有关的C语言问题

《数据结构》中用得最多的C语言概念：

指针和结构，及其相互关系

动态存储分配

函数，指针参数和结构参数

下面通过一些例子说明相关的问题

定义结构类型

```
typedef struct SeqList {
    int n;          /* 元素个数n < NMAX  */
    DataType  elements[NMAX];
} SeqList;      /* SeqList 成为一个自定义类型名 */
```

定义指向结构类型的指针类型：

```
typedef  SeqList  *PSeqList; /* PSeqList 是新定义的指针类型
```

定义和使用自定义类型的变量：

```
SeqList list1;   PSeqList pl = &list1;
list1.n = 0;          (*pl).n = 0;       pl->n = 0;
/* 三个语句做了同样事情，"." 和 "->" 都是C运算符，p->n
与 (*p).n 意义相同。 */
```


也可以写：

```
SeqList list1;   SeqList* pl = &list1;
list1.n = 0;  (*pl).n = 0;   pl->n = 0;
```

定义指针类型 PSeqList 更方便，也容易保持程序的一致性

希望大家在写程序时根据需要，定义适当的类型

到处使用 struct …，等等都不好

教科书和课堂幻灯片里大量使用了 -> 运算符，表示从指向结构的指针出发，去访问结构的成员

自定义类型的类型名可以像C语言内部类型名（如 int、char 一样使用，用于定义变量、函数参数和返回值，写类型转换，用在各种类型定义中（包括用于 typedef ）

定义有递归性质的类型

如果某类型里的一个部分引用了这个类型本身，就说它具有递归结构。例如单链表结点里有指向同样结点的指针

C里用 struct 定义这种类型，需要用"不完全 struct 说明"，即只给出结构标志，不给出完整的结构描述。例：

```
typedef struct Node Node; /* 定义结点类型，不完全说明 */
struct  Node {                  /* 单链表结点结构 */
    DataType  info;
    Node * link;
};
typedef Node *PNode, *LinkList; /* 定义指针类型 */
```

课程幻灯片采用的是另一种写法，与这里的定义完全等价

另一常见（在一些教材里可以看到）的定义方式：

```
struct  Node {                  /* 单链表结点结构 */
    DataType  info;
    struct Node * link;
};
typedef struct Node Node, *PNode, *LinkList;
```

这样写也对，但不容易看清 link 成分的类型就是 PNode

这种定义方式中不需要使用"不完全 struct"说明

动态存储分配

C语言的动态存储管理功能由标准库提供，有关功能函数在标准头文件 <stdlib.h> 和 <malloc.h> 里声明

程序里要动态存储分配，就必须 #include 上述头文件之一

C 标准库的动态存储管理系统提供了4个函数，最常用的是：

```
void* malloc(size_t size); /* size_t 是一个无符号整数类型 */
```

分配一存储块，其中足以存放大小为 size 的数据，返回所分配的存储位置的地址；若不能完成分配就返回 NULL

```
void free(void *p);
```

将 p 所指的存储块释放。要求 p 的值必须是以前通过动态存储分配函数得到的块的地址

malloc 使用

使用 malloc 的基本方式：用 sizeof 计算所需存储块的大小，对得到的块指针做类型转换后赋给适当类型的指针变量。应检查分配是否成功。例：

```
PSeqList createNullList_seq( void ) {
    PSeqList plist = (PSeqList) malloc( sizeof(SeqList) );
    if (plist != NULL)
        plist->n = 0;                    /*分配成功，空表长度为0 */
    else
        printf("Out of space!\n"); /*分配失败*/
    return plist;
}
```

由 malloc 得到的存储块处于不定状态，应注意对其中必要的部分赋值，使相应的存储区处于正确状态。

free 的使用

通过动态分配得到的存储块，如果不再有用了，就应该及时释放。释放存储块的操作通过 free 完成。例：

```
PSeqList plist = (PSeqList) malloc( sizeof(SeqList) );
… …
free(plist);
/* 注意，释放了 plist 指向的块之后，就不能再通过 plist 间接访问了（除非通过赋值让它有了正确的所指） */
```

如果一个指针（当前）的值不是通过 malloc 等得到的地址，就不能对它使用 free!!!

