第 1 章 实验概述

根据《数据结构》课程的技术特性，设置课程的同步实践环节是十分重要的。通过课程各知识点内容的实践训练，明确程序构造的原则与方法，加强程序设计的基本方法与实现的学习，突出构造性思维、模块化实现等训练的特征，对学生组织数据、数据结构及编写程序能力的提升有着极大的帮助作用。

本章将从《数据结构》课程实验的概念与基础、问题的求解与实现架构，以及数据结构在基于 C 语言实现的一般技术等方面进行阐述。

1.1 数据结构的基本概念与 C 语言的实现

1.1.1 数据元素

1．数据项

数据项（Data Item）用于描述客观事物的某种属性。因此，数据项是描述客观事物的最基本的、不可分的具有名称和值的数据单位，是具有独立含义的最小标识单位。

例如，姓名：张三，身高：1.73，这两个数据在 C 语言中的表示示例如下：

```
char  name[10];
float  height;
```

将其实例化，即赋值如下：

```
strcpy(name,"张三");
height=1.73;
```

2．数据元素

数据元素（Data Element）用于描述客观事物的综合属性，是数据项的集合体，在计算机中通常作为一个整体进行考虑和处理。我们通常称此时的数据元素为记录（Record）。

例如，学生是一类客观事物的存在，具有姓名和身高等属性。学生这样的客观事物在 C 语言中的表示示例如下：

```
struct person;                      // 结构 struct person 声明
typedef struct person student;      // 将结构 struct person 声明新类型名 student
struct person                       // 结构 struct person 定义
```

```
{   char  name[10];
    float  height;
};
```

将其实例化，即：

```
student stu1,stu2;                  // 定义结构型变量
strcpy(stu1.name, "张三");
stu1.height=1.73;
strcpy(stu2.name, "李四");
stu2.height=1.80;
```

1.1.2 数据与数据对象

1．数据对象

数据对象（Data Object）是性质相同的数据元素的集合，是数据元素关系的整体性逻辑描述。数据对象包含三个方面的内容，一是整体性，二是数据元素，三是数据元素间的位置关系。

例如，“一个班级全体同学的集合”是一个数据对象，在数据结构中可逻辑表示为一个线性表：((张三, 1.73), (李四, 1.80), …)。

2．数据

数据（Data）是计算机程序处理的基本对象，同一数据类型的数据元素的集合，是数据元素关系的整体性物理描述。

数据包含了数据对象整体性物理描述，数据对象和数据对象元素的集合定义，以及数据对象中每个元素在计算机中相互位置关系的表达方式。

例如，可以用定长顺序线性表表示“一个班级的所有同学的集合”，数据定义如下：

```
student stu[50];          // 类型 student 定义见 1.1.1
```

将其实例化，即：

```
strcpy(stu[0].name, "张三");
stu[0].height=1.73;
strcpy(stu[1].name, "李四");
stu[1].height=1.80;
```

从 student stu[50]中可看到具体的数据元素是 student 型的，具体的数据项是 name 和 height；每个数据元素的位置关系是通过数组 stu 的下标体现的，即位置关系的表达方式是数组下标。

又如，用链式线性表表示一个班级的所有同学，数据对象定义如下：

```
struct studentnode;                    // 声明结构 studentnode
typedef struct studentnode *pstu;      // 将结构指针定义为新类型 pstu
typedef struct studentnode stu;        // 将结构定义为新类型 stu
struct studentnode                  // 结构体的具体定义，即数据对象集合元素的内容结构
{   student Data;                   //集合元素中数据元素的定义，类型 student 定义见 1.1.1
    pstu  next;                     // 集合元素之间位置关系的定义，链式表结点的指针
}
```

将其实例化，即：

```
pstu pMyClass;      // 指针变量定义，即整体性定义，链式线性表首指针
pstu p;             // 指针变量定义，即集合元素定义
p=(pstu)malloc(sizeof(stu));
strcpy(p->name, "张三");
p->height=1.73;
p->next=NULL;
pMyClass=p;
p=(pstu)malloc(sizeof(stu));
strcpy(p->name, "李四");
p->height=1.80;
p->next=NULL;
pMyClass->next=p;
```

1.1.3　数据结构

数据结构（Data Structure）是指相互之间存在一种或多种特定关系的数据元素的集合及集合整体属性的描述。

数据结构比数据有更完善的描述能力，通常在含有数据描述的基础上，还包含关于集合层次的综合性相关描述。如在 1.1.2 节中，关于“一个班级全体同学的集合”的定长线性表的表示方法是：

```
student stu[50];
```

这种描述只是回答了某个班级的学生数据集合的数据类型是 student，最多是 50 位同学，但无法回答这个班到底有多少位同学，班级名称是什么。因此，具有整体描述功能的数据结构定义如下：

```
// 定义代码段
#define  MAXLEN=50
struct Class;
typedef struct Class OurClass;
struct Class
{  student stu[MAXLEN];
   int studentnum;          // 记录实际同学人数
   char classname[50];      // 班级名称
};
OurClass MyClass;
```

