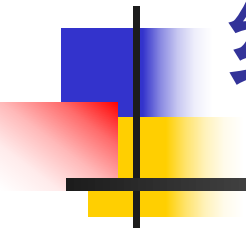
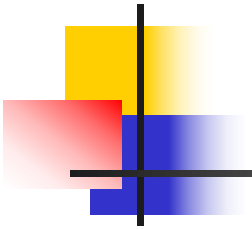


第12讲
结构与链表(Part I)

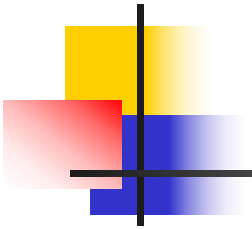
周水庚

2024年11月28日



提要

- 结构类型和结构变量
- 结构数组
- 结构形参和结构指针形参
- 链表及其应用
- 联合
- 位域
- 枚举
- 类型定义



提要

- **结构类型和结构变量**
- 结构数组
- 结构形参和结构指针形参
- 链表及其应用
- 联合
- 位域
- 枚举
- 类型定义

为什么要结构类型?

- 我们要处理一个班级的同学的信息，每个同学的信息包括:
  - 学号
  - 姓名
  - 性别
  - 年龄
  - 成绩
  - 家庭地址
- 怎么来组织、存储这些信息?

描述一个班级的同学的信息

- 用多个一维数组，假设最多100名学生
  - Unsgined stdno[100]; /* 学号 */
  - char *stdname[100]; /* 姓名 */
  - char stdgender[100]; /* 性别 */
  - int stdage[100]; /* 年龄 */
  - float stdscore[100]; /* 成绩 */
    - float stdscore[100][6]; /* 假设有多门课程 */
  - char *stdaddr[100]; /* 地址 */
- 缺点
  - 繁琐；丢掉了学号和其它信息的关联和单个同学信息的整体性，编程麻烦

描述一个班级的同学的信息

- 用一个二维数组?
- 不行! 学号、姓名、性别、年龄、成绩、地址等信息的类型不一致!
  - 数组中的元素要求具有相同的类型
  - 譬如: **int A[100], B[100][100]**等

为什么要结构类型?

- 结构类型
  - 一个数据对象常常包含多项不同类型的数据信息
    - 一个学生的数据实体可能要包含以下多项数据信息:*学号、姓名、性别、年龄、成绩、家庭地址*
  - 这类对象的数据因所包含的成分类型不同，不能用目前的单一数组来表示，也不便将它们的成分分拆成多个独立的单个数据项，因为这样会失去对象的整体性
  - *C*语言用结构数据类型描述由若干独立意义成分组成的数据对象

结构类型和结构变量

- 结构类型的定义
  - 定义一个结构类型的一般形式为

```
struct  结构类型名 {
  成分说明表
};
```

  - 其中，关键字“struct”引出结构类型的定义
  - struct之后任选的标识符是结构类型的名字
  - 用花括号括住的是该结构类型的成分说明表，指明组成此种结构类型全部成分

结构类型定义示例

例如，由日、月、年组成的日期结构类型为：

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

某厂职工信息管理系统中职工信息结构类型为：

```
struct person { /* 职工信息结构类型 */
  char  *name;      /* 姓名 */
  char  *address;  /* 地址 */
  float salary;      /* 基本工资 */
  float addition;   /* 附加工资 */
  float cost;        /* 房租水电费 */
  struct date hiredate;
   /*工作日期，说明结构类型可嵌套*/
};
```

一个结构类型中的某些成分可以是其它结构类型。这种嵌套不能包含自身，但允许在定义中引用自己，即可以有指针成分，该成分能指向当前正在定义的结构

结构类型定义示例（2)

学生信息结构类型为:

```
struct stdType { /* 学生信息结构类型 */
  unsgined  number;      /* 学号 */
  char  *name;       /* 姓名 */
  char  gender;       /* 性别 */
  int     age;        /* 年龄 */
  float  score ; /* 成绩 */
  char  *address;  /* 地址 */
};
```

结构变量

- 在结构类型定义中，详细列出了结构类型所包含的每个成分的名及其类型
- 结构类型定义只是表明一种数据类型，是定义一种数据结构的“模式”或“样板”，并不定义“实物”，不要求分配存储单元
- 程序要使用结构数据，必须定义结构变量
- 结构变量存在期间要占用存储单元。它的成分个数和各成分的类型与结构类型定义中的成分个数和各成分的类型相一致
- 具有结构类型的变量就是结构变量，结构变量也简称结构

结构变量的定义

- 定义结构变量有两种不同的方法
  - 方法一：先定义结构类型，再定义结构变量
  - 方法二：定义结构类型时，同时定义结构变量
- 方法一：先定义结构类型，再定义结构变量
  - 一般形式：struct 结构类型名 结构变量标识符列表;
  - 如：利用已定义的结构类型 struct stdType，代码“struct stdType st1, st2, stArray[200];”定义两个结构变量st1和st2，另定义结构数组 stArray[]，其元素类型为 struct stdType

结构变量的定义（续）

- 方法一：先定义结构类型，再定义结构变量（续）
  - 定义结构变量的写法与定义一个标准数据类型变量有区别
    - 定义结构变量时，不仅要明确指明变量的类型为结构 (struct)，同时要指明某一特定的结构类型名(如：stdType)
    - 定义标准数据类型的变量时，只需指明其类型，如“int i;”
  - 定义结构变量例子

```
struct date date1, date2;   /* 定义两个日期变量 */
struct person employee;     /* 定义一个职工变量 */
```

    - 定义两个struct date型变量date1和date2，一个struct person型变量employee

结构变量的定义（续）

- **方法二：定义结构类型时，同时定义结构变量**
  - 其一般形式： struct 结构类型名 {
    成分说明表
    } 结构变量标识符列表;
  - 如：代码
```
struct point{ /* 说明绘图程序的坐标类型 */
    int x;
    int y;
} p1, p2;
```
  - 定义struct point型变量p1、p2。在这种定义形式中，**如某种形式的结构类型只是临时性定义几个变量，可以省略结构类型名**，直接定义结构变量

结构变量的定义（续）

- 方法二：定义结构类型时，同时定义结构变量（续）
  - 如：

```
struct {
    char  name [20];        /* 产品名称 */
    int   num;              /* 产品编号 */
    float price;            /* 价格 */
    float quantity;         /* 数量 */
} part;                     /* 库存产品 */
```

结构变量的定义（续）

- 结构类型名、结构变量名和结构类型的成分名有不同的应用场合，分别出现在不同的程序上下文中，即使结构类型名、结构变量名、结构类型的成分名有相同的名字，编译程序能根据它们在源程序文件中出现的上下文，区别出名字的不同意义
  - 如：
```
struct s {
    int x;
    int s;
}s;
```
  - “struct s”中的s表示结构类型；“int s;”中的s为成分名。在语句中，对s的引用是结构变量s。编译程序根据标识符s 出现的上下文确定它是结构类型名，还是结构变量名或是结构类型的成分名