计数并清零的分配函数

```
void* calloc(size_t n, size_t size);
```

分配一存储块，其中足以存放 n 个大小为 size 的数据，将存储块中所有单元清零，返回所分配的存储位置的地址；若不能完成分配就返回 NULL。常用于分配“动态数组”

分配调整函数

```
void* realloc(void* p, size_t size);
```

重新分配函数，分配足以存放大小为 size 的数据的存储块，返回其地址。如果这个块不小于 p 所指的块，那么保证该块前面部分保存的内容与 p 所指的块一样；如果这个块比 p 所指的块小，那么保证其中保存了 p 所指块内容的前一部分。如分配完成，p 的值不能再用；若分配无法完成则返回 NULL，并保证 p 所指的块内容不变

函数与结构

使用函数对结构进行操作有两种基本方式：

1. 为函数定义结构参数和/或结构返回值
2. 为函数定义结构指针参数和/或结构指针返回值

结构指针：即指向结构的指针

两种方式语义不同，各有适用之处。我们需要弄清其语义

注意：C 语言函数的参数都是值参数，调用时总是把实在参数（实参）的值赋给形式参数（形参），而后执行函数体

结构和指针参数也不例外，在调用函数时，对结构参数做的是结构赋值（把整个实参结构赋给形参），对指针参数做的是指针赋值（把实参指针赋给形参）

如果函数 f 的原型是：

```
..... f (PSeqList  pl) { … … }
```

调用函数 f 的正确形式：

```
SeqList L1; /* 定义顺序表 L1 */
..... f(&L1);
```

在 f 里可以完成对 L1 的任何操作

如果函数 g 的参数用 SeqList 类型，调用形式

```
… … g (SeqList sl) { … … } … …
  SeqList L1; /* 定义顺序表 L1 */
  … … g( L1 );
```

调用时复制 L1 整体到 sl，函数里对 sl 的任何修改与 L1 无关

这种参数形式要做整个结构的赋值，不能完成修改结构的操作

几个表操作的例子

```
PSeqList createNullList_seq( void ) { /*创建空顺序表*/
    PSeqList palist = (PSeqList)malloc(sizeof(SeqList));
    … …
    return palist;
}
```

通过结构指针可以把函数里动态分配的结构传出来

```
int insert_seq(PSeqList palist, int p, DataType x) {  /* 表插入 */
    int q;
    … …
    for (q = palist->n - 1; q >= p; --q)  /* 元素逐一后移*/
        palist->elements[q+1] = palist->elements[q];
    palist->elements[p] = x;   palist->n++;
    return TRUE;
}

int delete_seq( PSeqList palist, int p ) { /* 表删除 */
   int q;    … …
   for (q = p + 1; q < palist->n; ++ q)         /* 元素逐个前移 */
       palist->elements[q - 1] = palist->elements[q];
   palist->n--;                                 /* 元素个数减1 */
   return TRUE;
}
/* 修改被操作的表本身，必须用结构指针参数 */
```


定义函数计算整数表中元素之和（假设表元素为整数）

```
int calc1 (SeqList sl) {
    int i, n, sum = 0;
    for (i = 0, n = sl.n; i < n; ++i) sum += sl.elements[i];
    return sum;
}

int calc2 (PSeqList pl) {
    int i, n, sum = 0;
    for (i = 0, n = pl->n; i < n; ++i) sum += pl->elements[i];
    return sum;
}
```

这两个定义都能完成工作。但它们的工作方式不同

调用 calc1 时将会复制整个表（结构），而调用 calc2 时只复制一个指针


```
int locate_seq(PSeqList palist, DataType x) { /* 求元素下标 */
    int q;
    for ( q = 0; q<palist->n; ++q)
        if (palist->elements[q] == x) return q;
    return  -1;
}

int locate_seq1(SeqList slist, DataType x) { /* 求元素下标 */
    int q;
    for ( q = 0; q < slist.n; ++q)
        if (slist.elements[q] == x) return q;
    return  -1;
}
```

对表这样的复杂结构，通常都应该用结构指针参数。幻灯片和教科书中的例子都是这样