Data Structure Lab Manual

Winter 2019

Session 2

The following is the code what we did last time. The subtract, multiply and divide are not implemented in the following code but it is assumed that the students have already implemented in the class complex.

```
#include<iostream>

using namespace std;

//---------------------------------------------------------------------
//---------------------------------------------------------------------
// Definition of complex
//---------------------------------------------------------------------
//---------------------------------------------------------------------

class complex
{
        double real;
        double imag;

public:

        void setReal(double);
        void setImag(double);
        double getReal();
        double getImag();

        void add(complex, complex);
};


void complex::setReal(double a) { real = a; }
void complex::setImag(double a) { imag = a; }
double complex::getReal() { return real; }
double complex::getImag() { return imag; }
void complex::add(complex a , complex b)
```

```
{
        real = a.real + b.real;
        imag = a.imag + b.imag;
}

//----------------------------------------------------------------------------
//----------------------------------------------------------------------------
// Main Program
//----------------------------------------------------------------------------
//----------------------------------------------------------------------------
int main()
{
        int size;
        double  r, im;
        complex c1, c2, c3;

        cout << "Enter the first complex number:";
        cin >> r >> im;
        c1.setReal(r);
        c1.setImag(im);

        cout << "Enter the second complex number:";
        cin >> r >> im;
        c2.setReal(r);
        c2.setImag(im);

        c3.add(c1, c2);
        cout << "Sum: " << c3.getReal() << "+i"<<c3.getImag();

        cin >> r;
        return 0;
}
```

First step

Modify the function add such that you are changing the function arguments by mistake .

```
void complex::add(complex a , complex b)
{
        a. real = 0.0 // this line will corrupt your result
        a.imag = 0.0 // this line will corrupt your result
        real = a.real + b.real;
        imag = a.imag + b.imag;
}
```

Whatever you do the result of add will be always wrong because the function is modifying your arguments.

Second step

What can we do such that our function does not modify our function arguments? Use the keyword **const** with the function arguments as

```
void complex::add(const complex a , const complex b)
{
        a. real = 0.0 // this line will corrupt your result
        a.imag = 0.0 // this line will corrupt your result
        real = a.real + b.real;
        imag = a.imag + b.imag;
}
```

also change the signature of your function in the class.

```
void add(const complex, const complex);
```

Compile the code and see what compilation error you get. The function arguments can no longer be modified. Remove the two lines where you are modifying the function arguments from add function.

```
void complex::add(const complex a , const complex b)
{
        real = a.real + b.real;
        imag = a.imag + b.imag;
}
```

Third step

Now we are going to study the **&** operator. This operator can give us the address of a variable or an object. Modify the code of the function add as below:

```
void complex::add(const complex a , const complex b)
{
    cout<<"The address of a is:"<<&a<<endl;
        real = a.real + b.real;
        imag = a.imag + b.imag;
}
```

and include the following line above the line where you are calling the add function

```
cout<<"The address of c1 is:"<<&c1<<endl;
c3.add(c1,c2);
```

So the add function basically creates a local object a in which it copies the content of c1. We do not want this copying. Instead we want the add function to reuse c1 for its operation. How can this be done? This can be done using the concept of reference variable. Change the add function as below:

```
void complex::add(const complex &a ,  const complex &b)
{
    cout<<"The address of a is:"<<&a<<endl;
        real = a.real + b.real;
        imag = a.imag + b.imag;
}
```

and also the function declaration in the class complex as

```
void add(const complex &, const complex &);
```

Run the code and you will see that the address of object "a" and "c1" are same.

IMPORTANT: Generally students have difficulty in getting the concept of **&** operator. You should always remember that if you have a datatype on the left as in "complex & a = b;" it will create a reference variable. This means there is only one object which has two names "a" and "b".

Whenever you do not have any datatype as in

```
cout<<"The address of a is:"<<&a<<endl;
```

where &a does not have a datatype, it will give the address of the object.

This should be very clear to you before you proceed.