1.1.4　数据类型

数据类型（Data Type）是数据结构与其上一组操作的总称。一般，数据类型中定义了两个集合：一是该类型元素的取值范围，二是该类型元素允许使用的一组运算。

例如，在 C 语言中常用扩展名为.h 的文件存放数据定义及一组运算的声明，常用.c 文件存放运算具体实现的相关函数。

1．一个.h 文件示例

顺序表操作头文件——SeqList_OP.h

```
// 数据定义部分
struct Elem;
```

```
typedef struct Elem ElemType;
struct Elem
{   char  name[10];
    float  height;
};
struct SeqList;
typedef struct SeqList *PSeqList;
struct SeqList
{   int  MAXNUM;        // 顺序表中可以存放元素的个数
    int  n;             // 实际存放线性表中元素的个数，n≤MAXNUM
    ElemType *Data;     // Data[0]，…，Data[n-1]存放线性表中的元素
};
// 相关运算的声明
PSeqList createNullList_seq(int m);         // 产生一个有 m 个空间的空线性表
int isNullList_seq(PSeqList palist);        // 判断线性表是否是空表
int locate_seq(PSeqList palist, ElemType x); //找到数据元素 x 在线性表中的位置
int insertPre_seq(PSeqList palist, int p, ElemType x);
                                    // 在线性表中在 p 所指数据前插入数据元素 x
int deleteP_seq(PSeqList palist, int p);    // 删除线性表中 p 所指的数据
int countElementNum(PSeqList palist)        // 统计线性表中元素个数
```

2．一个.c 文件示例

顺序表操作源程序文件——SeqList_OP.c

```
#include "SeqList_OP.h"
#include "stdio.h"
#include "stdlib.h"
PSeqList createNullList_seq(int m)
{// 创建新的顺序表
PSeqList palist = (PSeqList)malloc(sizeof(struct SeqList));
     if (palist!=NULL)
     {    palist->Elem=(ElemType *)malloc(sizeof(ElemType)*m);
          if (palist->Data!=NULL)
          {    palist->MAXNUM=m;
               palist->n=0;           // 空表长度为 0
          }
           else
          {    free(palist);
               palist=NULL ;
           }
     }
     if(palist==NULL)
         printf("Out of space!\n");     // 存储分配失败
     return  palist ;
}// end createNullList_seq
int  isNullList_seq(PSeqList palist)
{// 判断 palist 所指顺序表是否为空表
return ( palist->n == 0 );
}
```

```
int  locate_seq(PSeqList palist, ElemType x)
{// 求数据元素 x 在 palist 所指顺序表中的下标
     ……
}
int insertPre_seq(PSeqList palist, int p, ElemType x)
{// 在线性表 palist 中，在 p 所指数据前插入数据元素 x
     ……
}
int  deleteP_seq(PSeqList palist, int p)
{// 在 palist 所指顺序表中删除下标为 p 的数据
     ……
}
int countElementNum(PSeqList palist)
{// 统计线性表 palist 中数据（即结点）个数
     ……
}
```

1.1.5　抽象数据类型

抽象的本质是抽取或提取反映问题的共性内容，而忽略问题的具体细节，这正是人们从事计算机研究的本质所在。

1．数据的抽象层次

计算机中使用的是二进制数，这是最基本的数据形式，是计算机硬件运算的基本数据形式。但是二进制的数据使用起来有三个不便：一是数据位数太长，不宜识读；二是单一的数据形式不方便表达丰富的事物；三是不符合人们的使用习惯。因此，人们开始采用了各种丰富的数据，如十进制数、十六进制数、八进制数等。这样，在低级语言汇编编程中表示数据时，不用再考虑二进制数的细节，而直接采用十进制数等表示法即可。

在高级语言中，出现了高一级的数据抽象：一是基本数据类型，如整型、实型、字符型、指针类型等；二是组合数据类型，如数组、结构体、集合等。而在数据结构的概念中，则出现了更高一级的数据抽象，如各种表、队列、栈、树、图、窗体、管理器等。

综上所述，到此为止的数据层次有三层，即：低级数据、高级数据和抽象数据。从下往上看是由低一级数据构成高一级数据；从上往下看是高一级的数据表现内容比低一级的数据表现内容更丰富，使用上更方便。

每一层的数据都有与之相关联的一组操作。

低级的数据类型及相关操作：低级的数据即计算机使用的二进制数据，其相关数据操作只有几条操作指令，如 ADD、SUB、MUL、DIV 等，低级数据的表达方式与操作指令由 CPU 生产商提供。

高级的数据类型及相关操作：C 语言的数据有基本数据类型，如 int、char、float 等；组合数据，如数组、结构体等。其相关的操作非常丰富，如 +、–、*、/、&等。这些数据表达方式与操作由高级语言提供商提供。

