Module 9:
Operator Overloading

In this module we will cover

- **Overloading the + operator**
- **Overloading =**
- **Overloading ==**
- ***, / , +=, ++....**

Aims of this module

In this module we introduce the feature of C++ which allows you to custom define the action of operators for classes.

These include =, *, +, +=,....etc
This feature gets used for several simplifications which you need to know about.

9.1 Overloading the + operator

What do you do if you want to add together two objects ????

Eg: Suppose you have written a `Complex` class to represent complex numbers. You would naturally wish for the following to be possible:

```
Complex a, b, c, d ;

...

c = a + b ;
d = a * b ;
```

This is not automatically possible since `Complex` is <u>your class</u> and the compiler has no idea what + or * should do ???

Another example we have already met is the ThreeVector class.

Suppose we had written a cross(...) method to take the cross product of two vectors. To use this we would have had to write:

```
ThreeVector p, q, r;

...
r = p.cross( q ) ;
```

It might be much neater if we could write:

```
ThreeVector p, q, r;

...
r = p * q  ;
```

but again, this is obviously not automatically possible since * is not defined for ThreeVectors

Such operations are only possible if you can tell the compiler which procedure to invoke in each case.

This is the feature provided by:

Operator Overloading

This feature allows you to custom define the meaning of operators like :

```
=, +, -, *, / , +=, >>, .....
```

when applied to objects of a class.

[See text books for full list]

The feature is based upon the fact that an operator like:

+

is formally equivalent to a method called:

```
operator+
```

In other words the following two bits of code are identical in function:

```
c = a + b ;

c = a.operator+( b ) ;
```

therefore to define + we provide an overloaded method:

```
operator+( ... )
```

in the normal way

As example we will write the

Complex

class and provide the + operator.

Here is the class declaration:

```
class Complex
{
  public:
    //Constructor
    Complex( float re, float im ) ;

    //+ Operator
    Complex operator+( Complex& k ) ;

  private:
    float real ;
    float imag ;
};
```

```
class Complex
{
  public:
    //Constructor
    Complex( float re, float im ) ;

    //Operators
    Complex operator+( Complex& k ) ;

  private:
    float real ;
    float imag ;
};
```

