Chapter 0: Operator Overloading & C++ Class Friends
By Satish Singhal Ph. D.

Table of Contents

CS 2: C++ Data Structures
Topic 0: Operator Overloading & C++ Class Friends
Version 1.1

Introduction:

The C++ operators may be conveniently divided into four different categories. These are mathematical, relational, logical & bitwise and others. The Table 0.1 shows the operators as categorized above.

Operator category	List of operators
Mathematical	+ - / * % += -= *= /= %= ++ -- =
Relational	< > >= <= == !=
Logical & bitwise	& && | || ! >> << &= ^= |= <<= >>=
Other	, -> . new delete [ ] ( ) : ? ~ :: ->* ^ .* sizeof typeid

Table 0.1: The highlighted operators cannot be overloaded[1]

The operators shown in Table 0.1 have some standard C++ meaning. For example the operator = has the meaning that it assigns the value on the right side to the memory location named on the left side. However, the C++ allows the classes to give additional meanings and definitions to these operators, setting up a process called "*operator polymorphism*[2]". In setting up, operator polymorphism the C++ allows any operator to be defined as a member function[3] of a class and then the function definition provides the detailed workings of the "additional" meaning the operator has been given for that class. Understand that the operator polymorphism applies only to the objects of C++ class for which the operator has been redefined as a member function. This process, where C++ operators have been given additional

[1] The operators new and delete have following additional versions that can be overloaded: delete [ ], and new [ ], which respectively refer to deleting or allocating memory for an array.

[2] Literal meaning of polymorphism is "many forms". This word has been derived from Greek language. C++ has other forms of polymorphism as well. These include function name polymorphism also called function name overloading and polymorphism of class objects created by the inheritance relationship. Inheritance will be covered in future chapters.

[3] The operators can also be overloaded as friend functions or stand-alone functions. We show example of overloading as friend function later.

meanings is also called "*operator overloading*". Overloading process does not remove or compromise the original functionality of operators. It merely makes them *polymorphic* for a user-defined class. The operator shown in highlights (typeid, sizeof, ? :: . and .*) cannot be overloaded. Rest of the 41 operators are overloadable. Another way to divide the C++ operators is to classify then as *polymorphic* (the one that can be overloaded) and *monomorphic* (the one that cannot be overloaded). We begin by showing an example of operator overloading for a class that does arithmetic for complex numbers.

Complex Number Class and Uses of Operator Polymorphism

Operator overloading is very useful for those cases, where the extended meaning of the operator could result in writing a simpler client code. Take for example the *complex numbers*[4] in mathematics. Each complex number would have a *real* part and an *imaginary* part and they may be represented in the form:

$X = a + ib;$ where,
$i = (- 1)^{1/2}$

Where a is the real part, b is the imaginary part and i is square root of minus one. If we were to write a class called Complex to facilitate the arithmetical operations on complex numbers, then perhaps we may provide a member function called add, which can take another complex number as an argument, add it to the caller object, and return the sum. Notice that how the client code will look like in this situation. It may look something similar to the given below.

```
Complex  Num1;
Complex  Num2;
Complex  Num3;
Num3 = Num1.add (Num2); //Line 4
```

The last statement in above code fragment is supposed to add Num1 to Num2 and return the sum as a complex number and store in the memory location Num3. Writing the mathematical equations in the form like line four above is confusing and hard for the client[5]. Therefore it would be desirable if client can write the code such as below to add two or more complex numbers:

```
Num3 = Num1  +  Num2;
```

What one desires in this situation is that programmer of the class Complex gives additional meaning to the operator + so that client can use it to add complex numbers, thus simplifying the overall task of code writing for the client.

[4] Complex numbers are extremely useful, for example in Electrical Engineering. Impedance of an alternating current is for example can be represented by a real part and a imaginary part.
[5] Specially Mathematicians do not like to change the form of their equations and symbols when using computer program to solve mathematical equations.

Fortunately, C++ allows us to do that. We have already used some overloaded operators in C++ with out perhaps appreciating their use. Take for example the insertion operator << used by object cout to insert data into the output stream directed to the standard output. We use it with the syntax:

```
cout<<Var;
```

where Var could be any of the primitive C++ data types (integral[6] or floating point) or even a null terminated character array which is a C type string. What that means is that operator << has been overloaded to print many different types of data. You may recall from our previous discussion of relationship between cout and insertion operator[7] << that one may write the output statement as follows:

```
cout.operator <<(Var);
```

The dot operator invoking the extraction operator means that operator << is essentially a member function of the class whose object[8] is cout. Therefore, for C++ classes the operators can be made their member functions by using the keyword operator followed by the symbol for that operator. The body of the operator as member function then would provide the added functionality for that operator. We illustrate the process of operator overloading by designing a class representing complex numbers called Complex, in which the operators facilitating the complex number arithmetic and logical operations would be overloaded. Table 0.2 gives the summary of the design of the class Complex. (The *conceptual diagram* of how an object of class Complex is supposed to look like is given in Figure 0.2 on page 12).

[6] The enum data type is an exception as it cannot be directly outputted by the insertion (<<) operator.
[7] See Topic 5 (Standard input and output in C++) my Notes on CS1.
[8] cout is member of ostream class.

Class Complex defines a class to represent the Complex numbers in mathematics. Class would have private data members to store the real and imaginary parts of a Complex number. Class would also have the member functions and overloaded operators to input Complex numbers, print them and do arithmetical operations on them.	
Private Attributes (or data members)	
double	Real *The field Real holds the real part of a complex number.*
double	Imag *The field Imag holds the imaginary part of a complex number.*
Public Member Functions	
	Complex (double Real1=0.0, double Imag1=0.0) Overloaded constructor *with default arguments.*
	~Complex ( ) *Default destructor.*
bool	getData (istream &in, int flag) *getData takes an object of type istream, reads data either from the standard input or from a file, and fills the real and imaginary fields of the caller object.*
void	printData (ostream &out) const *Function printData prints the complex number to user directed output media.*
double	getReal ( ) const *Gets the real part of the* `Complex` *number.*

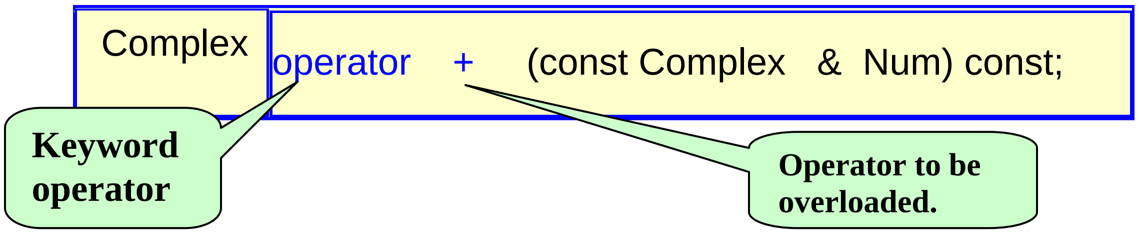
void	setReal (double Real1) *Sets the real part of the caller `Complex` number equal to the function argument.*
double	getImaginary ( ) const *Gets the imaginary part of the `Complex` number.*
void	setImaginary (double Imag1) *Sets the imaginary part of the caller `Complex` number equal to the function argument.*
Complex	operator+ (const Complex &Num) const *Overloaded operator + adds the caller `Complex` number to the `Complex` number passed in the argument and returns the sum.*
Complex	operator- (const Complex &Num) const *Overloaded operator - subtracts from the caller `Complex` number the `Complex` number passed in the argument and returns the difference.*
Complex	operator * (const Complex &Num) const *Overloaded operator * multiplies the caller `Complex` number with the `Complex` number passed in the argument and returns the product.*
Complex	operator / (const Complex &Num) const *Overloaded operator / divides the caller `Complex` number with the `Complex` number passed in the argument and returns the result.*

bool	operator== (const Complex &Num) const *The overloaded operator == tests if the caller and other* `Complex` *number are same or not.*
bool	operator!= (const Complex &Num) const *The overloaded operator != tests if the caller and other* `Complex` *number are equal to each other or not.*
bool	operator! ( ) const *Overloaded operator !(not) Tests to see if both, the real and imaginary parts of the caller* `Complex` *number are zero or not.*

<u>Table 0.2</u>

Notice that Table 0.2 is an expression of *pure design*. Nowhere do we indicate as how to implement the member functions, whose functionality is defined in the summary table above. *The process of expressing pure design before actual coding is done is a standard and powerful software engineering technique to reduce overall development time.*

The design indicates that we need two private data members of type double called Real and Imag, which would hold the real and imaginary part of a complex number. Since these two fields are private, and at times, client may need to change them or get an access to their current values. We therefore, need member functions getReal, setReal, and getImaginary and setImaginary for that purpose. To perform pure arithmetical operations we need to overload the operators + (plus) , – (minus), * (multiply), and / (divide). They must return the Complex number after the indicated mathematical operation. Let us discuss the design of one of the overloaded arithmetical operators to understand it further. For example let us take the Operator + (plus). The function header syntax for overloaded plus operator is designed as follows:



```
Complex    operator   +    (const Complex   &  Num) const;
```

Operator symbol after the keyword operator refers to operator to be overloaded. The space between the operator symbol and word operator is inconsequential. The argument list contains a Complex class type object, which is passed by reference. The reason for passing by reference is strictly to reduce the memory usage, which passing by value will increase. We pass a constant reference of the Complex type object to the operator +, because there is no need for us the change the object that is to be added to the caller object invoking the plus operator. *Therefore the access to object Num is kept strictly "read-only" access.* The return type is also Complex, indicating that the function should add the caller Complex number to the one in function argument and return their sum. The keyword const at the end of function header has the usual meaning that the function will not alter the state of the caller object and access to it also "read-only".

In Table 0.2, we overload two types of operators, the binary and unary. The operators +, -, *, /, ==, and != are binary operators as they must have two operands. The operator ! is a unary negation operator. You may have already noticed that the minimum number of arguments needed for the binary operators is one, whereas for unary operators it is zero (Table 0.2).

