第1章 绪 论

可能有人认为，随着计算机的功能越来越强大和运行速度越来越快，程序运行效率已变得越来越不重要了。然而，计算机功能越强大，人们就越要尝试解决更加复杂的问题，而更复杂的问题需要更大的计算量，这使得对程序的运行效率有更高的要求，工作越复杂越偏离人们的日常经验，使得从事软件开发的人必须学习和具备彻底理解隐藏在程序设计后面的一般原理——数据结构和算法。

从本质上讲，数据结构与算法的原理和方法独立于具体描述语言，然而只能使用具体的某种计算机语言才能在计算机上实现。本书采用目前普遍使用的 C++ 程序设计语言来描述各种数据结构与算法，假设读者具有程序设计基础，了解 C++ 的基本结构和语法。为了使读者更好理解，本章将对 C++ 的基本结构和语法进行介绍。

1.1 数据结构的概念和学习数据结构的必要性

对于数值计算问题的解决方法，主要是用数学方程建立数学模型，例如天气预报的数学模型为二阶椭圆偏微分方程；预测人口增长的数学模型为常微分方程。求解这些数学模型的方法是计算数学研究的范畴，例如采用差分算法、有限元算法和无限元算法等。

对于非数值计算问题，主要采用数据结构的方法建立数学模型，下面通过实例加以说明。

例 1.1 在人事管理系统中，包含有“员工基本信息”表格，包括了许多员工基本信息记录（例如，包含有编号，姓名，性别，籍贯，家庭住址，生日，如表 1.1 所示），将这些记录按照一定的顺序存放在“员工基本信息”表格中，每个员工基本信息记录按顺序排列，形成员工基本信息记录的线性序列，这是一种最简单的线性表结构。

表 1.1 员工基本信息

学 号	姓 名	性别	籍贯	家 庭 住 址	生 日
1001	刘靖	女	北京	人民北路 26 号	1985.12.18
1002	朱洪顺	男	成都	一环路北 3 段 56 号	1986.6.28
1003	李世红	男	太原	二环路东 6 段 168 号	1983.10.16
1004	陈冠杰	男	杭州	解放路 18 号	1982.11.29
1005	游倩华	女	苏州	人民西路 98 号	1988.6.8
1006	林键忠	男	青岛	人民西路 69 号	1986.2.28
1007	李代靖	女	太原	一环路东 6 段 16 号	1985.3.19
1008	刘茜	女	广州	一环路南 8 段 6 号	1981.8.18

例 1.2 典型的 UNIX 文件系统结构如图 1.1 所示，属于树结构，是一棵倒置的“树”，此处“树根”代表整个系统，用根目录“/”表示；下一层表示子系统，如 bin、lib 和 user 等，“叶子”就是文件，如 LinkList.h 和 SqList.h 等。

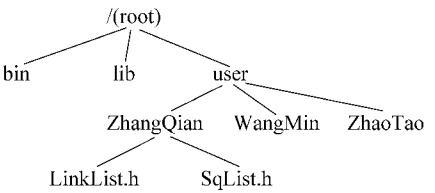

图 1.1 UNIX 文件系统结构图

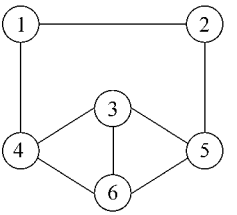

图 1.2 网站之间的“图”状结构

例 1.3 要在 n 个网站建立通信网格，要求使得网络中任一网站出现故障时，整个网络仍能正常通信。如图 1.2 所示，这些网站之间形成一种图结构。

从上面的实例可以看出，非数值计算问题的数学模型已不是数学方程，而是线性表、树和图等数据结构。简单地说，数据结构的研究范畴主要是非数值计算问题的操作对象及它们之间的关系，以及在计算机中的表示和实现。

如果有足够的存储空间存储一组操作对象，总可以查找指定的操作对象，显示不同的操作对象，或者将这些操作对象排序成任何期望的顺序，或者修改任何特定的操作对象。有可能对任何数据结构执行必需的操作，而选择不同的数据结构可能会产生很大的差异，同样的一个问题，选择一种数据结构可能在几秒内就运行完毕，而选择另一种数据结构可能需要几天时间才能完成运行。

在选择数据结构解决特别问题时，只有通过预先分析问题来确定必须达到的性能目标，才有可能挑选出恰当的数据结构。水平不高的程序设计人员往往忽视这一分析过程，而直接选用他们习惯使用的、但与问题不相称的数据结构，结果设计出效率低的程序。相反，当使用简单的设计就能达到目标时，选择复杂的数据结构来改进程序也没有必要。

