Roxana Dumitrescu

C++ in Financial Mathematics

What have we learnt?

- Arrays; relation between arrays and pointers..
- Returning arrays from functions
- Passing arrays to functions
- Intoduction to classes

Plan

- Classes
- Overloaded operators
- Study case: Complex class
- Working with multiple files

Classes

A **constructor** is a member function of a class that has the same name as the class. A constructor is called automatically when an object of the class is declared. Constructors are used to initialize objects.

Rules

- A constructor must have the same name as the class.
- A constructor definition cannot return a value. No return type, no *void*.

Classes

The class BankAccount

```
class BankAccount
{ public:
BankAccount (int dollars, int cents, double
   rate);
BankAccount();
double get_balance();
double get_rate();
void output(); // print
private:
double balance;
double interest_rate;
};
```

Classes

```
int main()
{
BankAccount account1(999,99,5.5), account2;
account1.output();
account2.output();
return 0;
}
```

Classes

Constructor 1

```
BankAccount::BankAccount(int dollars, int
   cents, double rate)
{ if((dollars<0)||(cents<0)||(rate<0))
{ cout<<"Illegal values for money or interest
   rate. \n";
exit(1);}
balance=dollars+0.01*cents;
interest_rate=rate;}
```

Classes

Constructor 2: Default constructor

```
BankAccount::BankAccount(): balance(0);
    interest_rate(0.0){};
```

Note that the last constructor definition is equivalent to

```
BankAccount::BankAccount()
{ balance=0; interest_rate=0.0;}
```

Classes

- We have **2** constructors. In other words, the constructor is *overloaded*.
- The first one is called **constructor with parameters** and the last constructor is called **default constructor.**
- **A default constructor** is simply a constructor that doesn't take parameters. If a default constructor is not defined in a class, the compiler itself defines one.
- Often times we want instances of our class to have specific values that we provide. In this case, we use **the constructor with parameters**.

Classes

- You can think of a constructor as a function that is automatically called before anyone is allowed to see the object. Technically speaking it isn't actually a function because it can **only** be called when the object is being initialised and because it doesn't have a return value.
- As we have seen in the example, inside the definition of the constructor you should set all **double**, **int** etc. fields to sensible default values. More generally, you should ensure that the object is in a consistent state before anyone sees it and you should perform whatever processing is required to achieve this.

Classes

Using constructors

Class_Name Object_Name(Arguments_for_Constructor);

```
BankAccount account1(999,99,5.5);
```

Classes

Copy constructors

We have seen:

- The default constructor
- The Parametrized constructor

The **copy constructor** is a constructor which creates an object by initializing it with an object of the same class, which has been created previously. The copy constructor is **used** to:

- Initialize one object from another of the same type
- Copy an object to pass it as an argument to a function
- Copy an object to return it from a function.

Classes

Copy constructors

Name_of_class Object_name (const Name_of_class & object){**...**}

Here, **object** is a reference to an object that is used to initialize another object**.**
If a copy constructor is not defined in a class, the compiler itself defines one**.**

Classes

Destructor

A **Destructor** is a special member of a class that is executed whenever an object of its class goes out of scope. Destructors are very useful for releasing resources in the case of dynamic allocation memory in the constructor (we'll see an example later!).

```
~Name_of_class(){...};\\
```

Example (in the case where is no dynamic allocation)

```
~BankAccount(){};
```

Classes

This pointer

Every object in C++ has access to its own address through a pointer called **this** pointer. It can be used inside a member function in order to refer to the invoking object.

```
BankAccount::BankAccount(int dollars, int
    cents, double rate)
{ if((dollars<0)||(cents<0)||(rate<0))
{ cout<<"Illegal values for money or interest
    rate. \n";
exit(1);}
*this.balance=dollars+0.01*cents;
*this.interest_rate=rate;}
```

Classes

This pointer

```
(*this).balance=dollars+0.01*cents;
(*this).interest_rate=rate;
```

has the **same meaning** as

```
this->balance=dollars+0.01*cents;
this->interest_rate=rate;
```

This is a pointer to the an object of the class BankAccount.
IMPORTANT RULE: To access the member variables through a pointer, use the operator ->!

Classes

This pointer

this always points to the object being operated on. More precisely, "this" is a const pointer (for e.g. in the previous example, **this** has the type **BankAccount * const**). You can change the value of the underlying object it points to, but you can not make it point to something else!