Implementation of Operator overloading functions

We show the implementation of plus (+) overloaded operator as an example. We know from the complex number algebra that when two complex numbers are added then the two real parts are added to form the real part of summed complex number. Same applies to the imaginary parts also. For example, consider two complex numbers given below C1 and C2;

$$C1 = A1 + iB1$$
$$C2 = A2 + iB2$$

Then the sum of C1 and C1, C3 is given as:

$$C3 = (A1+A2) + i(B1+B2)$$

Therefore, we may write the algorithm for the overloaded plus operator as follows (Figure 0.1).

Complex Complex::operator+(Const Complex & *Num*) const

Overloaded operator + adds the caller `Complex` number to the `Complex` number passed in the argument and returns the sum.
Precondition: The Complex number as an argument to the function has a non-null value.

Postcondtion: The sum of caller object and the object passed to the function is returned. The summed objects remain unchanged.
Algorithm:

```
Step1. Declare a temporary Complex object called Temp.
Step2. Set Temp.Real equal to Real plus Num.Real.
Step3. Set Temp.Imag equal to Imag plus Num.Imag.
Step4. return Temp.
```

Parameters:

Num	**is the `Complex` number to be added to the caller.**

Returns:

the `Complex` number which is sum of caller and the number passed in argument

<u>FIG. 0.1</u>

The Listing 0.1A shows the Complex.h file which includes the class definition, whereas the 0.1B shows the Complex.cpp showing the implementation of all member and helper functions. ***Compare the algorithm in Figure 0.1 with the implemented code for overloaded operator + in Listing 0.1B (lines 80 to 86 in Listing 0.1B) to see conversion of algorithm to C++ code.***

<u>Complex.h File</u>

```cpp
#ifndef Complex_H
#define Complex_H
#include <iostream>
#include <fstream>
using namespace std;
class Complex
{
private:
        double Real;
        double Imag;
public:
        Complex(double Real1=0.0,double Imag1 = 0.0);
        ~Complex();
        bool getData(istream& in, int flag );
        void printData(ostream& out) const;
        double getReal() const;
        void setReal(double Real1);
        double getImaginary() const;
        void setImaginary(double Imag1);
        Complex operator + (const Complex& Num)const;
        Complex operator - (const Complex& Num)const;
        Complex operator * (const Complex& Num)const;
        Complex operator / (const Complex& Num)const;
        bool operator == (const Complex& Num);
        bool operator != (const Complex& Num);
        bool operator ! ();
};
#endif //Listing 0.1A
```

Complex.cpp File

```
#include "Complex.h"
#include <cmath>
#include <string>
using namespace std;
const int ONE = 1;
const int TWO = 2;
const int THREE = 3;
//Helper functions
void Complex::printData(ostream& out) const
{
        if(Imag<0)
        {
                out<<Real<<" - "<<fabs(Imag)<<"*I"<<endl;
        }
        else
        {
                out<<Real<<" + "<<Imag<<"*I"<<endl;
        }
}
Complex::Complex(double RealPart, double ImagPart):
Real(RealPart), Imag(ImagPart)
{
}
Complex::~Complex()
{
}
bool Complex::getData(istream& in, int flag)
{
        double input = double();
        double num = double();
        bool is_input_ok = true;
        if(flag == ONE)
cout<<"Please enter the real part of the complex number: ";
        in>>num;
        if(!in.fail())
        {
                if(flag == ONE)
cout<<"Please enter the imaginary part of the complex number: ";
                in>>input;
                if(!in.fail())
                {
                        Real = num;
                        Imag = input;
                }
                else
                        is_input_ok = false;
        }
        else
                is_input_ok = false;

        return is_input_ok;
}
double Complex::getReal() const
{
        return Real;
}
void Complex::setReal(double real1)
{
        Real = real1;
}
double Complex::getImaginary() const
{
        return Imag;
}
void Complex::setImaginary(double Imag1)
{
        Imag = Imag1;
}
Complex Complex:: operator +(const Complex& number)const
{
        Complex Temp;
        Temp.Real = (Real + number.Real);
        Temp.Imag = (Imag + number.Imag);
        return Temp;
}
Complex Complex::operator - (const Complex& number)const
{
        Complex number3;
        number3.Real = (Real - number.Real);
        number3.Imag = (Imag - number.Imag);
        return number3;
}
Complex Complex:: operator *(const Complex& number)const
{
        Complex number3;
    number3.Real = ((Real*number.Real)-(Imag*number.Imag));
     number3.Imag = ((Real*number.Imag)+(Imag*number.Real));
        return number3;
}
Complex Complex::operator / (const Complex& number) const
{
        Complex number3;
        number3.Real =
((Real*number.Real)+(Imag*number.Imag))/(pow(number.Real,2)
+pow(number.Imag,2));
        number3.Imag = ((Imag*number.Real)-
(Real*number.Imag))/(pow(number.Real,2)+pow(number.Imag,2));
        return number3;
}
bool Complex::operator == (const Complex& rhs)
{
        if((Real == rhs.Real)||(Imag == rhs.Imag))
                return true;
        else
                return false;
}
bool Complex::operator != (const Complex& rhs)
{
        if((Real != rhs.Real)||(Imag != rhs.Imag))
                return true;
        else
                return false;
}
bool Complex::operator ! ()
{
        if((int(ceil(Real))==0)&&(int(ceil(Imag)) == 0))
                return true;
        else
                return false;
}
//Listing 0.1B
```

Figure 0.2 shows the conceptual diagram of as how an object of class Complex would look like.

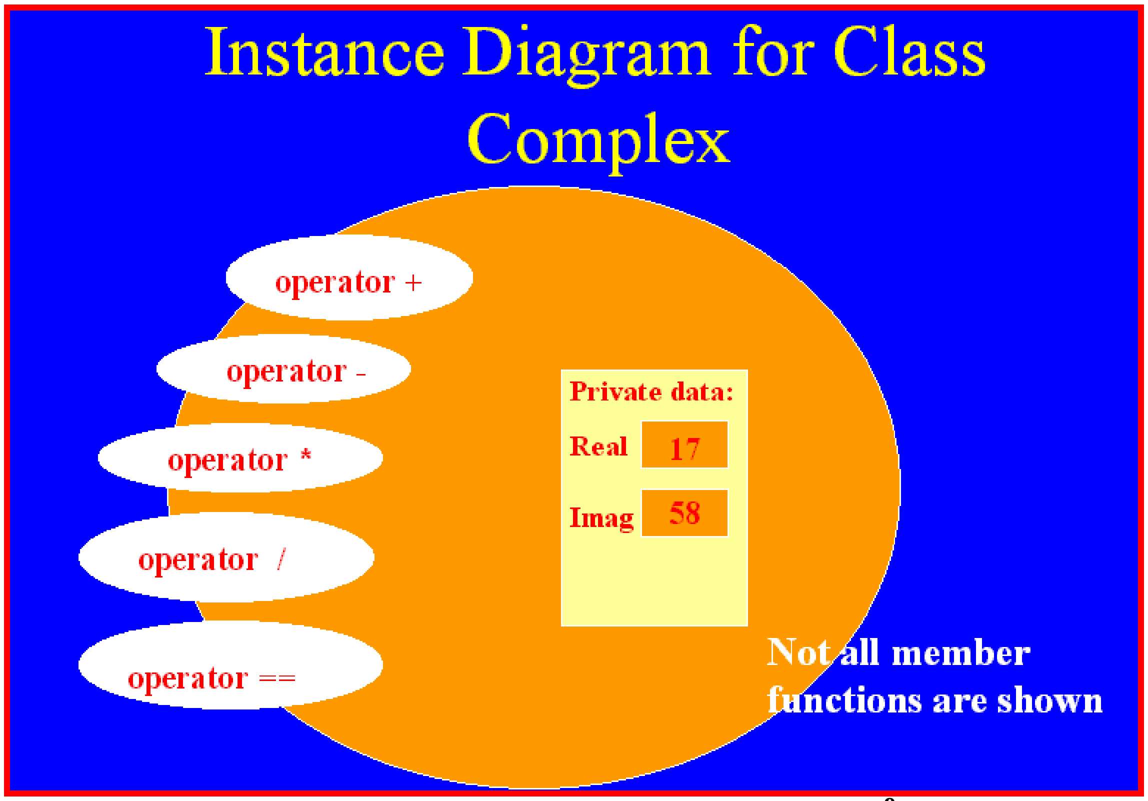


FIG. 0.2 Complex Number class – Conceptual Diagram[9]

The amber portion shows the class. The portion on yellow shows the private members of the class Complex. The public member functions are shown in white ovals lying partially outside the class indicating that they form the interface between the client and the private class members. Not all member functions are shown in Figure 0.2. We have given the algorithm for the operator +. For other operators we simply give the formula (Table 0.3), which can be easily converted to the algorithm and then to source code.

Complex Number C1	Complex Number C2	C1 – C2	C1/C2	C1*C2
a + ib	c + id	(a-b)+i(c-d)	(a*c+b*d)/(c^2+d^2) + i((b*c-a*d)/(c^2+d^2))	(a*c-b*d) + i(a*d+b*c)

Table 0.3: Complex Number Arithmetical Formulae

[9] I acknowledge Nell Dale's books for this type of visualization of a class object. See for example, the various editions of Programming and Problem Solving in C++ by Nell Dale and others.