结构变量初始化（续）

- 在定义结构变量时可同时给它置初值，称为结构变量初始化
- 结构变量初始化时，要按其结构类型定义中的各成分顺序逐一给出各成分的初值。如

```
struct  point{ /* 说明绘图程序的坐标类型 */
   int  x;
   int  y;
 }  p3 = { 20,   50};
```

  - 静态结构变量初始化如static struct date d3 = {24, 11, 2008};

结构变量初始化（续）

- 结构变量初始化对初值表达式的要求与数组变量初始化的要求相同，遵守相同的规则：
  - 静态和全局结构变量的初始化是在程序执行之前完成
  - 如果静态结构变量未指定初值，则自动置0值
  - 局部结构变量是每次控制进入它所属辖域时创建并初始化
  - 未指定初值的局部结构变量其初值是不确定的

结构指针变量

- 把结构变量s所占据的存储段开始地址赋给与其同类型的结构指针变量p，则说指针p指向结构变量s。指针p是一个结构指针变量，简称结构指针

- 定义结构指针的方法，与定义一般指针变量一样
  - 如代码 struct date *pd , date3; 定义结构指针pd和结构变量date3
  - 其中，指针变量pd能指向类型为 struct date的结构
  - 赋值pd = &date3，使指针pd指向结构date3

结构变量的引用

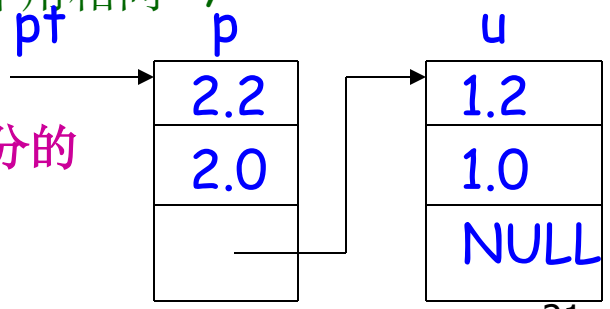
- 有两种引用结构的方式
  - 直接用结构变量名引用结构变量
    - 结构变量名.成分名
      - 例如，d1.year引用结构变量d1的year成分。因该成分的类型为int型的，可以对它施行任何int型变量可以施行的运算。例如赋值运算d1.year = 2008
  - 通过指向结构的指针间接引用结构变量
    - 指针变量名->成分名

引用结构成分的示例

例如：
```
struct  pNode {
        float  pos[2];
        struct  pNode  * next;
      } p, u, *pt;
```

以下代码的意义如它们的注释所示，产生如图所示的逻辑关系

```
pt = &p;              /* 使pt指向结构p */
p.pos[0] = 2.2;       /*结构p的pos数组成分的首元素pos[0]置值2.2 */
pt->pos[1] = 2.0;     /* 与代码“p.pos[1] = 2.0”的作用相同 */
p.next = &u;          /* 使结构p的next成分指向结构u */
p.next->pos[0] = 1.2; /* 与代码“u.pos[0] = 1.2”的作用相同 */
u.pos[1] = 1.0;
u.next = NULL;
```



引用结构成分的效果示意图

引用结构成分的示例(续)

- 上述例子说明结构的成分可以像普通变量一样使用。根据其类型决定其所有合法的运算
  - 进一步的例子

```
struct student{
  char name[20];
  int age;
  float score;
}stu1, stu2;

stu1.age ++ ;  /* 学生 stu1 的年龄增 1 */
 ++ stu2.age ;  /* 学生 stu2 的年龄增 1 */
 sum = stu1.score + stu2.score;
```

引用结构成分(续)

- 在结构类型struct pNode中，含有一个指向自身一样结构的指针成分，这是引用自身的结构形式
- 利用含有指针成分的结构可以实现结构的链接，用这样的结构能构成如链表、树、图等复杂数据结构
- 若结构成分本身又是结构类型，可继续使用成分运算符‘.’引用结构成分的结构成分，逐级向下引用到最低一级成分
  - 例如，对employee的某些成分的访问

```
employee.salary        = 560.00;
employee.hiredate.day  = 24;
employee.hiredate.month = 11;
employee.hiredate.year  = 2008;
scanf("%f", &employee.salary);
```

引用结构成分(续)

- ANSI C 还允许相同类型的结构变量相互赋值
- 表达式“* 指针变量”表示指针变量所指对象，所以通过指针引用其所指结构的成分也可写成：

  (* 指针变量). 结构成分名

  - 这里园括号是必需的，因运算符“*”的优先级低于运算符“.”
  - *pd.day等价于*(pd.day)，在这里这样的写法是错误的
  - 采用这种表记方法，通过pd引用date3的成分也可写成(*pd).day、(*pd).month、(*pd).year
  - 但很少场合采用这种表记方法

```
struct date *pd , date3;
```

提要

- 结构类型和结构变量
- **结构数组**
- 结构形参和结构指针形参
- 链表及其应用
- 联合
- 位域
- 枚举
- 类型定义

结构数组

- 用结构描述个体，用数组描述个体的集合
- 当数组的元素是结构时，这种数组就称为结构数组
  - 如用结构类型描述学生信息，而用结构数组表示一班学生。用结构数组表示全班学生能充分反映班级整体性，程序处理也较方便
- 在变量名之后指定元素个数，就能定义结构数组
  - 如 struct stdType students[30]; /*结构数组students有30个元素. 元素的类型是struct stdType */
  - struct person employees[200]; /*结构数组points有200个元素*/
  - 如 struct {

```
        int x;
        int y;
    } points[500];   /*一组坐标点*/
```

访问结构数组元素的成分

- 结构数组各元素在内存中顺序存放，也可初始化，对结构数组元素的访问也要利用元素的下标

- 访问结构数组元素的成分的标记方法为：
  - 结构数组名[元素下标].结构成分名
    - 首先是指定数组的元素，再指定该元素结构的成分

访问结构数组元素的成分(续)

- 结构指针变量也可指向结构数组的某个结构元素
  - 如有定义 struct stdType std[50], *ps, *p;
  - 赋值运算 ps = &std[2]使指针ps指向结构std[2]
- 以下代码序列实现在结构数组std中找最高分的那个结构，并由指针p指向该结构

```
p = std;  /* 等价于 p = &std[0], 使p指向结构std[0] */
for (ps = p+1 ;  ps < std+sizeof std/sizeof std[0]; ps++)
   if ( ps->score > p->score )  p = ps ;
```

  - 指针加n的意义是指针值增加n个单位，ps++使程序在循环执行过程中，指针ps顺序指向结构std[1]、std[2]、…

结构数组程序示例-1

- **根据输入的年份、月份和日期， 输出相应的月份英文名称和该日是该年份中的第几天**
  - 采用常用方法，严格检查输入数据合理性，发现不合理时，输出警告信息，并要求重新输入。程序框架如下

```
{  提示程序开始工作;
   for ( ; ; ) {
      输入年份;
      for ( ; ; ){ /* 月份输入不正确，继续循环*/
         输入月份;
         if (月份合理) break;
         输出警告信息;
       }
      输出月份英文名称;
```