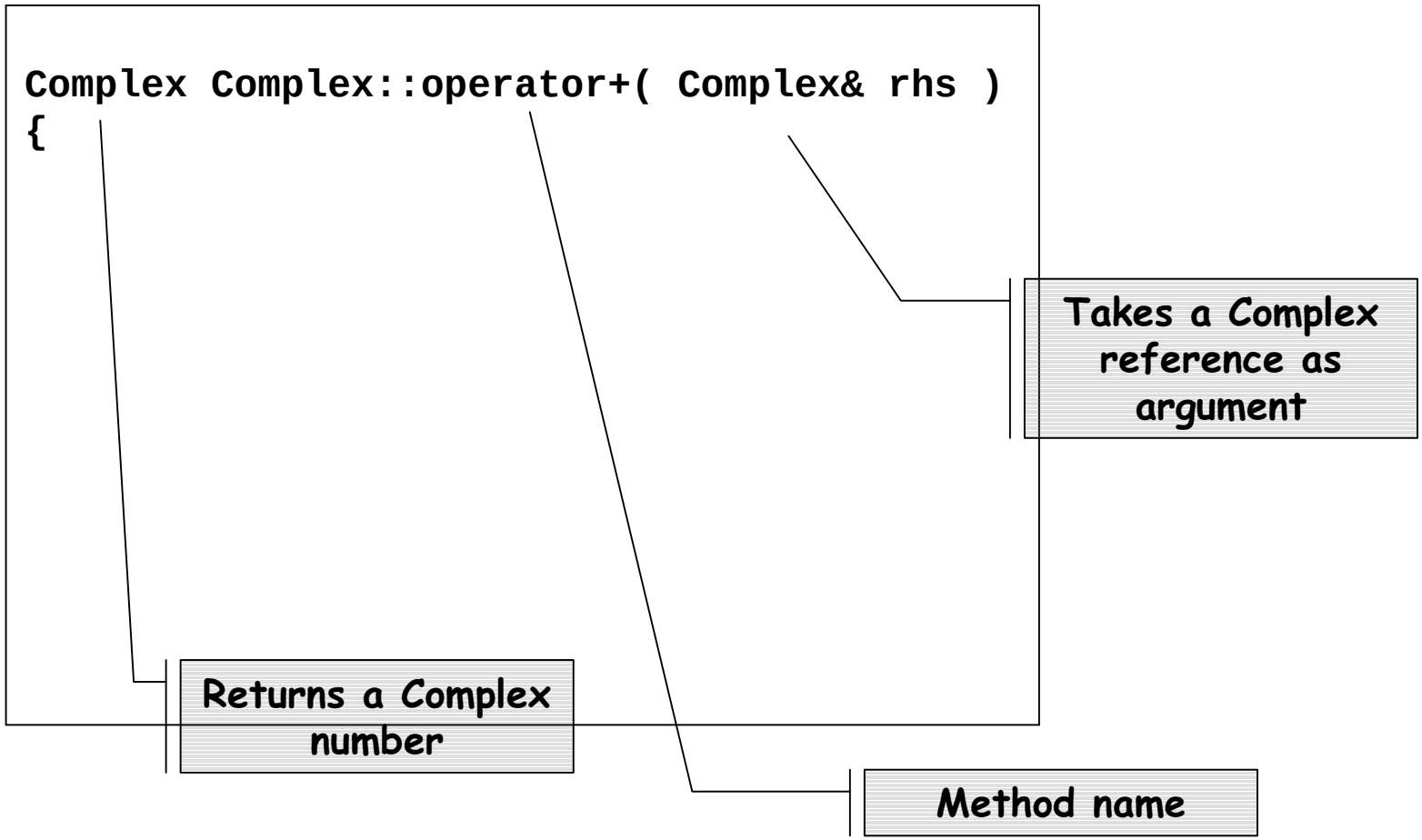
It has to return a Complex as a result of the addition, I.e. we will want to write
c = a + b;
Where a,b,c are Complex types

```
class Complex
{
  public:
    //Constructor
    Complex( float real, float imag ) ;

    //Operators
    Complex operator+( Complex& rhs ) ;

  private:
    float real ;
    float imag ;
};
```

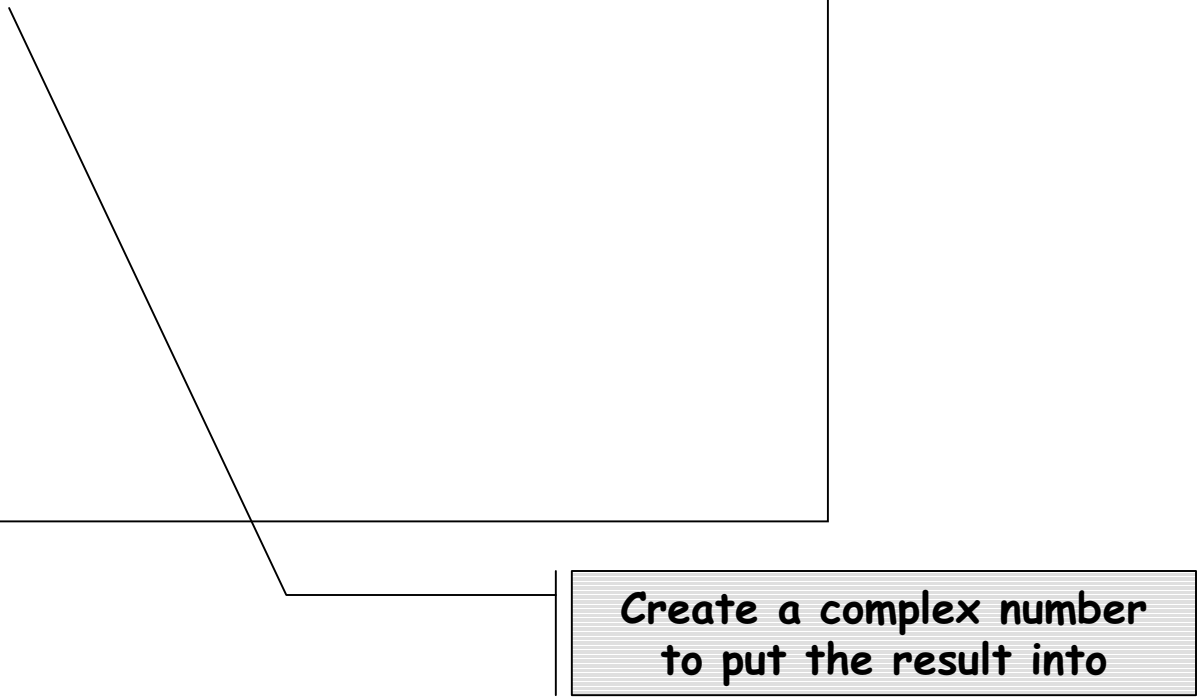
Declare we are going to overload two operators, + and *.

