信息实际上不是零散的，而是有很强的相关性。例如，一个学生完整的信息包含这样几项：编号、姓名、性别、年级和得分。这几项数据之中，编号是一个关键数据。一旦编号确定，那么其他几项也就跟着确定了。这说明一个事实：用于描述学生信息的多项数据间是有紧密联系的。反观我们设计的方案，单个数据项是描述清楚了，但数据之间的联系是松散的，并且这种联系只存在于设计这个解决方案的程序员的逻辑中，而没有在语法形式上清晰地表达出来。

实际上，在现实世界中，像学生 Peter 这样的人或事物常被称为**“对象（*object*）”**。一个对象拥有一些描述其特性的数据，例如用于描述学生信息的 5 项数据。这些数据称为对象的**“属性（*attribute*）”**。一个对象往往具有多于一个的属性，并且这些属性间有着紧密的联系。例如，在描述轿车这样的事物时，可以使用如表 9-1 的属性：

表 9-1 轿车的主要属性

序号	属性
1	牌照号
2	品牌
3	颜色
4	型号
5	车主姓名

这些属性中，**“牌照号”**是一个关键属性，它决定了其它所有的属性。

拥有相同特性的对象肯定不止一个。但是，这些对象都有一个共同点，就是拥有相同的属性。不过，不同对象的相同属性却有着不同的属性值，这样就将不同的对象区分开来。例如，笔者和朋友的轿车分别拥有不同的属性值，如表 9-2 所示。

表 9-2 不同的对象拥有不同的属性值

属性	笔者的轿车	朋友的轿车
牌照号	川 A ZXXXX	川 A XXXX7
品牌	斯柯达	标致
颜色	银色	白色
型号	明锐 1.6	408 2.0
车主姓名	BBB	CCC

表 9-2 中，**“斯柯达”**和**“标致”**就是不同对象在相同属性**“品牌”**下的不同属性值。其余属性以此类推。

虽然不同的属性值可以区分出不同的对象，但是这些对象都具有相同或相似的行为。例如，轿车都会有如表 9-3 所示的最基本的行为：

表 9-3 轿车的基本行为

序号	行为
1	启动
2	行驶
3	加速
4	换挡
5	制动

在上述关于轿车的例子中，读者可以看到，轿车不能只用单一的属性（值）来描述；或者反过来说，所有属性在一起才能完整描述轿车。这些属性之间有强烈的依赖关系，它们是一个不可分整体的组成部分。而所有的轿车都具有相同或相似的行为。这些都是对象的特征。

用计算机来仿真对象及其行为时，就可以利用对象的特性建立起计算机模型。一般地，模型中会用聚集在一起的数据来描述属性，用函数来描述行为。针对于参赛学生信息，以前建立的多数组方案就是一种模型。不过这种模型并不好，因为它只能区分出不同的对象，但不能清晰地描述对象属性间的联系。

要解决上述问题，必须在语言设计上加以支持。C 语言的结构体（*structure*）类型就是为此目的而设的。在本章中将会详细学习结构体的定义和使用，并结合函数、指针解决一些较为复

杂的问题。

9.2 结构体类型声明和使用

结构体是 C 语言提供的一种重要的数据类型，它拥有将属于某个实体的不同属性封装在一个语法成分内的能力。属性的封装在很大程度上描述了对象的特性，以及属性间的关系，从而使程序设计也变得清晰明了。

在下面的部分将学习如何声明和使用结构体。

9.2.1 结构体类型声明

可以这样描述结构体：它是一个或多个数据类型相同或不同的变量集合在一个名称下的用户自定义数据类型。

1．结构体类型声明的一般形式

C 语言严格规定了结构体类型的声明，以关键字 struct 开始，形式如下：

```
struct 结构标志名
{
    成员列表;
};//请读者注意这个分号的存在。漏写这个分号是初学者常犯的错误之一。
```

其中，成员列表非常像是一系列变量声明。例如：

```
struct student
{
    int  studentID;
    char name[10];
    enum GENDER gender;
};
```

在声明一个结构体类型的时候，需要注意以下事项。

（1）关键字 struct 是结构体类型的标识，用于区别其他类型。

（2）struct 关键字与结构标志名（本例中是 student）共同构成结构体类型的名称，也就是说，例中的结构体类型的名字是 struct student，与 int、char、float 具有同等地位。因为结构体类型名可能会被经常使用，所以许多程序员常使用 typedef 来为结构体类型取一个较短的别名。例如：

```
typedef struct student ST;
```

