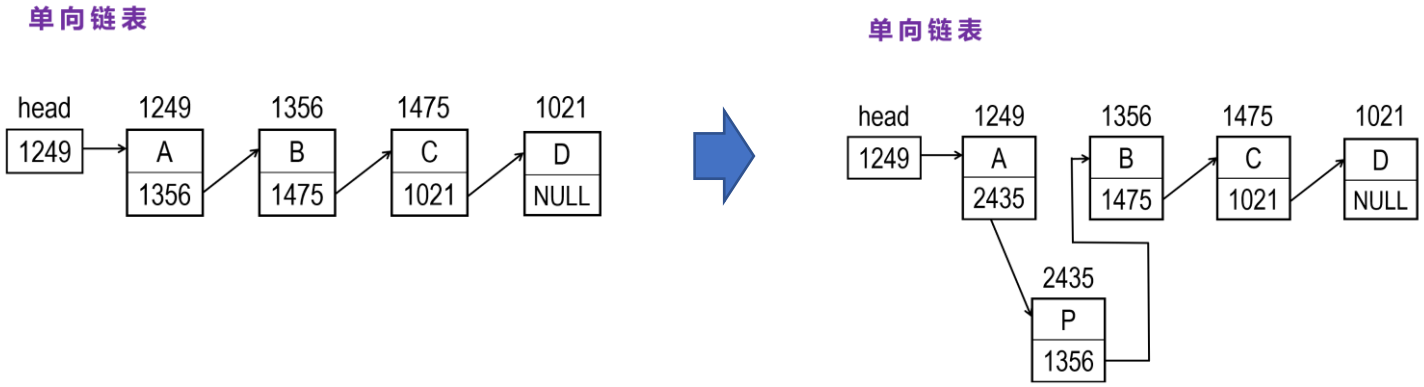
- 在链表中删除一个结点，比如删除结点C，只需使B指向D，并删除结点C所占内存。



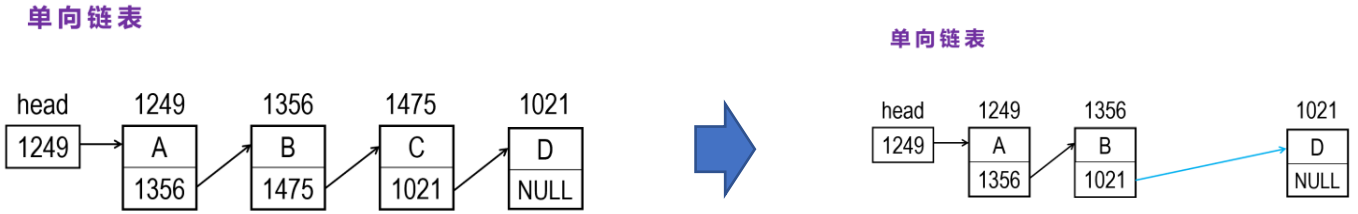
- 因此，链表的插入、删除非常方便。

单向链表

- 在链表中插入一个结点，比如，在结点A后插入结点P，只需使P指向B，使A指向P。



- 在链表中删除一个结点，比如删除结点C，只需使B指向D，并删除结点C所占内存。



- 因此，链表的插入、删除非常方便。

对比数组中要插入、删除数据，有什么优点?

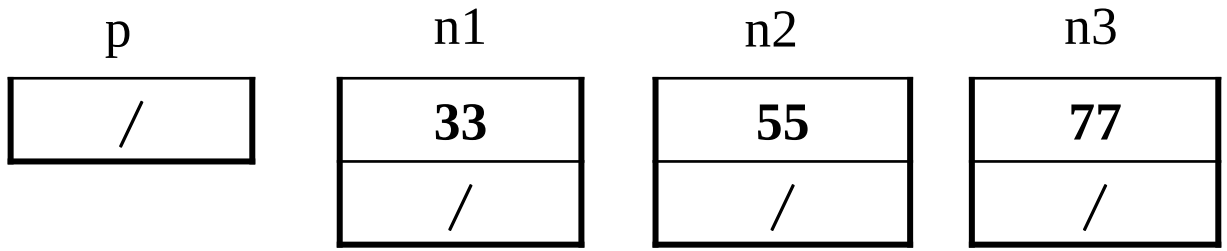
9.2.1 静态单向链表

用函数建立静态链表

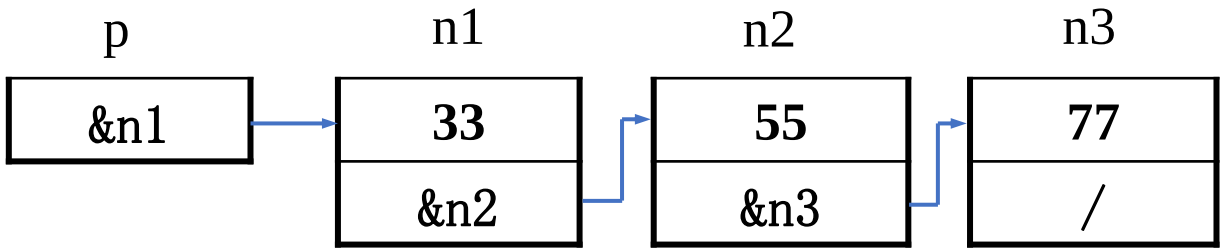
【例9-8】用函数建立单向链表

下图是某个单向链表的示意图。链表中的每个节点不但需要保存该节点的数据，而且需要保存后继节点的地址，因此使用结构来实现。

struct node_t 是一个只有一个int型数据域的单向链表节点的结构



各节点未链接之前的状态



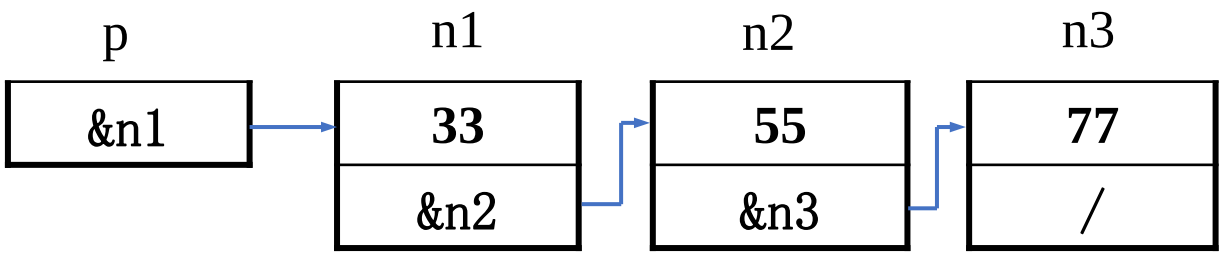
各节点链接成链表之后的状态

```
struct node_t{
    int value;
    struct node_t *next;
};
struct node_t *p,n1 ={33},n2={55},n3={77};

p = &n1,
n1.next = &n2,
n2.next = &n3;
```