1.2 数据结构的基本概念

本节将介绍数据结构的基本概念，根据作者的经验，初学者对这些概念在开始时会感到非常抽象，难于理解，但随着不断地学习，一定会融会贯通并加以深刻理解。

1.2.1 数据

数据是客观事物的符号表示，是计算机中可以操作的对象，也就是一切能输入到计算机中并能被处理的符号的总称。

数据是一个广义的概念，可以是数值型数据，例如整数、实数和复数等，主要应用于工程计算，相信读者都比较熟悉；也可以是非数值型数据，例如文字、图形和语音等。

1.2.2 数据元素和数据项

数据元素一般在计算机中能作为整体进行处理，是数据的基本单位。数据元素也称为

记录，有的数据元素由若干**数据项**所组成，例如在员工基本信息表中，每个员工记录是一个数据元素，而员工的编号、姓名、性别、籍贯、家庭住址和生日等内容为数据项，数据项是不可分割的最小单位。

1.2.3 数据结构

在现实世界中，不同数据元素之间不是独立的，而是存在着特定的关系，我们将这些关系称为**结构**，**数据结构**指相互之间存在着一定关系的数据元素的集合。

为了方便起见，用示意图表示数据结构，这种图称为**逻辑结构图**。具体表示为，用小圆圈表示数据元素，用小圆圈之间带有箭头的线段表示数据元素的有序对。具体地讲，对于有序对$<u, v>$，可表示为图 1.3 所示。

u 称为 v 的前驱，v 称为 u 的后继，数据元素之间的**关系**定义为有序对的集合。

根据数据元素之间关系的特性，有如下 4 类基本结构。

1. 集合结构

在数据结构中，如果不考虑数据元素之间的关系，这种结构称为**集合结构**。在集合结构中，各个数据元素是“平等”的，它们的共同属性是“同属于一个集合”，如图 1.4 所示。

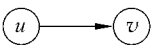

图 1.3　有序对$<u, v>$示意图

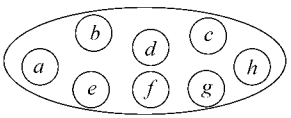

图 1.4　集合结构示意图

2. 线性结构

线性结构中的数据元素之间存在一个对应一个的关系，也就是除了第一个数据元素没有前驱，最后一个数据元素无后继以外，其他数据元素都有唯一的前驱和后继，如图 1.5 所示。

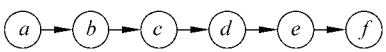

图 1.5　线性结构示意图

3. 树状结构

树状结构中的数据元素之间存在着一个对应多个的关系。数据元素之间存在着层次关系，也就是除了一个特殊的称为树根的数据元素无前驱外，其他数据元素都有唯一的前驱，如图 1.6 所示。

4. 图状结构

图状结构中的数据元素之间存在多个对应多个的关系，也就是任一数据元素可能有多个前驱和后继，如图 1.7 所示。

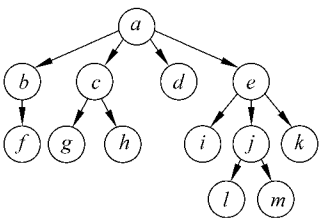

图 1.6　树状结构示意图

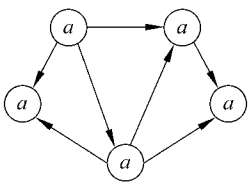

图 1.7　图状结构示意图

数据结构可以定义为如下形式的二元组：

$$\text{DataStructure}=(D, S)$$

其中，D 是一个数据元素的集合，S 是定义在 D 中的数据元素之间关系的有限集合。下面通过实例加以说明。

例 1.4 线性表可定义成如下形式的数据结构：

$$\text{List}=\{D, S\}$$

其中，$D=\{a_i \mid a_i \in \text{ElemSet}, 1 \leqslant i \leqslant n\}$，ElemSet 为某个数据元素的集合，$S=\{R\}$，$R=\{<a_{i-1}, a_i> \mid 2 \leqslant i \leqslant n\}$。

上面数据结构的形式定义实际上是一种数学描述，也就是从解决问题的实际出发，为实现必要的功能建立数学模型，其结构定义中的关系用于描述数据元素之间的逻辑关系，是面向问题的，称为**逻辑结构**。数据结构在计算机中的表示称为**物理结构**或**存储结构**，物理结构是面向计算机的。

本书后面讨论数据结构时，不但要讨论典型的逻辑结构，同时还应讨论逻辑结构相应的物理结构，以及数据结构的相关操作及其实现。

1.3 抽象数据类型及其实现

1.3.1 数据类型

在程序设计语言中已学过各种数据类型，如 C 语言的基本数据类型——整型和浮点型。这些数据类型规定了使用这些类型时的取值范围，同时还规定类型可以使用的不同操作，例如 32 位长的整型类型数据的取值范围是 $-2^{31} \sim 2^{31}-1$，能进行的操作有双目运算符＋、－、＊、/、％，单目运算符＋、－，关系运算符＜、＞、＜＝、＞＝、＝＝、！＝，赋值运算符＝等。在 C 语言中还提供了一些构造类型，例如数组类型、结构体类型等，程序员可以根据需要定义数据类型。在本书中，**数据类型**是指一组性质相同的值的集合以及定义在此集合上的一些操作的总称。

