周水庚

2017年12月14日

提要

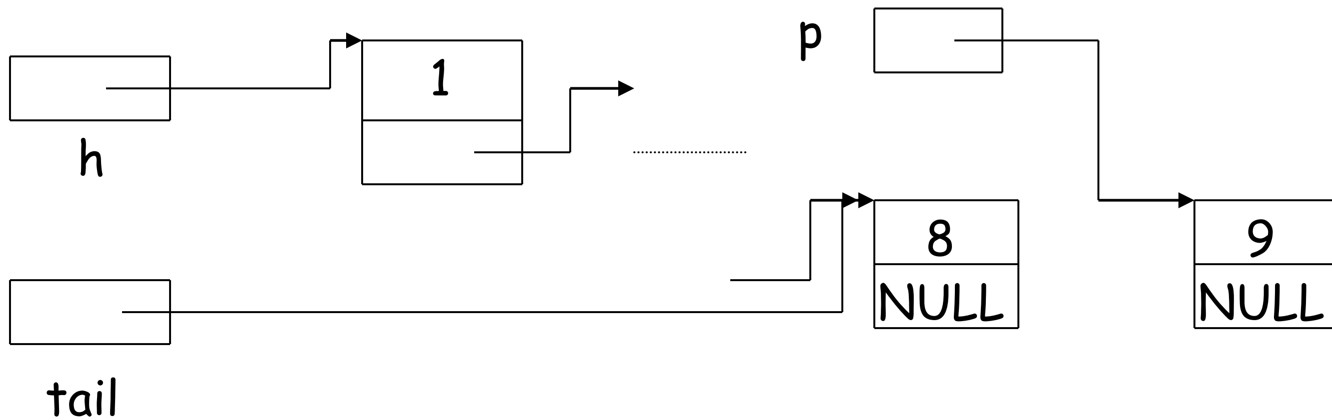
- 结构类型和结构变量
- 结构数组
- 结构形参和结构指针形参
- 链表及其应用
- 联合
- 位域
- 枚举
- 类型定义

提要

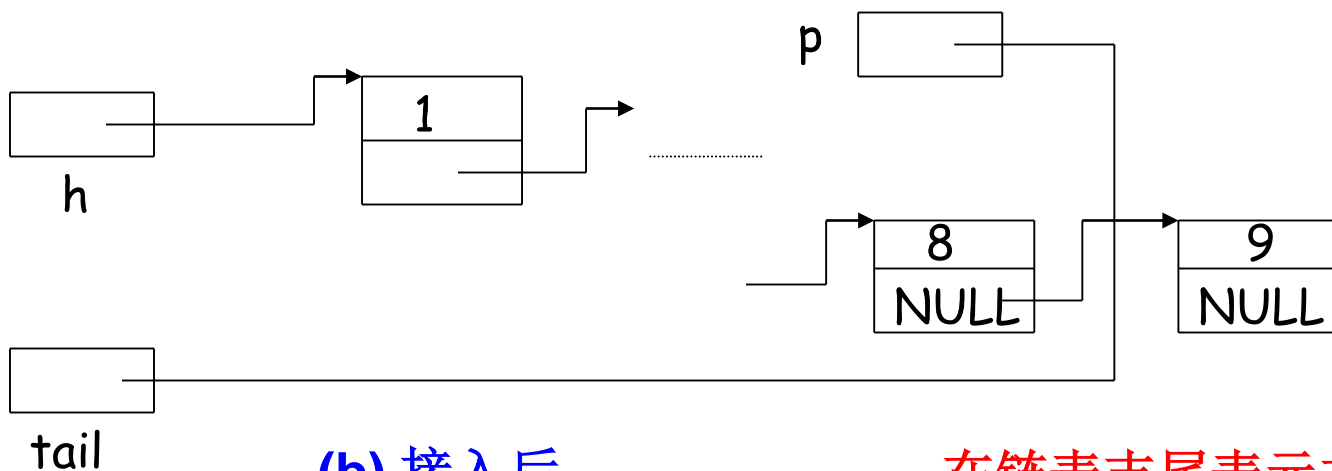
- 结构类型和结构变量
- 结构数组
- 结构形参和结构指针形参
- 链表及其应用
- 联合
- 位域
- 枚举
- 类型定义

链表程序设计实例-1

- 编写按整数输入顺序建立一个单向整数链表的函数
  - 从空链表开始，每输入一个整数，向系统申请一个表元存储空间，将输入整数存入新表元并将新表元接在链表末尾。当不再有整数输入时，函数返回生成的链表的头指针值
  - 为使新表元能方便地接在链表的末尾，引入指向链表的末尾表元的指针变量tail。建立空链表时，头指针h和末尾表元指针tail都置值NULL。将新表元接在链表末尾的工作要分两种情况。一是原链表为空链表；二是原链表有表元。下图是在链表末尾表元之后接一个表元的示意图，图中表元的整数用于区别不同表元



(a) 接入前



(b) 接入后

在链表末尾表元之后接一个新表元

链表程序设计实例-1(续)

- 设链表的表元类型为前面说明的**struct intNode**类型

```
struct  intNode  *createList()
   { struct intNode *h, /* 链表的头指针 */
          *tail, /* 链表末尾表元的指针 */
          *p;      int  n;
     h = tail = NULL;        printf("Input data.\n");
     while (scanf("%d", &n) > 0)  { /* 有整数读入 */
       p = (struct intNode *)malloc(sizeof(struct intNode));
       p->value = n;             p->next  = NULL;
       if (h == NULL)  h = tail = p;
       else  tail = tail->next = p;
     }
     return h;
   }
```

如果输入的整数是键值，程序对吗?