用函数建立静态链表

- 右侧的代码把三个独立的节点链接成一个如前图所示的链表，其中p是指向链表的指针；n1是链表的首节点；n3是链表的尾节点，其next的值是NULL
- 这个结构中的成员value用来保存int型的数据，next是一个指向同类节点的指针，用来保存后继节点的地址。
- 程序中通常以指向链表节点的指针表示一个链表，当链表为空时以NULL表示。
- struct node_t 结构的定义说明，我们也可以把一个单向链表看成由一个节点及其后继构成，而其后继也是一个单向链表。这样，单向链表就是一个递归定义的结构。



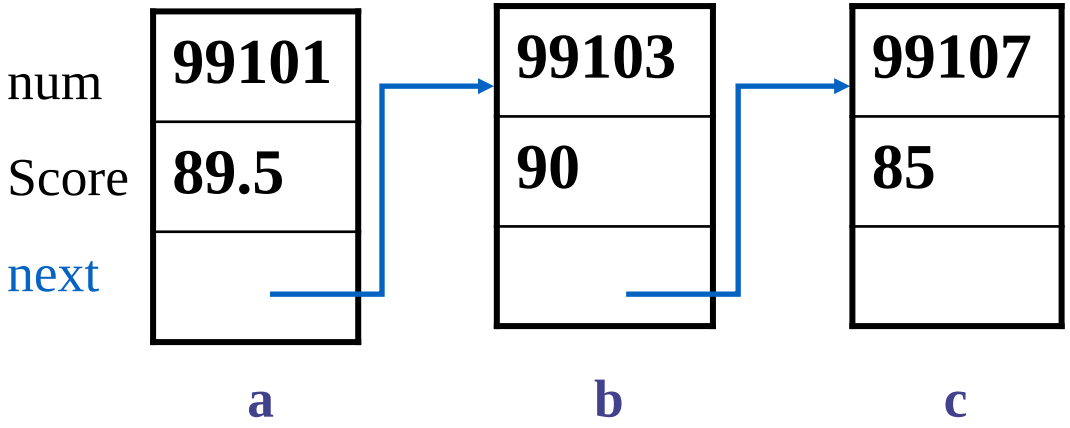

```
struct node_t{
    int value;
    struct node_t *next;
};
struct node_t *p,n1 ={33},n2={55},n3={77};

p = &n1,
n1.next = &n2,
n2.next = &n3;
```

建立静态链表实例

【例9-9】建立和输出一个简单静态链表

- next是struct student类型的成员，它又指向struct student类型的数据。
- 换句话说：next存放下一个节点的地址
- 各节点在程序中定义，不是临时开辟的，始终占有内容不放——“静态链表”



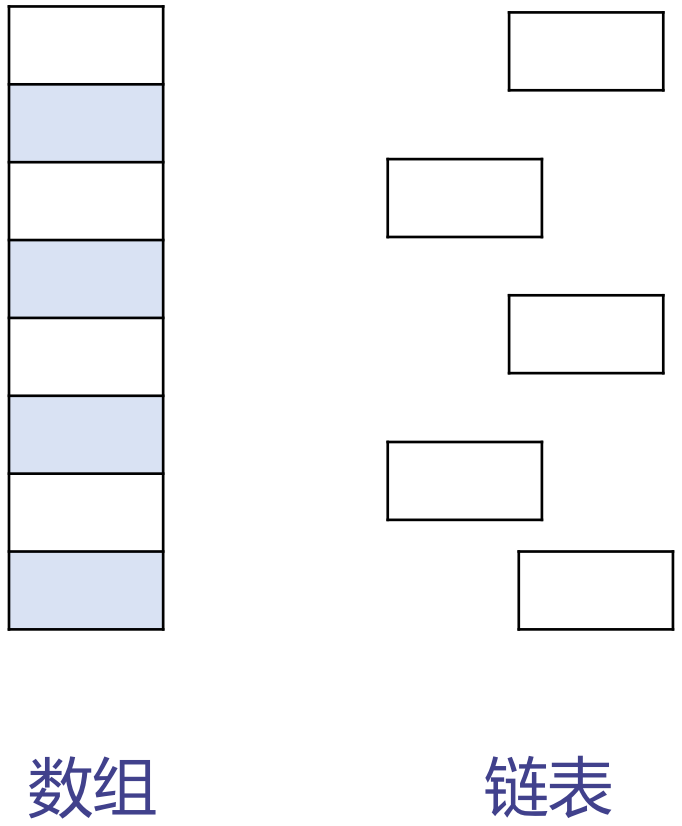

```c
#include <stdio.h>
struct student
{  long num;    float score;
    struct  student  *next;     };

int main()
{  struct 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;   a.next = &b;   b.next = &c;
    c.next=NULL;
    p=head;
    do
    {
       printf("%ld %5.1f\n",  p->num, p->score);
       p = p->next;
    } while(p != NULL);
     return 0;
}
```

9.2.2 动态单向链表

用函数建立动态链表*

- 链表更好的反映了数据和数据之间的关系
- 与数组相比，链表的长度不需事先确定；数据节点的插入和删除都很灵活
- 对于链表这样长度可变的动态数据结构，节点所需要的存储空间往往是通过动态内存分配的方式获得的



建立动态链表所需的函数*

- C 语言使用系统函数动态开辟和释放存储单元
- ANSI C 的三个函数(头文件 malloc.h)
  - void *malloc(unsigned int size)
  - void *calloc(unsigned n, unsigned size)
  - void free(void *p)

malloc 函数

函数原型：void *malloc(unsigned int size);
作用：在内存的动态存储区中分配 一个长度为size的连续空间
返回值：是一个指向分配域起始地址的指针（基本类型void）
执行失败：返回NULL

建立动态链表所需的函数*

- C 语言使用系统函数动态开辟和释放存储单元
- ANSI C 的三个函数(头文件 malloc.h)
  - void *malloc(unsigned int size)
  - void *calloc(unsigned n, unsigned size)
  - void free(void *p)

calloc 函数

函数原型: void *calloc(unsigned n, unsigned size);

作用：在内存动态区中分配 n个 长度为size的连续空间

返回值：指向分配域起始地址的指针；执行失败，返回null

主要用途：为一维数组开辟动态存储空间，其中n 为数组元素个数，每个元素长度为 size

建立动态链表所需的函数*

- C 语言使用系统函数动态开辟和释放存储单元
- ANSI C 的三个函数(头文件 malloc.h)
  - void *malloc(unsigned int size)
  - void *calloc(unsigned n, unsigned size)
  - void free(void *p)

注意

动态分配的存储单元在用完后一定要释放，否则内存会因申请空间过多引起资源不足而出现故障。

free 函数

函数原型： void free(void *p);

作用：释放由 p 指向的内存区

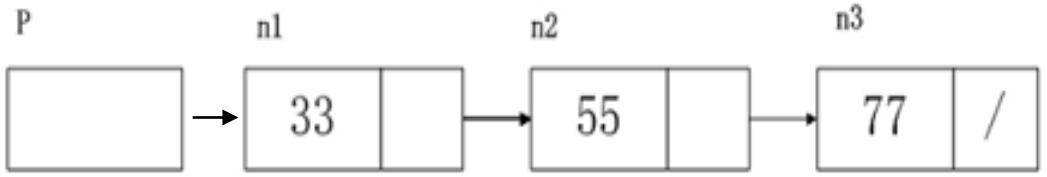
p 是最近一次调用 calloc 或 malloc 函数时返回的值

free 函数无返回值

用函数建立动态链表*

【例9-10】用函数建立动态链表

- 函数insert() 为新节点动态分配存储空间，并将其插入到链表头部。
- main()中的代码展示了如何使用该函数创建如图所示的链表，并按顺序将各个节点的值打印出来。
- 函数insert() 的参数list_p将指向待插入链表的头，value是新节点的值。insert()使用malloc()为新节点申请存储空间，将value保存到新节点中，将新节点的next指向原来链表的头，并返回新节点的地址。
- 函数main中的变量p必须被初始化为NULL，以保证创建的链表以NULL结尾。