1.3.2 抽象数据类型

抽象指抽取出事物具有普遍性的本质。据说在欧洲古代没有 3 及 3 以上的概念，他们在数数时，3 及 3 以上的数都称为 many，也就是说他们还没有抽象出 3 及 3 以上的自然数。可以看出，只有抽象才具有普遍性，**抽象数据类型**(abstract data type，ADT)指定义用于表示应用问题的数学模型，以及定义在此数学模型上的一组操作(也可称为服务或方法)的总称。

抽象数据类型可利用已有的数据类型来实现，并用已实现的操作构成新的操作。为使读者易于上机实践，本书采用 C++ 程序设计语言实现抽象数据类型，读者不但能学到数据结构与算法的知识，同时也加深了程序面向技术的领悟。

考虑到可能有读者还没学过 C++ 程序设计语言，下面对 C++ 进行简单讨论。读者最好将所有的实例上机运行，并在学习后面章节时随时查阅相关内容，这样随着学习进度的深入，自然就掌握了 C++ 。

1.3.3 C++ 程序的典型构架

下面以 C++ 的一个简单程序为例来说明 C++ 程序的典型构架，程序功能是输出 Hello world!。

此程序由如下 3 个文件组成：

```
//文件路径名:e1_1\hello.h
#ifndef__HELLO_H__                    //如果没有定义__HELLO_H__
#define __HELLO_H__                   //那么定义__HELLO_H__

void Hello(void);                     //函数原型

#endif


//文件路径名:e1_1\hello.cpp
#include <iostream>                   //包含 cout
using namespace std;                  //标准库包含在命名空间 std 中
#include "hello.h"                    //包含 Hello()的原型

void Hello(void)
//操作结果:显示 Hello, world!
{
    cout<<"Hello, world!"<<endl;      //用 cout 和<<输出,endl 表示回车换行
}

//文件路径名:e1_1\main.cpp
#include <stdlib.h>                   //包含库函数 system()的原型
#include "hello.h"                    //提供 Hello()函数的原型

int main()
{
    Hello();                          //调用 Hello()函数显示:Hello, world!

    system("PAUSE");                  //调用库函数 system()
    return 0;                         //返回值 0, 返回操作系统
}
```

C++ 的程序文件主要分为如下两类。

(1) 头文件(扩展名为.h)：一般用于存放函数原型，在上面实例中，hello.h 中存储有 Hello()的函数原型。

(2) 源程序文件(扩展名为.cpp)：在上面实例中，主程序文件 main.cpp 中通过 #include "hello.h"包含了头文件 hello.h，这样便实现了对函数原型或类的使用。

在 C++ 的典型构架中，源程序文件主要存放函数和类的实现。在本例中，hello.cpp 包含了函数 Hello()的实现，在 C++ 中一般将函数原型与函数实现分别存放在头文件与源程

序文件中。这两个文件的文件名相同，只是扩展名不同，在编译时对函数实现进行编译，在连接时实现对函数的引用。然而，对于用模板给出的参数化数据类型的函数，如果将函数体单独存放于一个源程序文件中，由于在编译这个源程序文件时还无法确定模板给出的参数化数据类型的具体类型，这时无法进一步编译，这样在连接时无法确定函数实现的代码，将会出现连接错误，这时只能将函数声明和实现放在同一个头文件中。

主程序文件 main.cpp 用于调用函数 Hello()实现打印“hello, world!”。

1.3.4 C++ 的类和对象

C++ 主要通过类和对象支持面向对象程序设计技术，C++ 的类在本质上就是 C 语言中结构体的扩充，对象实质是类型为类的变量。在类中不但可以包含数据成员，还可以包含函数成员，并且规定了对类中成员的三级访问权限：public、private 和 protected。