其中，名字 ST 就是类型 struct student 的别名，在使用时不需要在其前面加上 struct 关键字。

（3）{}内声明的变量称为结构体的**成员**（*member*），其类型可以是除了函数类型外的任意数据类型。函数指针类型因为不是函数类型，所以仍然可以作为结构体成员的类型。例如：

```
struct foo
{
     int f(); //error，结构体的成员不能是函数
     int (*p)(); //ok
}
```

结构体成员的声明方式与普通变量的定义方式相同。

（4）结构体类型声明是对对象属性的抽象，是一种描述，或者说是一个模板。编译器不会为这种声明分配存储空间。具体一点，就是结构体类型声明给出了该结构体有多少个成员，每个成员叫什么名字，分属于哪种数据类型，但并未实际占据相应大小的存储空间。

结构体类型声明可以在所有函数外部，称为**全局声明**，可以供所有函数使用。也可以声明在函数体或语句块内部，则该结构体声明被严格局限在包含它的函数（块）内部，只能在本函数（块）中使用，离开该函数（块）后，声明失效。这称为**局部声明**。例如有声明：

```
struct globally { }; //文件作用域中的类型声明

void foo()
{
     struct locally { }; //函数作用域中的类型声明
     …
```

```
}

void goo() {…}
```

那么结构体 golbally 可以被 foo()和 goo()函数使用，而结构体 locally 只能被 foo()使用，它对于 foo()之外的任何语法成分来说都是不可见的。既然是不可见的，当然就是无法使用的。一般来说，局部结构体声明是非常罕见的。

2．结构体可以是匿名的

在某些特殊的程序中，可能会声明一些匿名的结构体。匿名的意思是在声明时不给出结构体标志名。例如：

```
struct
{
    //some members
} s; //s 是该结构体的一个变量，详见 9.2.2
```

匿名结构体因其没有名字而无法使用其类型信息。所以，在声明匿名结构体的同时就必须定义其变量。

匿名结构体也是非常罕见的，所以不推荐读者使用这种声明方式。

3．结构体类型声明可以嵌套

前面讲到结构体成员变量可以是任意数据类型，那自然也包括结构体类型。这种结构体内部包含结构体的声明形式称为结构体类型的嵌套。

假设在 student 中增加一个成员 DOB（出生日期），它由年、月、日构成。据此，可以额外声明一个结构体来描述这种日期信息。例如：

```
struct birthday
{
    int year;
    int month;
    int day;
};

typedef struct birthday DOB;
struct student
{
    int  studentID;
    char name[10];
    enum GENDER gender;
     DOB    dob;
};
```

除了上述方式，可以将 birthday 结构体直接放在 student 内部，形成如下声明：

```
struct student
{
    int  studentID;
    char name[10];
    enum GENDER gender;
    struct birthday
     {
         int year;
         int month;
         int day;
     } dob;
};
```

以上两种声明是完全等效的。

> 在第二种声明中，虽然 birthday 结构体看起来是一个局部的声明，但是按照 C 语言的规定，这个声明并未被 student 局限在其内部，在 student 外部仍可以使用 birthday 结构体类型。也就是说，结构体 birthday 和结构体 student 有相同的作用域。

9.2.2 结构体变量声明

结构体类型是一种数据类型，和 int、char、float 等类型名具有同等的地位。不过，要在程序中使用该类型，必须定义该类型的变量。

C 语言中定义结构体变量有 2 种方式。

1．先声明结构体类型，再单独定义结构体变量

利用上一节声明的结构体类型 struct student，可以定义变量。例如：

```
struct student  stu1,stu2;
```

或者在使用 typedef 后，用如下方式定义变量：

```
ST stu1, stu2;
```

> 如果没有 typedef，直接使用结构体标志名作为类型名来声明变量是错误的。以下就是错误的变量声明：
>
> ```
> student stu; //error
> ```

一旦定义了变量 stu1、stu2，系统就为其分配了该类型大小的内存空间，此后就可以使用这些变量，并可以在其成员中存入数据。图 9-1 示意了结构体变量 stu1 的内存布局。可以看到，一个结构体变量是一个整体，占据了一定的内存；该内存块根据结构体声明的布局，被划分成若干区域，每个区域存储该结构体变量的一个成员。