结构数组程序示例-1(续)

- 算法(续)

```
for ( ; ; ) { /* 日期输入不正确，继续循环 */
    输入日期;
    if (日期合理) break;
    输出警告信息;
 }
计算日期是该年中的第几天;
输出结果;
输出是否继续的提示;
输入用户回答;
if (! 继续 ) break;      /* 结束程序 */
  }
}
```

结构数组程序示例-1(续)

- 主要数据结构
  - 日期结构变量：设日期变量包含有日、月、年、年中的天和月份名称字符串的首字符指针等成分
  - 月份与月份名称对照表：设它是一个结构数组，数组的每个元素是月份和月份英文名称组成的结构。因该对照表是固定不变的，可在定义它时初始化
  - 月份天数表：因月份天数有闰年和平年之分，将该数组设计成一个二维数组，其中第1行存储平年各月份的天数；第2行存储闰年各月份的天数，在定义时对它初始化

结构数组程序示例-1(续)

- 设计有关函数：将有关功能独立的程序段用函数来实现
  - 例如，确定日期是年中第几天的计算可编写成函数。为使该函数逻辑上独立于主函数，函数设有日期、月份、年份三个形参，返回日期是年中第几天的结果
  - 函数模型为int dayofYear(int，int，int)

结构数组程序示例程序1-P1

```
#include <stdio.h>
int dayTbl[][12] = {{31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
                    {31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}};
struct date { int day; int month; int year; int yearDay;
      char *monthName;
 } date;
struct { int mon; char *name;
 } month[] = {{1,"January"}, {2,"February"}, {3,"March"}, {4,"April"},
              {5,"May"}, {6,"June"}, {7,"July"}, {8,"August"},
              {9,"September"}, {10,"October"}, {11,"November"},
              {12,"December"}, {-1, ""}};
```

结构数组程序示例程序1-P2

```
char *monthName(int mon)  /* 寻找月份名称 */
{  int m;
   for(m = 0; month[m].mon > 0; m++)
      if(month[m].mon == mon) return month[m].name;
   return NULL;
}
int dayofYear(int d, int m, int y)  /* 计算年中第几天 */
{   int  i, leap, day = d;
    leap = (y%4 == 0 && y%100) || (y%400 == 0);
    for (i = 0; i < m-1; i++)  day += dayTbl[leap][i];
    return day;
}
```

结构数组程序示例程序1-P3

```
int main()
{  int leap, days; char ans; printf("\n\t\tDate Conversion Program\n");
   for(; ;)  {  printf("Year = ");  scanf("%d", &date.year);
      for(; ;) {  /* 输入月份 */
         printf("Month = ");  scanf("%d", &date.month);
         if (date.month >= 1 && date.month <= 12) break;
         printf("\tMonth must be between 1 to 12.\n");
       }
      date.monthName = monthName(date.month);
      printf("The name of the %d(th) month is %s.\n" ,  date.month,
date.monthName);
      leap=(date.year%4==0&&date.year%100)||date.year%400==0;
      days = dayTbl[leap][date.month-1];
```

结构数组程序示例程序1-P4

```
    for(; ;) {  /* 输入日 */
        printf("Day = ");   scanf("%d", &date.day);
        if (date.day >= 1 && date.day <= days) break;
        printf("\tDay must be between 1 to %d.\n", days);
      }
    date.yearDay = dayofYear(date.day, date.month, date.year);
    printf("\tThe days of the year are %d.\n", date.yearDay);
    printf("\tContinue?(y/n) ");
    do   scanf("  %c", &ans);
    while (!((ans >= 'A' && ans <= 'Z') || (ans >= 'a' && ans <= 'z')));
    if (ans != 'y' && ans != 'Y')  break;
  }
}
```

结构数组程序示例-2

- 设程序中有三个表示坐标点的变量，用户分别用1、2、3来称呼它们。程序给出以下命令供用户选择
  - 1.为某坐标点输入坐标值
  - 2.显示某点的坐标值
  - 3.指定两个点,让另外一个点成为它们的中点
  - 4.显示两点之间的矩离
  - 5.指定两点，判另一点是否在两点的连线上
  - 6.结束程序
  - 这里，将实现各种功能的代码、供用户指定坐标点序号的代码以及显示菜单并接受用户选择的代码分别编写成函数

结构数组程序示例2-P1

```
#include <stdio.h>
#include <math.h>
struct point{ int x;  int y;
 } p[3];  /* 结构数组，存储三个坐标点 */
int menu()  /* 菜单函数 */
{ int choice;
  do {  printf("\n\n\n");
    printf("  1.指定点，并为它输入坐标值\n");
    printf("  2.指定点，显示它的坐标值\n");
    printf("  3.指定两点,求出它们的中点坐标给另外一个点\n");
    printf("  4.指定两点，显示两点间的矩离\n");
    printf("  5.指定两点，判另一点是否在两点的连线上\n");
    printf("  6.结束程序\n");
```

结构数组程序示例2-P2

```
    printf("\t输入你的选择!\n");    scanf("%d", &choice);
    if (choice >= 1 && choice <= 6) return choice;  /* 返回选择 */
    printf("选择出错!  重新选择。\n");
  } while ( 1 );
}
int pointNo()   /* 接受用户指定的点的代号 */
{ int pno;
  while (1) {
     printf("指定点的代号:\n"); scanf("%d", &pno);
     if(pno >= 1 && pno <= 3)  return pno-1;
     printf("点的代号指定不正确, 请重新");
   }
}
```

结构数组程序示例2-P3

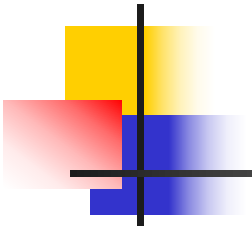
```
void inputPoint(int pno)  /* 输入指定点的坐标 */
{  printf("输入第 %d 点的 X 坐标和 Y 坐标\n", pno+1);
   scanf("%d%d", &p[pno].x, &p[pno].y);
}
void displayPoint(int pno) /* 显示指定点的坐标 */
{  printf("第%d点的X坐标=%d,它的Y坐标=%d\n", pno+1, p[pno].x, p[pno].y);
}
void midPoint() /* 求指定两点的连线中点为另一点 */
{ int pno1, pno2, pno3;
  pno1 = pointNo();  pno2 = pointNo();  pno3 = 3 - pno1 - pno2;
  p[pno3]. x = (p[pno1].x + p[pno2].x)/2;
  p[pno3]. y = (p[pno1].y + p[pno2].y)/2;
}
```

结构数组程序示例2-P4

```
void distance()  /* 指定两点，显示两点间的矩离*/
{ int pno1, pno2;  pno1 = pointNo(); pno2 = pointNo();
  printf("两点间矩离=%f\n",
  sqrt((double)(p[pno1].x-p[pno2].x)*(p[pno1].x-p[pno2].x)+
        (double)(p[pno1].y - p[pno2].y) * (p[pno1].y - p[pno2].y)));
}
void isInLine() /*判另一点是否在指定两点的连线上 */
{ int pno1, pno2, pno3;  double t;
  pno1 = pointNo();   pno2 = pointNo();   pno3 = 3 - pno1 - pno2;
  t = (double)(p[pno2].y-p[pno1].y)*(double)(p[pno3].x - p[pno1].x) -
       (double)(p[pno3].y - p[pno1].y)*(double)(p[pno2].x - p[pno1].x);
  if (fabs(t) < 1.0e-5)  printf("In same line\n");
  else printf("Not in same line\n");
}
```