链表程序设计实例-1(续)

- 上述函数createList()定义表明链表新表元总接在链表末尾
- 在有些情况下，对新表元放在链表中的位置没有要求，最简单的办法是将新表元插在链表首表元之前，即让新表元作为更新后链表的首表元。若要对函数createList()定义作这种改写，其中变量tail就不再需要了。另外，**while** 控制结构内的语句“p->next = NULL;”和if结构可用以下两个赋值代替：

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

- 第一个赋值表示新表元的后继表元为原链表中的首表元；第二个赋值表示让链表头指针指向新表元

- 新表元总是放在链表头

```
struct  intNode  *createList()   {
      struct intNode *h, *p; /* 链表的头指针 */
      int   n;
      h = NULL;
      printf("Input data.\n");
      while (scanf("%d", &n) > 0)  { /* 有整数读入 */
         p = (struct intNode *)malloc(sizeof(struct intNode));
         p->value = n;
         p->next  = h;
         h = p;
      }
      return h;
}
```

链表程序设计实例-2

- 编写一个函数，输入整数，建立一个按值从小到大顺序链接的链表
  - 函数以输入整数进行循环，对每个输入的整数，完成生成新表元、寻找插入位置和把新表元插入的工作。插入时，也要考虑插在首表元之前的情况。

```
struct intNode *cSortList()
{ struct intNode *u, *w, *p, *h = NULL; int n;
  printf("Input data.\n");
  while (scanf("%d", &n) > 0){/* 还能输入整数 */
      p=(struct intNode *)malloc(sizeof(struct intNode));
      p->value = n;
```

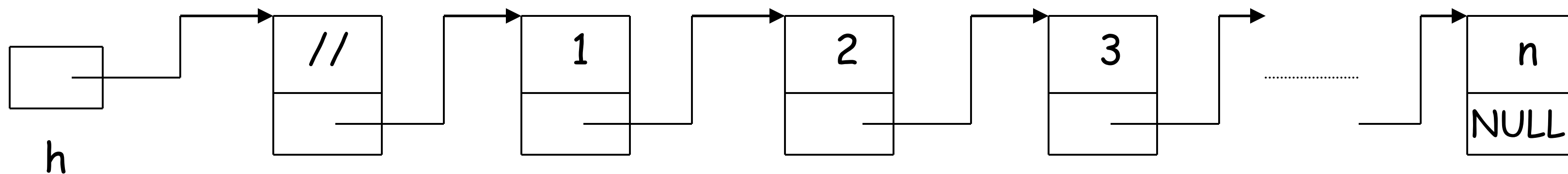
链表程序设计实例-2(续)

```
    u = h;  /* 寻找插入点 */
    while (u != NULL && n > u->value) {
        w = u;      u = u->next;
    }
    if (u == h)  h = p;
    else  w->next = p;
    p->next = u;
  }
  return h;
}
```

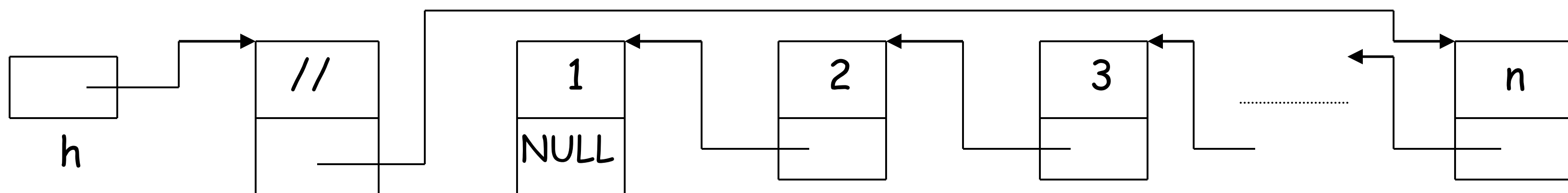
如果输入的整数是键值，程序该如何改?

链表程序设计实例-3

- 编制一个函数，实现将已知链表的表元链接顺序颠倒。即使链表的第一个表元变为最末一个表元，第二个表元变为最后第二个表元，……，最后一个表元变为第一个表元
  - 设链表为整数链表，且链表是带辅助表元的。下图表示链表颠倒前和颠倒后的情况



上(a) 颠倒前　　下(b) 颠倒后　　链表颠倒示意图



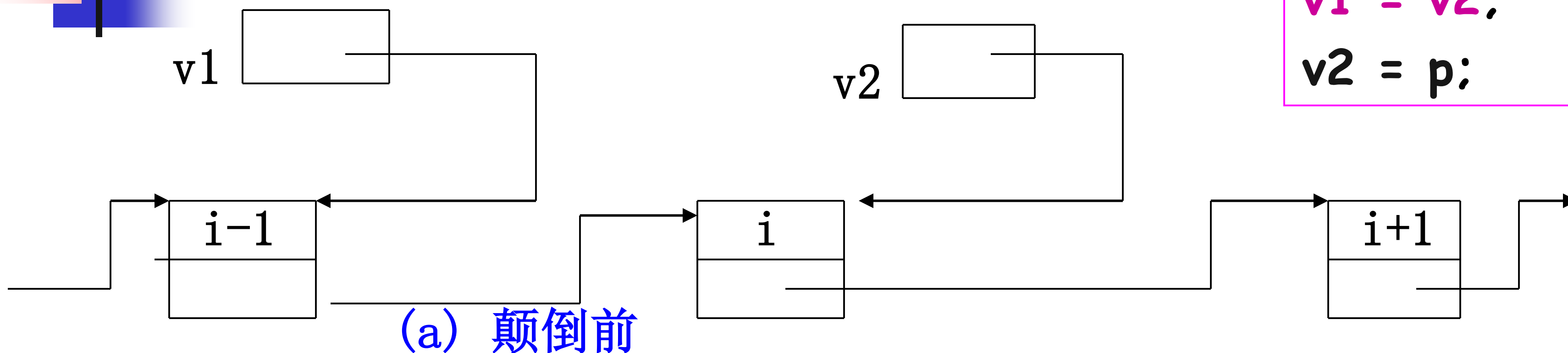
链表程序设计实例-3(续)