Classes

Some examples when you need the pointer this

(i) If you have a constructor (or member function) that has a parameter with the same name as a member variable, you should use "this" (if not, ambiguity!)

```
class YourClass{
private: int data;

public: YourFunction(int data){
this->data=data;
}
};
```

(ii) It will be used for the overloading of operators (you’ll see this just in a few minutes!).

Classes

Static members

- While most variables declared inside a class occur on an instance-by-instance basis (which is to say that for each instance of a class, the variable can have a different value), a static member variable has the same value in any instance of the class. More precisely, **static member variables and static functions are associated with the class, not with an instance.** For instance, if you wanted to number the instances of a class, you could use a static member variable to keep track of the last number used.

Classes

Static members

- Since the static member variables do not belong to a single instance of the class, you have to refer to the static members through the use of the class name.

```
class_name::x;
```

- You can also have static member functions of a class. Static member functions are functions that do not require an instance of the class, and are called the same way you access static member variables. Static member functions can only operate on static members, as they do not belong to specific instances of a class.

```
class_name::static_function;
```

Classes

Static members

- **Static functions** can be **used** to modify static member variables to keep track of their values : you might use a static member function if you chose to use a counter to give each instance of a class a unique id.

```
class user
{ private:
   int id;
   static int next_id;
  public:
// constructor
   user();
   static int next_user_id()
   {  next_id++;
        return next_id;}
};
```

Classes

Static members

```
int user::next_id = 0;

// constructor
 user::user()
   {
     id = user::next_id++; // or
    id=user::next_user_id();
   }
};
```

The line

```
user a_user;
```

would set id to the next id number not assigned to any other user.

Overloaded operators

Operator Overloading in C++

- In C++ the overloading principle applies not only to fonctions, but to operators too. The operators can be extended to work not just with built-in types but also classes.
- A programmer can provide his own operator to a class by overloading the build-in operator to perform some specific computation when the operator is used on objects of that class.
- Overloaded operators are functions with special names **the keyword operator** followed by the symbol for the operator being defined. Like any other function, **an overloaded operator has a return type and a parameter list**.

Operator Overloading in C++

Example 1.

```
int a=2;
int b=3;
cout<<a+b<<endl;
```

The compiler comes with a built-in version of the operator $(+)$ for integer operands - this function adds integers *x* and *y* together and returns an integer result. The expression $a + b$ could be translated to a function call which would take the following form

```
operator+(a,b)
```

Operator Overloading in C++

Example 2.

```
double c=2.0;
double d=3.0;

cout<<c+d<<endl;
```

The compiler also comes with a built-in version of the operator $(+)$ for double operands. The expression $c + d$ becomes fonction call operator+(c,d), and function overloading is used to determine that the compiler should be calling the double version of this function instead of the integer version.

Operator Overloading in C++

Example 3.
Add two objects of class **string** (we'll see this class more in detail later).

```
Mystring string1="Hello, ";
Mystring string2="world!";
std::cout<<string1+string2<<std::endl;
```

The intuitive expected result is that the string “Hello, World!” would be printed on the screen. However, because Mystring is a user-defined class, the compiler does not have a built-in version of the plus operator that it can use for Mystring operands. In this case the operand will give an error. **Conclusion**: it is needed an overloaded function to tell the compiler how the $+$ operator should work with two operands of type Mystring.

Operator Overloading in C++

- Almost any existing operator in C++ can be overloaded. The **exceptions** are: conditional (?:), sizeof, scope (::), member selector (.), and member pointer selector (.*).
- You can only overload the operator that exist. You can not create new operators or rename existing operators.
- At least one of the operators must be an user-defined type.
- Is not possible to change the number of operands an operator could support.
- All operators keep their default precedence and associativity.

When overloading operators, it’s best to keep the function of the operators as close to the original intent of the operators as possible.

Operator Overloading in C++

A first classification of operators

- Unary operators: they operate on a single operand and the examples of unary operators are the following:
  - The increment $(++)$ and decrement $(--)$ operators.
  - The unary minus $(-)$ operator.
  - The logical not $(!)$ operator.
- Binary operators have two operands, as for example the addition operator $+$, the subtraction operator $-$, the division operator $(/)$ etc.

Operator Overloading in C++

A second classification of operators

- Member operators of a class
  - Unary operators

```
Class_type X{...public:
Class_type operator++(){...}
}
```

  - Binary operators

```
Class_type X{...public:
Class_type operator+(const Class_type&
    c){...}
}
```

There are operators which can be only declared as member operators. Example: $=$, []...