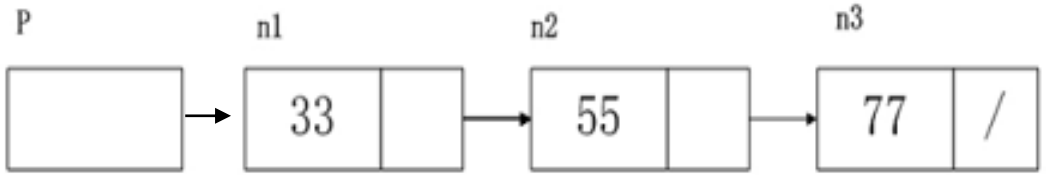

```c
#include <stdio.h>
struct node_t * insert(struct node_t * list_p, int value)
{
    struct node_t *tmp;
    tmp = malloc(sizeof(struct node_t));
    if(tmp == NULL) {  fprintf(stderr,"Out of memory\n");
        return list_p;}
    tmp->value = value;      tmp->next = list_p;
    return tmp;
}
void freelist(struct node_t * hd) {
    while(hd) {
          struct node_t *tmp = hd;   hd = hd->next;
          free(tmp);      }
}
int main() {
    struct node_t *p = NULL, *p1;
    p = insert(p, 77);   p = insert(p, 55);
    p = insert(p, 33);   p1 = p;
    for(; p != NULL; p = p->next)
       printf("%d ", p->value);
    putchar('\n');
    freelist(p1);
    return 0;
}
```

用函数建立动态链表*

【例9-10】用函数建立动态链表

- main()中的代码展示了如何使用该函数创建如图所示的链表，并按顺序将各个节点的值打印出来。
- 依次输出链表各个节点的数据域。

```
int main() {
    struct node_t *p = NULL, *p1;
    p = insert(p, 77);  p = insert(p, 55);
    p = insert(p, 33);  p1 = p;
    for(; p != NULL; p = p->next)
        printf("%d ", p->value);
    putchar('\n');
    freelist(p1);
    return 0;
}
```

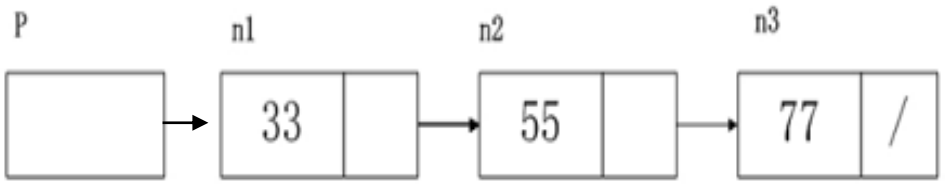


用函数建立动态链表*

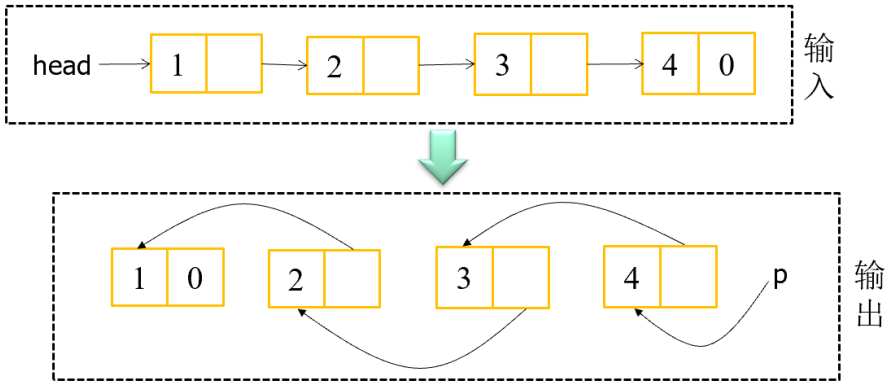
【例9-10】用函数建立动态链表

- 链表使用结束后，释放用malloc动态申请的内存。
- 使用临时指针tmp存放当前需要释放的指针。

```
void freelist(struct node_t * hd) {
    while(hd) {
        struct node_t *tmp = hd;
        hd = hd->next;
        free(tmp);
    }
}
```



使用递归函数简洁地处理链表



【例9-11】链表逆置函数

设计一个函数list_rev()，逆置用【例9-10】中的函数insert()生成的链表

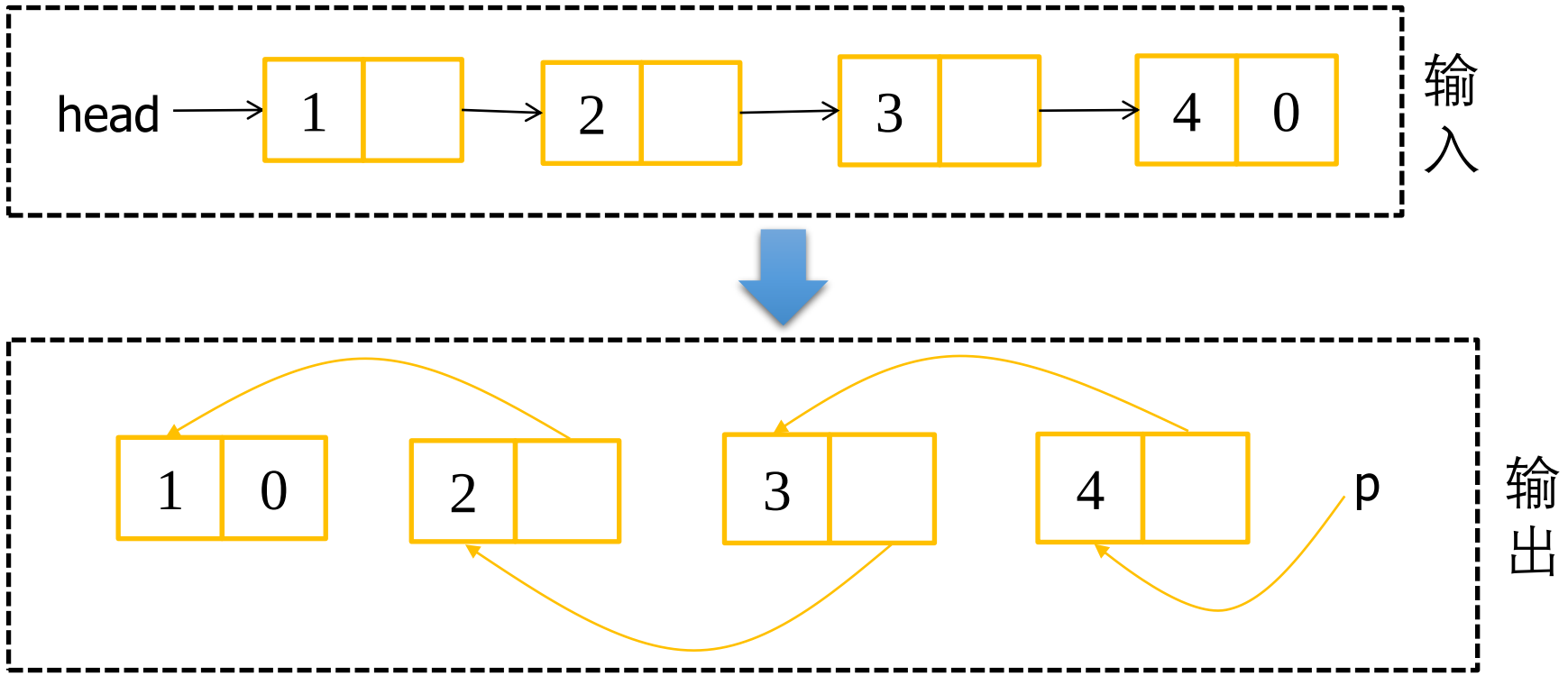
- 根据功能要求，函数以 struct node_t * 型参数的形式接受一个原始链表，将逆置后的链表以 struct node_t * 的值返回。
- 逆置可以分为三步：
  1. 逆置原始链表首节点的后继链表
  2. 将原节点接到逆置后的后继链表的末尾
  3. 将原节点的后继置为NULL
- 上述操作是递归的，其中第一步是递归调用，其返回值是原节点后继链表的逆置链表。递归过程的终止条件是链表为空，或者只有一个节点。