结构数组程序示例2-P5

```
int main()
{  int choice;
   while (1) {
     choice = menu();
     switch(choice) {
       case 1: inputPoint(pointNo()); break;
       case 2: displayPoint(pointNo()); break;
       case 3: midPoint(); break;
       case 4: distance(); break;
       case 5: isInLine(); break;
       case 6: return;
     }
   }
}
```



提要

- 结构类型和结构变量
- 结构数组
- **结构形参和结构指针形参**
- 链表及其应用
- 联合
- 位域
- 枚举
- 类型定义

结构形参

- 以函数dayofYear()为例，若以结构为形参，改写该函数有

```
int dayofYear(struct date d)
{ int  i, leap, day = d.day;
  leap = (d.year%4 == 0 && d.year%100) || d.year%400 == 0;
  for (i = 0; i < d.month-1; i++)  day += dayTbl[leap][i];
  return day;
}
```

- 调用带结构形参的函数，必须提供与形参相同类型的结构变量实参。所以主函数中对函数dayofYear()的调用应改写成：

```
date.yearDay = dayofYear(date);
```

- 函数设置结构形参，虽能向函数直接传递结构信息，但*函数调用时，系统要为结构形参分配存储单元，并要完成实参结构向形参结构传递值的工作等*

结构指针形参

- 函数设置结构指针形参：再改写函数dayofYear()，使其以日期结构指针为形参，并以此说明结构指针形参的用法

```
void dayofYear(struct date *dp)
{ int i, leap, day = dp->day;
  leap=(dp->year%4==0&&dp->year%100)||dp->year%400==0;
  for(i = 0; i < dp->month-1; i++)   day += dayTbl[leap][i];
  dp->yearDay = day;
}
```

- 改写后的函数通过指针形参引用结构成分，并将计算结果存放在结构的相应成分中，不再返回结果
- 对该函数的调用方式也需相应地改写成 dayofYear(&date);调用函数dayofYear()时，必须提供结构变量date的地址

结构形参 vs 结构指针形参

- 以结构为形参，函数dayofYear()必须用return 语句返回结果。函数的形参是函数的局部变量，函数调用时，将实参结构date的各成分值拷贝到形参结构d的各成分中，以后函数对d的改变与date无关

- 以结构指针为形参，函数调用时，虽只传递某结构的地址值，但函数能通过该指针间接引用外面实在的结构变量。函数引用形参所指结构，并把计算结果存储于形参所指的结构中

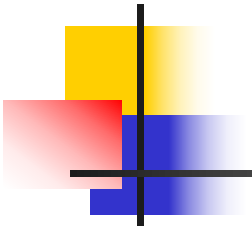
结构形参和结构指针形参

- C语言还允许函数返回结构类型值，如将函数dayofYear()改为设置 struct date类型的形参，并返回struct date类型的值。对函数dayofYear()的新的改写如下

```
struct date dayofYear(struct date d)
{ int i, leap;
  d.yearDay = d.day;
  leap = (d.year%4 == 0 && d.year%100)||d.year%400 == 0;
  for(i = 0; i < d.month-1; i++)  d.yearDay += day_tbl[leap][i];
  return d;
 }
```

结构形参和结构指针形参

- 主函数调用函数dayofYear()把返回的结构值赋给结构变量date，即 date = dayofYear(date);
- 调用有结构类型形参的函数时，实参结构的各成分值要全部拷贝给形参结构的相应成分，费时间又费空间
- 一般情况下，以传递结构指针为好。仅当为了程序的安全性，确保函数不修改实参结构成分值等的情况下，使用结构指针作为形参
- 函数返回结构值时，常需要将函数的返回值赋给结构变量，对于复杂的结构类型，需要执行一长串的传送指令才能实现这一要求
- 从程序执行效率和实用性说，以函数返回结构的指针更简便



提要

- 结构类型和结构变量
- 结构数组
- 结构形参和结构指针形参
- **链表及其应用**
- 联合
- 位域
- 枚举
- 类型定义

链表及其应用

- 程序用变量表示问题中的数据对象，变量需要通过变量定义而引入。程序执行时，不能显式地用语句随意生成变量和消去变量，并且变量之间的关系也不能由变量本身的成分来表达。但在现实问题中，问题所包含的数据对象的个数是变化着的，数据对象之间的关系也在不断地变化着的

- 按数据对象可能呈现的最多状态/情况来设定变量，称这种实现方法为静态方法

- 按需要用语句显式地生成和消去数据对象，数据之间的关系也由程序随时设定和改变，称这种实现方法为动态方法

动态数据结构

- 采用动态方法实现的数据结构被称为动态数据结构
- 动态数据结构中的数据对象是一种结构变量，除有一些数据信息成分，还有指针类型成分，并且这些指针成分能指向与自身同样类型的结构
  - 例如下面的结构类型定义：

```
struct intNode {  /* 整数链表表元类型 */
   int value;     /* 存放整数 */
   struct intNode *next; /* 存放后继表元的指针 */
  };
```

  - 这种结构又称为自引用结构。利用成分next可把一个struct intNode类型的结构与另一个同类型的结构链在一起，用于描述动态数据结构中两数据对象之间的关系
  - 在这种有成分引用自身结构类型的定义中，要求那些引用自身成分的类型只能是指针类型（为什么？）

内存动态分配和释放库函数

- 因动态数据结构中的数据对象可按需要由程序语句动态生成和消去，所以建立和维护动态数据需要内存动态分配和释放
- C 系统的函数库提供了供程序动态申请和释放存储单元的库函数。其中最经常使用的是下面介绍的三个库函数
  - void * malloc(unsigned size)
  - void * calloc(unsigned nelm, unsigned elsize)
  - void free(void *ptr)

内存动态分配和释放库函数

- void * malloc(unsigned size)
  - 函数调用malloc(size)向系统申请内存，从系统提供的动态存储区域中分配至少size个字节的连续空间，函数返回该连续空间的开始地址
  - 如果因动态存储区域已分配完，而不能满足这次申请要求，函数将返回0(NULL)值
  - 因函数malloc()的返回值是无类型指针值，程序可将该返回值强制转换成某种特定的指针类型
  - 如有变量定义 struct intNode *p;
  - p = (struct intNode *)malloc(sizeof(struct intNode));实现向系统申请能存储类型为struct intNode的一个结构，并将该结构的指针存于指针变量p中

