数据结构课程第一章部分习题解答

第一章 绪论

1-4．什么是抽象数据类型？试用 C++的类声明定义“复数”的抽象数据类型。要求

(1) 在复数内部用浮点数定义它的实部和虚部。

(2) 实现 3 个构造函数：缺省的构造函数没有参数；第二个构造函数将双精度浮点数赋给复数的实部，虚部置为 0；第三个构造函数将两个双精度浮点数分别赋给复数的实部和虚部。

(3) 定义获取和修改复数的实部和虚部，以及+、-、*、/等运算的成员函数。

(4) 定义重载的流函数来输出一个复数。

【解答】

抽象数据类型通常是指由用户定义，用以表示应用问题的数据模型。抽象数据类型由基本的数据类型构成，并包括一组相关的服务。

```
//在头文件 complex.h 中定义的复数类
#ifndef _complex_h_
#define _complex_h_
#include <iostream.h>

class comlex {
public:
   complex ( ){ Re = Im = 0; }                          //不带参数的构造函数
   complex ( double r ) { Re = r;    Im = 0; }          //只置实部的构造函数
   complex ( double r, double i ) { Re = r;    Im = i; }  //分别置实部、虚部的构造函数
   double getReal ( ) { return Re; }                    //取复数实部
   double getImag ( ) { return Im; }                    //取复数虚部
   void setReal ( double r ) { Re = r; }                //修改复数实部
   void setImag ( double i ) { Im = i; }                //修改复数虚部
   complex & operator = ( complex & ob) { Re = ob.Re;    Im = ob.Im; }    //复数赋值
   complex & operator + ( complex & ob );               //重载函数：复数四则运算
   complex & operator – ( complex & ob );
   complex & operator * ( complex & ob );
   complex & operator / ( complex & ob );
   friend ostream & operator << ( ostream & os, complex & c );  //友元函数：重载<<
private:
   double Re, Im;                                       //复数的实部与虚部
};
#endif

//复数类 complex 的相关服务的实现放在 C++源文件 complex.cpp 中
#include <iostream.h>
#include <math.h>
#include “complex.h”
```

```
complex & complex :: operator + ( complex & ob ) {
//重载函数：复数加法运算。
    complex * result = new complex ( Re + ob.Re,    Im + ob.Im );
    return *result;
}
complex & complex :: operator – ( complex & ob ) {
//重载函数：复数减法运算
    complex *result = new complex ( Re – ob.Re,    Im – ob.Im );
    return * result;
}
complex & complex :: operator * ( complex & ob ) {
//重载函数：复数乘法运算
    complex *result =
        new complex ( Re * ob.Re – Im * ob.Im,    Im * ob.Re + Re * ob.Im );
    return *result;
}
complex & complex :: operator / ( complex & ) {
//重载函数：复数除法运算
    double d = ob.Re * ob.Re + ob.Im * ob.Im;
    complex * result = new complex ( ( Re * ob.Re + Im * ob.Im ) / d,
        ( Im * ob. Re – Re * ob.Im ) / d );
    return * result;
}
friend ostream & operator << ( ostream & os, complex & ob ) {
//友元函数：重载<<，将复数 ob 输出到输出流对象 os 中。

    return os << ob.Re << ( ob.Im >= 0.0 ) ? "+" : "-" << fabs ( ob.Im ) << "i";

}
```

1-7 试编写一个函数计算 $n!*2^n$ 的值，结果存放于数组 *A*[*arraySize*]的第 *n* 个数组元素中，$0 \le n \le arraySize$。若设计算机中允许的整数的最大值为 *maxInt*，则当 $n > arraySize$ 或者对于某一个 k ($0 \le k \le n$)，使得 $k!*2^k > maxInt$ 时，应按出错处理。可有如下三种不同的出错处理方式：