抽象的数据类型及操作：如顺序表的类型定义；其上的操作：如产生一个空线性表、增加一个数据、删除一个数据、判断线性表是否为空等，都由程序设计人员自行定义和实现。而如何在实际应用中对数据进行抽象和操作，正是数据结构课程研究的内容和目标。

2．抽象数据类型

抽象数据类型（Abstract Data Type，ADT）是指基于一类逻辑关系的数据类型，以及定义在这个类型之上的一组操作。

抽象数据类型的定义取决于客观存在的一组逻辑特性，而与其在计算机内部如何表示与实现无关，即不论其内部结构如何变化，只要它的数学特性不变，都不影响其外部操作。形式化的 ADT 的定义如下：

```
ADT <ADT 名>
{   数据对象：<数据对象的定义>
    结构关系：<结构关系的定义>
    基本操作：<基本操作的定义>
}ADT <ADT 名>
```

这种定义的形式，结合 1.1.4 节有关学生的信息，C 语言的对应实现如下。

（1）数据元素的抽象与具体

```
struct Elem;                        // struct Elem 声明
typedef struct Elem ElemType;       // 定义 ElemType 为 struct Elem 类型
struct Elem                         // struct Elem 定义
{   char name[10];
    float height;
};
```

在这个定义程序段中，有如下两个内容。

① 数据元素抽象方法：语句“typedef struct Elem ElemType; ”解决数据的抽象问题，后续使用数据元素可以用类型名 ElemType 直接定义。

② 数据元素具体方法：语句“struct Elem{ …… }; ”完成抽象数据的具体化。这种具体化要求在解决实际问题时，根据客观事物的属性，具体定义。

（2）数据结构关系定义

```
struct SeqList;                         // struct SeqList 声明
typedef struct SeqList *PSeqList;       // 定义 PSeqList 为 struct SeqList 指针类型
struct SeqList                          // struct SeqList 定义
{   int  MAXNUM;                        // 顺序表中可以存放元素的个数
    int  n;                             // 实际存放线性表中元素的个数，n≤MAXNUM
    ElemType *elem;                     // elem[0],…,elem[n-1]存放线性表中的元素
};
```

在这个顺序表的结构关系的定义中，包含了以下信息。

① 数据元素的表示。每个学生是 ElemType 类型的数据，是线性表中的元素。可定义为具体数据变量 Elem[i]，此处实现了抽象数据的使用。

② 数据元素的组织。以动态数组的形式组织学生数据，具体语句是：

```
ElemType  *elem;
```

③ 动态数组的大小。语句“int MAXNUM; ”回答了该问题。

④ 动态数组实际的大小。语句“int n;”记录了该值。

⑤ 如何操作顺序表。通过语句“typedef struct SeqList *PSeqList; ”

定义了一个数据类型 PSeqList，以后可以非常方便地通过该类型定义的具体变量操作这种类型的顺序表。此处实现了顺序表的抽象数据定义。

（3）基本操作定义

数据操作，一般以函数声明的方式表示。根据 C 语言的特点和方法，基本操作的声明一般放在头文件中（如 SeqList_OP.h）；而基本操作的实现，一般放在源程序文件中（如 SeqList_OP.c）。

思考：为什么在使用 typedef 时采用这种顺序：第一条是声明语句；第二条是 typedef 类型定义语句；第三条是定义语句。

1.2　问题的求解与 C 语言的实现架构

1.2.1　问题的求解与实现

一个问题的求解一般涉及四个方面的内容：一是数据元素的具体化与抽象化，二是数据的表示内容与表示方法，三是数据结构的表示内容与表示方法，四是与数据结构相关的操作。

1．数据元素的具体化、抽象化与使用

（1）数据元素的具体化

数据元素的具体化是具体数据项组合的过程，形成一个组合体。在 C 语言中，使用结构体的方法表示这个组合体，通常具体化的表示方法为：

```
struct  Elem          // 结构体 struct Elem 定义
{
    数据项            // 各种数据类型定义，即具体化
};
```

（2）数据元素的抽象化

数据元素的抽象化是将具体数据用一种形式表示，也就是通常的数据类型，以备后续使用。在 C 语言中，抽象化的表示方法为：

```
typedef  struct  Elem  ElemType;
```

（3）数据元素具体化与抽象化的综合定义过程

数据元素具体化与抽象化是在定义过程中连续而综合定义的，具体如下：

```
struct  Elem;                          // 声明结构体 struct Elem
typedef  struct  Elem  ElemType;       // 将结构体定义为新类型 ElemType
struct  Elem                           // 结构体 struct Elem 定义
{
    数据项                             // 各种数据类型定义，即具体化
};
```

（4）数据元素抽象化的说明

在事物的描述中，事物需描述的属性决定了该事物数据项的个数，当该数据项的个数超过 1 个，我们很容易使用上述结构体的方法将数据具体化和抽象化，当数据项个数只有 1 个时，此时可以不需要上述结构体的方法将其组合成一个数据。但为了保持数据使用方式的一