内存动态分配和释放库函数

- void * calloc(unsigned nelm, unsigned elsize)
  - 函数调用calloc(nelm, elsize)也可用来向系统申请内存单元，在动态存储区中分配nelm*elsize个字节的连续空间，并将该空间的内容清0。函数返回该连续空间的开始地址；如果分配不成功，则返回Null
  - 如以下函数调用都返回能存储100个整数的存储块，利用函数返回的存储块的首地址，可象数组一样访问这存储块中的100个元素

```
p = (int *) malloc(100 * sizeof(int));
q = (int *) calloc(100, sizeof(int));
```

内存动态分配和释放库函数

- void free(void *ptr)
  - 函数调用free(ptr)释放由ptr所指向的存储块。要求ptr的值是调用函数malloc()或calloc()申请到的连续存储空间中的地址
- 以上三个函数中，形参size、nelm和elsize为unsigned类型，ptr为某种指针类型
- 函数malloc()和calloc()返回的是系统分配的连续存储空间的首地址，其类型为void *。程序将返回值赋给某个指针变量，然后用该指针变量间接引用存储空间的相应成分

线性表

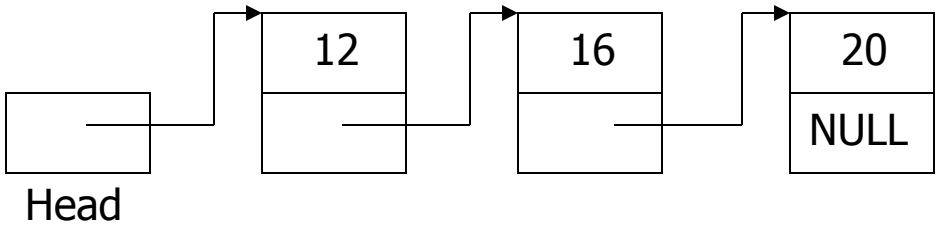
- 线性表是一种最简单又最常使用的数据结构
- 它是由相同类型的结点组成的有限序列，线性表中的结点习惯称为元素或表元
  - 一个有n个表元 e_0、e_1、…、e_{n-1}组成的线性表记为 (e_0、e_1、…、e_{n-1})
- 线性表中的表元个数称为线性表的长度，长度为0的线性表称为空表
- 线性表中表元之间的关系由表元在线性表中的位置所确定，用相邻表元构成的表元对 (e_i,e_{i+1}) (0 <= i <=n-2) 表示线性表上存在的线性关系

线性表（续）

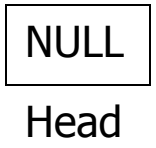
- 即使两个线性表有相同的表元集合，但它们的表元在线性表中出现的顺序不同，则它们是两个不相同的线性表
- 线性表的表元可以由多个成分组成。如果线性表有能唯一确定其表元的成分或成分组，则该成分或成分组称为该线性表的关键字，简称键
- 为了讨论简便，往往只考虑表元的关键字，而忽略表元的其它成分
- 线性表可以用顺序存储的数组实现，也可采用链表来实现

链表

- 链表概述
  - 有一个指针指向链表的第一个表元，称其为“头指针”或“链表头”
  - 链表中的每个表元都包含两部分内容：表元的实际数据信息和后继表元指针
- 下图表示最简单的一种链表结构
  - 图中，头指针head指向第一个表元，第一个表元又指向第二个表元，…，直到最后一个表元，该表元不再指向其他表元。称链表的最后一个表元为“表尾”



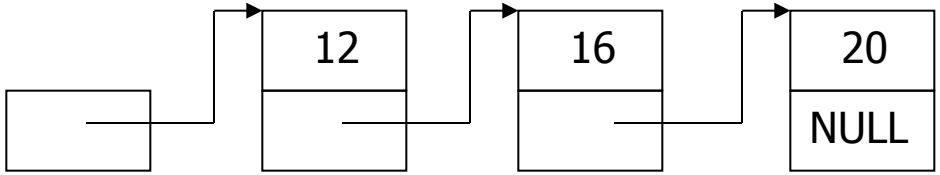
(a) 一个非空链表



(b) 一个空链表

链表（续）

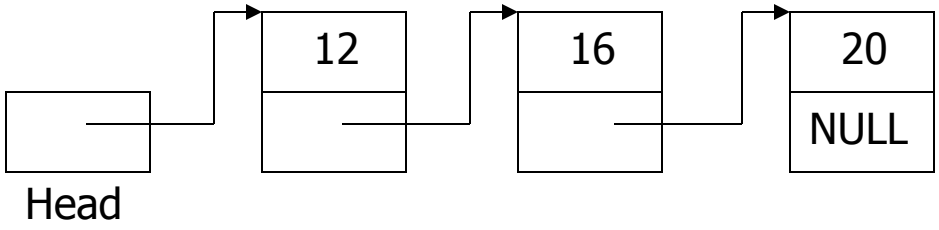
- 表元的后继表元位置由表元所包含的指针所指明。链表中各表元在内存中的存放位置是可以任意的。如要寻找链表中某表元，必须由链表头指针所指的第一个表元开始、顺序查找
- 下图中所示的链表结构是单向的，即每个表元只知道它的后继表元位置，而不能知道它的前驱表元
- 在某些应用中，要求链表的每个表元都能方便地知道它的前驱表元和后继表元，这种链表的表元应设有两个指针成分，分别指向它的前驱和后继表元，这种链表称为双向链表
- 为适应不同问题的特定要求，链表结构也有多种变形



链表与数组的主要区别

- 数组的元素个数是固定的，而组成链表的表元个数可按需要增减
- 数组元素的存储单元在数组定义时分配，链表表元的存储单元在程序执行时动态向系统申请
- 数组元素的顺序关系由元素在数组中的位置或下标确定，链表中的表元顺序关系由表元所包含的指针来体现
- 长度不固定的线性表，用可能最大长度的数组来描述，会浪费许多内存空间；用链表实现可按需要动态改变
- 用数组实现线性表，当有表元插入或删除时，要移动部分表元的存储位置；用链表实现不需要移动表元

链表（续）



- 上图中是一个有三个表元的整数链表，该链表的表元类型就可用前述的类型struct intNode来描述
- 结构类型struct intNode中的成分next是指针类型，它能指向类型struct intNode的结构。在结构类型struct intNode的定义中，为定义成分next的类型，使用了还未完全定义的类型，并且正是自已要定义的类型
- 在有成分引用自身结构类型的定义中，限制此成分的类型只能是指针类型
  - 如类型 T1中有成分t1引用类型T2，T2类型中有成分t2引用类型T1，其中t1和t2都是指针类型

链表（续）

- 为了定义类型T1和T2，需要引入不完整的结构类型说明

```
struct T1 {
    int      ival;
    struct T2 *t1;
  };
struct T2 {
    char     cval;
    struct T1 *t2;
  };
```

- 其中类型struct T1定义中的成分struct T2 *t1就包含不完整的结构类型struct T2

链表（续）

- 对于不完整的结构类型定义，可以在其辖域之内稍后给出它的完整的结构类型定义，如上面示例
- 但是，如果在上述定义的前面某处还有一个结构类型T2的定义，则类型T1中对T2的引用指的是前面的那个类型T2，而不是其后定义的类型T2
- 在这种情况下，如要在T1中引用其后定义的类型T2，可在T1类型定义之前加一个空的结构类型说明 `struct T2;`
  - 该空的类型说明的作用是屏蔽前面关于T2的定义