(1) 在 public 中声明的成员可以在程序中直接进行访问。

(2) 在 private 和 protected 中声明的成员可以被此类的成员函数及声明为友元(friend)的函数所访问。

(3) 在 protected 中声明的成员可以被此类派生的类所访问，而在 private 中声明的成员则不能被此类派生的类所访问。

下面是 Rectangle 类的声明和实现，在 C++ 中一般将类声明放在头文件中，将类实现放在源程序文件中。

```
//文件路径名:e1_2\rectangle.h
#ifndef __RECTANGLE_H__                    //如果没有定义__RECTANGLE_H__
#define __RECTANGLE_H__                    //那么定义__RECTANGLE_H__

class Rectangle
{
//私有成员
private:
    int length, width, height;             //长方体的长宽高

//公有成员
public:
    Rectangle(int len, int wd, int ht);    //构造函数
    virtual ~Rectangle(void){}             //析构函数
    int Volume(void);                      //返回长方体的体积
};

#endif



//文件路径名:e1_2\rectangle.cpp
#include "rectangle.h"                     //包含类 Rectangle 的声明

Rectangle::Rectangle(int len, int wd, int ht)
```

```
//操作结果：由 len、ed、ht 构造长方体
{
    length=len;
    width=wd;
    height=ht;
}

int Rectangle::Volume(void)
//操作结果：返回长方体的体积
{
    return length * width * height;
}

//文件路径名:e1_2\main.cpp
#include <stdlib.h>                          //包含库函数 system()的原型
#include <iostream>                          //包含 cout
using namespace std;                         //标准库包含在命名空间 std 中
#include "rectangle.h"                       //提供类 Rectangle 的声明

int main(void)
{
    Rectangle thisRectangle(6, 8, 9);        //构造一个长方体

    int volume=thisRectangle.Volume();       //计算长方体的体积
    cout<<volume<<endl;                      //显示长方体的体积

    system("PAUSE");                         //调用库函数 system()
    return 0;                                //返回值 0，返回操作系统
}
```

在定义一个对象时，将自动调用构造函数，构造函数名与类名相同。上面实例中构造函数原型 Rectangle(int len, int wd, int ht)表示由 len、wd 和 ht 分别作为长方体的长、宽和高定义一个长方体对象。

当对象被释放时，将自动地调用析构函数，析构函数名由~后面接类名，析构函数主要用于含有指针的数据成员中释放动态数据成员。上面实例中由于没有动态数据成员，所以析构函数不作任何操作。

1.3.5 C++ 的友元函数

友元(friend)函数需在类声明中用关键字 friend 加以声明，友元函数不是类的函数成员，但是友元函数能引用类的私有成员(private)和保护成员(protected)。下面将上例中类 Rectangle 的成员函数 Volume()改为友元函数，希望读者加以体会。

```
//文件路径名:e1_3\rectangle.h
#ifndef __RECTANGLE_H__                      //如果没有定义__RECTANGLE_H__
#define __RECTANGLE_H__                      //那么定义__RECTANGLE_H__
```

```
class Rectangle
{
//私有成员
private:
    int length, width, height;                    //长方体的长宽高

//公有成员
public:
    Rectangle(int len, int wd, int ht);           //构造函数
    virtual ~Rectangle(void){};                   //析构函数
    friend int Volume(Rectangle oRectangle);      //返回长方体的体积
};

#endif


//文件路径名:e1_3\rectangle.cpp
#include "rectangle.h"                            //包含类 Rectangle 的声明

Rectangle::Rectangle(int len, int wd, int ht)
//操作结果: 由 len、ed、ht 构造长方体
{
    length=len;
    width=wd;
    height=ht;
}

int Volume(Rectangle oRectangle)
//返回长方体的体积
{
    return oRectangle.length*oRectangle.width*oRectangle.height;
}


//文件路径名:e1_3\main.cpp
#include <stdlib.h>                               //包含库函数 system()的原型
#include <iostream>                               //包含 cout
using namespace std;                              //标准库包含在命名空间 std 中
#include "rectangle.h"                            //提供类 Rectangle 的声明

int main(void)
{
    Rectangle thisRectangle(6, 8, 9);             //构造一个长方体
```

```
    int volume=Volume(thisRectangle);          //计算长方体的体积
    cout<<volume<<endl;                        //显示长方体的体积

    system("PAUSE");                           //调用库函数 system()
    return 0;                                  //返回值 0,返回操作系统
}
```

1.3.6 运算符重载

在 C++ 语言中,运算符是作为函数来处理的,用户可用关键字 operator 加上运算符来表示函数,这种函数称为运算符重载函数。例如,两个复数相加可定义如下函数:

```
Complex Add (const Complex &a, const Complex &b);
```

但人们习惯用“+”表示相加,用运算符重载表示如下:

```
Complex operator+ (const Complex &a, const Complex &b);
```

运算符与普通函数使用的不同之处是普通函数参数出现在圆括号内,而运算符的参数出现在运算符的左右两侧。运算符重载有两种方式。

(1) 将运算符重载为全局函数。这时只有一个参数的运算符称为单目运算符,有两个参数的运算符叫做双目运算符,这种情况常声明为类的友元,以便引用类的私有成员。例如,复数加法运算符“+”可重载如下:

```
friend Complex operator+ (const Complex &a, const Complex &b);
```

(2) 运算符被重载为类的成员函数。此时对象自己成了左侧参数,所以单目运算符没有参数,双目运算符只有一个右侧参数。例如,复数加减法运算符“+”与“-”的可重载如下:

```
Complex operator- (const Complex &a);
```