Operator Overloading in C++

A second classification of operators

- Non-member operators of a class
  - Unary operators

```
Class_type X{...}

Class_type operator++(Class_type& c){...}
```

  - Binary operators

```
Class_type X{...}
Class_type operator+(const Class_type& c,
    const Class_type& d){...}
```

Since the unary operators only operate on the object they are applied to, unary operator overloads are generally implemented as member functions!

Operator Overloading in C++

Rules concerning operator overloading

- If you are overloading a unary operator, do so as member function.
- If you are overloading assignement $(=)$, subscript [], function call $(())$ or member selection $(->)$, do so as member function.
- If you are overloading a binary operator that modifies its left operand (e.g. operator $+=$) do so as a member function.
- If you are overloading a binary operator that does not modify its left operand (e.g. operator $+$), do so as a normal function or friend function.

Study case: Complex class

Complex class

- Making a class for complex numbers is a good educational example
- C++ already has a class **complex** in its standard template library (STL) - use that one for professional work

```
#include <complex>
complex<double> z(5.3,2.1), y(0.3);
cout<<z*y+3;
```

- However, writing your own class for complex numbers is a very good exercise for novice C++ programmers!

Complex class

How would we like to use the Complex Class?

```
void main()
{
Complex a(0,1);
Complex b(2), c(3,-1);
Complex q=b;
}
cout<<"q="<<q<<",a="<<a<<",b="<<b<<endl;
q=a*c+b/a;
cout<<"Re(q)="<<q.Re()<<",
    Im(q)="<<q.Im()<<endl;
}
```

Complex class

Basic contents of class Complex

- **Private** data members: real and imaginary part
- Some **public** member functions:
  - Constructors (in order to construct complex numbers)

```
Complex a(0,1); //imaginary unit
Complex b(2), c(3,-1);
Complex q=b;
```

  - Other functions (not the complete list, just examples):

```
cout<<c.Get_Re();
cout<<c.abs();
```

Complex class

Basic contents of class Complex

- Some **operators** declared in the public part:
  - In order to write out complex numbers

```
cout<<"q="<<q<<",a="<<a<<",b="<<b<<endl;
```

  - In order to perform arithmetic operations:

```
q=a*c+b/a;;
```

Complex class

```
class Complex
{
private:
    double re,im; //real and imaginary part
public:
    Complex();
    Complex(double re, double im); // Complex
   a(4,3);
    Complex (const Complex &c); // Complex
   q(a);
    ~Complex () {}
    double Get_Re() const;
    double Get_Im() const;
```

Complex class

```
    void Set_Re(double);
    void Set_Im(double);

    double abs () const; //double m=a.abs();
   // modulus
    /*member operator*/
    Complex& operator= (const Complex& c); //
   a=b;
};
```

Complex class

```
/*non-member operator, defined outside the
class*/
Complex operator+ (const Complex& a, const
Complex& b);
 Complex operator- (const Complex& a, const
Complex& b);
 Complex operator/ (const Complex& a, const
Complex& b);
 Complex operator* (const Complex& a, const
Complex& b);
```

Complex class

The simplest functions

- Extract the real and imaginary part (recall: these are private, i.e. invisible for users of the class; here we get a copy of them for reading)

```
double Complex::Get_Re() const {return re;}
 double Complex:: Get_Im() const {return
   im;}
```

- Computing the modulus:

```
double Complex::abs() const {return
   sqrt(re*re+im*im);}
```

Complex class

Inline functions

In the case of inline functions, the compiler replaces the function call statement with the function code itself (process called expansion) and then compiles the entire code.

- There are two ways to do this:
  1. Define the member-function inside the class definition.
  2. Define the member-function outside the class definition and use the explicit keyword **inline**:

```
inline double Complex::Get_Re() const
    {return re;}
```

When are inline functions useful? Inline functions are best for small functions that are called often!