致性，建议采用上述结构体的方法，这样虽然描述复杂一点，但关联数据结构的基本算法可以不变。不管采用哪种数据表现形式，建议不要忘记了 ElemType 的作用，例如，数据只有一个数据项是整型数据，建议可使用如下语句将其抽象为 ElemType 数据类型：

```
typedef  int  ElemType;
```

（5）数据元素的使用

通过上面抽象化的定义，在 C 语言中就有了新的数据类型 ElemType，可以像使用一般的数据类型（如 int、char 等）一样使用，如 ElemType element。

这样，在高级程序设计语言基础上设计了适用于数据结构的抽象型数据元素类型 ElemType，而不用考虑数据元素类型的多样性。

2．数据的表示内容与表示方法

（1）数据的表示内容

数据需要表示两个内容：一是数据的数据元素域，二是数据间相互关联的位置域。

（2）数据的表示方法

数据元素域的类型表示方法利用 ElemType 定义类型，格式与 C 语言数据类型定义格式相同，例如：

```
ElemType  stuent;
```

数据间相互关联的位置域就是表示数据的位置或相互关联数据的位置。位置域有 4 种表示方法：一是顺序表表示法，二是链式表示法，三是块链表示法，四是链块表示法。此处重点介绍顺序表表示法和链式表示法。

① 顺序表表示法。顺序表表示法就是一维数组表示，一维数组有静态数组和动态数组，通过数组下标实现数据的位置及数据间的相关位置。具体表示法如下：

```
ElemType  element[10];          // 静态数组表示法
ElemType  *element;             // 动态数组表示法
```

② 链式表示法。链式表示法就是通过指针实现数据位置与指针域的具体指向实现数据间的相关位置。例如，链式线性表的数据（即结点）表示法如下：

```
struct Data;                    // 声明结构体 struct Data
typedef struct Data *Pnode;     // 将结构体指针定义为新类型 Pnode
typedef struct Data  Node;      // 将结构体定义为新类型 Node
struct Data                     // 结构体 struct Data 定义
{  ElemType element;            // 数据元素域定义
   Pnode next;                  // 位置域，即指针域、链域
};
```

3．数据结构的表示内容、表示方法与使用

（1）数据结构的表示内容

数据结构的表示内容包含两个方面的内容：一是数据群体的属性描述，二是每个数据的描述。

（2）数据结构的表示方法

在 C 语言中，一般采用结构体的方式描述。例如，顺序表数据结构如下：

```
struct SeqList;                          // 声明结构体 struct SeqList
typedef struct SeqList  *PSeqList;       // 将结构体指针定义为新类型 PSeqList
struct SeqList
{// 数据群体的属性
    int  MAXNUM;                         // 顺序表中可以存放数据的个数
    int  n;                              // 实际存放线性表中数据的个数，n≤MAXNUM
    // 数据属性
    ElemType  *element; // element[0]，…，element[n-1]存放线性表中的数据
};
```

在这个结构体中，MAXNUM 和 n 属于这个顺序线性表数据结构的群体属性的描述，MAXNUM 记录的是该顺序线性表规定最大的数据个数，n 记录的是该顺序线性表中实际的数据个数；而 element 则表示了该群体中每个数据的属性，即数据的数据元素类型是 ElemType 型，element 是 ElemType 型指针，即每个数据的位置是指针 element 加下标。

（3）数据结构的使用

通过语句“typedef struct SeqList *PSeqList”的抽象化定义，在 C 语言中就具有了新的数据类型 PSeqList，可以像使用一般的数据类型一样使用，例如：

```
PSeqList list1;
```

4．数据结构相关的操作

数据结构的操作是对该数据结构完成某种功能，具体的思想称为算法，具体的表现形式为 C 语言中的函数。

与数据结构相关的算法又可以分为两类：一类是针对所使用的数据结构的基本算法，另一类是针对具体问题的求解算法，也称为实例算法。

（1）基本算法

对于任何一种数据结构（如线性表、二叉树等），基本算法就是由数据结构本身依赖的生成、判断、增加、修改、删除、查找等算法。

这类算法涉及的都是抽象化的数据和数据元素，一般不用考虑数据元素中数据项的特例，因此，这类算法对同一种数据结构是通用的。

（2）实例算法

对于任何一种数据结构，在其基本算法的基础上，根据解决问题的要求而必须实现的一系列具体的求解算法，即实例算法，才能真正实现问题的求解。

这类算法一般会涉及数据元素中具体的数据项。

1.2.2　C 语言的实现架构

C 语言的实现架构一般由数据结构实现与实例实现两部分组成。

1．数据结构的 C 语言实现

数据结构的实现包含数据结构的定义和基本算法的实现。一般数据结构的定义包含在.h 文件中，基本算法的实现在相应的.c 文件中，两个文件取同名。以下仍以本书 1.1.4 节介绍的顺序表描述。