下面是实例程序。

```
//文件路径名:e1_4\complex.h
#ifndef __COMPLEX_H__                          //如果没有定义__COMPLEX_H__
#define __COMPLEX_H__                          //那么定义__COMPLEX_H__

class Complex
{
private:
   double dRealPart;                           //实部
   double dImagePart;                          //虚部

public:
    Complex(double rp=0, double ip=0);
        //构造函数,构造复数,其实部和虚部分别被赋以参数 rp 和 ip 的值
    virtual ~Complex(void) {};                 //析构函数,复数被销毁
    void Show(void);                           //将实部赋为 e
```

```
    friend Complex operator + (const Complex &a, const Complex &b);
        //作为全局函数加法运算符"+"的重载,一般作为友元函数
    Complex operator - (const Complex &a);    //作为成员函数减法运算符"-"的重载
};

#endif
```

//文件路径名名:e1_4\complex.cpp

```
#ifdef _MSC_VER                               //表示是 VC
#if _MSC_VER==1200                            //表示 VC6.0
#include <iostream.h>                         //包含 cout
#else                                         //其他版本的 VC++
#include <iostream>                           //包含 cout
using namespace std;                          //标准库包含在命名空间 std 中
#endif    // _MSC_VER==1200
#else                                         //非 VC
#include <iostream>                           //包含 cout
using namespace std;                          //标准库包含在命名空间 std 中
#endif    // _MSC_VER
#include "complex.h"                          //提供类 Complex 及友元函数 operator+ ()的声明

Complex::Complex(double rp, double ip)
//操作结果:构造复数,其实部和虚部分别被赋以参数 rp 和 ip 的值
{
    dRealPart=rp;
    dImagePart=ip;
}

void Complex::Show(void)
//操作结果:显示复数
{
    cout<<dRealPart;                          //显示实部
    //对虚部进行讨论,分别加以显示
    if (dImagePart<0 && dImagePart!=-1)
        cout<<dImagePart<<"i"<<endl;
    else if (dImagePart==-1)
        cout<<"-i"<<endl;
    else  if (dImagePart>0 && dImagePart!=1)
        cout<<"+ "<<dImagePart<<"i"<<endl;
    else if (dImagePart==1)
        cout<<"+i"<<endl;

}
```

```
Complex operator + (const Complex &a, const Complex &b)
//操作结果：作为全局函数加法运算符"+"的重载，一般作为友元函数
{
    Complex c;
    c.dRealPart=a.dRealPart+b.dRealPart;
    c.dImagePart=a.dImagePart+b.dImagePart;
    return c;
}
```

```
Complex Complex::operator - (const Complex &a)
//操作结果：作为成员函数减法运算符"-"的重载
{
    Complex c;
    c.dRealPart=dRealPart-a.dRealPart;
    c.dImagePart=dImagePart-a.dImagePart;
    return c;
}
```

```
//文件路径名:e1_4\main.cpp
#include <stdlib.h>                    //包含库函数 system()的原型
#ifdef _MSC_VER                        //表示是 VC
#if _MSC_VER==1200                     //表示 VC6.0
#include <iostream.h>                  //包含 cout
#else                                  //其他版本的 VC++
#include <iostream>                    //包含 cout
using namespace std;                   //标准库包含在命名空间 std 中
#endif    // _MSC_VER==1200
#else                                  //非 VC
#include <iostream>                    //包含 cout
using namespace std;                   //标准库包含在命名空间 std 中
#endif    // _MSC_VER
#include "complex.h"                   //提供类 Complex 及友元函数 operator+ ()的声明

int main()
{
    Complex z1(6, 8);                  //通过构造函数自动地生成复数 z=6+8i
    z1.Show();

    Complex z2(8, 9);                  //通过构造函数自动地生成复数 z=8+9i
    z2.Show();

    Complex z3;
    z3=z1+z2;                          //z3=z1+z2=14+17i
    z3.Show();
```

```
    Complex z4;
    z4=z2-z1;                          //z4=z2-z1=2+i
    z4.Show();

    system("PAUSE");                   //调用库函数 system()
    return 0;                          //返回值 0,返回操作系统
}
```

注意：经过实际测试，在重载运算符“+”与“-”时，Visual C++ 6.0 只在包含 iostream.h 时才能通过编译，而 Visual C++ 2005、Visual C++ 2005 Express、MinGW Developer Studio 与 Dev-C++ 4.9.9.2 只能在包含 iostream 时通过编译，并且 Visual C++ 2005 只支持 iostream，并且在 VC 中有宏_MSC_VER 表示版本号。对于 Visual C++ 6.0，_MSC_VER 的值为 1200，因此本题中通过条件编译方式使程序能自适应各种 C++ 的编译器。