- 颠倒操作是一个循环过程，设从第一个表元开始颠倒，每循环一次颠倒一个表元的链接关系，直至最后一个表元颠倒完成。设某次循环前已颠倒了前(i-1)个表元， 本次循环欲颠倒第i个表元的链接关系。为实现从颠倒前状态变为颠倒后的状态，可用以下四个赋值操作实现：

```
p = v2->next; /* 保护 v2 所指表元的后继指针 */
v2->next = v1; /* 使*v2后继表元是v1所指表元 */
v1 = v2;          /* 调整 v1 */
v2 = p;           /* 调整 v2 */
```

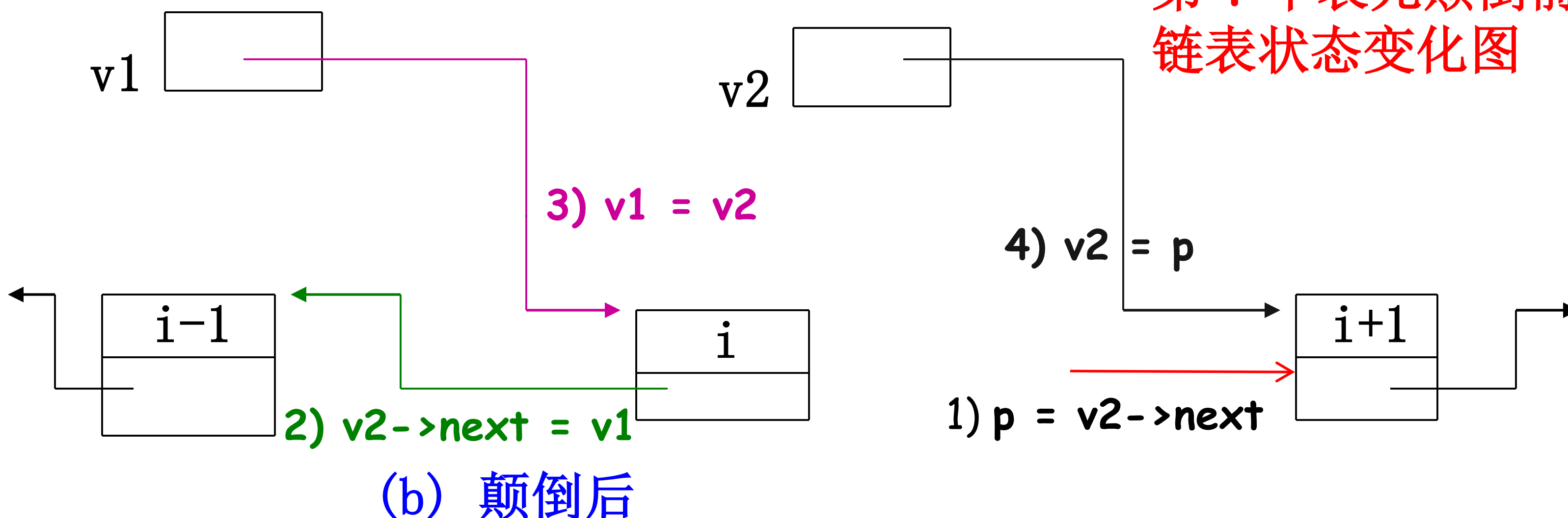
链表程序设计实例-3(续)

```
p = v2->next;
v2->next = v1;
v1 = v2;
v2 = p;
```



(a) 颠倒前

第 i 个表元颠倒前、后链表状态变化图



(b) 颠倒后

链表程序设计实例-3(续)

```
/* 带辅助表元的链表颠倒 */
void reverse (struct intNode *h)
{  struct intNode *p, *v1, *v2;
   v2 = h->next;  /* v2 指向链表的首表元 */
   v1 = NULL;  /* 开始颠倒时，已颠倒部分为空 */
   while (v2 != NULL) { /* 还未颠倒完，循环 */
     p = v2->next;  v2->next = v1;
     v1 = v2;       v2 = p;
   }
   h->next = v1;
}
```

链表程序设计实例-3(续)