(1) 用 **cerr<<**及 **exit** (1)语句来终止执行并报告错误；

(2) 用返回整数函数值 0, 1 来实现算法，以区别是正常返回还是错误返回；

(3) 在函数的参数表设置一个引用型的整型变量来区别是正常返回还是某种错误返回。

试讨论这三种方法各自的优缺点，并以你认为是最好的方式实现它。

【解答】

```
#include "iostream.h"
#define arraySize 100
#define MaxInt 0x7fffffff

int calc ( int T[ ], int n) {
    int i, value = 1;
    if ( n != 0 ) {
        int edge = MaxInt / n / 2;
```

```
            for ( i = 1; i < n; i++) {
                value *= i*2;
                if ( value > edge ) return 0;
            }
            value *= n * 2;
        }
        T[n] = value;
        cout << "A[" << n << "]=" << T[n] << endl;
        return 1;
    }

    void main ( ) {
        int A[arraySize];
        int i;
        for ( i = 0; i < arraySize; i++)
            if ( !calc ( A, i ) ) {
                cout << "failed at " << i << " ." << endl;
                break;
            }
    }
```

1-9 (1) 在下面所给函数的适当地方插入计算 *count* 的语句：

```
    void d (ArrayElement x[ ], int n ) {
        int i = 1;
        do {
            x[i] += 2;    i +=2;
        } while (i <= n );
;       i = 1;
        while ( i <= (n/2) ) {
            x[i] += x[i+1];    i++;
        }
    }
```

(2) 将由(1)所得到的程序化简。使得化简后的程序与化简前的程序具有相同的 *count* 值。

(3) 程序执行结束时的 *count* 值是多少？

(4) 使用执行频度的方法计算这个程序的程序步数，画出程序步数统计表。

【解答】

(1) 在适当的地方插入计算 count 语句

```
void d ( ArrayElement x [ ],    int n ) {
    int i = 1;
    count ++;
    do {
        x[i] += 2;    count ++;
        i += 2;    count ++;
        count ++;                    //针对 while 语句
    } while ( i <= n );
```

```
    i = 1;
    count ++;
    while ( i <= ( n / 2 ) ) {
        count ++;                    //针对 while 语句
        x[i] += x[i+1];
        count ++;
        i ++;
        count ++;
    }
    count ++;                        //针对最后一次 while 语句
}
```

(2) 将由(1)所得到的程序化简。化简后的程序与原来的程序有相同的 count 值：

```
    void d ( ArrayElement x [ ],  int n ) {
    int i = 1;
    do {
        count += 3;  i += 2;
    } while ( i <= n );
    i = 1;
    while ( i <= ( n / 2 ) ) {
        count += 3;  i ++;
    }
    count += 3;
}
```

(3) 程序执行结束后的 count 值为 3n + 3。

当 n 为偶数时，count = 3 * (n / 2) + 3 * (n / 2) + 3 = 3 * n + 3

当 n 为奇数时，count = 3 * ((n + 1) / 2) + 3 * ((n – 1) / 2) + 3 = 3 * n + 3

(4) 使用执行频度的方法计算程序的执行步数，画出程序步数统计表：

行 号	程 序 语 句	一次执行步数	执行频度	程序步数
1	void d (ArrayElement x [ ], int n) {	0	1	0
2	int i = 1;	1	1	1
3	do {	0	⌊(n+1)/2⌋	0
4	x[i] += 2;	1	⌊(n+1)/2⌋	⌊(n+1)/2⌋
5	i += 2;	1	⌊(n+1)/2⌋	⌊(n+1)/2⌋
6	} while (i <= n);	1	⌊(n+1)/2⌋	⌊(n+1)/2⌋
7	i = 1;	1	1	1
8	while (i <= (n / 2)) {	1	⌊n/2+1⌋	⌊n/2+1⌋
9	x[i] += x[i+1];	1	⌊n/2⌋	⌊n/2⌋
10	i ++;	1	⌊n/2⌋	⌊n/2⌋
11	}	0	⌊n/2⌋	0
12	}	0	1	0
			(n ≠ 0)	3n + 3