> 一个结构体变量所占内存字节数可以粗略地认为是各成员变量的字节数之和。实际上，编译器为了方便存取，结构体变量的实际大小往往比这个和要大。因此，请程序员一定不要去手工统计一个结构体变量的大小，而使用运算符 sizeof 来计算。

2．在声明结构体类型的同时声明结构体变量。

例如：

```
struct student
{
    int   studentID;
    char  name[10];
    enum GENDER gender;
} stu1, stu2;
```

如果结构体类型声明是全局的，那么声明的两个变量也是全局变量。这并不是一种好的声明方式，因为编码原则中的一条就是尽量少用全局变量。

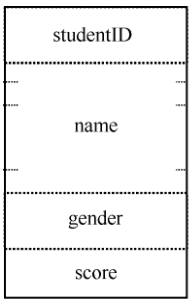

图 9-1　结构体变量 stu1 的内存布局

9.2.3 结构体变量的使用和初始化

结构体变量拥有多个成员。因此，在使用中常常是对单个成员进行操作。

1．访问结构体变量的成员

访问结构体变量的成员要使用成员选择运算符“.”，其语法为：

```
结构体变量名.成员名
```

例如：访问 stu1 中的 studentID，可以用这样的表达式：

```
stu1.studentID
```

虽然这是一个由两个名字构成的表达式，但却是一个整体，它和普通变量的使用完全一样，可以输入、输出、赋值、运算等。例如：

```
stu1.studentID=5;
scanf(“%d”,&stu1.studentID);
printf(“%d”,stu1.studentID+1);
```

在上面两个输入、输出函数中，由于“.”运算符具有很高的优先级，所以首先被计算，其次才会使用“&”或“+”运算符。

对于嵌套结构体的变量，可以使用级联的“.”运算符来访问其成员。比如要访问 stu1 的 dob

的 year 成员，可以用如下表示。

```
stu1.dob.year=1992;
```

在上述表达式中，由于“.”运算符的结合性是从左至右的，因此这个表达式可以理解为：(stu1.dob).year。其中，stu1.dob 是一个 birthday 类型的结构体变量，而 year 是该结构体变量的一个成员。

2．结构体变量的初始化

在定义结构体变量时可以对其进行初始化，语法为：

```
struct 标志名  变量名= { 初始化值列表 };
```

其中，初始化值列表是一个用逗号“,”隔开的值序列。该序列中的每个值在顺序、类型上与成员变量声明的顺序和类型完全一致。

【例 9-1】结构体变量的初始化。

```
//9-1.c
#include <stdio.h>

enum GENDER { Male='M', Female='F' };

struct student          //结构体类型声明一般采用全局声明
{
   int    studentID;
   char   name[10];
   enum GENDER gender;
   int    score;
};

int main()
{
   struct student stu1 = {1, "Peter", Male, 95};
   printf("学号：%d\n 姓名：%s\n 性别：%c\n 成绩：%d\n",
          stu1.studentID, stu1.name, stu1.gender, stu1.score);

   return 0;
}
```

程序的运行结果为：

```
学号：1
姓名：Peter
性别：M
成绩：95
```

对于嵌套的结构体，可以用{}将属于内部结构体的初始化值列表括起来。下例示意了这种初始化方式：

```
struct student stu1 = {1, "Peter", Male, {1993,12,3}, 95};
```

C 语言允许只初始化结构体变量中的部分成员，但有一个要求是：未被初始化的成员只能是在成员列表中靠后的那些，并且这些成员的值被自动初始化为 0。不允许在两个已被初始化的成员之间有未初始化的成员出现。例如：

```
struct student stu1 = {1, "Peter"}; //ok
struct student stu2 = {2, "Linda", ,  95}; //error
```

> C99 标准提出一种更好的初始化方式，就是可以初始化指定成员而不管其顺序如何，其语法如下例所示：
>
> ```
> struct student stu2 = { .studentID = 2, .gender = Female };
> ```
>
> 其中，类似于.studentID 的语法成分称为“指定子(designator)”。

3．结构体变量的赋值

两个同类型的结构体变量之间可以直接互相赋值。例如：

```
struct student stu1 = {1, "Peter", Male, 95},stu1Copy;
```

```
stu1Copy=stu1;
```

实际进行的操作是将 stu1 所占据的内存直接复制到 stu1Copy 的内存中。这样一来，stu1Copy 成为 stu1 的一个副本。不过，在程序中出现的更多情况是给结构体变量的成员赋值。例如：

```
stu1Copy.studenID = stu1.studentID;
```