Testing the class Complex

The minimal test we need to do is to ascertain that all overloaded operators give intended results and functions to get and display data work. Listing 0.1C shows the main program to test the class Complex.

```
#include "Complex.h"
#include <string>
using namespace std;
void openOutFile(ofstream& Out);
void openInFile(ifstream& input);
int main()
{
cout<<"This program will do complex number algebra for you."
<<" You would be able to add,   subtract, divide and multiply "
 <<" the complex numbers like ordinary numbers.\n";
cout<<"Preparing to get the data for the first complex number.\n";
        Complex Num1;
        if(Num1.getData(cin,1))
        {
              cout<<"The complex number you entered is: ";
                Num1.printData (cout);
        }
```

Gets and prints the complex number.

```
cout<<"Preparing to enter the data for the Second”
<<“ complex number.\n";
        Complex Num2;
        if (Num2.getData (cin, 1))
        {
               cout<<"The complex number you entered is: ";
                Num2.printData (cout);
        }

        Complex Num3 = Num1 + Num2;
```

Uses the overloaded operator +

```
       cout<<"The sum of complex numbers you entered is: ";
        Num3.printData (cout);

        Num3 = Num1 - Num2;
```

Uses the overloaded operator -

```
cout<<"The first complex number minus the second one is: ";
        Num3.printData (cout);

        Num3 = Num1/Num2;
```

Uses the overloaded operator /

```
     cout<<"The first complex number divided by second one is: ";
00061         Num3.printData (cout);
00062
00063         Num3 = Num1*Num2;
00064         cout<<"The product of two complex numbers is: ";
00065         Num3.printData (cout);
00066
00067         Complex Num4 = Num3;
00068         cout<<"Num3  = ";
00069         Num3.printData (cout);
00070
00071         cout<<"Num4  = ";
00072         Num4.printData (cout);
00073
00074         if (Num3 == Num4)
00075           cout<<"Num3 and Num4 are equal to each other.\n";
00076         else
00077       cout<<"Num3 and Num4 are not equal to each other.\n";
00078
00079         Complex Num5 = Num3 + Num4;
00080
00081         cout<<"Num5 = ";
00082         Num5.printData (cout);
00083
00084         if(Num5 != Num4)
00085                 cout<<"Num5 is not equal to Num4.\n";
00086         else
00087                 cout<<"Num5 is equal to Num4.\n";
00088
00089         Complex Num6 (0,0);
00090         cout<<"Num6 = ";
00091         Num6.printData (cout);
00092
00093         if (!Num6)
00094                cout<<"Negation of Num6 results in true.\n";
00095         return 0;
00096
00097 }
```

Uses the overloaded operator *

Tests the overloaded operator ==

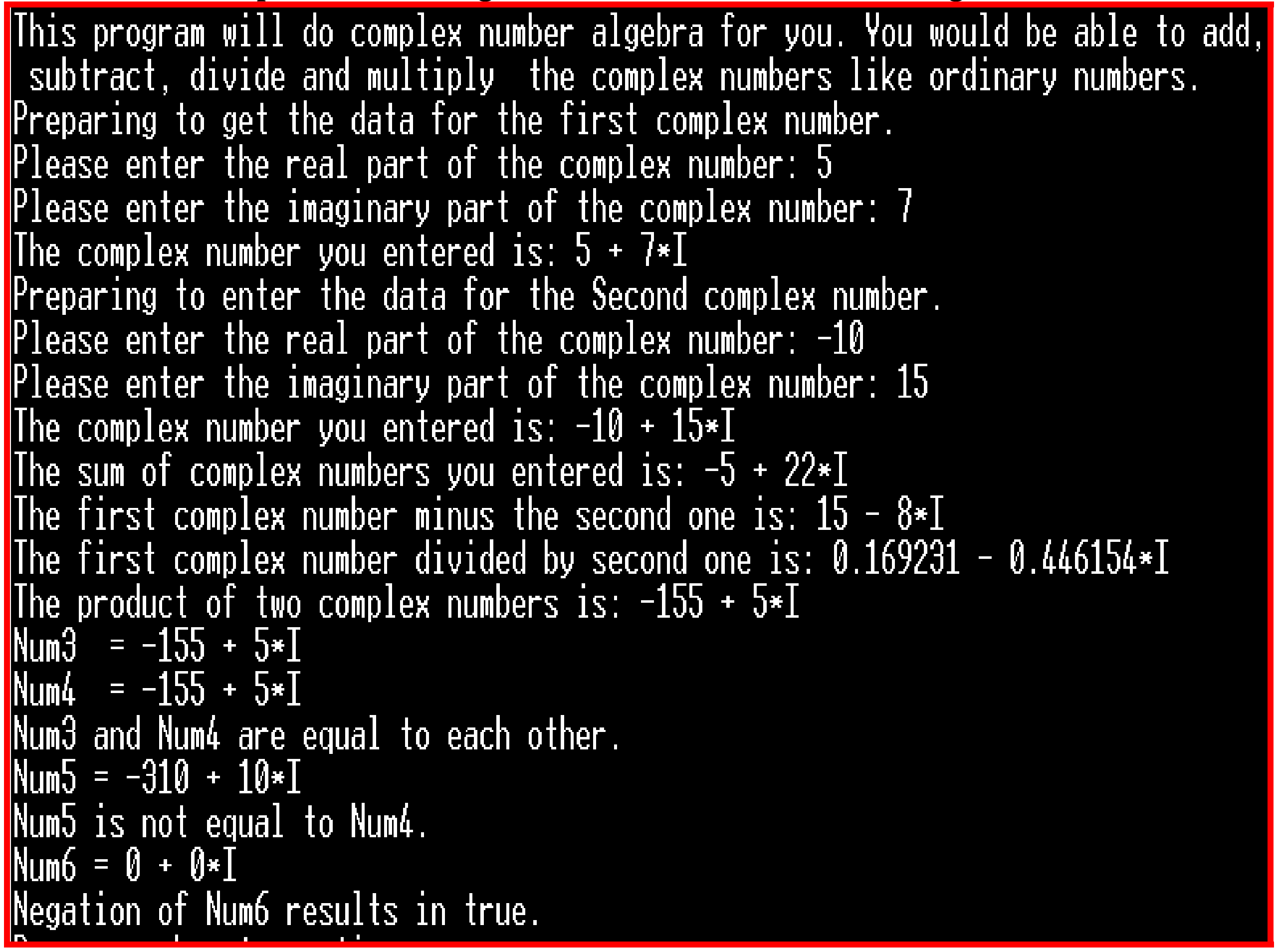
Tests the overloaded operator !=

Tests the overloaded operator !

//Listing 0.1C

We intentionally omit the code for function openOutFile and openInFile. The code for them is available in original file. These functions are to be used if client wishes input data from a file or output to a file, a feature we do not present here. Code line 33 to 38 show the procedure for getting and printing the user inputted complex

number. The code lines 48. 53, 58, 63, 74, 84 and 93 respectively test +, -, /, *, ==, !=, and ! overloaded operators. The Figure 0.3 shows the results of Listing 0.1C.



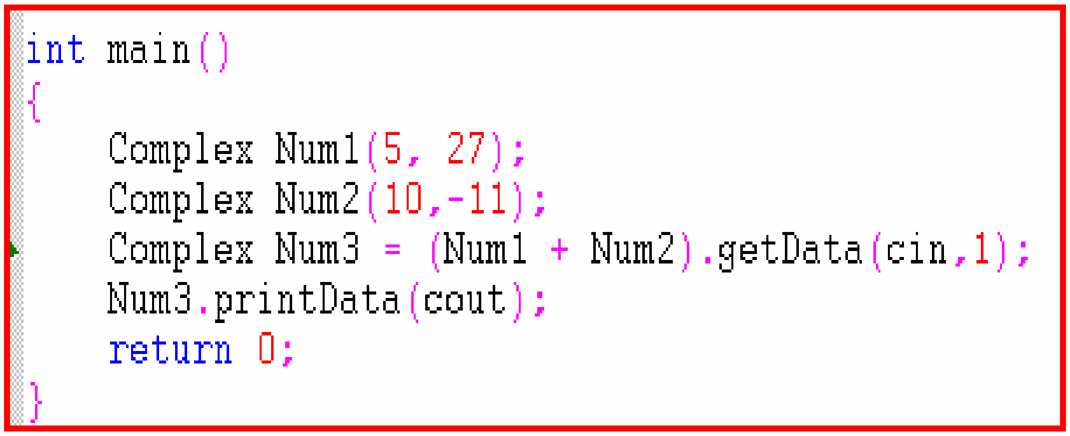
```
This program will do complex number algebra for you. You would be able to add,
 subtract, divide and multiply  the complex numbers like ordinary numbers.
Preparing to get the data for the first complex number.
Please enter the real part of the complex number: 5
Please enter the imaginary part of the complex number: 7
The complex number you entered is: 5 + 7*I
Preparing to enter the data for the Second complex number.
Please enter the real part of the complex number: -10
Please enter the imaginary part of the complex number: 15
The complex number you entered is: -10 + 15*I
The sum of complex numbers you entered is: -5 + 22*I
The first complex number minus the second one is: 15 - 8*I
The first complex number divided by second one is: 0.169231 - 0.446154*I
The product of two complex numbers is: -155 + 5*I
Num3  = -155 + 5*I
Num4  = -155 + 5*I
Num3 and Num4 are equal to each other.
Num5 = -310 + 10*I
Num5 is not equal to Num4.
Num6 = 0 + 0*I
Negation of Num6 results in true.
```

<u>FIG. 0.3C</u>

By applying the formulae given in Table 0.3, one can confirm that the overloaded operators indeed are giving the correct results. The negation operator simply tests if real and imaginary parts are actually zero. If they are then it, returns true, as the negation of both of them would be a boolean true value. This is so, as Num6 is set to zero and zero for both of its real and imaginary parts, whose negation results in true boolean value.

<u>Returning *constant objects*: Saving Client from Inadvertent Errors</u>

It is unlikely that client will write a code similar to the Listing 0.2, but inadvertent mistakes do happen.



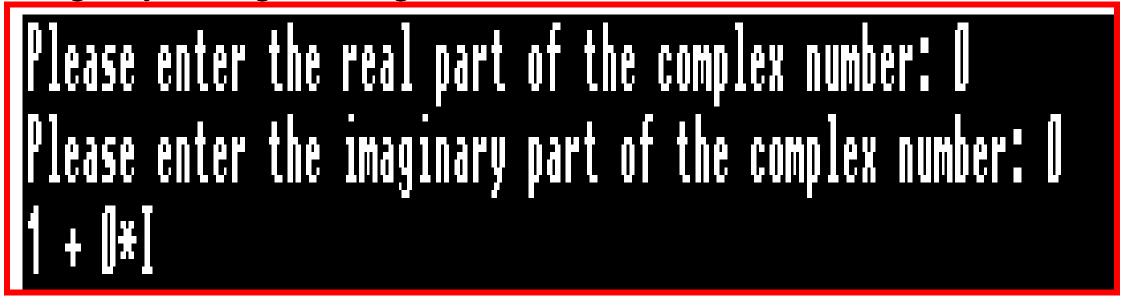
```
int main()
{
    Complex Num1(5, 27);
    Complex Num2(10,-11);
    Complex Num3 = (Num1 + Num2).getData(cin,1);
    Num3.printData(cout);
    return 0;
}
```

Listing 0.2

The affect of such inadvertent mistake can be serious at times. The output of Listing 0.2 shown in Figure 0.4 shows that printing Num3 does not give expected results of

15 + 16*i

because the object (Num1 + Num2) was not a constant object, thus it could be changed by making a call to getData member function.



```
Please enter the real part of the complex number: 0
Please enter the imaginary part of the complex number: 0
1 + 0*I
```

FIG. 0.4

Making the overloaded operator or other object returning functions return a constant object can eliminate this type of inadvertent error. This requires that the header of overloaded operator functions must be changed as follows:

const Complex operator + (const Complex& Num) const;

Makes the object to be returned a "read-only" object during return!