使用递归函数处理链表

- 确定递归函数接口功能

struct node_t * p = list_rev(struct node_t * head);

将head指向的链表逆序，并返回逆序后的链表头指针给p

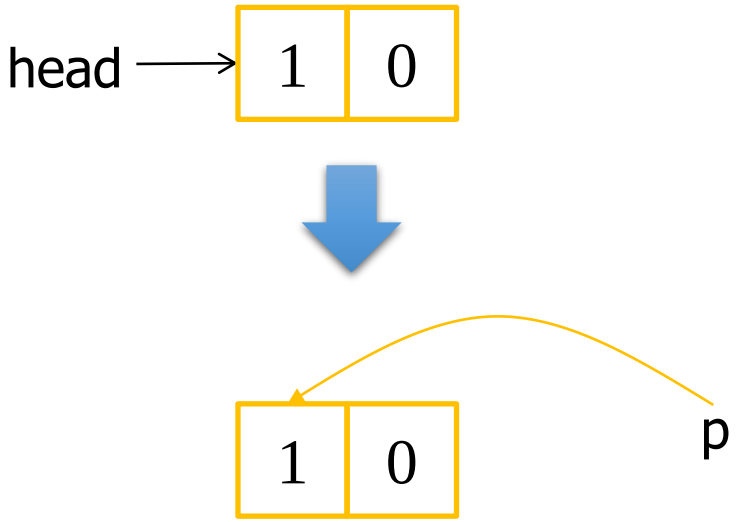


使用递归函数处理链表

- 确定递归函数的基本情况

```
struct node_t * p = list_rev(struct node_t * head);
```

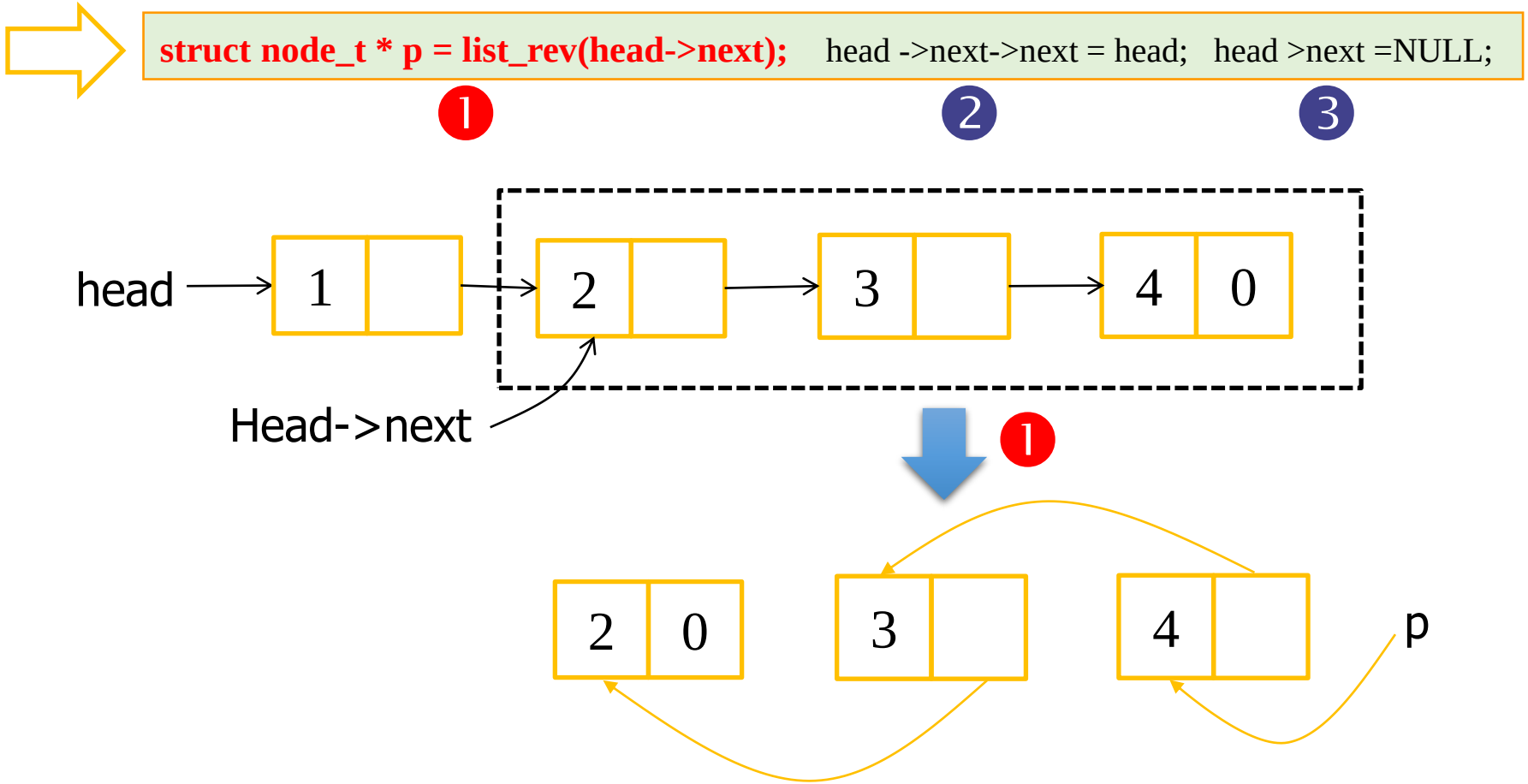
当head指向的链表只有一个节点或head本身为NULL时，直接返回head(单节点链表逆序就是它自己)