> 由于是内存复制，因此，如果在 student 结构体中含有数组成员（如 name），那么这个数组也会被复制。但要注意：两个数组之间是不能直接赋值的。所以，以下成员赋值是错误的：
>
> ```
> stu1Copy.name = stu1.name;
> ```

4．结构体变量的输入、输出

C 语言规定，不能将一个结构体变量作为一个整体进行输入、输出等操作，只能对其每一个成员分开进行操作。例如，对以下操作编译器都会报错：

```
printf("%d",stu1);  //error
scanf("%d%s%c%d",stu1); //error
```

正确的操作只能如【例 9-1】那样，逐一输入、输出结构体变量的成员。

9.2.4 何时使用结构体

当程序员分析出需要描述的对象拥有多个属性（分量）时，面临着两种选择：数组和结构体。的确，数组和结构体在某些方面是相似的：它们都是元素（成员）的集合。但二者也有很大的不同：数组的所有元素都属于相同的类型；而结构体的成员可能分属于不同的类型。

基于此，如果对象的属性都具有相同的类型时，选择数组或结构体来描述对象都可行。数组的优势是能够与循环结构结合采用相同的一段代码对属性进行处理，但其劣势是由于元素采用统一命名（加索引）的方式来访问，因此对属性的描述不够清晰，需要程序员记住每个元素代表哪个属性，这对他人阅读程序增加了难度。结构体的成员不能使用循环来处理，但单独命名的成员可以使程序更加清晰。例如，在描述三维空间点的坐标时，3 个坐标的类型相同，因此使用数组和结构体进行描述的方法都常被用到。

```
double point1[3]; //x=point1[0], y=point1[1], z=point1[2]
struct Point
{
    double x, y, z;
} point2;
```

采用以上哪种方式需要根据程序的需求来定。如果对象的属性较多，并且具有相同的处理模式，那么选择数组是很合适的。而在属性较少并且要求程序的可读性情况下，选择结构体更合适一些。

如果对象的多个属性具有不同的类型，那么只能选择结构体来描述，例如上面提到的学生信息。

9.3 结构体数组

一个结构体变量只能描述一个对象，而对于多个同类对象，就可以使用结构体数组。

结构体数组的定义方式与普通数组的定义一样。例如：

```
struct student stu[30];
```

在上述声明中，变量 stu 不是结构体变量，而是一个一维数组，其长度为 30，每个数组元素都是一个结构体变量。

结构体数组与普通数组一样，面临初始化和使用的问题。

1．结构体数组的初始化

```
struct student stu[30]={{1, "Peter", Male,85},
```

```
                                {2,”Linda”, Female, 98},
                                {3, “Ken”, Male, 96},
                                };
```

在上述初始化过程中，如果值列表的长度小于数组的长度，那么数组中剩下的那些未指定初始值的元素将被编译器按照结构体变量的初始化原则自动初始化。

2．结构体数组变量的使用

我们都知道，可以使用下标来索引数组元素。例如，要访问 stu 的第 i 个元素，那么语法为：

```
stu[i]
```

根据定义，这个元素实际上是一个结构体变量，因此对它的成员访问方式为：

```
printf(“%d”,stu[i].studentID);
```

在上述表达中，**“[]”**和**“.”**运算符具有相同的优先级，而它们的结合性是从左至右的，因此变量 stu 最先和**“[]”**运算符结合，得到一个结构体变量，然后再用**“.”**运算符获得它的一个成员变量。

表达式

```
stu.studentID[i]
```

是错误的。首先，**“.”**运算符先与 stu 结合，这要求 stu 是个结构体变量，但本例中它并不是，因此是错误的；其次，**“[]”**运算符与 studentID 结合，这要求 studentID 是个数组，而这在本例中仍是一个错误。

【例 9-2】初始化 5 个学生信息，并计算出他们的平均成绩，并输出不及格学生的姓名。

```
//9-2.c
#include <stdio.h>

struct student
{
     int    studentID;
     char   name[10];
     char   gender;
     float  score;
};

int main()
{
   struct student stu[5]={ {1,"李强",'M',85.5},
           {2,"张燕",'F',96},
           {3,"罗云",'F',75},
           {4,"马良",'M',47.5},
           {5,"王世强",'M', 56}};
   float avg, sum = 0;
   int i;

   printf("不及格名单如下：\n");
   for (i =0; i <5; ++i)
   {
          sum += stu[i].score;
          if (stu[i].score < 60)
             printf("%d %s\n", stu[i].studentID, stu[i].name);
   }
   avg = sum / 5;
   printf("5 个学生的平均成绩=%.2f\n",avg);


   return 0;
}
```