Understand that now the entire header: ***const Complex operator + (const Complex& Num) const;*** **provides the** ***three-level read-only-access*** **to the objects involved during the invocation of member function operator +:**

- **The caller object or the receiver has "read-only" access. This is enforced by the end const modifier.**
- **The returned object has 'read-only" access. This is enforced by the leading const modifier.**
- **Unless the constancy of reference is broken in the body of member function operator + or the object being passed has mutable members, the object Num,**

has "read-only" access. This is enforced by the const preceding the data type for the function argument.

Once the object returning functions are made to return constant objects, the object such as (Num1 + Num2) in Listing 0.2 cannot be modified, thus a call such as

```
Num3 = (Num1 + Num2).getData (cin, 1);
```

will cause a compile error similar to as follows:

error C2662: 'getData' : cannot convert 'this' pointer from 'const class Complex' to 'class Complex &' Conversion loses qualifiers.

What compiler is saying here is that before the result of operation Num1 + Num2 is stored into a storage location of type Complex, it cannot be modified by the function getData.

When overloaded operators do not return a const class object, then confusion may also be created when for example in the Listing 0.1C a code fragment such as below would compile.

```
Complex Num1, Num2, Num3;
Num1 + Num2 = Num3;
```

The reason that above would compile is that C++ considers (Num1 + Num2) as a *mutable object* of type Complex. Therefore, it feels free to assigns a value to it as mutable objects have L value. Of course the whole affect is more or less meaningless because the object on left hand side has no name so there is not much we can do with it. Yet it causes the confusing code to be compiled. This would not happen if overloaded operator + returns a const object, in which case a compile error will be issued if a statement such as: Num1 + Num2 = Num3; is attempted.

It is imperative that we modify the return type of overloaded operators, which returns a Complex object as shown in the Table 0.4 below.

Modified Headers for object returning overloaded operators for class Complex
const Complex operator+ (const Complex &Num) const
const Complex operator- (const Complex &Num) const
const Complex operator * (const Complex &Num) const
const Complex operator/ (const Complex &Num) const

Table 0.4

The headers of corresponding overloaded operator functions in the implementation file (Complex.cpp) will change accordingly. One example is shown below.

```
const Complex Complex::operator+( const Complex & Num ) const
```

The return is now qualified by words const and Complex jointly. Notice that we do not need this kind of protection when we are returning the primitives such as int, float or bool etc, because unlike objects, they are returned as literals. On the other hand, objects have functions, data, and other objects embedded in them and some functions can alter caller object.

Perhaps the value printed for Num3 in Figure 0.4 as 1 + 0*i will surprise some readers. Why the value set for real part is is one when the call to getData specifies it to be zero? This is because the function getData returns a true, which is internally represented as literal integer 1. Therefore in the program statement

```
Num3 = (Num1 + Num2).getData (cin, 1);
```

if function getData returns a true (which would be the case if input was successful), then the above statement in affect becomes as follows:

```
Num3 = 1;
```

The above assignment sets the real part of the Num3 to one and imaginary part to zero. This happens because the constructor for class Complex has default arguments for both real and imaginary parts. C++ examines the right side of the assignment in the statement Num3 =1; and does its best to make the assignment by making a constructor call by a process called *automatic type conversion*. It then treats the above statement as:

Automatic type conversion

```
Num3 = Complex (1);
```

Since the second argument is by default 0.0, it assigns the value of one to the real part of Num3. *In absence of default arguments in the constructor, the statement such as Num3 =1; will cause compile error, because C++ will not find a suitable constructor to make the assignment.*

Constructors have no return type but can make assignments

Some of you may be puzzled to see the constructor call we made above in the statement;

```
Complex Num3 = Complex (1);
```

It almost seems like that constructor is acting like a value returning function, even though constructor has no return type. This is exactly the case. The constructor

returns the constructed object even though there is no return type specified. Therefore, constructor call may be made in the right hand side of an assignment. With respect to the Listing 0.1, the following constructor calls are valid:

```
Complex Num1;
Complex Num2 = Complex ( );
```

Calls the default constructor with default values for both arguments.

```
Complex Num3 = Complex (10);
Complex Num4 (15)
```

Calls the explicit constructor with default value for the imaginary part

```
Complex Num5 = Complex (10, 11)
Complex Num6 (5, 7);
```

Calls the explicit constructor with given values for real and imaginary parts.

Understand that a statement such as;

```
Complex Num7 ( );
```

will compile! However, it does not create an object of type Complex with name Num7. *The reason is that compiler takes the above statement as a prototype of a function named Num7, which would return an object of type Complex.* Therefore, be careful as even though a constructor call like;

Complex Num6 (5,7) ;

is acceptable, because it does not resemble a function prototype declaration, the one such as Complex Num7 ( ); is only considered to be a declaration of function prototype.

The member functions, such as overloaded operators may return the object of type Complex by making a *constructor call directly in the return statement.* For example we may modify and shorten the code for overloaded operator + (plus) as follows:

```
const Complex Complex:: operator +(const
Complex& number) const
{
  return Complex (Real + number.Real, Imag +
number.Imag);
}//Listing 0.3
```

One could modify other overloaded operators, which return an object of type Complex accordingly by direct call to the constructor in the return statement. We leave that as an exercise for now.

Overloading Unary operators

We overloaded one unary operator in the Listings 0.1 and 0.2. Additional unary operators may be overloaded. However, we must understand their meanings and definitions carefully for the complex numbers. Table 0.5 describes five more unary operators we may overload for the class Complex.

Unary Operator	Example of application	Meaning when applied to a complex number
Unary negation (-)	-Num1	Sets both real and imaginary parts to opposite sign
Pre-increment	++Num	Pre-increments the real part by one.
Pre-Decrement	--Num	Pre-decrements the real part by one
Post-increment	Num++	Post-increments the real part by one
Post-decrement	Num--	Post-decrements the real part by one.

Table 0.5

Understand that post and pre fix operators only operate on the real part of the imaginary number, whereas the unary negation operates on both parts.

We saw in overloading the not unary operator (!) that when overloaded as a member function, the unary operator takes no arguments. However, one problem which would confound us is that how shall we then differentiate in the function prototype for pre-increment and post-increment operators? Similar problem exists in differentiating the function proto-types for pre-decrement and post-decrement operators. For example, we could use the following proto-type for the pre-increment operator:

```
const Complex operator ++( );
```

Proto-type for pre-increment member function. Same style is followed for pre-decrement.

Since no two function proto-types in C++ can be exactly alike, we cannot use the same proto-type for post-increment. Therefore we must differentiate the prototype for post-increment by using an extra argument, even though this argument plays no

role whatsoever in the function definition. ***This extra argument is just a marker[10] for the compiler, and in reality the function body for post-increment or post-decrement operators does not use this argument.*** **Table 0.6 gives the summary description of the five overloaded operators shown in Table 0.5.**

const Complex	operator++ ( ) *Overloaded operator ++ pre-increment increases the real part by one first and returns the reconstituted `complex` number.*
const Complex	operator++ (int marker) *Overloaded operator ++ post-increment first returns the current value of `complex` number and then increments its real part by one. The argument marker is not used.*
const Complex	operator-- ( ) *Overloaded operator -- pre-decrement decreases the real part by one first and returns the reconstituted `complex` number.*
const Complex	operator-- (int marker) *Overloaded operator -- post-decrement first returns the current value of `complex` number and then decrements its real part by one. The argument marker is not used.*
const Complex	operator- ( ) *Negation operator - simply changes the signs of Real and Imaginary parts to opposites, and then returns the `Complex` number, which is negative of its previous form.*

Table 0.6

Listing 0.4A shows the new source code for the Complex class.

[10] **Marker can only be used in post (increment or decrement) operators.**

```cpp
#ifndef Complex_H
#define Complex_H
#include <iostream>
#include <fstream>
using namespace std;

class Complex
{
private:
        double Real;
        double Imag;
public:
        Complex(double Real1=0.0,double Imag1 = 0.0);
        ~Complex();
        bool getData(istream& in, int flag );
        void printData(ostream& out) const;
        double getReal() const;
        void setReal(double Real1);
        double getImaginary() const;
        void setImaginary(double Imag1);
        const Complex operator + (const Complex& Num)const;
        const Complex operator - (const Complex& Num)const;
        const Complex operator * (const Complex& Num)const;
        const Complex operator / (const Complex& Num)const;
        bool operator == (const Complex& Num) const;
        bool operator != (const Complex& Num) const;
        bool operator ! () const;
       const Complex operator ++(); //pre-increment
    const Complex operator ++(int marker); //post-increment
        const Complex operator --(); //pre-decrement
    const Complex operator --(int marker); //post-decrement
        const Complex operator - ();//unary negation
};

#endif
//Listing 0.4A
```

The relevant portion of Comple.cpp implementation file is shown in Listing 0.4B, where we only show the code for the five new overloaded operators listed in Table 0.5.

```cpp
const Complex Complex::operator ++() //pre-increment
{
        return Complex (++Real, Imag);
}
```

```
const Complex Complex::operator ++(int marker) //post-increment
00142 {
00143         double temp_real = Real;
00144         double temp_imag = Imag;
00145         Real++;
00146         return Complex (temp_real, temp_imag);
00147 }
00148 //---------------------------------------------------------
00149 const Complex Complex::operator --() //pre-decrement
00150 {
00151         return Complex (--Real, Imag);
00152 }
00153 //-------------------------------------------------------
const Complex Complex::operator --(int marker) //post-decrement
00155 {
00156         double temp_real = Real;
00157         double temp_imag = Imag;
00158         Real--;
00159         return Complex (temp_real, temp_imag);
00160 }
00161 //--------------------------------------------------
00162 const Complex Complex::operator - ()
00163 {
00164         return Complex (-Real, -Imag);
00165 }//Listing 0.4B
```

Save the current values of real and imaginary parts to be used to return the current state of the object.