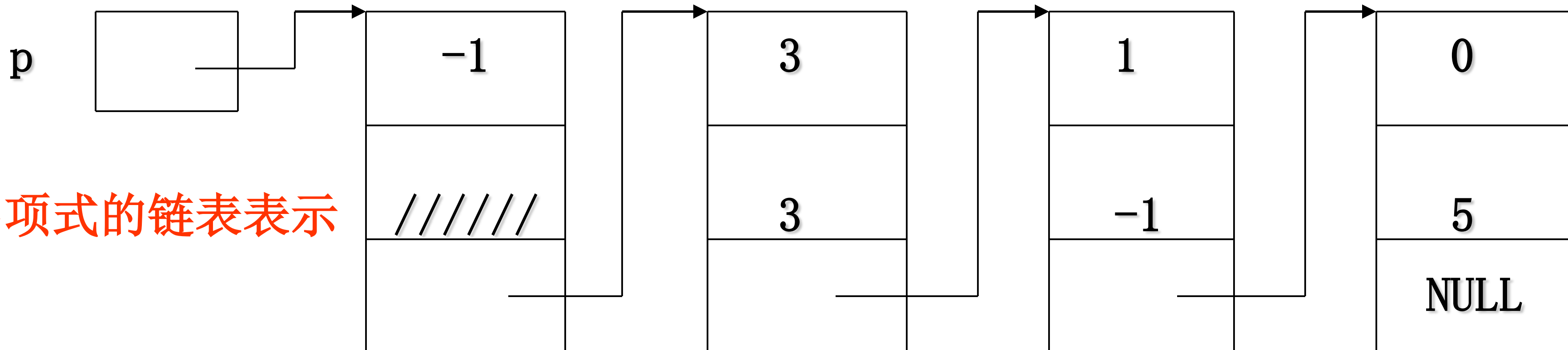
```
/* 不带辅助表元链表颠倒，已知链表首指针的指针 */
void reverse (struct intNode **hpt)
{  struct intNode *p, *v1, *v2;
   v2 = *hpt;  /* v2 指向链表的首表元 */
   v1 = NULL;  /* 开始颠倒时，已颠倒部分为空 */
   while (v2 != NULL) { /* 还未颠倒完，循环 */
     p = v2->next;  v2->next = v1;
     v1 = v2;       v2 = p;
   }
   *hpt = v1;
}
```

链表程序设计实例-3(续)

```
/* 不带辅助表元链表颠倒，已知链表首指针,返回链表首指针 */
struct intNode * reverse (struct intNode *h)
{  struct intNode *p, *v1, *v2;
   v2 = h;  /* v2 指向链表的首表元 */
   v1 = NULL;  /* 开始颠倒时，已颠倒部分为空 */
   while (v2 != NULL) { /* 还未颠倒完，循环 */
     p = v2->next;  v2->next = v1;
     v1 = v2;        v2 = p;
   }
   return  v1;
}
```

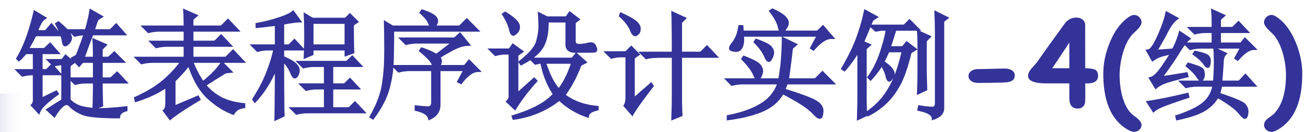
链表程序设计实例-4

- **编写一个多项式相加的函数。**一个多项式 $p_n(x)=a[n]*x^n+a[n-1]*x^{n-1}+ \dots + a[1]*x^1+a[0]$ 用一个链表来表示。如 $p(x)=3*x^3-1*x^1+5$ 用如下图所示的带辅助表元链表来表示。其中表元是由幂次、系数和后继表元指针三个成分组成的结构。现编写用这种形式表示的**两个多项式相加函数**addpoly()



链表程序设计实例-4(续)

- 设函数addpoly()实现 l(x) + k(x) => l(x)
- 引入两个指针变量 lpt 和 kpt，分别指向l(x)链表和k(x) 链表的当前考察项。从高次项到低次项，逐项考察它们的项的指数：
  - 如 (*lpt)的幂次与 (*kpt) 的幂次相等， 则 (*lpt) 的系数变为它们的和。但当和为零时， 项 (*lpt) 应从链表中删去。
  - 若 (*lpt) 的幂次大于 (*kpt) 的幂次，则 (*lpt) 不变。
  - 若 (*lpt) 的幂次小于 (*kpt) 的幂次，则在 l(x) 的链表中插入一项， 其幂次与系数均与 (*kpt) 的相同




```c
#include  <stdio.h>
#define EPSILON  1.0e-5
#include <math.h>
struct node  {
   int      power;
   double  coef;
   struct  node *link;
};
void addpoly(struct  node *l, struct  node *k)
{ struct node *p, *lpt, *kpt, *q;
  p = l;  lpt  = l->link;   kpt  = k->link;
```

链表程序设计实例-4(续)

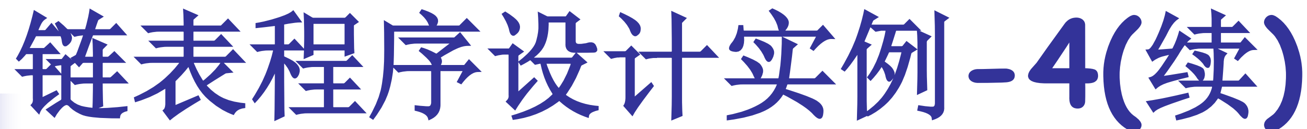
```
while (kpt) {
   if (lpt == NULL)  { /* 第一个多项式为空 */
     q=(struct node *)malloc(sizeof(struct node));
     q->power = kpt->power;
     q->coef  = kpt->coef;
     q->link  = NULL;   p->link  = q;
     p        = q;      kpt      = kpt->link;
   }
   else if (lpt->power == kpt->power){
     lpt->coef+=kpt->coef; /*等幂次项系数相加*/
     if (fabs(lpt->coef) <= EPSILON)  {
        p->link = lpt->link;  free(lpt); /* p是最后一个表元的前驱 */
     }
    else  p = p->link;
```

链表程序设计实例-4(续)

```
    lpt = p->link;   kpt = kpt->link;
  }
  else if (lpt->power > kpt->power) {  /* 跳过 (*lpt) */
       p = lpt;    lpt = lpt->link;
  }
  else { /*lpt->power<kpt->power 复制(*kpt)，插在*p之后*/
    q = (struct node *)malloc(sizeof(struct node));
    q->power = kpt->power; q->coef  = kpt->coef;
    p->link  = q; q->link  = lpt;
    p = q;  kpt = kpt->link;
  }
 }
  return ;
}
```