程序的运行结果为：

```
不及格名单如下:
4 马良
5 王世强
5 个学生的平均成绩=72.00
```

9.4 结构体与指针

在很多应用当中，结构体和指针是紧密结合在一起使用的。这里讨论一下二者有哪些相结合的方式。

9.4.1 指针变量作为结构体的成员

与其他类型一样，指针类型的变量可以成为某个结构体的成员。例如：

```
struct Memblock
{
    int  size;     //内存块的大小
    char *mem;   //在内存中的位置
};
```

在上述声明个，mem 就是结构体 struct Memblock 的一个指针成员。

一般情况下，都是动态为指针成员分配内存，或者是通过赋值的方式让它指向一个已经存在的内存块。例如：

```
struct Memblock block;
block.size = 1024;
block.mem = (char *)malloc(block.size);
```

当 block 不再使用后，可以通过显式的方式来释放这块内存：

```
free(block.mem);
```

在实际的应用当中，常会看到这样的结构体声明：

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

读者可能会感到奇怪：难道一个结构体可以用自己定义自己（即一种递归定义）吗？要知道，递归定义的类型是一种未完成类型，是不能通过编译的。

实际上，这不是一种递归定义，而以下代码中才存在递归定义：

```
struct _node_r
{
    int data;
    struct _node_r next; //error, recursive definition
};
```

编译器确定一个类型是否已定义完成的一个重要的指标就是该类型（的变量）的大小是否能被确定。前面已经提到过，一个结构体（变量）的大小是根据它的所有成员的大小来确定的。据此来观察_node_r 结构体，可以发现，其成员 next 的类型依赖于_node_r 类型本身，而_node_r 又依赖于成员 next，这不可避免地造成了递归定义，因此编译器无法确定_node_r 结构体的大小，而认为这是一种错误的未完成类型定义。

但在_node 结构体的定义中，next 成员的类型不是结构体，而是一个指针。指针变量的大小是可以确定的（一个指针变量的大小与一个整型变量的大小相同）。因此，_node 结构体的定义是完全合法的。

9.4.2 指向结构体变量的指针

结构体类型和其他类型一样，可以定义变量，可以定义数组，当然也可以定义一个指针用于

指向一个结构体变量或数组的指针，用来存放一个结构体变量所占的内存段的起始地址。

1．指向结构体变量的指针的定义

如有结构体及其变量声明：

```
struct student stu={1,“Peter”,‘M’,85.5};
```

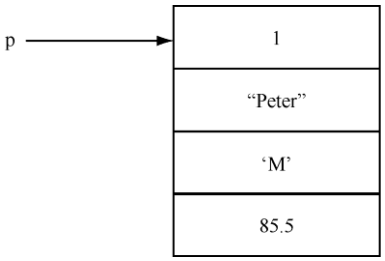

图 9-2　指针 p 指向了结构体变量

那么定义指向结构体变量 stu 的指针为：

```
struct student *p;
p=&stu;
```

可以说，指针 p 指向了结构体变量 stu，如图 9-2 所示。

正如读者看到的那样，指针 p 也恰恰指向了 stu 的第一个成员 studentID。这种现象比较容易解释，因为 studentID 的地址和 stu 的地址在数值上是一样的。但即使如此，也不能说指针 p 指向了成员 studentID，因为它们的基类型不同。

2．使用指向结构体变量的指针访问结构体的成员

通过指针变量访问结构体成员仍然需要使用成员选择运算符。例如，访问 name 成员，读者可能会想到这种方式：

```
p.name
```

但是，这是不正确的语法，因为“.”运算符要求其左边的变量一定是结构体变量。那么，下面的访问方式又如何呢？

```
*p.name
```

由于“.”运算符具有最高的优先级，因此上述表达式被解析为：

```
*(p.name)
```

可以看到，这仍然不正确。而正确的访问方式是：

```
(*p).name
```

通过()使“*”运算符首先与 p 结合，从而产生正确的表达。

读者可能已经想到，上述的表达方式稍嫌麻烦，特别是在多次使用这种方式的情况下。因此，C 语言提供了一种简化的方式，就是使用另一个成员选择运算符：->。表达式(*p).name 可以简化为：

```
p->name
```