链表的基本操作

- 链表的基本操作主要包括：建立空链表、生成指定值的新表元、插入一个表元、遍历、查找和删除一个表元等
- 这些操作都有固定的模式，利用它们并结合循环等控制能实现各种有关链表的复杂操作

- 下面以整数单向链表为例，说明各种操作的实现
  - 假定链表表元类型是前面定义的struct intNode类型，并另有以下变量说明

```
struct intNode *head, *p, *q, *u, *v, *w;
```

  - 其中变量head是整数链表头指针，其余变量是工作指针

链表的基本操作（续）

- 建立空链表
  - 链表的建立过程是从空链表(没有链表表元)出发，逐渐插入链表新表元的过程
  - 要使以变量head为头指针的链表为空链表，让head取NULL值即可
  - 语句：head = NULL；就建立了以head为链表头指针的空链表

链表的基本操作（续）

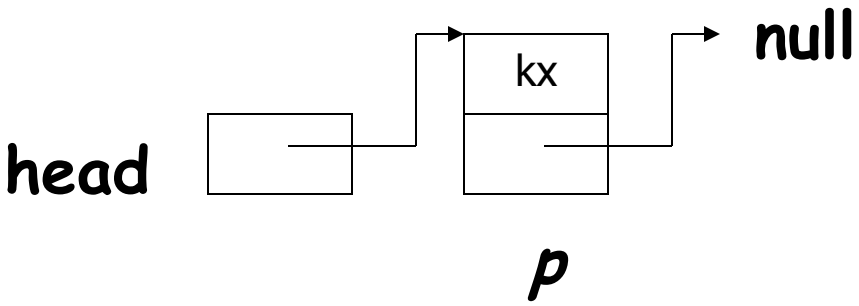
- 生成指定值的新表元
  - 要生成值为kx的新表元，先向系统申请一个表元大小的存储块，然后将指定值填入即可。如代码

```
p = (struct intNode *)malloc(sizeof(struct intNode));
p->value = kx; p->next = null;
```

  - 这里，有对函数malloc()的调用，实参sizeof(struct intNode)是一个表达式，sizeof运算的结果是运算分量所需(或占用)的字节数，即存储一个表元所需字节数
  - malloc()返回分配给新表元的存储空间的开始地址

生成由指定值的表元构成的新链表

```
p = (struct intNode *)malloc(sizeof(struct intNode));
p->value = kx;
head=p;
p-next=null;
```



链表的基本操作（续）

- 生成指定值的新表元(续)
  - 申请获得的存储空间开始地址需转换成存储对象类型的指针类型，再对该存储空间以数据类型所具有的成分进行存储
    - 例如，对函数malloc()调用的返回值作强制类型转换(struct intNode *)，使函数调用返回的(void *)类型值转换成struct intNode * 类型的值，以便通过转换后的指针值引用成分value和next

链表的基本操作（续）

- 向链表插入一个新表元
  - 向链表插入一个新表元是建立链表的主要操作之一，按新表元要插入位置不同有：
    - 在首表元之前插入新表元
    - 在某指针所指的表元之后插入新表元
  - 在首表元之前插入新表元
    - 在首表元之前插入新表元，插入后将使新表元成为新的首表元。这个工作包括将原链表的首表元接在新表元的后面，和修改链表首指针，使链表的首指针指向新表元。设新表元由指针p所指，下面是实现这个功能的代码`p->next = head;  head = p;`

链表的基本操作（续）

- 向链表插入一个新表元(续)
  - 在某指针所指的表元之后插入新表元
    - 对于单向链表，因为一个表元的前驱表元不是直接可引用的，所以新表元一般要求插在某已知表元之后。设已知表元由指针w所指，且w的值不是NULL，而待插入的新表元由指针p所指
  - 插入工作包含两个基本动作
    - 首先，将w所指表元的后继表元指针复制给p所指表元的后继指针域，即原先w所指表元的后继表元成为p所指表元的后继表元，这可用代码p->next = w->next实现
    - 然后，是将p的值复制给w所指表元的后继指针域，即让p所指表元成为w所指表元的后继表元，这可用代码w->next = p实现

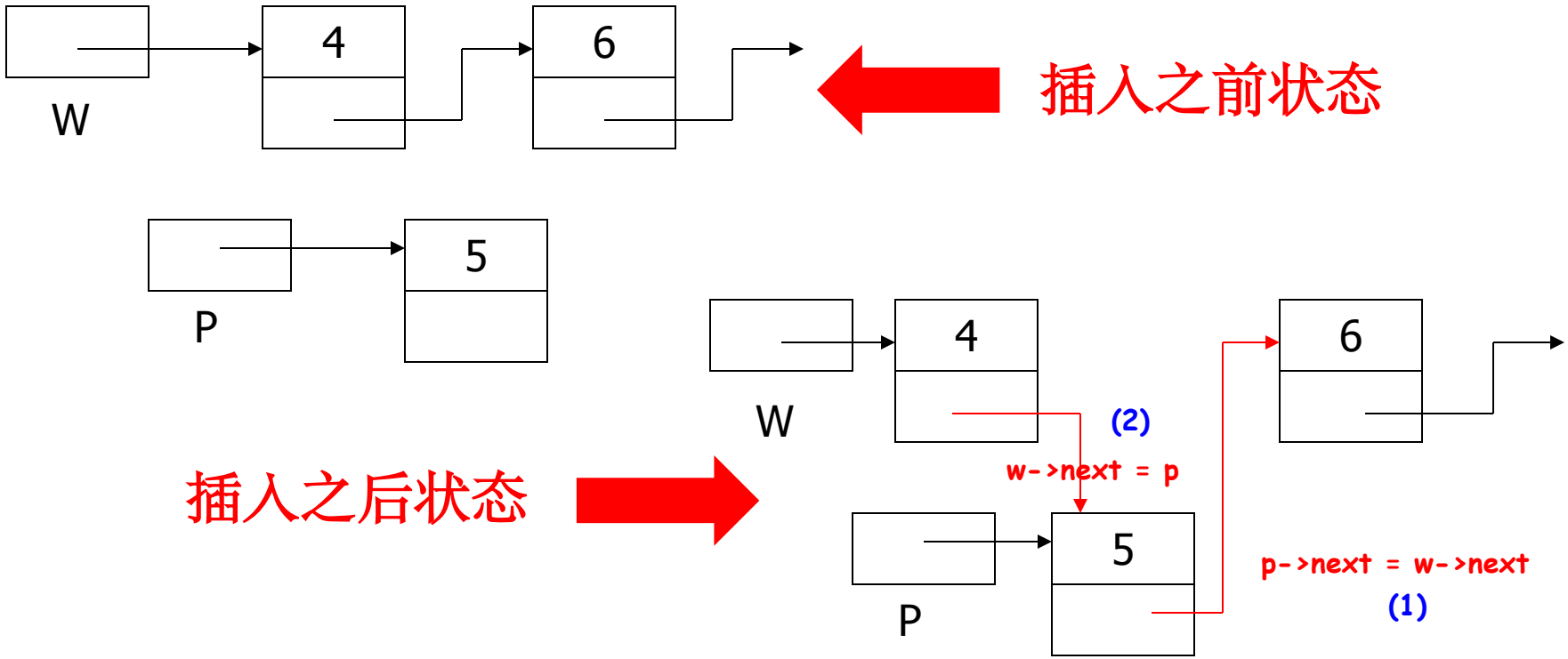
链表的基本操作（续）

- 向链表插入一个新表元(续)
  - 在某指针所指的表元之后插入新表元(续)
    - 以下代码就能实现在w所指表元之后插入p所指表元

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

链表的基本操作（续）

- 向链表插入一个新表元(续)