1.3.7 C++ 的参数传递

C++ 有两种参数传递方式：传值和引用。

(1) 传值方式。传值方式是默认的方式，在调用函数时，将实参的值传递给函数局部存储区相应参数的副本，函数对副本进行操作，只修改副本的值，而不会修改实参的值。

(2) 引用方式。引用方式需在形参声明时参数名前加上符号 &，在调用函数时，被传递的不是实参的值，而是实参的地址，函数通过地址存取实参，这样函数可修改实参的值。在 C 语言中，引用是通过指针实现的，通过指针编程难度更高。

下面是一个引用 & 方式和指针方式传递参数的实例。

```
//文件路径名:e1_5\swap.h
#ifndef __SWAP_H__                     //如果没有定义__SWAP_H__
#define __SWAP_H__                     //那么定义__SWAP_H__

void Swap(int &a, int &b);             //函数原型，用引用 & 传递方式实现
void Swap(int *pa, int *pb);           //函数原型，用指针传递方式实现

#endif

//文件路径名:e1_5\swap.cpp
#include <iostream>                    //包含 cout
using namespace std;                   //标准库包含在命名空间 std 中
#include "swap.h"                      //包含 Swap()的原型

void Swap(int &a, int &b)
//操作结果：用引用 & 传递方式实现交换参数值
{
    int tem;
```

```
    //循环赋值
    tem=a;
    a=b;
    b=tem;
}
```

```
void Swap(int *pa, int *pb)
//操作结果:用指针传递方式实现交换参数值
{
    int tem;

    //循环赋值
    tem=*pa;
    *pa=*pb;
    *pb=tem;
}
```

```
//文件路径名:e1_5\main.cpp
#include <stdlib.h>                    //包含库函数 system()的原型
#include <iostream>                    //包含 cout
using namespace std;                   //标准库包含在命名空间 std 中
#include "swap.h"                      //提供 Swap()函数的原型

int main()
{
    int a=2, b=3;
    cout<<"交换前:"<<a<<","<<b<<endl;
    Swap(a, b);
    cout<<"交换后:"<<a<<","<<b<<endl;

    int u=6, v=8;
    cout<<"交换前:"<<u<<","<<v<<endl;
    Swap(&u, &v);
    cout<<"交换后:"<<u<<","<<v<<endl;

    system("PAUSE");                   //调用库函数 system()
    return 0;                          //返回值 0,返回操作系统
}
```

上面的函数 Swap()通过引用 & 方式或指针方式传递参数,都达到了交换参数值的目的,但用指针时更容易出错。

说明:

(1) 将数组作为值参传递时,由于数组名实际表示数组的起始存储地址,传递的是数据第一个元素的地址,这样对数组元素的修改本质是对实参数组元素的修改。

(2) 当对象作为值参传递时,为了节省传递时资源开销,本书一般声明为 const Type

&,也就是常值引用,这样在函数体内部不能修改参数。

1.3.8 C++ 的输入输出

C++ 中通过流(stream)操作实现标准输入输出,在 C++ 中有两个类:istream 和 ostream。istream 类对应于输入流,cin 是 istream 类对应的标准输入对象;ostream 类对应于输出流,cout 是 ostream 类对应的标准输出对象。操作符<<用于输出,操作符>>用于输入,下面是实例程序。

```
//文件路径名:e1_6\main.h
#include <iostream>
using namespace std;                          //标准库包含在命名空间 std 中
#include <stdlib.h>                           //包含库函数 system()原型

int main(void)
{
    int n, s=0;

    cout<<"请输入自然数 n:";                  //标准输出
    cin>>n;                                   //标准输入
    while (n<0)
    {
        cout<<"n 不能为负!请重新输入:"<<endl;     //标准输出,endl 表示回车换行
        cin>>n;                                   //重新输入 n
    }

    for (int i=1; i <=n; i++)
        s=s+i;
    cout<<"1+2+...+"<<n<<"="<<s<<endl;            //标准输出

    system("PAUSE");                              //调用库函数 system()
    return 0;                                     //返回值 0,返回操作系统
}
```

C++ 的文件输入输出流包含在头文件<fstream.h>中,执行输入流的对象应声明为 ifstream 类,执行输出流的对象应声明为 ofstream 类,执行输入输出流的对象应声明为 fstream 类,下面是实例程序。

```
//文件路径名:e1_7\main.cpp
//显示文本文件
#include <iostream>
#include <fstream>
using namespace std;                          //标准库包含在命名空间 std 中
#include <stdlib.h>                           //包含库函数 system()及 exit()原型的
                                              //头文件
```

```
#define MAX_LINE_LENGTH 100

int main(int argc, char * argv[])
{
    if (argc!=2)
    {
        cout<<"用法：ShowFile 文本文件名"<<endl;
        system("PAUSE");                          //调用库函数 system()
        exit(1);                                  //非正常退出
    }

    ifstream inFile(argv[1]);                     //输入文本文件
    if (inFile.fail())
    {
        cout<<"打开文件"<<argv[1] <<"失败!"<<endl;
        system("PAUSE");                          //调用库函数 system()
        exit(2);                                  //非正常退出
    }

    char str[MAX_LINE_LENGTH];
    inFile>>str;                                  //输入字符串 str
    while (!inFile.eof())
    {   //文件 inFile 未结束,则循环
        cout<<str<<endl;                          //输出字符串 str
        inFile>>str;                              //输入字符串 str
    }

    inFile.close();                               //关闭文件 inFile

    system("PAUSE");                              //调用库函数 system()
    return 0;                                     //返回值 0, 返回操作系统
}
```