使用递归函数处理链表

- 确定递归函数的递归逻辑

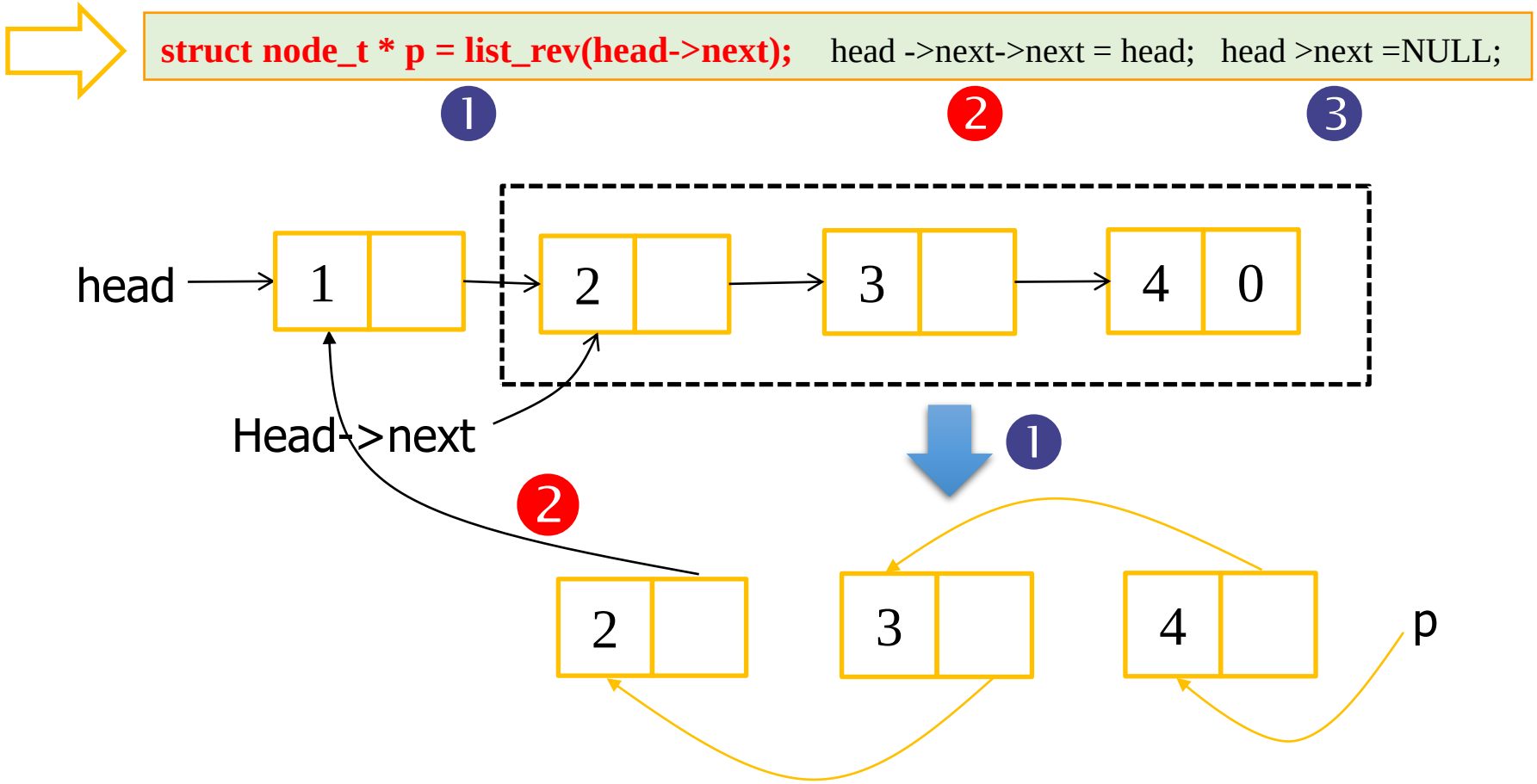
struct node_t * p = list_rev(struct node_t * head);



使用递归函数处理链表

- 确定递归函数的递归逻辑

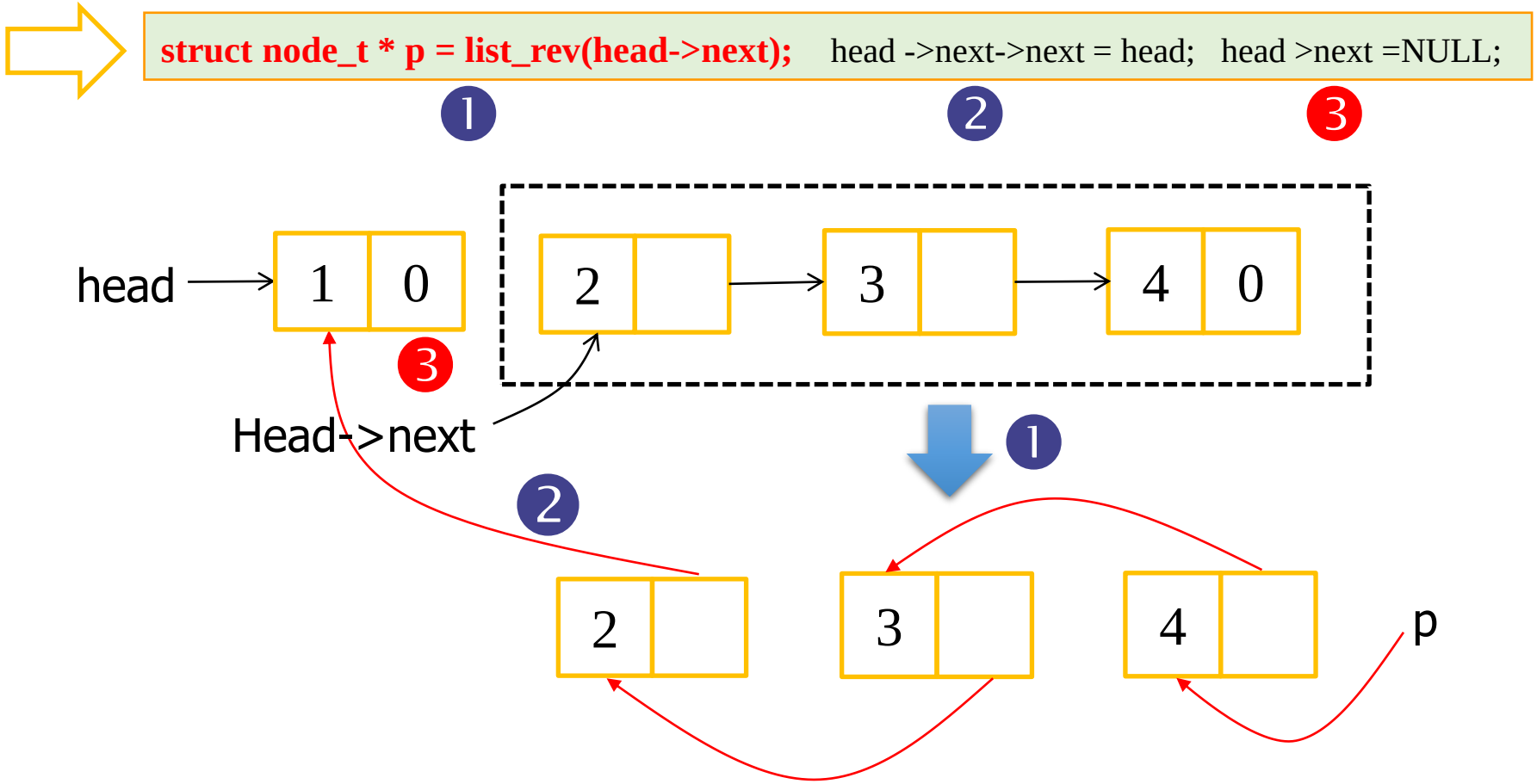
struct node_t * p = list_rev(struct node_t * head);