链表的基本操作（续）

- 遍历链表
  - 从链表的首表元开始，沿着链表表元的链接顺序逐一考察每个表元，直至链表结束
  - 用工作指针p遍历整个链表，访问表元只做输出表元值的工作。p的初值为链表头指针，在p不等于NULL值时循环，输出p所指表元的值，并准备考察下一个表元(即p = p->next)。能实现遍历链表功能的函数定义如下

```
void travelLink(struct intNode *h)
   {  struct intNode *p = h;
      while ( p != NULL ) {
         printf("%4d ", p->value); /* 输出表元的值 */
         p = p->next;  /* 准备访问下一个表元 */
      }
      printf("\n");
   }
```

链表的基本操作（续）

- 在链表中查找指定值的表元
  - 在链表中查找指定值表元可能有不同目的
    - 找到后以获取该表元的详细信息，称为简单查找
    - 为了进一步对链表的修改，或将查到的表元删除，或在查到的表元之前插入一个新表元等，称为动态查找
  - 查找过程的实现又可分两种情况
    - 在无序链表上的查找
    - 在有序链表上的查找

链表的基本操作（续）

- 在链表中查找指定值的表元
  - 无序链表上的简单查找：从链表头指针所指第一个表元出发，顺序查找：1）发现有指定值的表元，以指向该表元的指针值为查找结果；2）查找至链表末尾，未发现指定值的表元，查找结果为NULL

```
struct intNode * searchSLink(struct intNode *h, int key)
{  struct intNode *v = h;
   while (( v != NULL ) && ( v->value != key))    v = v->next;
   return  v; }
```

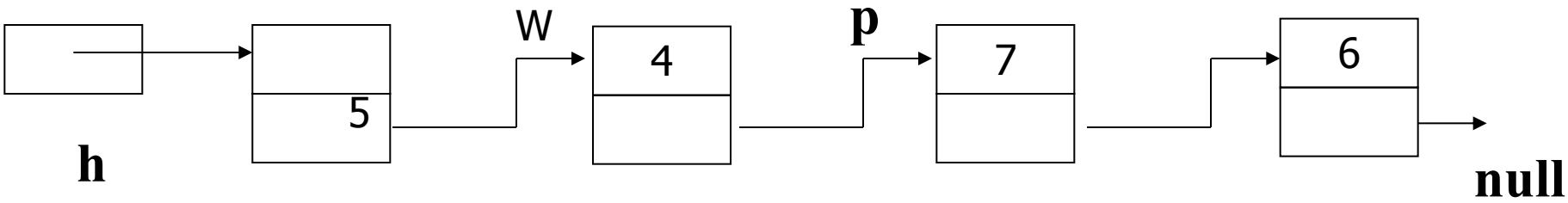
形参h是已知链表的头指针，形参key是要找表元的值。函数使用指向链表表元的工作指针变量v，它的初值为链表头指针。在v不等于NULL值，且v所指表元的值不等于指定值情况下，考察下一个表元(即v = v->next)

链表的基本操作（续）

- 在链表中查找指定值的表元
  - 有序链表上的简单查找
  - 如链表的表元是按值从小到大顺序链接的，则在顺序考察链表表元的查找循环中，当发现还有表元，并且该表元的值比**key**小时，应继续查找。反之，若不再有表元，或当前表元值不比**key**值小，则应结束查找。查找循环结束后，仅当还有表元，且表元的值与**key**值相同情况下，才找到，函数返回当前表元的指针。否则，链表中没有值为**key**的表元，函数返回**NULL**值。

```
struct intNode *searchSOLink(struct intNode *h, int key)
{   struct intNode *v = h;
    while (( v != NULL ) && ( v->value < key))
      v = v->next;
    return  v != NULL && v->value == key ? v : NULL;
}
```

动态查找



为了支持后续的插入或者删除操作，
当搜索到p指向的表元（k=7）时，无论是删除表元（k=7）还是在它之前插入新表元，我们都要知道w

链表的基本操作（续）

- 无序链表上的动态查找

  动态查找为插入或删除做好必要的准备工作。由于是单向链表，函数除要回答查找结束时的当前表元的指针外，还应回答该表元的前驱表元指针。令函数的返回值是查找结束时的当前表元的指针，函数另设一个指针形参，将找到的前驱表元的指针存于环境中的某指针变量中，所以该形参的类型是intNode**的。

```
struct intNode *searchDSLink(struct intNode *h, int key,
                                        struct intNode ** pp)
{ struct intNode *v = h, *u = NULL;
    while (( v != NULL ) && ( v->value != key)) {
        u = v ;   v = v->next; /* 后继表元成为当前表元 */
    }
    *pp = u;   return  v;
}
```

链表的基本操作（续）

- 有序链表上的动态查找

  设链表的表元按值从小到大顺序链接，与无序链表上的动态查找类似，但查找循环当发现查找值不比链表当前表元的值小时，就应提早结束查找循环。

```
struct intNode * searchDOLink(struct intNode *h,
                    int key,   struct intNode ** pp)
 {      struct intNode *v = h, *u = NULL;
        while (( v != NULL ) && ( v->value < key)) {
             u = v;       v = v->next;
        }
        *pp = u;      return  v;
 }
```

链表的基本操作（续）

- 如在头指针为head的有序链表上找值为5的表元，并要求找到时，值为5表元的指针存于指针变量p，而前驱表元的指针存于指针变量q，则函数的调用形式为

  p = searchDOLink(head, 5, &q);