1.3.9 有关 C++ 的动态存储分配

在 C++ 中,不但具有 C 语言中的动态存储分配函数 malloc()和 free(),而且还提供了更易使用的新命令：new 和 delete。在 C++ 中使用 new 分配的存储单元必须用 delete 进行释放。

new 命令的使用格式如下：

new 被建立对象的数据类型

或

new 被建立对象的数据类型[分配存储单元个数]

new 命令返回指向被建立对象的指针。第一种格式适合于建立单个对象,而第二种格

式实际是建立一个对象数组。

delete 命令的使用格式如下：

```
delete 指向被释放对象的指针
```

或

```
delete []指向被释放对象数组的指针
```

第一种格式适合于释放单个对象，而第二种格式实际是释放对象数组。

下面是实例程序。

```
//文件路径名:e1_8\main.cpp
//显示文本文件内容
#include <iostream>
#include <fstream>
using namespace std;                          //标准库包含在命名空间 std 中
#include <stdlib.h>                           //包含 exit()及 system()函数原型的头文件
#include <string.h>                           //包含 strcpy()函数原型的头文件

#define MAX_LINE_LENGTH 100

int main(int argc, char * argv[])
{
    ifstream * inFilePtr;
    char * strPtr, * strFileNamePtr;
    strPtr=new char[MAX_LINE_LENGTH];                    //动态分配存储空间
    strFileNamePtr=new char[MAX_LINE_LENGTH];            //动态分配存储空间

    if (argc!=2)
    {
        cout<<"请输入要显示的文本文件名:";
        cin>>strFileNamePtr;
    }
    else
    {
        strcpy(strFileNamePtr, argv[1]);
    }

    inFilePtr=new ifstream(strFileNamePtr);              //输入文件
    if (inFilePtr==NULL)
    {
        cout<<"打开文件"<<strFileNamePtr <<"失败!"<<endl;
        exit(1);                                         //非正常退出
    }

    while (!inFilePtr->eof())
    {   //当文件 inFilePtr 未结束时循环
```

```
        inFilePtr->getline(strPtr, MAX_LINE_LENGTH);     //读入一行字符
        cout<<strPtr<<endl;                              //显示一行字符
    }

    delete []strPtr;                                     //动态释放字符串
    delete []strFileNamePtr;                             //动态释放字符串
    delete inFilePtr;                                    //动态释放文件对象

    system("PAUSE");                                     //接收任一字符
    return 0;                                            //正常退出
}
```

1.3.10 结构与类

C++ 中的类(class)实际上是结构(struct)的扩充。在 C++ 中,类与结构的区别在于:在结构中默认的访问权限是 public,而在类中默认的访问权限是 private。除此之外,类与结构是等价的。