使用递归函数处理链表

- 确定递归函数的递归逻辑

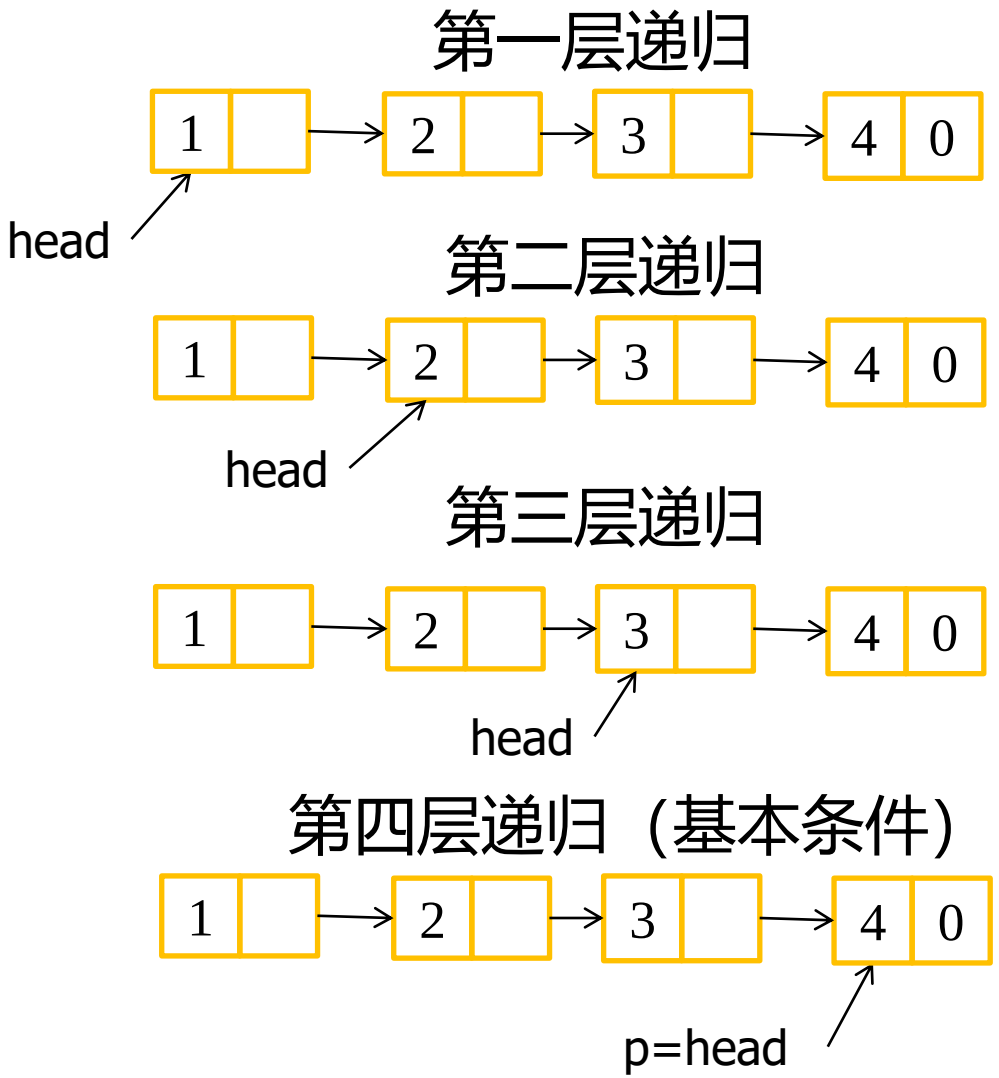
struct node_t * p = list_rev(struct node_t * head);



使用递归函数处理链表

- 根据以上分析写出递归函数

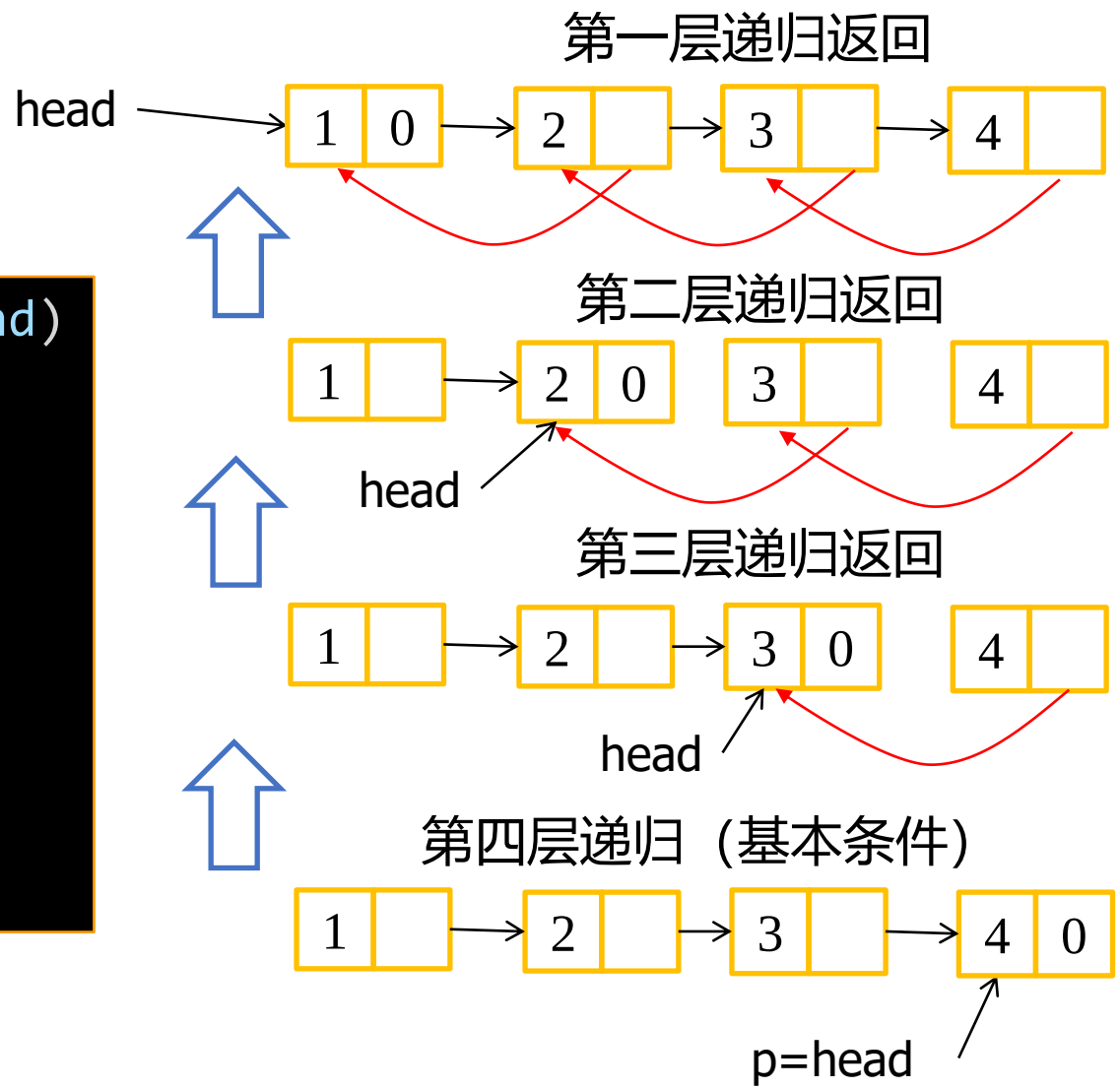
```
struct node_t *list_rev(struct node_t *hd)
{
    struct node_t *p;
    if(hd==NULL || hd->next == NULL)
        return hd;
    p=list_rev(hd->next );
    hd->next->next = hd;
    hd->next = NULL;
    return p;
}
```