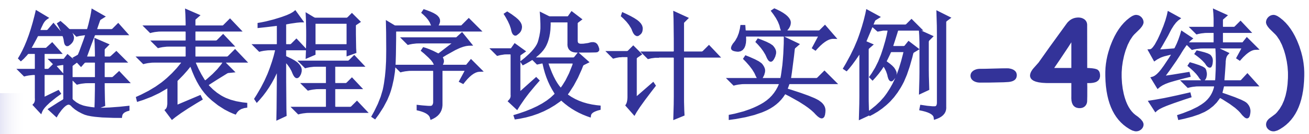
链表程序设计实例-4(续)

```
struct node *create_list() /* 产生一个多项式链表 */
{  struct node *u, *w, *p, *h;    int  n;
   /* 建立空的带哨兵链表 */
   h = (struct node *)malloc(sizeof(struct node));
   h->link = NULL;
   printf("Input data.(less zero: finish)\n");
   scanf("%d", &n);  /* 输入幂次 */
   while (n >= 0) {  double  coef;
     p = (struct node *)malloc(sizeof(struct node));
     p->power = n; scanf("%lf", &coef); /* 输入系数 */
     p->coef = coef; w = h; u = h->link;
     while (u != NULL && n < u->power) {/* 幂次从高到低 */
        w = u;  u = u->link;
     }
```



链表程序设计实例-4(续)

```
    w->link = p;   p->link = u;
    scanf("%d", &n);  /* 输入幂次 */
  }
  return h;
}
void main()
{ struct node *h1, *h2, *p, *q;
  h1 = create_list();   h2 = create_list();
  addpoly(h1, h2);  q = h1->link;
  while (q){ printf("%d  %f\n", q->power, q->coef);
             q = q->link;
  }
```



链表程序设计实例-4(续)

```
q = h1;
while (q)   {
   p = q->link;   free(q);  q = p;
}
q = h2;
while (q) {
  p = q->link;   free(q);       q = p;
}
}
```

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

联合

- 在某些特殊应用中，要求某些数据对象在程序执行的不同时期能存储不同类型的值。如某高级语言解释程序，可能需要一个变量表，用于存储变量的值。因变量有整型、字符型或浮点型之分，该成分应能根据需要或存储整数、或存储字符、或存储浮点数。联合就是迎合类似这种需要而引入的
- 联合使在同一内存区域中，几种不同类型的值选其中之一存放。如联合的成分可能是一个整数、或是一个字符，或是一个实数，它们占用同一个存储区域，由最近放入的内容决定该区域究竟是整数、还是字符、或是浮点数
- 存于联合变量中的只能是一种数据，联合是多种数据类型值的覆盖存储。几种不同类型的数据从同一地址开始存储，但任一时刻只存储其中一种数据，而不是同时存放多种数据。分配给联合的存储区域大小，要求至少能存储其中最大一种数据

联合类型定义

- 定义联合类型的一般形式为：

```
union  联合类型名 {
      成分说明表
};
```

- 如下面定义的联合类型(**union udata**)能存储整型，或字符型，或浮点型的数据：

```
union udata {
  int   ival;
  char  chval;
  float fval;
};
```

- 定义了联合类型**union udata**，就可用该联合类型定义变量

联合类型定义(续)

- 如 union udata x, y, g;
- 也可将联合类型定义与变量定义合在一起，如

```
union  udata  {
  int     ival;
  char    chval;
  float   fval;
 } x, y, g;
```

- 如类型名不再被引用，上述**union**之后的标识符**udata**还可省略。从以上解释看到，联合与结构的定义形式非常相似。但它们的含义是不相同的

引用联合成分

- 引用联合成分的书写形式也类似于引用结构成分的书写形式
  - 如 x.ival (视联合变量 x 为 int 型的)
    y.chval (视联合变量 y 为 char 型的)
    g.fval (视联合变量 g 为 float 型的)
- 联合是把同一存储区域当作不同类型的变量来使用
  - union udata联合类型可用作解释程序的变量表中表元素某个成分的类型。假设变量只有整型，字符型和浮点型。联合能存储几种类型中的任何一种类型值，不管赋的是那一种类型的值，总是占用联合的全部存储空间

引用联合成分(续)

- 联合也可出现在结构和数组中，联合也可包含有结构和数组。引用结构中的联合，或联合中的结构的书写形式与引用嵌套结构成分的书写形式一样

```
struct {
    char  name[30];     /* 标识符 */
    int   class_flag;   /* 标识符属性类别 */
    int   uflag; /* 存于联合成分中的值的类型 */
    union               /* 存储变量值 */
    { int    ival;  /* 当变量为整型时 */
      char   chval; /* 当变量为字符型时 */
      float  fval;  /* 当变量为浮点型时 */
    } uval;
  } sym_tbl[1000];  /* 变量表 */
```

定义了一个结构数组 sym_tbl

用sym_tbl[50].uval.fval引用结构数组sym_tbl中的第50 个结构的联合成分uval的fval(视其中的联合为浮点型数据)

使用联合变量注意事项-1

- 一个联合可以存放多种不同类型数据，但在每一瞬间只能存放其中一种数据，而不是同时存放多种数据。存于联合中的值是最近一次存入的值。存入新值后，原有的值就全部或部分被新值所覆盖，原有的值就不再存在
  - 如有以下赋值：

```
x.ival = 1;
x.fval = 2.0;
x.chval= '?';
```

  - 完成以上三个赋值后，只有**x.chval**是有效的，以**x.ival**或**x.fval**引用其值已经变成不确定的了

使用联合变量注意事项-1(续)