Finally Change the add function as below:

```
void complex::add(const complex &a ,  const complex &b)
{
        real = a.real + b.real;
        imag = a.imag + b.imag;
}
```

and also the function declaration in the class complex as

```
void add(const complex &, const complex &);
```

Modify also the subtract, multiply and divide to use reference varilables as argument.

Fourth Step

The objective of this step is to learn how to use return by reference. In order to do this we implement following function as a member of the class complex. Include these lines in your code

```
complex & complex::maxComplex(complex &a, complex &b)
{
        double magnitude1, magnitude2;

        //compute the square of the magnitude of two complex numbers
        magnitude1 = a.real * a.real + a.imag*a.imag;
        magnitude2 = b.real * b.real + b.imag*b.imag;

        if (magnitude1 > magnitude2)
                return a;
        else
                return b;
}
```

The corresponding function declaration which should go in the class is

```
complex & maxComplex(complex &a, complex &b);
```

The maxComplex function returns the reference of the complex object whose magnitude is greatest of the two. In order to test this function implement following lines in the main program

```
    complex &c4 = maxComplex(c1, c2);
        cout << "Maximum: " << c4.getReal() << "+i" << c4.getImag();
        cout << "The address of c1 is:" << &c1<<endl;
        cout << "The address of c2 is:" << &c2 << endl;
        cout << "The address of c4 is:" << &c4<<endl;
```

Test this that c4 will have the same address as that of the greatest number of c1 or c2.

Fifth Step

Define a pointer of type complex in the main program and use this pointer to refer the add function of the complex class.

```
    complex * p;
        p = &c3;
        p->add(c1, c2);
        cout << "Sum:" << p->getReal() << "+i" << p->getImag()<<endl;
```

This step shows how to use a pointer to a user defined datatype in C++.

Sixth Step

You can use dynamic allocation of array of objects using a complex number using a pointer. The keyword which you can use is "new". Write a loop to take an array of complex numbers as an input from the user.

```
    int size;
        double a, b;
```

```
cout << "Enter the size of the array:";
cin >> size;
p = new complex[size];
//this loops takes the input
c3.setImag(0);
c3.setReal(0);
for (int i = 0; i < size; i++)
{
        cout << "Enter the " << i + 1 << " complex number";
        cin >> a >> b;
        p[i].setReal(a);
        p[i].setImag(b);
}
```

Seventh Step

Define the three constructors for the complex number in the class and test them in the main program.

```
//default constructor
    complex() { real = 0; imag = 0; };
    //copy constructor
    complex(complex &c) { real = c.real; imag = c.imag; };
    //parameterized constructor
    complex(double r, double i) { real = r; imag = i; };
```

Test these constructors to initialize the data members of the class in main program.

Eigth Step

Introduce a static member in the private section of the class

```
static int numberOfObjects;
```

The static data member is created only once for a class. This data member unlike "real" and "imag" will be shared by all objects of the class complex. Just Before main program include the following line

```
int complex::numberOfObjects = 0;
```

IMPORTANT: This line tells the compiler to set the memory for the static member. This should not be confused with the private member of a class is being accessed by the external world. Define a static member function which returns this value to a calling program as

```
static int getNumberOfObjects() { return numberOfObjects; };
```

The static member functions can access only static member functions. If you try to access the "real" or "imag" in the above function, it will give you error. Now modify the constructors as following

```
//default constructor
          complex()
          { real = 0; imag = 0; numberOfObjects++; };
          //copy constructor
          complex(complex &c)
          { real = c.real; imag = c.imag; numberOfObjects++;};
          //parameterized constructor
          complex(double r, double i)
          { real = r; imag = i; numberOfObjects++;};
```

Write the following line in the main program and test whether it prints the number of objects which has been created out of the class complex

```
cout << "The number of objects created from the complex class are "
<< complex::getNumberOfObjects();
```

Ninth Step

You can also define your parameterized constructor using an initialization list as

```
complex(double r, double i):real(r), imag(i) {  numberOfObjects++; };
```

This way of initializing the data will be seen a lot in the data structures assignments.