运算符->要求其左边必须是个结构体变量的指针，其右边必须是该结构体的成员，否则将会出错。

> 运算符->由减号“-”和大于号“>”组成，两个符号间不能有空格。

如果结构体中有其他嵌套的结构体变量，例如：

```
struct Outer
{
    struct Inner { int x; } in;
    struct Inner *q;
} out, *p = &out;
```

那么通过指针 p 来访问 in 的 x 成员的方式为：

```
p->in.x
```

由于“->”和“.”运算符具有相同的优先级，且它们的结合性是从左向右的，因此该表达式被解析为：

```
(p->in).x
```

p->in 是一个结构体变量，因此访问它的成员 x 应该使用“.”运算符。

如果有如下赋值：

```
out.q = &out.in;
```

那么可以通过指针 p 和 q 来访问 x 成员：

```
p->q->x
```

上述表达式因被解析为(p->q)->x 而正确。

3．动态生成结构体变量

前面的例子中，结构体变量是在编写程序时定义的，那么这个变量在编译阶段就已经确定了，在程序运行时已经存在。但在有些特别的场合，需要在运行时动态地产生一些结构体变量。为达到这个目标，需要在程序中使用 malloc()和 free()函数。例如，对于_node 结构体，可以用如下的方法动态生成一个变量：

```
typedef struct _node NODE;
NODE *head;
head = (NODE *)malloc(sizeof(NODE));
```

此后可以往指针 head 指向的动态对象中存储数据。

```
head->data = 1;
head->next = NULL;
```

现在如法炮制，再生成两个动态对象：

```
NODE *p, *q;
p = (NODE *)malloc(sizeof(NODE));
p->data = 2;
p->next = NULL;
q = (NODE *)malloc(sizeof(NODE));
q->data = 3;
q->next = NULL;
```

接下来，将这 3 个对象通过它们的指针域 next 链接在一起：

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

图 9-3 示意了 3 个动态对象的链接情况。可以看到，它们形成了一种链式结构。

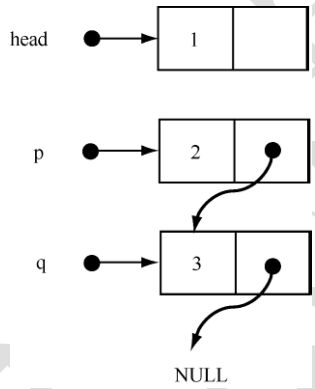

图 9-3　3 个结构体变量通过指针域链接

一旦链接完成，那么从 head 出发，沿着每个变量的 next 指针往下，可以将链中的所有变量访问一遍。

【例 9-3】运行时动态生成结构体变量。

```
//9-3.c
#include <stdio.h>
#include <malloc.h>
#include <string.h>

struct _node
{
    int data;
    struct _node* next;
};
typedef struct _node NODE;

int main()
{
    NODE *p, *q, *head;

    //first node
```

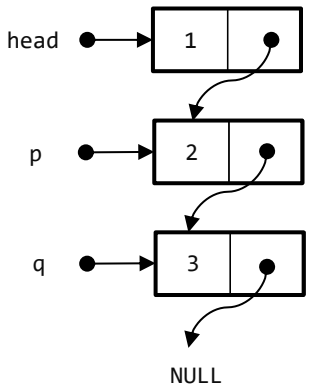

图 9-3 三个结构体变量通过指针域链接在一起

```
    head = (NODE *)malloc(sizeof(NODE));
    head->data = 1;
    head->next = NULL;

    //second node
    p = (NODE *)malloc(sizeof(NODE));
    p->data = 2;
    p->next = NULL;
    head->next = p;

    //third node
    q = (NODE *)malloc(sizeof(NODE));
    q->data = 3;
    q->next = NULL;
    p->next = q;

    p = head;
    while (p != NULL)
    {
        printf("%4d", p->data);
        p = p->next;
    }

    //destroy all nodes
    p = head;
    while (p != NULL)
    {
        q = p;
        p = p->next;
        free(q);
    }

    return 0;
}
```

程序的运行结果为：

```
   1   2   3
```

实际上，本例中描述的链式数据结构称为**“链表**（*linked list*）**”**，这是一种非常有用的存储结构。程序中的_node 结构体称为链表的**“节点”**，while 语句构成的循环称为**“**链表的**遍历”**。在《数据结构》这门课中将对链表进行详细描述，这里就不再深入讨论了。