- 实际使用联合时，为了记住最近存于联合中的值的类型，通常另外设有一个存于联合中的数据的类型标志量。如前面变量表**sym_tbl**的结构成分中的**uflag**。下面的代码输出变量表中表元**sym_tbl[50]**的值

```
switch (sym_tbl[50].uflag)  {
    case INT: printf("INT VALUE   = %d\n", sym_tbl[50].uval.ival);
              break;
    case CHAR: printf("CHAR VALUE  = %c\n", sym_tbl[50].ural.chval);
              break;
    case FLOAT: printf("FLOAT VALUE = %f\n", sym_tbl[50].uval.fval);
              break;
    default: printf("BAD TYPE \n");
   }
```

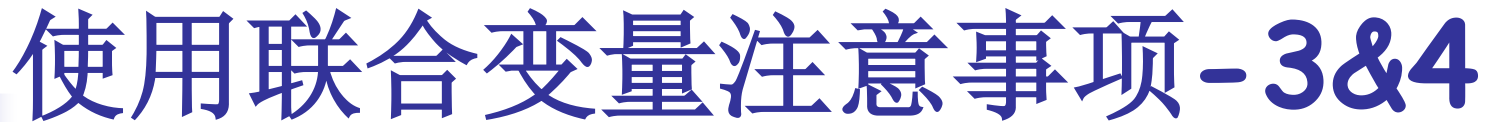
在描述中，假定符号INT，CHAR，FLOAT是某处已定义的常量，成分uflog只能取这三个值之一，分别代表变量取整型、字符型或浮点型值

使用联合变量注意事项-2

- 联合变量的开始地址和其成分变量的开始地址相同。如&x，&x.ival，&x.chval 都是同一地址值
  - 利用联合的所有成分都从同一地址开始存储的规定，可实现长数据的自然分拆
  - 如有

```
union {
    char s[4]; long lint;
} u;
long n;
…
n = -5L; u.lint = n; /* 给u赋long型值 */
```

  - 则u.s[0]、u.s[1]、u.s[2]、u.s[3]是对long型值按字节的分拆



使用联合变量注意事项-3&4

- 定义联合类型的变量时，也可为变量给出初值，对联合的初始化只能对其成分表中列举的**第一个成分置初值**
- 函数的形参不能是联合类型，函数的结果也不能是联合类型。但指向联合的指针可以作为函数形参，函数也可以返回指向联合的指针

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

位域

- 要标志某些对象的状态或特征,可用机器字中的一位二进制位或连续若干二进制位代表不同属性的状态。例如，某台计算机配置的磁盘机中的控制状态寄存器的字长为16位(自右至左，第0位至第15位)。设其中某些位的意义如下：

  第15位：置 1 表示数据传送发生错误；

  第7位 ：置 1 表示设备已准备好，可传送数据；

  第6位 ：置 1 允许响应中断；

  第2位 ：置 1 表示读；

  第1位 ：置 1 表示写。

- 实现上述要求可以给对应字中的某些二进位定义一系列表示特征的代码

位域(续)

```
#define  ERROR 0100000 /* 对应第 15 位错误标志 */
#define  READY    0200  /* 对应第 7 位准备好 */
#define  IENABLE 0100  /* 对应第 6 位允许中断 */
#define  READ     04       /* 对应第 2 位读 */
#define  WRITE    02       /* 对应第 1 位写 */
```

符号表中,为了区分每个标识符的类别属性，如变量名，函数名，类型名，及全局的，静态的等。可在描述类别属性字符中的某些二进位作为标志位使用。例如

```
#define  VARIABLE  01  /* 第 0 位表示变量名 */
#define  FUNCTION  02  /* 第 1 位表示函数名 */
#define  TYPE      04  /* 第 2 位表示类型名 */
#define  EXTERNAL  010 /* 第 3 位表示外部的 */
#define  STATICAL  020 /* 第 4 位表示静态的 */
```

位域(续)

- 通常称这种表示法为字位标志法，所有的数字必定是2的若干次幂。对这些位可以进行移位、屏蔽、求补等运算，就能实现对属性值的测试，存储等
  - 如语句 flg = VARIABLE | EXTERNAL | STATICAL; 表示将**flg**置成变量，全局，静态的
  - 而语句 flg &= ~(VARIABLE | EXTERNAL | STATICAL); 将**flg**置成非变量，非全局，非静态的。
- 更好的方法是利用C语言提供的位域机制，能直接定义和存取字中字段。字段是机器字存储单元中的一串连续的二进位。字段的定义和存取的方法建立在结构基础上

位域(续)

- 字段是一种特殊的结构成分，说明该成分只要在成分名后附上冒号‘:’后随一个数字，用来指出该字段有几个二进位

```
struct id_atr {
  unsigned variable  :1;
  unsigned function  :1;
  unsigned type      :1;
  unsigned external  :1;
  unsigned statical  :1;
};
```

  - 设有变量定义 struct id_atr flg ;
  - flg包含五个字段，其中每个字段的长度均为1。为特别强调它们是无正负号的量，定义它们是unsigned型的

位域(续)

- 又如用字段把某台计算机的指令字定义为

```
struct ins_type {
    unsigned op :6;          /* 操作码占前 6 位 */
    unsigned flg :2;         /* 特征码占 2 位 */
    unsigned operand1 :4; /* 第一操作数占 4 位 */
    unsigned operand2 :4; /* 第二操作数占 4 位 */
} instruction ;
```

- 引用字段的方法类似于引用结构的成分。如flg.variable、instruction.op分别表示引用flg的variable字段和引用instruction的op字段

位域(续)