SeqList_OP.h 和 SeqList_OP.c 两个文件构成了数据结构的抽象数据类型（ADT）。

（1）SeqList_OP.h 文件内容

① 具体数据元素定义与其抽象

```
struct Elem;
typedef struct Elem ElemType;
struct Elem
{
    ……
};
```

② 数据结构定义与其抽象

```
struct SeqList;
typedef  struct  SeqList   *PSeqList;
struct SeqList
{   int  MAXNUM;         // 顺序表中可以存放元素的个数
    int  n;              // 实际存放线性表中元素的个数，n≤MAXNUM
    ElemType *element;  // element[0]，…，element[n-1]存放线性表中的元素
};
```

③ 相关的基本功能函数（算法）声明

此处声明的功能函数是数据结构相关的基本功能，是提供给使用某种数据结构的编程人员使用的功能函数。

```
PSeqList createNullList_seq(int m ); // 产生一个有 m 个空间的空线性表
int isNullList_seq(PSeqList palist); // 判断线性表是否是空表
int locate_seq( PSeqList palist, ElemType x ); // 找到数据 x 在线性表中的位置
int insertPre_seq( PSeqList palist, int p, ElemType x );
                                        // 在线性表中 p 所指数据前插入数据元素 x
int deleteP_seq( PSeqList palist, int p );      // 删除线性表中 p 所指的数据
int countElementNum(PSeqList palist);           // 统计线性表中元素个数
```

④ 方便性、简明性、可读性等定义

这部分内容根据需要定义，一般放在.h 文件的最前面。例如，C 语言没有逻辑型数据，可使用如下宏定义：

```
#define  True  1
#define  False  0
```

（2）SeqList_OP.c 文件内容

```
// 第一部分：引用文件声明
include  "SeqList_OP.h"
……
// 第二部分：根据 C 语言的必要声明与定义
// 本部分的内容是仅供基本功能函数使用的，不提供给其他程序使用
……
// 第三部分：数据结构涉及的基本算法
PSeqList createNullList_seq(int m);
{// 产生一个长度为 m 的空表
    ……
}
```

```
int  isNullList_seq(PSeqList palist);
{// 判断线性表是否是空表
    ……
}
……
```

2．实例的 C 语言实现

一种数据结构可以表示多个具体的问题，即多个实例。如 1.1.3 节的班级数据类型 OurClass，即可以定义两个班级：

```
OurClass Class1,Class2;
```

根据问题的具体要求，实例算法在不同于基本算法的.c 文件中。根据问题的大小，可以定义多个.h 文件和多个.c 文件，但必须有且仅有一个含 main()函数的.c 文件，称之为主程序文件。假如实例算法的 C 语言实现是在一个.c 文件中，该文件一般由以下 3 部分组成：

① 引用数据结构的 ADT；

② 必须具有一个 main()函数；

③ 相关实例算法。

所以，以上述的顺序线性表为例，实例的 C 语言程序架构如下：

```
Include  "SeqList_OP.c"
……
相关函数声明语句
……
相关函数定义
……
main ()
{
……
}
……
```

1.2.3　应用程序主菜单的设计

应用程序的界面是与用户交流最直接的渠道。应用程序的界面设计应该遵循以下原则：使用方便、布局合理、颜色和谐、界面文字简洁规范。常见的应用程序界面有字符型界面和窗体型界面。本节简要介绍字符型界面的一般设计与实现方法。

在字符型界面中，用户只能通过键盘符号与计算机交流，这种风格的界面通常用于一些传统的面向过程的程序设计。用户通过选择输入程序窗口的提示字符，控制程序流程，以决定程序下一步的操作。

1．界面背景颜色设置

界面背景颜色设置需要调用函数 system(color)，并且包含头文件 stdlib.h。颜色属性由两个十六进制数字指定，第一个为背景色，第二个为前景色（字体的颜色）。

例如，要设置背景颜色为蓝底白字的模式，system 函数的设置格式为：

```
system("color 1f");
```

其中，color 表示颜色属性、1 代表背景色为蓝色，f 代表前景字体色为亮白色。表 1-1 是前景色和背景色取值含义的对照表，颜色的取值共有 16 种，从 0~F。函数中的两个数字，可以为表 1-1 中的任何数字。

表 1-1 颜色取值对照表

0	1	2	3	4	5	6	7
黑色	蓝色	绿色	湖蓝色	红色	紫色	黄色	白色
8	9	A	B	C	D	E	F
灰色	淡蓝色	淡绿色	淡浅绿色	淡红色	淡紫色	淡黄色	亮白色

2．界面背景大小设置

界面背景大小的设置，即调整系统控制台（Console）显示的宽度和高度，需要调用函数 system(mode con)，并且包含头文件 stdlib.h。

例如，要设置背景的大小为：长度（lines）35 行，宽度（cols）78 列。函数的设置格式为：

```
system("mode con: cols=78 lines=35");
```