Increment the real part.

Return the object before incrementation.

Algorithm is same as for post-increment operator.

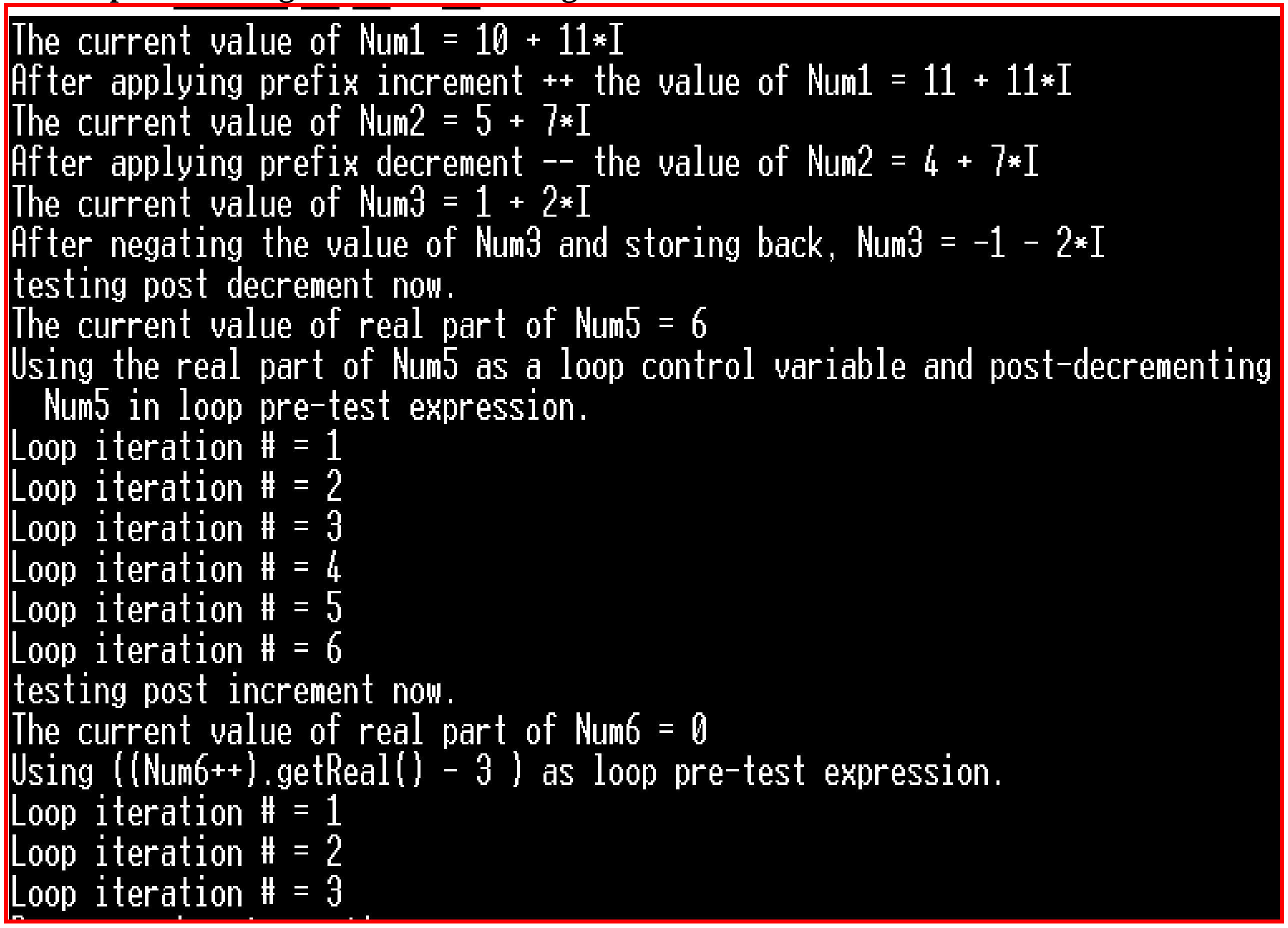
In Listing 0.4B, the algorithms for prefix increment or decrement and for pure negation are trivial. The post-fix increment or decrement operator algorithms need some explanation. As shown for the postfix increment operator, one would need to save the current values of the real and imaginary parts of the Complex object before incrementing the real part (Lines 143 & 144 in Listing 0.4B). Then we increment the Real part. Finally, we return the Complex object reconstituted by the real and imaginary parts saved on lines 143 &144 of code in Listing 0.4B. This ascertains that the real part is incremented after the original object copy is returned[11]. The postfix decrement operator is overloaded in the similar manner. Listing 0.4C shows the main function to only test the five-overloaded operator discussed here.

[11] Think of this in the following manner: it is as if the object is returned but post-fix operator stays attached to the real part of the object. Example is shown later.

```cpp
#include "Complex.h"
#include <string>
using namespace std;
int main()
{
        Complex Num1 (10,11);
        Complex Num2 (5,7);
        Complex Num3 (1,2);
        Complex Num4 (6,22);
        Complex Num5 (6,2);
        //Testing prefix operators
        cout<<"The current value of Num1 = ";
        Num1.printData (cout);
cout<<"After applying prefix increment ++ the value of Num1 = ";
        ++Num1;
        Num1.printData (cout);
        cout<<"The current value of Num2 = ";
        Num2.printData (cout);
cout<<"After applying prefix decrement -- the value of Num2 = ";
        --Num2;
        Num2.printData (cout);
        //Testing negation
        cout<<"The current value of Num3 = ";
        Num3.printData (cout);
cout<<"After negating the value of Num3 and storing back, Num3 = ";
        Num3 = -Num3;
        Num3.printData (cout);
        //Testing post-decrement
        cout<<"testing post decrement now.\n";
        cout<<"The current value of real part of Num5 = ";
        cout<<Num5.getReal ()<<endl;
cout<<"Using the real part of Num5 as a loop control variable "
                <<"and post-decrementing    "
                <<"  Num5 in loop pre-test expression.\n";
        int counter = 0;
        while ((Num5--).getReal())
        {
           cout<<"Loop iteration # = "<<++counter<<endl;
        }
        //Testing post increment operator
        cout<<"testing post increment now.\n";
        Complex Num6 (0,12);
        cout<<"The current value of real part of Num6 = "
                <<Num6.getReal ()<<endl;
        cout<<"Using ((Num6++).getReal() - 3 ) as loop "
00047                 <<"pre-test expression.\n";
00048          counter = 0;

00049          while ((Num6++).getReal() - 3 )
00050          {
00051               cout<<"Loop iteration # = "<<++counter<<endl;
00052          }
00053
00054          return 0;
00055 }//Listing 0.4C
```

The output of Listing 0.4C is shown in Figure 0.5.



```
The current value of Num1 = 10 + 11*I
After applying prefix increment ++ the value of Num1 = 11 + 11*I
The current value of Num2 = 5 + 7*I
After applying prefix decrement -- the value of Num2 = 4 + 7*I
The current value of Num3 = 1 + 2*I
After negating the value of Num3 and storing back, Num3 = -1 - 2*I
testing post decrement now.
The current value of real part of Num5 = 6
Using the real part of Num5 as a loop control variable and post-decrementing
  Num5 in loop pre-test expression.
Loop iteration # = 1
Loop iteration # = 2
Loop iteration # = 3
Loop iteration # = 4
Loop iteration # = 5
Loop iteration # = 6
testing post increment now.
The current value of real part of Num6 = 0
Using ((Num6++).getReal() - 3 ) as loop pre-test expression.
Loop iteration # = 1
Loop iteration # = 2
Loop iteration # = 3
```

FIG. 0.5

Once again, the testing of pre-fix and negation operators is trivial as they can be applied and printed in the normal manner. The results for pre-fix and negation operators are shown in first six lines of Figure 0.5. In testing post-decrement operator we create Num5 as the Complex object with the real part being equal to six (Line 11 in Listing 0.4C). Then we use the expression:

```
(Num5--).getReal()
```

as loop pre-test expression. Notice that in this expression, first, the current value of Num5 is accessed and then it is decremented. Since the real part of the Num5 was

six, the loop should run six iterations. That is exactly the result shown in Figure 0.5. In order to test the post-increment we use the following loop pre-test expression:

```
(Num6++).getReal() - 3
```

We set the value of real part of Num6 to zero when we construct the object Num6 (Line 43 in Listing 0.4C). The loop should exit when real part of Num6 becomes three as the difference in the pre-test expression will then be zero. This would run the loop for three iterations – a result confirmed by Figure 0.5.

Friend Functions and/or Polymorphic operators, which are members of other classes!

We have used the function getData and printData in order to input the values, and print a Complex object respectively. The user however may simply wish to use the standard extraction operator >> for data input and insertion operator << for output. Can we overload them for class Complex so that a code similar to the one given below may be written?

```
Complex  Num1;
cin>>Num1; // Allows user to input the real and imaginary parts
cout<<Num1;//Prints the complex number to standard output
```

The problem in doing so is that the extraction and insertion operators are already member functions of istream and ostream classes respectively. The same operator cannot be member of two classes and work as messenger on their objects at the same time as a binary operator! Yet, exactly this kind of functionality is needed in overloading the extraction and insertion operators to make them work for class Complex. C++ allows a remedy for this situation. The remedy is that the *members of any class can become friends of another class.* A class can also become a friend of another class. Therefore, a public member function of class Class1 can become a friend function of the Class2. The friend functions have same privileges in a class as member functions do[12]. Therefore, friend functions have access to all the other resident members of the friend class including the private members. *The syntax of declaring the friend function includes using the keyword friend followed by the function proto-type.* The friend functions can be placed in any of the private, public or default areas of a class. However, since friend functions cannot be private members, it makes sense to declare them in the public portion of a class. The example below shows the syntax.