- 另外，字段可以认为是一个无符号整数，像别的整数一样可出现在算术表达式中
  - 如

    ```
    flg.variable = flg.external = flg.statical = 1;
    ```

    将**flg**的相应位设置成**1**
  - 在以下形式的**if**语句中，条件

    ```
    if (flg.extermal==0 && flg.statical==0) …;
    ```

    是测试**flg**的相应位是否都为**0**

位域使用注意事项

- 一个字段只能在同一个整数字中，即限制字段不能跨越整数字的边界。如果剩余的位太少不够下一个字段时，下一个字段将占用下一个整数字
- 另外，字段可以不命名，称作无名字段，无名字段用于填充。 定义无名字段时，只有冒号和占用的位数。特别地，当无名字段的位数为0时，表示下一字段占用下一整数字，与整数字边界对齐
- 使用字段时，要注意具体机器分配字段的方向，有的从左向右，也有的从右向左
- 另外，约定字段值无符号和字段无地址，不能对字段施取地址运算(&)

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

枚举

- 除数字，文字信息之外，还有专用名称信息，如反映电梯运行状态的有上(UP)，下(DOWN)，停(STOP)；又如表示星期几的名称等。为提高程序描述问题时的直观性，引入枚举类型的机制
- 程序用枚举方法列举一组标识符作为枚举类型的值的集合。当一个变量具有这种枚举类型时，它就能取枚举类型的标识符值
- 枚举类型定义的一般形式为：

```
enum   类型名   {标识符1，标识符2，…，标识符n};
```

- 例如定义枚举类型enum weekday

```
enum weekday {SUN, MON, TUE, WED, THU, FRI, SAT};
```

- 用它可以定义变量，如

```
enum  weekday  today, yesterday, tomorrow;
```

  - 定义枚举类型enum weekday的变量today，yesterday，tomorrow。它们能取SUN到SAT之一的值，如

```
today     = SUN;
tomorrow  = MON;
yesterday = SAT;
```

- 也可以在定义枚举类型同时，定义枚举类型变量，如

```
enum {RED, YELLOW, BLUE} color;
```

枚举常量

- 枚举类型中的标识符称为枚举常量，每个标识符都表示一个有意义的值。对于编译系统来说，枚举类型中的标识符只是一组互不相同的标识符而已，标识符本身的字面意义只是供阅读程序的人便于理解程序。编译系统将枚举类型中的标识符值作为常量处理，故称枚举常量。程序不能对它们赋值
  - 如 SUN = 0 或 SAT = 6 都是错误的
- 为了便于处理枚举类型，编译系统将每个枚举常量与一个整数相联系，即枚举常量在内部被视作一个整数，值的大小由它们在枚举类型中出现的顺序确定，依次为 0，1，2，…
  - 如上面定义，SUN 值为0，MON值为1，…，SAT 值为6
  - 如有赋值语句 today = SUN; 使变量today的值为0

枚举

- 枚举类型变量的值也可输出。如
  - printf("%d\n"，today); 将输出整数0
- 枚举常量的对应整数也可由程序直接指定
  - 如

    ```
    enum weekday {SUN=7,MON=1,TUE,WED,THU,FRI,SAT};
    ```
  - 指定SUN为7，MON为1，后面未指定对应整数的枚举常量所代表的整数，是前一个枚举常量代表的整数加1。所以在上述定义中，TUE为 2，…，SAT 为 6

枚举

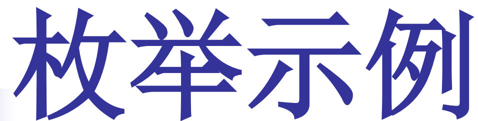
- 因枚举常量代表一个整数，同一枚举类型的变量与常量或整数可以作关系比较。比较时，以枚举常量所代表的整数为基础
  - 如 `if (today == SUN) yesterday = SAT;`
- 通常枚举类型变量有与整型变量一样的使用方法。但为了强调枚举类型变量值的特别含义，当一个整型表达式的值赋给枚举类型变量之前，应插入类型强制转换，将整值转换成枚举常量后再赋值。但是多数C语言的编译对此没有强制类型转换要求

枚举

- 如在TURBO C或HP机上的C系统中，下面C程序将输出
  Red = 5, Yellow = 6, x = 1, y = 3, z = -1

```
#include <stdio.h>
void main()
{ int i = 2, j = 1;
  enum  en {RED = 5, YELLOW, BLUE} x, y, z;
  x = i-j;    y = j+i;    z = j-i;
  printf("Red=%d,Yellow=%d,x=%d,y=%d,z=%d\n",RED,YELLOW,x,y,z);
}
```

- 使用枚举类型，除能命名反映其含义的标识符外，对标识符值的内部实现，程序员可以不必考虑。另外，一个变量具有枚举类型，还能反映变量值的有限性



枚举示例

- 枚举类型变量常用于循环的控制变量，枚举常量用于多路选择控制情况。如下面的C代码实现读入每月的月收入金额，求出年收入总金额。

```
{
  enum  {Jan = 1,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec} month;
  int  yearearn, monthearn;
  for(yearearn=0,month=Jan;month<=Dec;month++){
    printf ("Enter the monthly earning for ");
    switch  (month) {
      case Jan : printf("January.\n");  break;
      case Feb : printf("February.\n"); break;
      case Mar : printf("March.\n");    break;
      case Apr : printf("April.\n");    break;
```

```
    case May : printf("May.\n");      break;
    case Jun : ptintf("June.\n");     break;
    case Jul : printf("July.\n");     break;
    case Aug : printf("August.\n");   break;
    case Sep : printf("September.\n");break;
    case Oct : printf("October.\n");  break;
    case Nov : printf("November.\n"); break;
    case Dec : printf("December.\n"); break;
   }
   scanf("%d", &monthearn);   yearearn += monthearn;
   printf("\n");
  }
 printf("The total earnings for the year are %d\n", yearearn);
}
```