使用递归函数处理链表

- 根据以上分析写出递归函数

```
struct node_t *list_rev(struct node_t *hd)
{
    struct node_t *p;
    if(hd==NULL || hd->next == NULL)
        return hd;
    p=list_rev(hd->next );
    hd->next->next = hd;
    hd->next = NULL;
    return p;
}
```

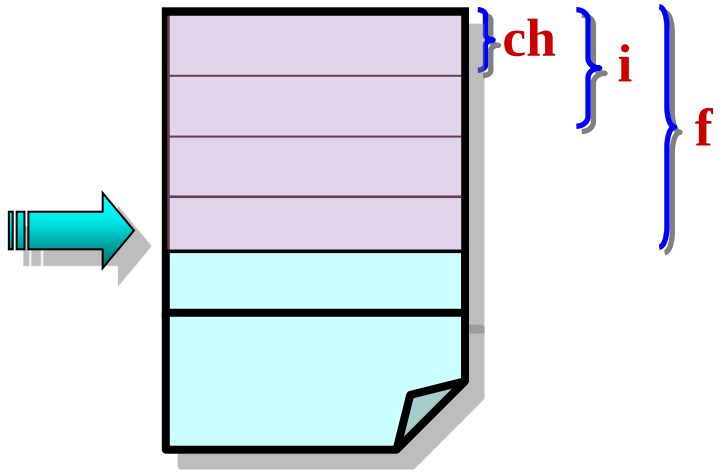


9.3 联合

9.3 联合(union)

- 联合类型在初级编程中较少使用。
- 在一些复杂的程序中，如编译系统中，联合是一种不可缺少的数据类型。
- 联合的作用是使一组类型不同的变量共享同一块存储空间。
- 联合使得一个变量可以根据需要存储不同类型的数据，或者可以对同一个数据按不同的类型进行解释。
- 定义和使用联合类型的语法以及相关的术语与结构类型相似，只是将关键字struct换成union。


```
union UData
{
    short i;
    char ch;
    float f;
};
```



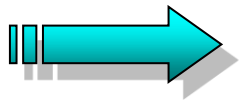
联合类型的定义

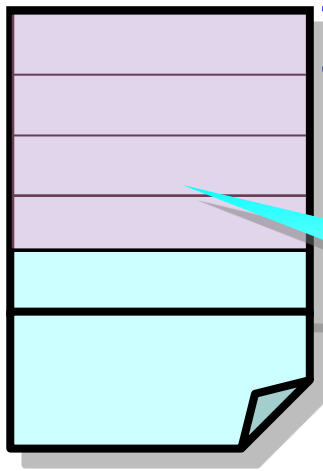
- 构造数据类型，也叫共用体或联合体
- 用途：使几个不同类型的变量共占一段内存(相互覆盖)

```
union [联合类型名]
{
   数据类型名1   成员名1;
   数据类型名2   成员名2;
   … …
   数据类型名n   成员名n;
};
```

类型定义不分配内存

```
union UData
{
    short i;
    char ch;
    float f;
};
```





联合的大小是成员中占内存最大的成员的大小

共占4字节

sizeof(union UData) vs sizeof(f)

注意

“共用体”与“结构体”的定义形式相似。但它们的含义是不同的。

结构体变量所占内存长度是各成员占的内存长度之和。每个成员分别占有其自己的内存单元。而共用体变量所占的内存长度等于最长的成员的长度。几个成员共用一个内存区。

联合成员的访问

联合类型变量名.成员名

联合类型指针名->成员名 或 (*联合类型指针名).成员名

```
union data
{
    int i;
    char ch;
    float f;
};
union data a, b, c, *p, d[3];
```

a.i　　a.ch　　a.f

p->i　　p->ch　　p->f

(*p).i　　(*p).ch　　(*p).f

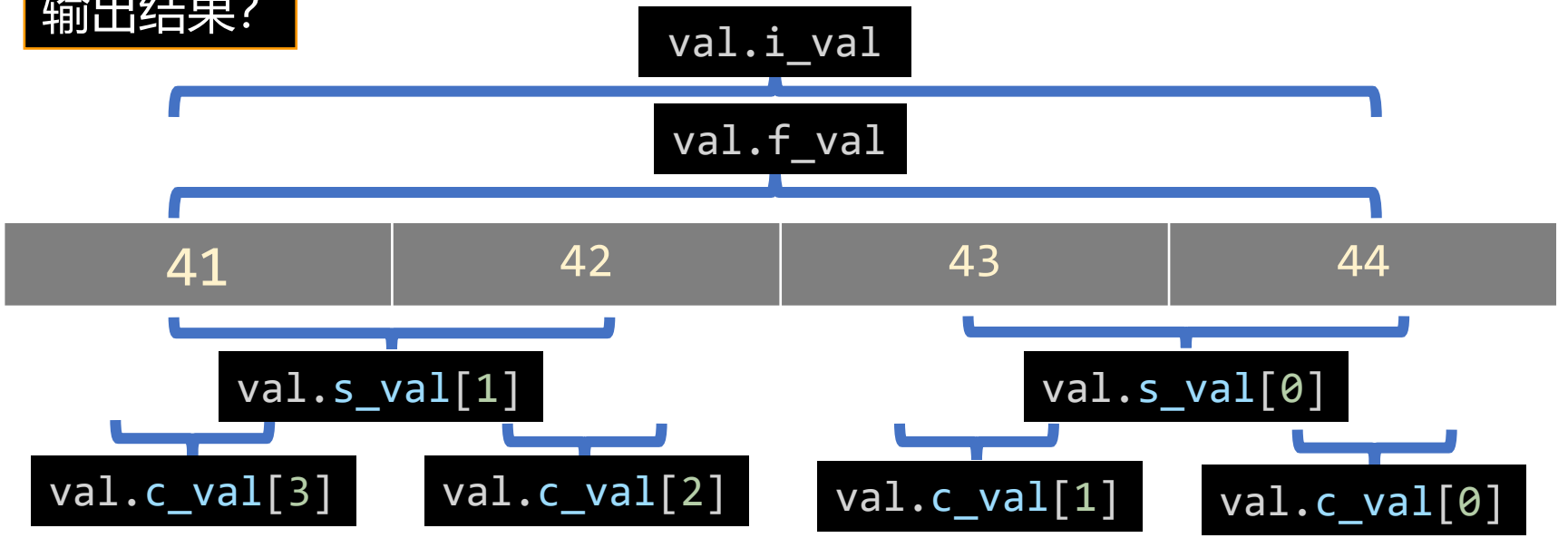
d[0].i　　d[0].ch　　d[0].f

联合成员的访问

【例9-12】联合成员的访问　通过成员变量改变对联合中内容的解释方式

```
union val_t {
    int i_val;
    float f_val;
    short s_val[2];
    char c_val[4];
 } val;
```