[12] Friend functions cannot use the modifiers such as const in the end.

```
class SomeClass
{

        private:
                //data members
        public:
                friend int someFunction (int value, char data);
};
```

Friend functions may be placed in the default access area, or in areas marked private.

Generally, it is best to place friend functions in the public access area.

Keyword friend is needed as first word.

Followings are the *requirements for declaring friend functions*:

- The keyword friend is required as the first word in the function prototype (but is skipped when full definition is written in the implementation file).
- Friend functions can be placed with in the scope of any of the access control keywords. However, it is best to put them under public access control as such access is the only one accorded to them.
- In the implementation file the friend functions do not need the scope resolution operator to bind them to the class as they are actually not the members of the class. (They are just friends).
- Friend functions have access to the private data members of the class they are friends to.
- Unlike the standard C++ function name overloading, name of friend function can be overloaded.

Therefore in the implementation file, the source code for the friend function is written in a manner similar to the one written for stand-alone functions. For example, the code for someFunction shown in the example above will be written as follows in the implementation file.

```
int someFunction (int value, char data)
{
        //code here
}
```

Implementation of friend function does not need the keyword friend or the scope resolution operator.

In addition, the friend function has to have the same name as it does in its member class. However, its return type and argument list can be altered by the class, which defines it as a friend. *The friend functions are not allowed to have an end modifier*. For example, it would be a compile error if we defined a friend function as:

```
int someFunction (int value, char data) const; //compile error
```

The limitation that friend functions are not allowed const end modifier makes their excessive use unsafe, as they actually are allowed to modify the private data members of the class and may inadvertently do so.

Function Returning a Reference

Before we can show the method of overloading the insertion and extraction operators of iostream class, we need to discuss one more topic, where functions return values by reference with the ampersand (&) attached to the return type. The proto-type of such a function may look like the one given below:

int& myFunction (int& value);

For generalization, replace int by other data types.

The above example shows only one case and there would be cases that function has to take no parameters as arguments. *General rule about functions returning a reference is that either the variable they return a reference to must have global access, or the reference to that variable must be passed as an argument to the function*[13]. Listing 0.5 shows one example of the function using the reference return mechanism.

```
#include <iostream>
using namespace std;

int my_int = 10;
int& myFunction();

int main()
{
        cout<<"The value of my_int = "<<my_int<<endl;
        my_int = 99;
        int  your_int = myFunction();
        cout<<"The value of your_int = "<<your_int<<endl;
        cout<<"Setting myFunction() = 55\n";


        myFunction() = 55;


cout<<"Value of myFunction printed as cout<<myFunction =  "
                <<myFunction()<<endl;
        cout<<"The value of my_int =  "<<my_int<<endl;

        return 0;
```

Global integer my_int defined

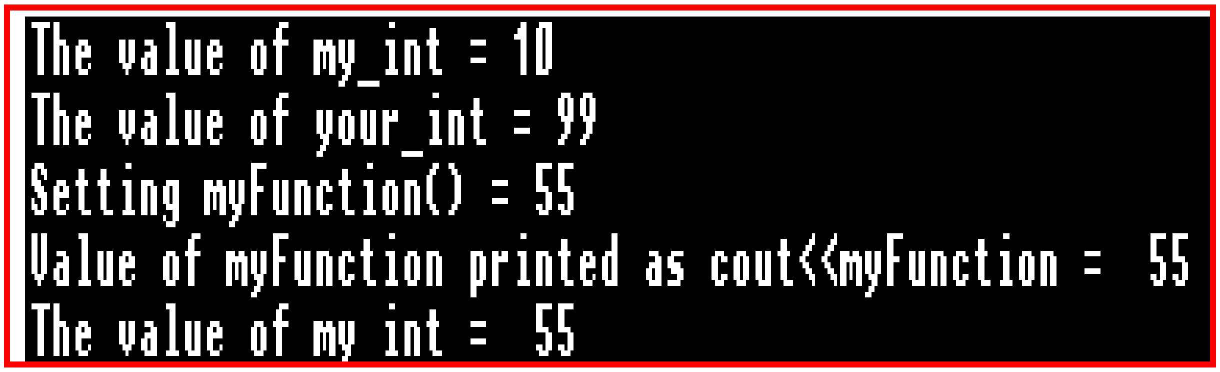
Notice that because myFuction is an int&, it can be assigned a value!

[13] This rule is related to the fact that once the function call is complete local variables are destroyed. Therefore any reference to local variables cannot be returned.

```
00020 }
00021
00022 int& myFunction()
00023 {
00024         return my_int;
00025 }
//Listing 0.5
```

Returns the reference to global my_int.

In Listing 0.5 there is a global integer my_int defined (Line #10). The definition of the function myFunction (lines 22 to 25) shows that this function returns the my_int. ***Since the function returns an int&, it actually returns a reference to global my_int.*** **Figure 0.6 shows the output of Listing 0.5.**



```
The value of my_int = 10
The value of your_int = 99
Setting myFunction() = 55
Value of myFunction printed as cout<<myFunction =  55
The value of my int =  55
```

<u>FIG. 0.6</u>

The line #9 of Listing 0.5 simply prints the current value of global my_int, which is 10. In line #10 we set the global my_int to 99, and then in next line set your_int equal to the value returned by the function myFunction. After that, outputting your_int (line #12) prints a value of 99. A new thing, allowed by C++ reference return type, happens when in line #14 we are able to set myFunction( ) equal to a value of 55! This unthinkable possibility, where a function call may be assigned a value happens because the ***function name myFunction ( ) actually becomes an alias for whatever variable the function is returning.*** **Thus the name myFunction ( ) is actually an alias for global my_int. Therefore when we print the values of myFunction ( ) and my_int on lines #16 and 17, they both print 55.** ***The functions, which have a reference return type, acquire an L-value, which is actually the variable they are returning. Therefore, they may appear in the left hand side of an assignment expression.***

<u>Overloading of Iostream Operators</u>

Now we proceed to overload the insertion (<<) and extraction (>>) operators for the class Complex, so that we can automate the Complex number input and output. Before we do that however, it is instructive to see as to how the classes' istream and ostream actually define their member operators. The Listing 0.6A shows portions of

class istream, where the extraction operator (>>) is overloaded. The Listing 0.6B shows the overloading style for insertion operator (<<).

```
istream& operator>>(short &);
istream& operator>>(unsigned short &);
istream& operator>>(int &);
istream& operator>>(unsigned int &);
istream& operator>>(long &);
istream& operator>>(unsigned long &);
istream& operator>>(float &);
istream& operator>>(double &);
istream& operator>>(long double &);
//Listing 0.6A
ostream& operator<<(short);
ostream& operator<<(unsigned short);
ostream& operator<<(int);
ostream& operator<<(unsigned int);
ostream& operator<<(long);
ostream& operator<<(unsigned long);
ostream& operator<<(double);
ostream& operator<<(long double);
//Listing 0.6B
```

We notice that for both extraction as well as insertion operators the return type is either istream& or ostream&. This return of reference makes the cascaded input and output possible. After each operator application, the original stream is returned back, therefore the input and output may be cascaded. Thus the following types of input or output become possible:

cin>>var1>>var2>>var3………;
cout<<var1<<var2<<var3……..;

If we did not wish to have the capability for cascaded input and output as shown above, *we could overload the insertion and extraction operators as void functions.* However, there is no reason for not wanting cascading capability. Therefore, we may wish to overload the extraction operator << as a friend function for class Complex with the following proto-type:

friend istream& operator >> (istream& input, Complex& Num);

One may inquire as to why we need the extra argument *istream& input* in above function? The reason is that at times we may need to use the above operator for reading the data from a file. There is no single global object of type istream, which can be returned by reference for cases where input is taken from a file or from a keyboard. Therefore, it is best to pass the reference to istream object as an argument and return the same one back as a reference return type. *General rule when reference return type is invoked is to return the reference to either a globally available variable, or a variable passed to the function by reference, or to a variable created dynamically on the heap portion of memory*[14]. Understand that the Complex number Num needs to be passed by reference also, because the extraction operator needs to change it and populate its data fields from user input.

The overloaded insertion operator proto-type is written as follows:

```
friend ostream&  operator << (ostream& out, const Complex& Num);
```

We pass the Complex number to be outputted by the insertion operator by reference only because it is memory efficient. However, we need to ascertain that the operator << does not change the Num as it is merely supposed to print it. Therefore, we pass a constant reference to the Num to be printed (unless such constancy is removed by a constant cast on purpose). We must be in a hurry and see what the code for the above two overloaded operators may look like. However, we need to wait, just a bit longer as we wish to explain the logic for overloading the previously overloaded operators +, -, *, /, ==, and != also as friend functions.

Automatic Type conversion by overloading member functions as friend functions

The way we wrote the overloaded binary operators +, -, *, /, ==, and != in Listing 0.4, the following operations are certainly possible:

```
Complex Num1 (10,12);
Complex Num2;

Num2 = Num1 +1;
Num2 = Num1 – 2;
Num2 = Num1*3;
Num2 = Num1/4.4;
bool is_equal = Num1 == 5;
bool not_equal = Num1 != 4;
```

Literals on the right hand side of overloaded binary operators are automatically converted to type Complex because we have a constructor, which takes the default arguments.

[14] The concept of dynamically created variables on heap will become clear after we discuss pointers in future.