此外，在菜单设计中，常用的 system 函数还有：

① system("cls")用于清屏，便于清除程序上一次在屏幕输出的内容；

② system("pause")用于冻结屏幕，便于观察程序的运行结果。

3．字符菜单设计实例

（1）假设程序界面的主菜单内容如下。

```
1: Red
2: Green
3: Blue
4: Yellow
5: Gray
0: Goodbye!
Please Input choose(0-5):
```

使用数字 0～5 选择菜单项。当用户选择 1～5 对应的选项时，屏幕界面分别显示 Red、Green、Blue、Yellow、Gray。输入数字 0 时，退出应用程序。代码如下：

```
[参考程序 1]
#include "stdio.h"
#include "stdlib.h"
void main()
{   int SelNum;
    system("color 1f");                          // 设置界面为白字蓝底
    system("mode con: cols=78 lines=35");        // 设置界面大小
    do
    {// 循环语句保证主菜单始终出现在界面中
        printf("\t 1: Red\n");
        printf("\t 2: Green\n");
        printf("\t 3: blue\n");
        printf("\t 4: Yellow\n");
```

```
            printf("\t 5: Gray\n");
            printf("\t 0: Goodbye!\n");
            printf("\n      请输入功能编号(0～5): ");
            scanf("%d",&SelNum);
            switch(SelNum)
            {  case 1: printf("Red\n");    break;
               case 2: printf("Green\n");  break;
               case 3: printf("blue\n");   break;
               case 4: printf("Yellow\n"); break;
               case 5: printf("Gray\n");   break;
               case 0: exit(0);
               default: printf("\n\t ERROR! Please  Input again\n"); break;
            }
        }while(SelNum!=0);
    }// endmain
```

（2）在实际应用中，switch 语句中 case 语句后面对应的功能选择，通常都是通过具体的函数来实现的。因此，可以对 [参考程序 1] 中的 switch 语句进行修改，添加对应的 5 个功能函数 Red()、Green()、Blue()、Yellow() 和 Gray() 的调用。

[参考程序 2]中，增加了 5 个函数，它们实现了分别按颜色的名称，改变界面背景为对应颜色的功能。代码如下：

```
[参考程序 2]
#include "stdio.h"
#include "stdlib.h"
void Red()
{    printf("Red\n");  system("pause");
     system("color 4f");
}
void Blue()
{    printf("Blue\n"); system("pause");
     system("color 9f");
}
void Green()
{    printf("Green\n"); system("pause");
     system("color 2f");
}
void Yellow()
{    printf("Yellow\n"); system("pause");
     system("color 6f");
}
void Gray()
{    printf("Orange\n"); system("pause");
     system("color 8f");
}
void main()
{   int SelNum;
    system("color 1f");                          // 设置界面为白字蓝底
```

```
    system("mode con: cols=78 lines=35");       // 设置界面大小
    do
    {   // 循环语句保证主菜单始终出现在界面中
        system("cls");
        printf("\n\n        欢迎使用本示例菜单\n\n");
        printf("\n******请根据功能输入菜单编号******\n");
        printf("\t 1: Red\n");
        printf("\t 2: Green\n");
        printf("\t 3: blue\n");
        printf("\t 4: Yellow\n");
        printf("\t 5: Gray\n");
        printf("\t 0: Goodbye!\n");
        printf("\n      请输入功能编号(0～5): ");
        scanf("%d",&SelNum);
    switch(SelNum)
    {   case 1:  Red();     break;
        case 2:  Green();   break;
        case 3:  Blue();    break;
        case 4:  Yellow();  break;
        case 5:  Gray();    break;
        case 0:  printf("\n\t  谢谢使用!\n"); exit(0);
        default: printf("\n  输入有错! 请重新输入!\n\n"); break;
    }// end switch
    }while(SelNum!=0);
}// endmain
```

1.3 C 语言实现的相关技术

1.3.1 程序设计的原则

在进行程序设计时，不能仅以完成功能为目标，应结合软件工程的指导思想和 C 语言的特点，遵循数据结构程序设计的三个原则，即：功能的模块性、算法的通用性、驱动的实例性。

1．功能的模块性

功能的模块性是以功能块为单位进行程序设计，实现其求解算法的方法称为模块化。模块化程序设计即模块化设计，简单地说就是程序的编写不是开始就逐条录入计算机语句和指令，而是首先用主程序、子程序、子过程等框架把软件的主要结构和流程描述出来，并定义和调试好各个框架之间的输入、输出链接关系。逐步求精的结果是得到一系列以功能模块为单位的算法描述。

在 C 语言程序设计中，有两个层次的模块化设计：一个层次是函数级模块，各个函数均是单一功能的函数；另一个层次是文件级模块，由若干.h、.c 文件组成。一般包含数据结构的定义和基本操作文件，实例操作和驱动性操作文件。