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

类型定义

- 在使用标准类型名定义或说明变量时，可只写标准类型名指明变量的数据类型。而用结构，联合，枚举等类型定义或说明变量时，要冠以表明数据类型类别的关键字，如 **struct**、**union**、**enum** 等
- C 语言提供类型定义**typedef**为类型命名的机制
- 用 **typedef** 定义新类型名后，对于结构，联合或枚举类型，使用它们定义或说明变量时不再冠类型类别关键字

类型定义示例

```
typedef struct {
    int  num;
    char *name;
    char sex;
    int  age;
    int  score;
} STD_TYPE;  /* 结构类型 STD_TYPE */
typedef int INTARRAY[20];/*含20个整数的数组类型INTARRAY*/
typedef enum{RED,YELLOW,BLUE} COLOR; /*枚举类型*/
```

利用以上类型定义，可定义变量如下：

```
STD_TYPE  std1, std2;  /* 定义两个结构变量 */
INTARRAY  v1,v2; /* 定义两个各含20个整数的数组 */
COLOR    c1, c2;     /* 定义两个枚举变量 */
```

类型定义

- 在以上变量定义中，对于结构，枚举等类型，不必再冠相应的类型类别关键字。特别对于数组类型，当有多个数组变量成分类型相同，数组元素个数也相同时，先用 **typedef** 定义一个数组类型，然后再定义数组变量就比较方便，简洁
  - 如类型定义 `typedef int ARRAY[100];`
  - 可用类型 ARRAY 定义多个相同模式的数组，如

    `ARRAY x, y, z;`
  - 若不用由类型定义提供的类型名 ARRAY，等价的定义是

    `int x[100], y[100], z[100];`

类型定义之别名

- 用 typedef 除能在定义新类型时给类型命名外，也可以给已有类型命名一个别名。如类型定义

```
typedef int   INTEGER;
typedef float REAL;
```

- 给类型 int 指定别名 INTEGER，给类型 float 指定别名 REAL。在能用 int 的地方可用 INTEGER，能用 float 的地方可用 REAL 代替。如以下变量定义

```
int   n, m;   float  x, y;
```

- 可等价地用以下变量定义代替：

```
INTEGER  n, m;    REAL     x, y;
```

类型定义之别名

- 为已有类型定义别名是为了让程序员能使用更习惯的类型名。也有为类型定义别名，使类型名更能迎合该类型的意义
  - 如程序中变量 counter 用作计数，通过以下类型和变量定义，可能会更能反映变量 counter 的意义

```
typedef int COUNT;
COUNT  counter;
```

- 给类型定义别名也能提高程序的可移植性
  - 如A计算机系统中，int型数据用2个字节，long型数据用4字节。B计算机系统中，int和long型数据都用4个字节。若将B系统上的C程序正确移植到A系统上，需要将程序中所有int改为long

类型定义

- 若原来的C程序有类型定义 typedef int INTEGER; 且程序中都用**INTEGER** 定义整型变量，而没有用**int**。则移植时只需修改关于类型 **INTEGER** 的定义，如改为

  typedef long INTEGER;
- 通常，在组织复杂的程序时，不同源程序文件中用到的同一数据类型，如数组，结构，联合，指针等，常用**typedef**定义给有关数据类型命名，并将这些类型定义单独放在一个源文件中，凡要用到它们的源文件，就用**#include**预处理命令将它包含进来
- C语言提供的**typedef**类型定义更重要的应用是为了清晰构造各种复杂的数据结构

类型定义

- C语言提供了丰富的数据结构构造机制，并利用指针，通过反复应用这些构造手段，能构造各种复杂数据结构，为程序设计带来很大方便。但是在程序中如何清晰明了地定义复杂结构也是程序员在实际编程时应注意的问题
  - 比如，程序中变量apt被定义为

    ```
    char (*(*(* apt)[5])())[6];
    ```

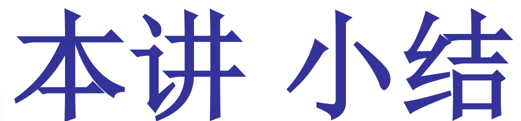
  - 它表示 apt 是一个指向指针数组的指针，其中指针数组有5个元素、元素是指向函数的指针，而它能指向的函数返回值类型是一种指针类型，指向有6个元素的字符数组。这种通过反复构造构成的类型比较难于直观理解，若用类型定义，由已定义的类型定义新类型，通过逐步定义就会比较好理解

类型定义

- 如对上述定义可写成以下一系列的类型定义

```
typedef char CHA[6];/* 有6个元素的字符数组类型 */
typedef CHA *CHAP; /* 指向CHA的指针类型 */
typedef CHAP CHAPF();/*返回CHAP类型值的函数类型 */
typedef CHAPF *CHAPFP; /* 指向CHAOF的指针类型 */
typedef CHAPFP CHAPFPA[5];/*元素类型为CHAPFP有5个元素的数组
    类型*/
typedef CHAPFPA *CHAPFPAP;/*指向CHAPFPA指针类型 */
CHAPFPAP apt; /* 说明类型为CHAPFPAP的变量 */
```

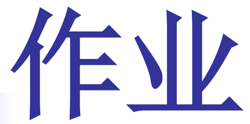
- 程序用typedef定义类型，只是为类型命名，或为已有类型命名别名。作为类型定义，它只定义数据结构样板，并不要求分配存储单元。用typedef定义的类型名来定义变量与直接写出变量的类型再定义变量具有完全相同的效果



- 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

第七章 小结

- 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



作业

- 习题七：第6-15题