The reason for that is that C++ looks at the argument on right hand side of the binary operators and even though it is not of type Complex, it has a constructor available which takes the default arguments. Therefore, using that constructor the C++ converts the literals on the right hand side of the binary operators into a complex number. In doing so, the real part is set equal to the value of the literal and the imaginary part to zero (the second default argument). ***This is one example of automatic type conversion.*** **The reason C++ can do that is because the code line such as:**

```
Num2 = Num1 +1;
```

is executed as;

```
Num2 = Num1.operator +(1);
```

Now the argument for the overloaded operator + must be a Complex type instead of a literal integer float or some other. What C++ does in this case is to see if it can find a constructor in Complex class, which can actually convert the literal 1 to a Complex type. Lo and behold, it does find the constructor, which takes the default values for both real and imaginary parts. Thus, C++ uses that constructor to convert literal 1 into the complex number of the form;

1 + 0*i

Therefore, the net affect of the above operation is that one is added to the real part of Num1 and the result is stored in Num2. Same procedure is repeated to convert the other binary operator applications on previous page. Thus, the ***automatic type conversion works for overloaded operators as member functions.*** **The literal or variable of primitive type to be converted is passed as the argument to the operator function which also means that such literal or primitive variable must be the right hand side operand.** ***However, when such literal or primitive variable is made a left hand operand, then the automatic type conversion fails.*** **For example in Listing 0.4, the following will cause a compile error:**

```
Num2 =  1 + Num1;  //Compile Error!
```

It is important to understand why something like that will fail. Understand that the right hand side of above assignment purports an operation similar to:

```
1.operator+(Num1);
```

However, the literal 1 is not a type Complex and it cannot call the operator +. One may argue that why C++ cannot automatically type convert 1 to Complex object here. ***The reason it cannot do so is that it has no way of knowing as what is coming after the + operator.*** **For all it knows it could be another literal integer or float. For addition of primitives, it can upgrade the storage on the fly, but it has no way of coming back and re-converting the literal 1 to complex. Therefore, compiler does**

not even try and issues a compile error as it parses the expression. The automatic type conversion of left-hand operand of the binary overloaded operators can be done if we overload the binary operators as friend functions, instead of the member functions. For example, the proto-type for the overloaded plus (+) operator would be altered as follows:

friend const Complex operator +(const complex& Num1, const Complex & Num2);

Therefore, when a binary operator is overloaded as a member function, it must take minimum of two arguments, one for each operand. With an overloaded + operator as above, both of the statement below become legal and give proper results:

Num2 = Num1 +1;
Num2 = 1 + Num1;

This is so because both operands are being passed as function arguments, and if one of them is a literal or primitive variable, C++ can call a constructor to convert it to type Complex by using the default argument constructor. Same logic holds for overloading other binary operators as friend functions. The automatic type conversion further simplifies the task of writing client code.

Modified Complex class with friend functions and overloaded operators:

Listing 0.7A gives the modified Complex class with the new friend functions and operators. We deleted the two former member function getData and printData, as they are no longer needed. We added a new member function called absolute which finds the absolute value of a complex number[15].

```
#ifndef Complex_H
#define Complex_H
#include <iostream>
#include <fstream>
using namespace std;
class Complex
{
private:
        double Real;
        double Imag;
public:
        Complex(double Real1=0.0,double Imag1 = 0.0);
        ~Complex();
        double getReal() const;
```

[15] **The absolute value of a complex number = (real*real + imaginary*imaginary)$^{1/2}$.**

```
       void setReal(double Real1);
       double getImaginary() const;
       void setImaginary(double Imag1);
      friend const Complex operator + (const Complex&
Num1, const Complex& Num2);
      friend const Complex operator - (const Complex&
Num1, const Complex& Num2) ;
      friend const Complex operator * (const Complex&
Num1, const Complex& Num2);
      friend const Complex operator / (const Complex&
Num1, const Complex& Num2) ;
      friend bool operator == (const Complex& Num1, const
Complex& Num2) ;
      friend bool operator != (const Complex& Num1, const
Complex& Num2) ;
       bool operator ! () const;
       const Complex operator ++(); //pre-increment
       const Complex operator ++(int marker); //post-
increment
       const Complex operator --(); //pre-decrement
       const Complex operator --(int marker); //post-
decrement
       const Complex operator - ();
   friend ostream&  operator << (ostream& out, const
Complex& Num);
   friend istream&  operator >> (istream& input, Complex&
Num);
       double absolute() const;
};

#endif
```
//Listing 0.7A

The summary of new member and friend functions is given in Table 0.7 below.

double	**absolute ( ) const**
	Member function absolute finds and returns the absolute value of the Complex number using the following formula: absolute value = sqrt(Real*Real + Imag*Imag).
Friends	
const Complex	**operator+ (const Complex &Num1, const Complex &Num2)**

	Overloaded operator + adds the Num1 `Complex` number to the Num2`Complex` number passed in the arguments and returns the sum.
const Complex	**operator- (const Complex &Num1, const Complex &Num2)**
	Overloaded operator - subtracts from the Num1 `Complex` number the Num2 `Complex` number, both passed as the arguments and returns the difference.
const Complex	**operator * (const Complex &Num1, const Complex &Num2)**
	Overloaded operator (*) multiplies the Num1 `Complex` number with the Num2 `Complex` number (both passed as the arguments) and returns the product.
const Complex	**operator/ (const Complex &Num1, const Complex &Num2)**
	Overloaded operator (/) divides the Num1 `Complex` number with the `Complex` number Num2 (both passed as the arguments) and returns the result.
bool	**operator== (const Complex &Num1, const Complex &Num2)**
	The overloaded operator == tests if the Num1 and Num2 `Complex` numbers are same or not.
bool	**operator!= (const Complex &Num1, const Complex &Num2)**
	The overloaded operator != tests if the Num1 and Num2 `Complex` numbers are equal to each other or not.
ostream &	**operator<< (ostream &out, const Complex &Num)**

	Overloaded insertion operator prints the `Complex` number by using the variable name as it would for primitive data types. ***Pre-condition: The object Num must not be a null reference.*** ***Post-condition: The object Null is not altered.***
istream &	**operator>> (istream &input, Complex &Num)**
	Overloaded extraction operator takes the user input for the real and imaginary parts of the `Complex` number Num either from the keyboard or from the file. ***Pre-condition: The Object Num must not be a null reference.*** ***Post-condition: depending on user input the object Num may be altered.***
Table 0.7	

We also show implementation of only the member and friend functions listed in Table 0.7, as all other functions remain unaltered (Listing 0.7B).

```
//-----------------------------------------------------
const Complex operator + (const Complex& Num1, const Complex& Num2)
{
return Complex ((Num1.Real + Num2.Real),(Num1.Imag + Num2.Imag));
}
//--------------------------------------------------------
const Complex operator - (const Complex& Num1, const Complex& Num2)
{
return Complex((Num1.Real - Num2.Real),(Num1.Imag - Num2.Imag));
}
//--------------------------------------------------------
const Complex operator * (const Complex& Num1, const Complex& Num2)
{
return Complex(((Num1.Real*Num2.Real)-(Num1.Imag*Num2.Imag)),
((Num1.Real*Num2.Imag)+(Num1.Imag*Num2.Real)));
}
//--------------------------------------------------------
```

```
const Complex operator / (const Complex& Num1, const Complex& Num2)
{
        return Complex(((Num1.Real*Num2.Real)+(Num1.Imag*Num2.Imag))/(pow(Num2.Real,2)+pow(Num2.Imag,2)),
        ((Num1.Imag*Num2.Real)-(Num1.Real*Num2.Imag))/(pow(Num2.Real,2)+pow(Num2.Imag,2)));
}
//-----------------------------------------------------------
bool operator == (const Complex& Num1, const Complex& Num2)
{
   if((Num1.Real == Num2.Real)&&(Num1.Imag == Num2.Imag))
                return true;
        else
                return false;
}
//-----------------------------------------------------------
bool operator != (const Complex& Num1, const Complex& Num2)
{
     if((Num1.Real != Num2.Real)||(Num1.Imag != Num2.Imag))
                return true;
        else
                return false;
}
//---------------------------------------------------------
ostream&  operator << (ostream& out, const Complex& Num)
{
        if(Num.Imag<0)
        {
          out<<Num.Real<<" - "<<fabs (Num.Imag)<<"*I"<<” “;
        }
        else
        {
         out<<Num.Real<<" + "<<Num.Imag<<"*I"<<”  “;
        }

        return out;
}
//---------------------------------------------------------
istream& operator >>(istream& input, Complex& Num)
{
        input>>Num.Real>>Num.Imag;
        return input;
}
//----------------------------------------------------------
```

```
double Complex::absolute() const
{
        return (sqrt(Real*Real +Imag*Imag));
}
//----------------------------------------------
//Listing 0.7B
```

Testing the Class Complex in Listing 0.7

We test the class Complex for normal functioning as well as for automatic type conversion when left hand side of binary operators includes literals. The Listing 0.7C gives the driver program for testing.

```
#include "Complex.h"
using namespace std;


void testAutomaticConversion();
void testUnaryOperators();
int main()
{
        Complex number1;
        Complex number2;
        Complex number3;
        bool sentinel = true;

        cout<< "To perform Complex number arithmetic,
please enter 1, or enter 0 to exit."<<endl;
        cin>>sentinel;

        while (sentinel)
        {
cout<<"You will be prompted in set of two calls to enter”
<<“ each time the"<<endl;
cout<<"Real and Imaginary part of your two Complex “
<<“ numbers on which you "<<endl;
cout<<"need the arithmetical operations
<<” performed"<<endl<<endl;

  cout<<"Please enter the arguments of first Complex”
<<" number first real & then imaginary."<<endl;
                cin>>number1;

 cout<<"Please enter the arguments of second “
<<” complex number first real & then imaginary."<<endl;