2．算法的通用性

算法的通用性指在进行算法功能设计时，使其功能代码能够完成同一功能不同量的实际要求。例如，1.2.2 节中介绍的关于顺序表定义与算法，对任何形式的顺序表，这些定义（除数据元素中数据项外）和基本算法都是完全适用的。

3．驱动的实例性

驱动的实例性是程序运行的必要条件，一般具备三个方面的内容：一是客观事物的属性定义，二是通用类型的实例变量定义，三是实例问题的求解程序。

1.3.2　数据基本属性与数据处理

任意一个数据都有两个基本属性：数据的值及存放。无论是在纸上或计算机系统中，还是在任何其他的数据使用方式中，这两个属性必须同时具有。“存放”是空间概念，具有尺寸（即大小）和位置属性。应该对数据的属性有比较清晰的认识。

软件技术的核心是如何对数据进行处理。例如，“读数据”即采取某种形式从该数据的存放处取出其值；“写数据”即采取某种形式将该数据的值存放到某个位置；“改写数据”即采取某种形式将其新值存放到该数据已在的位置，“删除数据”即采取某种形式解除该数据与原存放位置之间的关系等。

1.3.3　数据结构与算法

数据结构是研究一组数据类型相同的数据的位置关系及相关操作的一门技术。在数据结构中，对数据的操作方式和方法，是用算法进行描述的。因此，数据结构与算法之间存在必然的联系。

算法一般包含两个行为：如何找到数据与如何处理数据。对数据的处理有两类：数据操作和数值计算。数据操作是指数据的增、删、改、查等操作；数值计算是指对数据的各种数学计算。无论是数据操作还是数值计算，显然，都必须知道该数据的存放位置。所以在算法中，找到数据的存放位置是进行数据处理的第一步。如果是单个数据，就应该知道该数据的位置；如果是一组相关数据，就应该知道某个数据的存放位置，并能通过该数据的存放位置找到其他数据的存放位置。

综上所述，算法是数据结构研究的重要方面。

1.3.4　数据结构的 C 语言描述

从软件工程的角度看，数据结构的算法思想是通用的，数据结构的算法软件也是通用的，使用 C 语言实现数据结构技术时，有如下使用经验。

1．数据元素的表示

数据元素用于描述客观事物的属性。这种属性有时是简单的，简单到一个数据项就可以描述该事物；而更多的时候是复杂的，复杂到需要由多个数据项才能描述该事物。C 语言的结构体技术可以实现无论是一个数据项还是多个数据项，都可以通过如下结构语句包装成一个数据元素：

```
struct  DataElement
{
    ……
};
```

并且结合 C 语言的 typedef 命令，将该结构体定义为数据类型 DataElementType，这样，就可如同使用数据类型 int 一样使用。

（1）定义语句

```
typedef  struct  DataElemen  DataElementType;
```

（2）使用语句

```
DataElementType  x;
```

2．数据结构的表示

（1）数据结构的一组数据整体包装技术

数据结构的一组数据及这组数据的描述信息可以通过 C 语言的结构体和 typedef 技术，实现一组数据的整体定义。如 1.2.2 节中数据结构定义与其抽象的内容。

（2）数据结构的定义与基本操作的整体包装技术

数据结构设计时建议借助 C 语言的多文件共享技术，使用.h 和相应.c 两个文件，提供一个数据结构的定义和相应操作的文件包。

（3）数据结构中数据容纳量的通用性设计技术

数据结构必须支持一类数据的操作，这类数据的数据量不是静态分配的，而是在程序运行时才能确定下来，即在实例化的过程中才能确定。这个问题的解决必须采用 C 语言的动态空间技术。C 语言动态空间技术的几个相关函数如下。

① malloc()函数

函数原型为：

```
void  *malloc(unsigned  int  size);
```

malloc()函数是一个内存分配函数，在动态区中分配一个长度为 size 的连续空间，如果分配成功，函数返回一个指向该空间首地址的指针；如果分配失败，函数返回空指针（NULL）。例如，分配一个 DataElementType 类型的空间给指针变量 p，代码如下：

```
DataElementType  *p;
p=(DataElementType *)malloc(sizeof(DataElementType)*30);
// 申请空间大小是 30×DataElementType 的字节数，返回的首地址赋给指针变量 p
```

② calloc()函数

函数原型为：

```
void *calloc(unsigned int n, unsigned int size);
```

calloc()函数也是一个内存分配函数，在动态区中分配 n 个长度为 size 的连续空间，如果分配成功，函数返回一个指向该空间首地址的指针；如果分配失败，函数返回空指针（NULL）。例如：

```
DataElementType  *p;
P=( DataElementType *)calloc(30, sizeof(DataElementType));
// 申请空间大小是 30×DataElementType 的字节数，返回的首地址赋给指针变量 p
```

③ realloc()函数

函数原型为：

```
void  *realloc(void *p, unsigned int size);
```

如果已经通过 malloc()或 calloc()函数获得了动态空间，想改变其大小，可以用 realloc()函数重新分配空间。

用 realloc()函数将 p 所指向的动态空间的大小更改为 size 大小。p 的值不变。如果重新分配不成功，函数返回空指针（NULL）。例如：

```
realloc(p, sizeof(DataElementType)*50);
```