Both are binary operators therefore need an argument

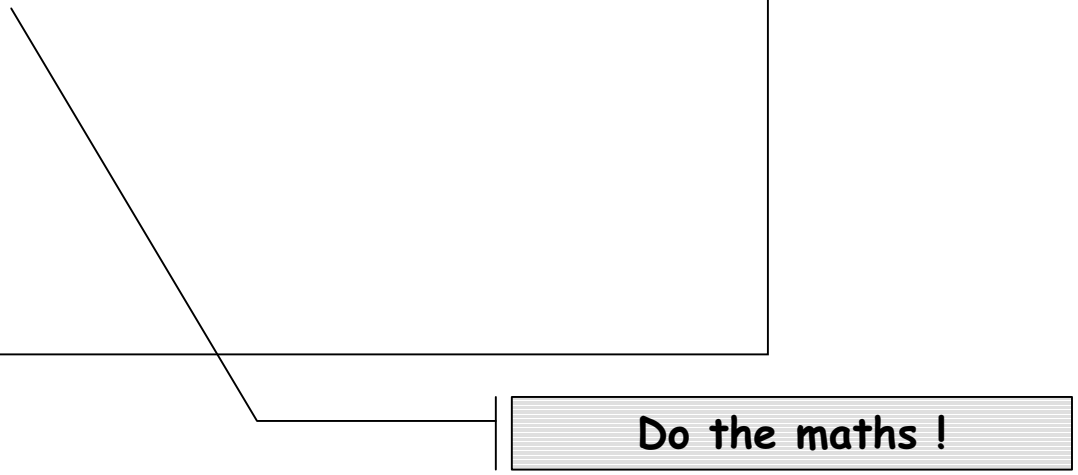
...now lets write the operator+ method....



```
Complex Complex::operator+( Complex& rhs )
{

  Complex result ;
```



```
Complex Complex::operator+( Complex& rhs )
{

  Complex result ;

  result.real = real + rhs.real ;
  result.imag = imag + rhs.imag ;
```



```
Complex Complex::operator+( Complex& rhs )
{

  Complex result ;

  result.real = real + rhs.real ;
  result.imag = imag + rhs.imag ;

  return result ;

}
```

Why did we make this method return a Complex number ?

You might think its obvious (good if so !!)

i.e `a + b` must produce a result, and what else can you do with it but return it ?

In case you dont think its obvious, lets look at what in effect actually happens

For the purposes of this explanation suppose we have created three complex numbers

```
...

Complex  a, b, c ;

....
```

Later in the program there is a line like this:

```
c = a + b ;
```

Now imagine the + operation replaced by the method equivalent:

```
c = a.operator+ ( b ) ;
```

It should be clear from this that the method called `operator+` needs to return a Complex object to be placed into the variable called `c`.

Student exercise

Write a Complex class as already indicated (I.e. with a constructor and a + method)

Add the

-

*****

operators

Write a program to demonstrate their use

15.4 Operator =

We glossed over the = operation earlier

i.e: when we wrote (the equivalent of) :

```
Complex a, c ;

...

c = a ;
```

I "implicitly" asserted that setting `object1 = object2` required no thought, i.e. that it was automatic.

```
Complex a, c ;

...

c = a ;
```

In fact for "simple" objects the compiler does provide a quite sensible default action for =

It simply does a "memberwise" copy

I.e. each member variable in a is copied into the corresponding member variable in c

In most cases this is fine.... but not always.…beware of pointers….