输出结果?




```
val.i_val = 0x41424344;
printf("i: %d(%x),%f\n", val.i_val, val.i_val, val.f_val);
printf("s[0]:%d(%x), s[1]:%d(%x)\n",
            val.s_val[0], val.s_val[0], val.s_val[1], val.s_val[1]);
printf("c[0]:%c(%x), c[1]:%c(%x), c[2]:%c(%x), c[3]:%c(%x)\n",
            val.c_val[0], val.c_val[0], val.c_val[1], val.c_val[1],
            val.c_val[2], val.c_val[2], val.c_val[3], val.c_val[3]);
```

联合成员的访问

【例9-12】联合成员的访问　通过成员变量改变对联合中内容的解释方式

输出结果?

```
union val_t {
    int i_val;
    float f_val;
    short s_val[2];
    char c_val[4];
 } val;
```

```
i: 1094861636(41424344),12.141422
s[0]:17220(4344), s[1]:16706(4142)
c[0]:D(44), c[1]:C(43), c[2]:B(42), c[3]:A(41)
```

所有变量的内容，即十六进制值是相同的，说明所有成员变量共享相同的内容，不同的是对这些内容的解释。

```
val.i_val = 0x41424344;
printf("i: %d(%x),%f\n", val.i_val, val.i_val, val.f_val);
printf("s[0]:%d(%x), s[1]:%d(%x)\n",
           val.s_val[0], val.s_val[0], val.s_val[1], val.s_val[1]);
printf("c[0]:%c(%x), c[1]:%c(%x), c[2]:%c(%x), c[3]:%c(%x)\n",
           val.c_val[0], val.c_val[0], val.c_val[1], val.c_val[1],
           val.c_val[2], val.c_val[2], val.c_val[3], val.c_val[3]);
```

联合应用举例*

【例9-13】有若干个人员的数据，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：姓名、号码、性别、职业、职务。要求用同一个表格来处理。

num	name	sex	job	class(班) position(职务)
101	Li	f	s	19374
102	Wang	m	t	prof

- 要求输入人员数据，然后输出。
- 分析：学生数据的class(班级)和教师数据的position(职务)类型不同，但在同一表格中，可使用“联合”数据结构。

联合应用举例*

【例9-13】有若干个人员的数据，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： 姓名、号码、性别、职业、职务。要求用同一个表格来处理。

（能根据前面的输入，决如何选定后面的输入，加强了程序的通用性）

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

num	name	sex	job	class(班) / position(职务)
101	Li	f	s	19374
102	Wang	m	t	prof

联合应用举例*

【例9-13】有若干个人员的数据，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：姓名、号码、性别、职业、职务。要求用同一个表格来处理。

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

```
#include <stdio.h>
int main()
{   int i;
    for(i=0;i<2;i++)
    {   printf("please enter the data of person:\n");
        scanf("%d %s %c %c",&person[i].num,person[i].name,
                            &person[i].sex,&person[i].job);
        if(person[i].job=='s')
            scanf("%d",&person[i].category.class); //如是学生，输入班级
        else if(person[i].job=='t')
            scanf("%s",person[i].category.position); //如是教师，输入职务
        else        printf("Input error!"); //如job不是's'和't'，显示“输入错误”
    }
    printf("\n");
    printf("No.namesex job class/position\n");
    for(i=0;i<2;i++)
    {   if (person[i].job=='s') //若是学生
            printf("%-6d%-10s%-4c%-4c%-10d\n",person[i].num,person[i].name,
                   person[i].sex,person[i].job,person[i].category.class);
        else          //若是教师
            printf("%-6d%-10s%-4c%-4c%-10s\n",person[i].num,person[i].name,
                   person[i].sex,person[i].job,person[i].category.position);
    }
    return 0;
}
```

9.4 类型定义语句

9.3 类型定义语句(typedef)

- 在阅读标准库函数联机手册和一些大型程序源代码时，常看到一些陌生的数据类型，如size_t , time_t, INT16, UINT32等。

- 其实这些数据类型很多都是已学过的基本数据类型，只是由程序员使用 typedef（type define) 定义了新的类型名称，以增加程序的可读性和可移植性。

- C语言中 typedef 不直接创建新的数据类型，而只是为已有的数据类型提供别名。如， size_t 常等价于unsigned int , INT16 常等价于signed short, UINT32常等价于unsigned long。

typedef语句格式

typedef <标识符说明>;

```
typedef  int    Lenth;
typedef  char *  String;

typedef  struct    pt_2d{
        int x, y;
} pt_2d;

typedef  union   u_t{
        char *  word;
        int   count;
        double value;
} u_t;
typedef  u_t *   u_ptr;
```

- typedef定义的类型名使用方式与任何类型完全相同
- typedef说明的新类型名称既可以是基本类型，也可以是构造类型或函数类型
- 有了pt_2d和u_t之类的类型定义，在使用这些类型时就不需要再写struct和union这样的关键字了，使得代码简洁易懂，因此在定义struct和union同时使用typedef定义新名称已成为一种惯例

本章小结

- 掌握结构类型与结构变量的定义方法
- 掌握结构成员的访问方法
- 掌握结构数组与结构指针的用法
- 了解联合的概念
- 理解链表的定义
- 静态链表的创建
- 动态链表的创建与逆置等相关操作
- 掌握typedef语句

课后作业

- 根据课件和教材内容复习
- 教材章节后相关习题
- 通过OJ平台做练习赛
- 预习
  - 使用文件的方法：打开、创建与关闭
  - 文件的二进制格式读写
- 上机实践题
  - 把本课件和书上讲到的所有例程输入计算机，运行并观察、分析与体会输出结果。
  - 编程练习课后习题内容。

课堂作业

1. 在纸上画几个有两三个节点的链表的例子，看看例9-6操作过程中的每一步对它们产生的影响。

2. 对单向链表的逆置也可以使用非递归的方法，程序如何写？比较一下两者的代码，测试并分析一下两者的优缺点。

```
struct node_t *list_rev(struct node_t *hd)
{
    struct node_t *p;
    if(hd==NULL || hd->next == NULL)
        return hd;
    p=list_rev(hd->next );
    hd->next->next = hd;
    hd->next = NULL;
    return p;
}
```