其中，p 是已分配空间的指针，size 是调整后的空间大小。当 p 指向的后续空间能够满足新的空间大小要求，增加的空间在其已分配空间的后续接着连续添加分配，此时 p 的值不变，当 p 指向空间的后续空间不能满足新空间大小要求，则系统在堆中找到一块满足要求大小的空间，并将已分配空间的内容移动到新的存储空间，函数返回一个指向新空间首地址的指针，此时 p 的值不变，但 p 悬空了。

如果重新分配不成功，函数返回空指针（NULL），p 值不变。该函数的一般使用方法如下：

```
DataElementType  *q;
q=( DataElementType *)realloc(p, sizeof(DataElementType)*50);
if(q!=NULL)
    p=q;
```

思考：这段程序可确保 p 是指向有数据的空间，为什么？

④ free()函数

函数原型为：

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

free()函数的功能是释放指针变量所指向的动态空间，使这部分空间能重新被其他变量使用。例如：

```
free(p);
```

以上 4 个函数的声明在 stdlib.h 头文件中，在使用这些函数时需使用如下指令把 stdlib.h 头文件包含到程序文件中。

```
#include <stdlib.h>;    // 文件引用
```

或

```
#include ”stdlib.h”;    // 文件引用
```

思考：#include <文件名>与#include “文件名”有何区别。

3．指针的使用

指针是 C 语言支持的一种数据访问技术，也是 C 语言的一个重要功能。由于数据结构的通用性、动态性和模块性等技术特点，掌握指针概念及技术是非常重要的。

（1）什么是指针

指是指向，所谓“指向”是通过地址来体现的。即通过某种指向形式，借助指向内容，可以和物理上相分隔的事物发生某种关系。具体地说，在 C 语言中要访问或使用计算机中存

储的数据，有两种访问方式：一是可以通过变量名直接访问、二是通过该数据的存储地址间接访问，这个数据的地址，也称为它的“指针”，即指针的内容是地址。

（2）指针变量

既然指针是地址，即指针有内容，所以指针必须以某种形式存放在某个位置。在 C 语言中以变量的形式存放指针内容，这种变量称为指针变量。显然，指针变量不同于一般的变量，它是专门用来存放地址的，所以，必须定义一个“指针类型”以区别其他变量类型。

例如，在 C 语言中，整型变量 t 的指针就是变量 t 的存放地址。可以定义一个“指针类型”的变量 p，来存放变量 t 的地址；这个变量 p，即为指针变量。则对变量 t 的访问，既可以通过变量名 t 直接访问，也可以通过指针变量 p 间接访问。具体实现代码如下：

```
int t, *p;  // t 为 int 型变量，p 为 int 型指针变量
p=&t;       // 将变量 t 的存储地址赋给指针变量 p
t=3;        // 将变量 t 赋值 3
*p=3;       // 将 3 存放在 p 指定的地址空间，效果同 t=3
```

（3）指针变量与其他变量的区别

C 语言可以通过指针变量访问所指向空间的数据。既然指针变量是变量，它就具备一般变量的特性。下面就指针变量与非指针变量做对比分析。

① 空间地址属性。变量存放在内存中都有存放地址，指针变量的地址属性与非指针变量的地址属性是一样的。

② 空间大小属性。空间的大小一般由数据类型决定。如 char 型变量只占 1 个字节，int 型变量占 4 个字节等，而指针变量本身，虽然有不同的类型，如有 char 型的指针变量、int 型的指针变量等，但所有指针变量所占的存储空间的大小是一致的，在 32 位系统和 64 位系统中，分别占 4 个字节和 8 个字节。

③ 类型对指针变量的作用。在计算机内部，无论什么数据都是以二进制形式存放的，指针变量的类型直接反映了数据的读取方式。具体表现在以下几个方面。

第一，指针变量的类型决定了从指针变量指向的地址开始操作多少个字节空间的数据。

第二，指针变量的类型决定了读出二进制数据时，以何种类型的数据形式表示；写入时将按何种数据类型的格式转换为二进制数据保存。

第三，指针变量的类型决定了指针变量做增减运算时的结果。例如，指针变量的值增加 1，虽然其运算的结果都是原指针变量指向地址空间的下一个地址空间的首址，但由于指针变量类型的不同，下一个地址空间的首址也是不一样的。

试分析如下代码段中地址空间的变化：

```
char  *pc;
int   *pi;
……
pc=pc+1;
pi=pi+1;
……
```

假定指针变量 pc 的值是 1000，则 pc+1=1001，pc+2=1002，因为一个 char 数据只占 1 个字节。

假定指针变量 pi 的值也是 1000，则 pi+1=1004，pi+2=1008，因为一个 int 数据占 4 个字节。