So in general you should provide your own = operator

Here is how you might overload the = operator

```
class Complex
{
  public:
    //Constructor
    Complex( float re, float im ) ;

    //Operators

    Complex operator+( Complex& rhs ) ;

    void operator=( Complex& rhs ) ;

  private:
    float real ;
    float imag ;
};
```

First declare the overloading

Note: in practice it is not quite done like this - we come to the difference in a few slides

....and here is the corresponding code...

```
void Complex::operator=( Complex& rhs )
{

  real = rhs.real ;
  imag = rhs.imag ;

  return ;

}
```

The members of the object for which the method is called (i.e. the LHS) get set to the members of the argument (the RHS). Recall the equivalence:

```
c = a ;

c.operator=( a ) ;
```

Why did we say that it is not quite done this way ?

It is connected with "chaining" of operations.
We require the following to work:

```
Complex a,b,c ;

...

c = b = a ;
```

If you interpret this as:

```
c = ( b = a ) ;
```

then it is clear that the `b = a` operation had better return a complex number in order for it to appear as the RHS of `c = (some complex number) ;`

If still in any doubt then consider the equivalence:

```
c = b.operator=( a )
```

...therefore here is a better overloaded = operator

```
class Complex
{
  public:
    //Constructor
    Complex( float re, float im ) ;

    //Operators

    Complex operator+( Complex& rhs ) ;

    Complex operator=( Complex& rhs ) ;

  private:
    float real ;
    float imag ;
};
```

....and here is the corresponding code...

```
Complex Complex::operator=( Complex& rhs )
{
  //Do the maths on the object which = is called for
  real = rhs.real ;
  imag = rhs.imag ;

  //Make a copy to return
  Complex result( real, imag ) ;
  return result;
}
```

For the advanced:

In fact for an object to make a copy of itself for this purpose is a waste of space and time

It is not necessary, instead it can simply use `this`

```
Complex Complex::operator=( Complex& rhs )
{
  //Do the maths on the object which = is called for
  real = rhs.real ;
  imag = rhs.imag ;

  return *this;
}
```

You might need to think about this if it is the first time you have used `this`

`return *this` causes the object itself to be copied as the return value.

For the advanced:

you can also avoid the copy on return by returning a reference to your self instead of a copy of yourself.

```
Complex& Complex::operator=( Complex& rhs )
{
  //Do the maths on the object which = is called for
  real = rhs.real ;
  imag = rhs.imag ;

  return *this;
}
```

You can do this because

(i) you(the object) contain the required return value

(i) you are stable and will not vanish upon return

If you do not understand this now - do not worry

15.5 Boolean operator ==

You would want to overload the boolean == operator so that you can do tests like:

```
Complex a, c ;

...

if( c == a )
{
  ...
}
```

♦ **Whole Class excercise on whiteboard** ♦

15.6 Discussion on returning of references

Discussion 1:
why wouldn't this have worked ?

```
Complex& Complex::operator=( Complex& rhs )
{
  //Do the maths
  real = rhs.real ;
  imag = rhs.imag ;

  //Make a copy to return
  Complex result( real, imag ) ;
  return result;

}
```

Related discussion 2:

why can you only return a reference for some operators =, +=
and not others +, -, *

I.e why can you not find some code to implement this..

```
Complex& Complex::operator+( Complex& rhs )
{

  ??? some code to do the maths ???

  return ???something??? ;

}
```

The answer to both questions is essentially the same:

Provided the result of the operation is held in the one of the operand objects (i.e.

`c = a`

places the result in `c`)

then the resulting object can always pass a reference to itself as it by defn exists outside the scope of the method.

If however the result of the operation is not actually held in one of the operands, i.e.

`a + b`

does not put the result in either `a` or `b`, then you have to make a new object for the result and pass that object as a return argument not a reference as the new object will go out of scope upon returning.

Student exercise

Add the following operators to your Complex class

/

(if you can remember how to divide complex numbers)

=

+=

Complex/
complex.h
complex.cpp

Summary of Module 9: Operator Overloading

In this module we have covered the following topics.

- Overloading +
  - equivalence of "+" with "operator+"
  - declaring and writing the operator+ method
  - return values from operator+
- Overloading of =
  - return value from operator=
  - use of *this
- Overloading of ==
- Some others needed for the Complex class