Data Type II

Abstract Data Type

An abstract data type (ADT) is a specification of a set of data and the set of valid operations on it. The specification is independent of the actual implementation and so is abstract. The operations can be specified as mathematical functions (interface) and a set of axioms satisfied by them.

Data Type complex

We have already talked about the data type complex. It may be viewed as a collection of ordered pair of real numbers. The essential operations on this data type are:

Operations on complex

- I/O operations: read and write of complex number.
- Basic operations: addition, subtraction, multiplication, division, modulus, conjugate, test for equality etc. on complex numbers.
- Other operations: initializing a complex number, constructing a complex number from a pair of real numbers, projecting the real and the imaginary parts of a complex number etc.

Axioms on complex

The set of axioms satisfied by the basic operations are specified in mathematics.

- Addition and multiplication operations are associative and commutative. There are identity elements for both the operations.
- For every complex number z, there is an additive inverse of z. If $z \neq 0$, then there is also a multiplicative inverse of z. In fact the datatype complex forms a field.

Axioms on complex

- Two complex numbers z_1 and z_2 are equal *iff* their real parts are equal and their imaginary parts are also equal.
- There are interesting axioms that are not explicitly mentioned in mathematics e.g. for all complex number z, $z = \text{makeComplex}(\text{real}(z), \text{imag}(z))$.

Implementation of complex

We have already seen how the datatype complex (an approximation) can be implemented as a product (structure) of two floating-point numbers. Due to the approximation of real numbers by floating-point numbers, some of the original axioms of complex may fail.

Implementation of Operations

It would have been nice if we could have overloaded the usual operators to implement the mathematical operations e.g. addition, subtraction, test for equality etc. But in the language C that is not possible and we implement the operations as functions.

Interfaces of a Few Operations

```
complex readComplex() ;
void    readComplex1(complex *) ;
void    writeComplex(complex) ;
complex addComplex(complex, complex) ;
complex subComplex(complex, complex) ;
complex multComplex(complex, complex) ;
complex divComplex(complex, complex) ;
complex makeComplex(float, float) ;
float   realPart(complex) ;
float   imaginaryPart(complex) ;
int     isEqComplex(complex, complex) ;
```

How to Organize the Implementation

- Header or interface file.
- Operation implementation file.

Header File: `complex.h`

We put the type definition and the function interfaces (prototypes) in a header or interface file `complex.h`. There is no executable code in the header file. It contains type definition, function interfaces, macro definitions and inline functions. It is to be properly guarded against multiple inclusion.

complex.h

```
#ifndef _MYCOMPLEX_H
#define _MYCOMPLEX_H
#include <stdio.h>
#include <math.h>
typedef struct complexType {
        double real, imag;
} complex ;

complex readComplex() ;
void    readComplex1(complex *) ;
void    writeComplex(complex) ;
complex addComplex(complex, complex) ;
```

```
complex subComplex(complex, complex) ;
complex multComplex(complex, complex) ;
complex divComplex(complex, complex) ;
complex makeComplex(double, double) ;
double  realPart(complex) ;
double  imaginaryPart(complex) ;
int     isEqComplex(complex, complex) ;

#endif
```

Implementation file `complex.c`

All operations are implemented as functions.
The functions are collected in `complex.c` file.

complex.c

```
#include "complex.h"
#define MAXLEN 100

complex readComplex() { // complex.c
        complex temp ;

        scanf("%lf%lf",&temp.real, &temp.imag);
        return temp ;
}

void readComplex1(complex *cp) {
     scanf("%lf%lf",&cp -> real, &cp -> imag);
```

```
}

void writeComplex(complex c) {
     char s[MAXLEN], sign = '+', j = 'j' ;

     if(c.imag < 0.0) {
        c.imag = - c.imag ;
        sign = '-' ;
     }
     sprintf(s, "%f%c%c%f%c", c.real,
                 sign, j, c.imag, '\0') ;
     printf("%s", s) ;
}
```

```
complex addComplex(complex x, complex y) {
        complex temp ;

        temp.real = x.real + y.real ;
        temp.imag = x.imag + y.imag ;
        return temp ;
}

complex subComplex(complex x, complex y) {
        complex temp ;

        temp.real = x.real - y.real ;
        temp.imag = x.imag - y.imag ;
        return temp ;
```

```
}

complex multComplex(complex x, complex y) {
        complex temp ;

        temp.real = x.real*y.real -
                    x.imag*y.imag ;
        temp.imag = x.real*y.imag +
                    x.imag*y.real;
        return temp ;
}
complex divComplex(complex x, complex y) {
        complex temp ; // y cannot be zero
        double deno = y.real*y.real +
```

```
                    y.imag*y.imag ;

        temp.real = (x.real*y.real +
                     x.imag*y.imag)/deno ;
        temp.imag = (x.imag*y.real -
                     x.real*y.imag)/deno;
        return temp ;
}

complex makeComplex(double r, double i) {
        complex temp ;

        temp.real = r ; temp.imag = i ;
        return temp ;
```

```
}

double realPart(complex c) { return c.real;}
double imaginaryPart(complex c) { return c.imag;}
int isEqComplex(complex x, complex y) {
    return (x.real==y.real && x.imag==y.imag);
}
```

User program `testComplex.c`

Now we consider the user program `testComplex.c` that will use the data type. Note that user is interested about the interface (header) and the code, but not the detail of the implementation.

User program testComplex.c

```
#include "complex.h"
int main() // testComplex.c
{
    complex c, d, e, f ;

    c = readComplex() ;
    writeComplex(c) ;
    printf("\n") ;

    readComplex1(&d) ;
    writeComplex(d) ;
    printf("\n") ;
```

```
e = addComplex(c,d) ;
writeComplex(e) ;
printf("\n") ;

e = subComplex(c,d) ;
writeComplex(e) ;
printf("\n") ;

e = multComplex(c,d) ;
writeComplex(e) ;
printf("\n") ;

e = divComplex(c,d) ;
```

```
    writeComplex(e) ;
    printf("\n") ;

    f = makeComplex(3.0, 4.0) ;
    writeComplex(f) ;
    printf("\n") ;

    if(isEqComplex(e,f))
      printf("\n e = f\n") ;
    else
      printf("\n e != f\n") ;

    return 0 ;
}
```

Hiding the Detail of Implementation

The question is how to hide the detail of the implementation from the user. There are different ways of doing it.

- User includes the header file and links the object module corresponding to the implementation.
- User includes the header file and links the static or dynamic library.

Separate Compilation

```
$ cc -Wall -c complex.c
```

This gives the object-module `complex.o`.

Compiling `testComplex.c`

```
$ cc -Wall testComplex.c complex.o
```

This gives the executable file `a.out`.

Makefile for Compilation

```
a.out   :        testComplex.o complex.o
                 cc testComplex.o complex.o
testComplex.o  :        testComplex.c
                 cc -Wall -c testComplex.c
complex.o :      complex.c
                 cc -Wall -c complex.c
clean :
        rm a.out *.o
```

Creating Archive/Static Library

```
$ ar -rcs libcomplex.a complex.o
```

This gives the static library called libcomplex.a.

```
$ file libcomplex.a
libcomplex.a:  current ar archive
```

Compiling the User Program

$ cc -Wall testComplex.c -L. -lcomplex

The datatype library `libcomplex.a` is linked with the user program to create the executable file `a.out`.

- -L. - current directory is also searched for the library.
- -lcomplex - the library file name is `libcomplex.a`.