1.3.11 C++ 的模板

对于序列排序,可以是整数序列、字符串序列等,如按 C 语言的方法只能分别编写程序。在 C++ 中使用模板(template)代替任意类型的序列,这样就不需分别编写程序,也就是实现了代码的复用。在有模板的算法中,只能将函数声明与函数实现放在同一个文件中,为方便与统一起见,本书以后都将函数声明与函数实现放在头文件中。下面是序列起泡排序的程序。

```
//文件路径名:e1_9\bubblesort.h
#ifndef __BUBBLESORT_H__
#define __BUBBLESORT_H__

#define MAX_SIZE 100
#include <iostream>
using namespace std;

//类的声明部分
template <class ElemType>
class DataList
{
private:
    ElemType data[MAX_SIZE];                         //存储数据元素值
    int size;                                        //元素个数
    void Swap(const int mark1, const int mark2);     //交换 data[mark1]与 data[mark2]
public:
    DataList(int sz=6): size(sz) {};                 //构造函数
    ~DataList(void) {};                              //析构函数
    void SelectSort(void);                           //选择排序
```

```
    void Input(void);                                    //输入数据元素值
    void Show(void);                                     //显示数据元素值
};

//类的实现部分
template <class ElemType>
void DataList<ElemType>::Swap(const int mark1, const int mark2)
//操作结果：交换 data[mark1]与 data[mark2]
{
    //使用循环赋值交换 data[mark1]与 data[mark2]
    ElemType temp;
    temp=data[mark1];
    data[mark1]=data[mark2];
    data[mark2]=temp;
}

template <class ElemType>
void DataList<ElemType>::SelectSort(void)
//操作结果：选择排序
{
    for (int i=0; i<size-1; i++)
    {   //在 data[i]~data[size-1]之间选出最小元素 data[currLargePos]
        int currLargePos =i;                             //假设 data[i]为当前的最小元素
        for (int j=i+1; j<size; j++)
        {   //在 data[i]~data[size-1]之间找出真正的最小元素
            if (data[j]<data[currLargePos])
            {  //如果 data[j]小于当前的最小元素 data[currLargePos],则将 j 赋给 currLargePos
                currLargePos=j;
            }
        }
        if (currLargePos!=i)
        {   //当 data[i]不为最小元素,则交换 data[kcurrLargePos]与 data[i]
            Swap(currLargePos, i);
        }
    }
}

template <class ElemType>
void DataList<ElemType>::Input(void)
//操作结果：输入数据元素值
{
    cout<<"输入元素个数:";
    cin>>size;
    for (int i=0; i<size; i++)
    {
```

```
        cout<<"输入第"<<i+1<<"个数据元素:";
        cin>>data[i];
    }
}

template <class ElemType>
void DataList<ElemType> ::Show(void)
//操作结果:显示数据元素值
{
    cout<<"元素:";
    for (int i=0; i<size; i++)
        cout<<data[i]<<" ";
    cout<<endl;
    cout<<"元素个数:"<<size<<endl;
}

#endif


//文件路径名:e1_9\main.cpp
#include <iostream>
#include "selectsort.h"
#include <stdlib.h>                          //包含库函数 system()原型的头文件
const int SIZE=6;
using namespace std;

int main(void)
{
    DataList<int>  List(SIZE);

    List.Input();                            //输入元素个数及元素值
    cout<<"排序前:"<<endl;
    List.Show();                             //显示排序前的元素值
    List.SelectSort();                       //排序
    cout<<"排序后:"<<endl;
    List.Show();                             //显示排序后的元素值

    system("PAUSE");                         //调用库函数 system()
    return 0;                                //返回值 0,返回操作系统
}
```

1.4 算法和算法分析

1.4.1 算法

算法是解决特定问题求解步骤的描述,在计算机中为指令的有限序列,并且每条指令表

示为一个或多个操作。对于给定的问题可以有多种算法来解决，本书的有些问题给出了多种算法。

一个算法应具有如下几条性质才能称为是一种解决特定问题的算法。

1. 正确性

正确性指必须完成所期望的功能，也就是应当满足以特定的“规格说明”方式给出的需求。对算法是否“正确”的理解可以有如下 4 个层次。

(1) 程序中不含任何语法错误，这一层次很容易达到，只要源程序能编译成功就没有语法错误了。

(2) 程序对于几组输入数据能够得出满足要求的结果，这一层次也容易达到。一般程序员编写完程序后都要进行测试，都要输入几组数据，只有输出都能达到正确的结果才算编写完算法。

(3) 程序对于精心选择的、典型的、苛刻的并带有刁难性的几组输入数据能够得出满足要求的结果，这一层次一般也能达到。一般软件公司都有专门的测试小组，都会选择如边界数据、非法数据、有效数据进行测试，经过测试后的算法应能达到要求。

(4) 程序对于一切输入数据都能得出满足要求的结果，这一层次只有简单的算法才能达到，对于大型算法一般都无法达到。

软件公司通常以第(3)层次意义的正确性作为衡量一个算法是否合格的标准。

2. 具体性

一个算法必须由一系列具体操作组成，这里的“具体”指所有操作都必须经过已实现的基本操作有限次来实现，并且所有操作都是可读的、可执行的，每一操作必须在有限时间内完成。

3. 确定性

算法中的所有操作都必须有确切的含义，不能产生歧义，算法的执行者或阅读者都能明确其含义及如何执行。

4. 有限性

算法必须在执行有限步后结束，并且每一步都在有限时间内完成，如果一个算法要由无限步操作才能结束(例如无限循环)，这样的算法在计算机上无法在有限时间内完成，也就是说算法没实际意义。

5. 可读性

算法应具备良好的可读性，这样的算法有利于算法的查错及对算法的理解。一般算法的逻辑必须清楚、结构简单，所有标识符必须具有实际含义，在算法中必须加入适当的注释说明算法的功能、输入输出参数的使用规则以及算法各程序段的功能描述等内容。

6. 健壮性

健壮性指输入数据非法时，算法能作适当的处理并作出反应，而不应死机或输出异常结果。软件公司在测试程序时通常都会有意输入无效数据，例如在输入数值时有意输入字符，要求程序必须作出适当的处理。

1.4.2 算法分析

对于一个算法的评价，首先应考虑算法的正确性，其次是运算量(即运行效率的高低)，