函数返回后，根据p和q的值，可分出以下多种不同情况：

A. 若q == NULL且p == NULL，则链表是空链表；

B. 若q == NULL，且p != NULL，则在考察链表的首表元时，就结束查找，是否找到由表达式p->value == key为真确定。

C. 若q != NULL，且p == NULL，则链表中没有要找的表元，q指向链表的末表元；

D. 若q != NULL，且p != NULL，则在考察链表的中间表元时，就结束查找，是否找到也由表达式

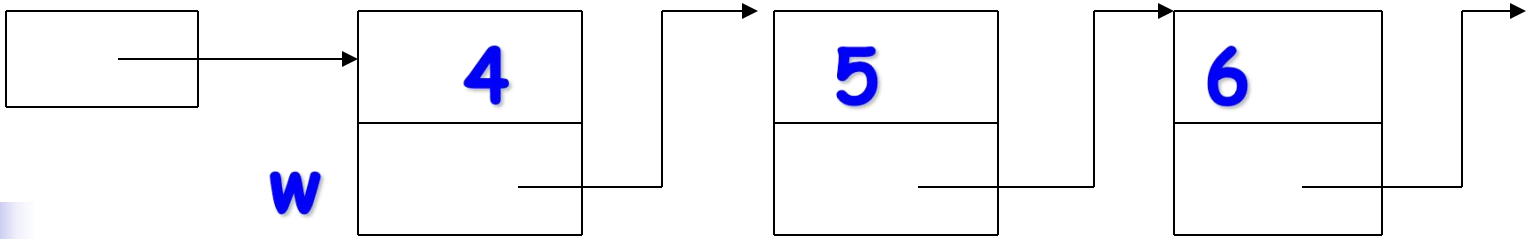
p->value == key为真确定。

链表的基本操作（续）

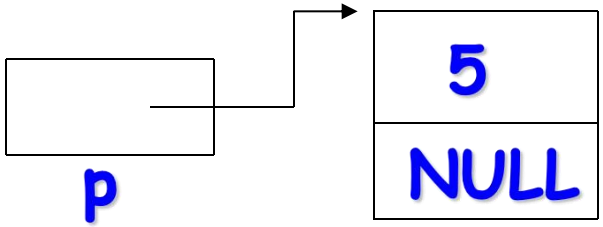
- 从链表删除指定表元的后继表元
  - 在动态数据结构的应用中，除产生新表元和插入到动态数据结构中的操作以外，还有改变动态数据结构中表元之间的关系、删除原动态数据结构中的表元等基本操作。
  - 设要删除链表中由指针变量w所指表元的后继表元。如后图所示，欲删去值为5的表元。经删除后，w所指表元的后继表元变为值为6的表元。完成这样的删除操作，可用以下语句实现：

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

(a) 删除前



(b)删除后



操作操作：

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

从链表删除的表元可有两种不同的处理：调用函数free()回收删除表元所占的存储单元；或将删除表元插入到其他链表等

链表的基本操作（续）

- 从无序链表删除指定值的表元
  - 首先要查找指定值的表元。若未找到，则不做删除工作；若找到，则将它从链表中删除，并返回被删除表元的指针
  - 删除时要考虑两种情况：如删的是首表元，要更改链表头指针；否则更改前驱表元的后继指针
  - 因在删除链表首表元时，要修改链表的头指针。为此，函数将对应链表头指针的形参类型说明为指向指针的指针，即二级指针

```
struct intNode*sDelete(struct intNode **hpt, int key)
{ struct intNode  *u, *w;
  u = *hpt;     /* 让 u 指向链表的首表元 */

 while ((u != NULL) && (u->value != key)) {
   w = u;     u = u -> next;
 }
 if (u != NULL){ /* 链表中有值为key的表元 */
     if (u == *hpt)
       *hpt = u->next; /* 修改链表头指针 */
     else  w->next = u->next;
     u->next = NULL ;
 }

  return  u; /* 返回指向被删除表元的指针 */
}
```

如有某链表头指针为 head，要删除表元值为 5 的表元，可用代码p = sDelete(&head, 5)来实现。

链表的基本操作（续）

- 调用前面的无序链表动态查找函数，上述函数又可改写为：

```
struct intNode *sDelete(struct intNode  ** hpt, int key)
{ struct intNnode  *u, *w;
  if ((u = searchDSLink(*hpt, key, &w)) == NULL)
     return NULL;
  if(w==NULL)*hpt=u->next; /* 修改链表头指针 */
  else w->next = u->next;
  u->next = NULL;
  return u;
}
```

链表的基本操作（续）

- 在有序链表上，利用动态查找函数，重写删除函数：

```
struct intNode *rDelete(struct intNode ** hpt, int key)
{ struct intNode  *u, *w;
  if ((u = searchDOLink(*hpt, key, &w)) == NULL
          || u->value != key)
      return NULL;
  if(w==NULL)*hpt=u->next; /* 修改链表头指针 */
  else w->next = u->next;
  u->next = NULL;
  return u;
}
```

链表的基本操作（续）

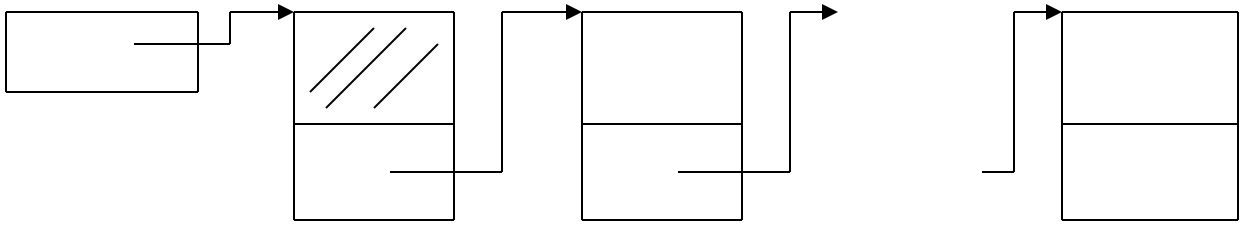
- 在有序链表中插入指定值的表元
  - 搜索指定表元，若找到，不再插入；否则，生成新表元并插入之。

```
int rInsert(struct intNode  ** hpt, int key)
{ struct intNode  *u, *w, *p;
  if ((u = searchDOLink(*hpt, key, &w)) == NULL
      || u->value != key) {
      p=(struct intNode*)malloc(sizeof(struct intNode));
      p->value = key;    p->next = u;
      if (w==NULL) *hpt = p; /* 修改链表头指针 */
      else w->next = p;
      return 1;  /* 插入新结点成功 */
  }
  return 0; /* 链表已有值为key的表元 */
}
```

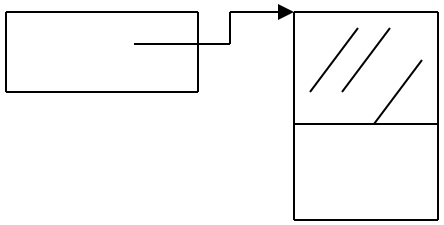
链表的基本操作（续）

- 前面的函数 **rInsert()**返回整型值，0表示链表中已有指定值的表示，本次函数调用未插入新表元；返回1，表示正确完成插入工作
- 插入新表元要考虑插入在首表元之前，删除表元也要考虑删除链表首表元
- 为了避免判别是否是链表的首表元和修改链表头指针，单链表通常增设一个辅助表元，它出现在链表有效表元的首表元之前

链表的基本操作（续）



带辅助表元的非空链表



带辅助表元的空链表

图7-10带辅助表元的链表

在带辅助表元链表上的插入或删除表元的操作，因总是在辅助表元之后进行，不再需要判别是否是链表的首表元，也不会对链表头指针作修改