```

```
                cin>>number2;

                number3 = number1 + (number2);

    cout<<endl<<"("<<number1 <<")"<<" + "<<"("<<number2<<")"<<" = "<<number3<<endl<<endl;
cout<<"The absolute Value is = "<<number3.absolute()<<endl<<endl;

                number3 = number1 - (number2);
                cin.get();

           cout<<"("<<number1<<")" <<" - "<<"("<<number2<<")"<<" = "<<number3<<endl<<endl;
cout<<"The absolute Value is = "<<number3.absolute()<<endl<<endl;

                number3 = number1*(number2);
                        cin.get();

                cout<<"("<<number1 <<")"<<"*"<<"("<<number2<<")"<<" = "<<number3<<endl<<endl;
cout<<"The absolute Value is = "<<number3.absolute()<<endl<<endl;

                number3 = number1/(number2);
                        cin.get();

                cout<<"("<<number1 <<")"<<"/"<<"("<<number2<<")"<<" = "<<number3<<endl<<endl;
cout<<"The absolute Value is = "<<number3.absolute()<<endl<<endl;

                cout<<"Another set of numbers? Please enter 1 to continue or 0 to end"<<endl;
                cin>>sentinel;
        }
        cin.get();
        testAutomaticConversion();
        cin.get();
        testUnaryOperators();
        cout<<"End of the program"<<endl;
        return 0;
}
void testAutomaticConversion()
{
        Complex Num1(-1, 12);
        cout<<"Num1 = "<<Num1<<endl;
        Num1 = 1 + Num1;
        cout<<"After setting Num1 = 1 + Num1, the Num1 = "<<Num1<<endl;
        Num1 = 1-Num1;
        cout<<"After setting Num1 = 1 - Num1, the Num1 = "<<Num1<<endl;
        Num1 = 2/Num1;
        cout<<"After setting Num1 = 2/Num1, the Num1 = "<<Num1<<endl;
        Num1 = 2*Num1;
        cout<<"After setting Num1 = 2*Num1, the Num1 = "<<Num1<<endl;
        if(1==Num1)
                cout<<"Num1 is equal to 1.\n";
        else
                cout<<"Num1 is not equal to 1.\n";
        if(1 !=Num1)
                        cout<<"Num1 is not equal to 1.\n";
        else
                cout<<"Num1 is equal to 1.\n";

}
void testUnaryOperators()
{
        Complex Num1(10,11);
        Complex Num2(5,7);
        Complex Num3(1,2);
        Complex Num4(6,22);
        Complex Num5(6,2);
        //Testing prefix operators
        cout<<"The current value of Num1 = ";
        cout<<Num1<<endl;
cout<<"After applying prefix increment ++ the value of Num1 = ";
        ++Num1;
        cout<<Num1<<endl;
        cout<<"The current value of Num2 = ";
        cout<<Num2<<endl;
cout<<"After applying prefix decrement -- the value of Num2 = ";
        --Num2;
        cout<<Num2<<endl;
        //Testing negation
        cout<<"The current value of Num3 = ";
        cout<<Num3<<endl;
cout<<"After negating the value of Num3 and storing back, Num3 = ";
        Num3 = -Num3;
        cout<<Num3<<endl;
        //Testing post-decrement
```

```
        cout<<"testing post decrement now.\n";
        cout<<"The current value of real part of Num5 = ";
        cout<<Num5.getReal()<<endl;
cout<<"Using the real part of Num5 as a loop control variable "
                <<"and post-decrementing    "
                <<"  Num5 in loop pre-test expression.\n";
        int counter = 0;
        while((Num5--).getReal())
        {
           cout<<"Loop iteration # = "<<++counter<<endl;
        }
        //Testing post increment operator
        cout<<"testing post increment now.\n";
        Complex Num6(0,12);
        cout<<"The current value of real part of Num6 = "
                <<Num6.getReal()<<endl;
        cout<<"Using ((Num6++).getReal() - 3 ) as loop "
                <<"pre-test expression.\n";
        counter = 0;
        while ((Num6++).getReal() - 3 )
        {
            cout<<"Loop iteration # = "<<++counter<<endl;
        }
        Complex Num7 (9,2);
        Complex Num8 (-2,-9);
        Complex Num10 = Num7++ + --Num8;
        cout<<Num10<<endl;
}//Listing 0.7C
```

The Figure 0.7 below gives the output from the Listing 0.7C.

```
//*****************************************************************
To perform Complex number arithmetic, please enter 1, or enter 0 to exit.
1
You will be prompted in set of two calls to enter each time the
Real and Imaginary part of your two Complex numbers on which you
need the arithmetical operations performed

Please enter the arguments of first Complex number first real & then imaginary.
5 7
Please enter the arguments of second complex number first real & then imaginary.

10 -15

(5 + 7*I   ) + (10 - 15*I   ) = 15 - 8*I

The absolute Value is = 17
```

```
(5 + 7*I   ) - (10 - 15*I   ) = -5 + 22*I

The absolute Value is = 22.561
(5 + 7*I   )*(10 - 15*I   ) = 155 - 5*I

The absolute Value is = 155.081


(5 + 7*I   )/(10 - 15*I   ) = -0.169231 + 0.446154*I

The absolute Value is = 0.477171

Another set of numbers? Please enter 1 to continue or 0 to end
0
Num1 = -1 + 12*I
After setting Num1 = 1 + Num1, the Num1 = 0 + 12*I
After setting Num1 = 1 - Num1, the Num1 = 1 - 12*I
After setting Num1 = 2/Num1, the Num1 = 0.0137931 + 0.165517*I
After setting Num1 = 2*Num1, the Num1 = 0.0275862 + 0.331034*I
Num1 is not equal to 1.
Num1 is not equal to 1.

The current value of Num1 = 10 + 11*I
After applying prefix increment ++ the value of Num1 = 11 + 11*I
The current value of Num2 = 5 + 7*I
After applying prefix decrement -- the value of Num2 = 4 + 7*I
The current value of Num3 = 1 + 2*I
After negating the value of Num3 and storing back, Num3 = -1 - 2*I
testing post decrement now.
The current value of real part of Num5 = 6
Using the real part of Num5 as a loop control variable and post-decrementing
  Num5 in loop pre-test expression.
Loop iteration # = 1
Loop iteration # = 2
Loop iteration # = 3
Loop iteration # = 4
Loop iteration # = 5
Loop iteration # = 6
testing post increment now.
The current value of real part of Num6 = 0
Using ((Num6++).getReal() - 3 ) as loop pre-test expression.
Loop iteration # = 1
Loop iteration # = 2
Loop iteration # = 3
6 - 7*I
```

End of the program
```
//*****************************************************************
```
FIG. 0.7

The main function tests the normal functioning of class Complex, while it calls the function testAutomaticConversion and testUnaryOperators to test the automatic type conversion and unary operators respectively. We have already analyzed the code in function testUnaryOperators in the Listing 0.4C. The code to test the binary operators (+, -, *, and /) in main function gives expected results. The function testAutomaticConversion tests the binary operators with expressions such as:

```
Num1 = 1 + Num1; // Line 64
if(1==Num1) // Line 72
if(1 !=Num1) //Line 76
```

In all cases, the expected results are obtained, indicating the correct functioning of overloaded friend operators and automatic type conversion.

We do not show more operators overloading in this chapter. Rather in forthcoming chapters, we will use the technology developed in this section as needed. We discuss the last topic in this chapter, which is friend class.

Friend Class

Similar to the way a function can be a friend of a class, a class can also be a friend to another class. When a class MyClass becomes a friend of class YourClass, then all member functions of MyClass also become friend functions of YourClass. In order to make MyClass as a friend of YourClass, it must be declared as a friend inside YourClass. If definition of MyClass is not given before the definition of YourClass, then a *forward declaration* of MyClass is needed. The Code fragment in Listing 0.8 shows the overall procedure.

```
//*******************************************************
class MyClass; //Forward declaration

class YourClass
{
        public:
                friend class MyClass;
//------------ More code--------------
};

class MyClass
{
        //code
};//Listing 0.8
//*******************************************************
```

Rules for Operator overloading

We have not been able to show every example of rules of operator overloading. Therefore, we summarize all the rules below.

1. **If overloaded operator takes parameters, then at least one of the parameters must be of class data type. Exceptions are the post-fix increment and decrement.**
2. **Most operators can be overloaded in one of the following ways: member of the class, friend function, or stand-alone function.**
3. **The following four operators, =, [ ], ->, and ( ) can only be overloaded as non-static class members.**
4. **C++ does not allow creation of the new operators. Only the allowed operators from the Table 0.1 can be overloaded.**
5. **You cannot change the number of arguments an operator must take. For example, the unary operator ++ cannot be converted into a binary operator by including two arguments.**
6. **Overloading does not allow the precedence of an operator to be altered.**
7. **The list of operators that cannot be overloaded is given in Table 0.1.**
8. **The overloaded operators cannot have default arguments like other functions can.**

References

http://www.ul.ie/~flanagan/ee6721/basoper/basoper.html#distinction
Absolute C++ by Walter Savitch

Chapter Index

Chapter Glossary

Automatic Type Conversion: *Converting from one data type to another automatically.*

Complex Numbers: *The numbers in mathematics that are composed of real and imaginary parts. Imaginary part is multiplied by square root of minus one.*

Conceptual Diagram: *The diagram showing the concept of a class object. This would include all data members and member functions as well as their access control.*

Constant Object: *An object that has read-only access when it is returned from a function or when it invokes certain member functions. Constant objects need not be immutable. However the immutable objects are always constant objects.*

Forward Declaration: *A declaration that tells compiler that the function name, struct, or class specified in the declaration would be fully defined later and allow its use in the source code preceding the full definition.*

Friend Class: *A class declared as friend to certain class has access to the private data members of the latter.*

Friend Function: *A stand alone or a member function (of some class) when declared a friend of a class, can access the private data members of the latter.*

Marker: *A dummy argument passed to a function to alter its signatures to distinguish from another function with same name. Markers are used in overloading post-fix unary operators,*

Monomorphic Operators: *The C++ operator that cannot be overloaded. This includes following operators:* **typeid sizeof ? :: . and .***

Mutable Object: *An object whose attributes are modifiable. See also mutable keyword for C++.*

Operator Overloading[1]: **Having more than one operator with the same name in the same scope. Built-in operators, such as + and *, are overloaded for types such as int and float. Users can define their own additional meanings for user-defined types. It is not possible to define new operators or to give new meanings to operators for built-in types. The compiler picks the operator to be used based on argument types based overload resolution rules.**

Operator Polymorphism: *See Operator Overloading.*

Polymorphic Operators: *The C++ operators (nearly 41 in number) that can be overloaded.*

<u>**Post-fix Operators:**</u> ***The operators such as post-increment and post-decrement are called post-fix operators.***

<u>**Pre-fix Operators:**</u> ***The operators such as pre-increment or pre-decrement are called pre-fix operators.***

<u>**Three-level Read-only-access:**</u> ***The read-only access being enforced for the caller object, the object being passed as function argument, and the object being returned, by the class member function with prototype such as below***

const Type functionName(const Type& Data) const;

Type is the data type and functionName is the name of member function.

…………………………………………………………………………………….

1. **From C++ glossary published by Bjarne Stroustrup (http://www.research.att.com/~bs/glossary.html)**