Complex class

The **const** concept

A **const member function** is a member function that guarantees it will not modify the object.
As we have seen, to make a member function **const**, we simply append the const keyword to the function prototype, after the parameter list, but before the function body.

```
double Complex::Get_Re() const {return re;}
```

Complex class

The **const** concept

Any const member function that attempts to change a member variable or call a non-const member function will cause a compiler error to occur.

```
void Complex::Set_Re() const {re=0;} //
   compile error, const functions can't change
   member variables.
```

Rule: Make any member function that does not modify the state of the class object const.
Remark: Note that **constructors cannot be marked as const.**

Complex class

The **const** concept

- Recall that **const** variables cannot be changed:

```
const double p=3;
p=4; // ILLEGAL!!  compiler error
```

- **const** arguments (in functions)

```
void myfunc (const Complex& c)
{c.re=0.2; /* ILLEGAL!! compiler error }
```

Complex class

The **const** concept

- **const** Complex arguments can only call **const** functions:

```cpp
double myabs (const Complex& c)
{return c.abs();} // ok, because c.abs() is
   a const function.
```

- Without const in

```cpp
double Complex::abs () {return
   sqrt(x*x+y*y);}
```

the compiler would not allow the c.abs call in myabs

```cpp
double myabs (const Complex& c)
{return c.abs();}
```

because Complex::abs is not a const member function

Complex class

Question: how to create a complex number which is the sum of two complex numbers?

```
Complex c1 (1,2);
Complex c2(2,3);

Complex sum=c1+c2;
```

Answer: Overload the operator "+"

Complex class

Overloading the "+" operator

- To overload the $+$ operator, first notice that the $+$ operator will need to take two parameters, both of them of Complex type. To be more precise, these parameters must be **const** references to Complex.
- The operator $+$ will return a Complex containing the result of the addition.
- To overload the $+$ operator, we write **a function** that performs the necessary computation with the given parameters and return types. The only particular thing about this function is that it must have the name **operator+**.

Complex class

- The meaning of $+$ for Complex objects is defined in the following function

```
Complex operator + (const Complex& c1,
    const Complex& c2 )
```

- The compiler translates

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

into

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

Complex class

There are several ways to define the operator $+$.

First possibility:

```
Complex operator+ (const Complex& a,  const
   Complex& b)
{ Complex temp;
  temp.Set_Re(a.Get_Re()+b.Get_Re());
 temp.Set_Im(a.Get_Im()+b.Get_Im());
 return temp;}
```

Complex class

Second possibility

```
Complex operator+ (const Complex& a,  const
   Complex& b)
{ return Complex (a.Get_Re()+b.Get_Re(),
   a.Get_Im()+b.Get_Im());}
```

Complex class

Third possibility

```
Complex operator+ (const Complex& a,  const
   Complex& b)
{ Complex temp;
  temp=a;
  temp+=b;
  return a;
}
```

Here we use the following idea: we can first overload the assignment operator ($=$) and the operator $+=$ as member operators. Using these operators, one can overload the non-member operator $+$.

Complex class

The assignement operator

- Writing

```
a=b;
```

implies a call

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

- this is the definition of assignement

Complex class

The assignement operator

- We implement operator= as a part of the class:

```
Complex& Complex::operator= (const Complex&
    c)
{
    re=c.re;
    im=c.im;
    return *this;
}
```

- If you forget to implement operator=, C++ will make one (this can be dangerous)

Complex class

The multiplication operator

- First attempt

```
Complex operator* (const Complex& a, const
   Complex& b)
{
Complex h;  // Complex()
h.re=a.re*b.re-a.im*b.im;
h.im=a.im*b.re+a.re*b.im;
}
```

Complex class

The multiplication operator

- Alternative (avoiding the *h* variable)

```
Complex operator* (const Complex& a, const
   Complex& b)
{
return Complex(a.re*b.re-a.im*b.im,
   a.im*b.re+a.re+b.im)
}
```

Complex class

Remark

- The member operators $+=$, $-=$ can be implemented in the same way as $=$
- The non-member operators $-$, $/$ can be implemented in the same way as $+$ and $*$.

Complex class

Constructors

- Recall that constructors are **special** functions that have the same name as the class
- The declaration statement

```
class q;
```

calls the member function Complex()

- A possible implementation is

```
Complex:: Complex {re=im=0.0;}
```

In this case, declaring a complex number means making the number $(0, 0)$.

Complex class

Constructors with arguments

- The declaration statement

```
class q(-3,1.4);
```

calls the member function Complex(double, double)

- A possible implementation is

```
Complex:: Complex (double re_, double im_)
{re=re_; im=im_; }
```

Complex class

Constructors with arguments

- A second possible implementation is

```
Complex:: Complex (double re,  double im)
{this->re=re; this->im=im; }
```

Note that in this case we use the pointer **this**, since we have parameters with the same name as the private members.

Complex class

Copy constructor/Assignment operator

- The statements

```
Complex q=b;
Complex q(b);
```

  makes a new object *q*, which becomes a copy of *b*. In this case, the **copy constructor** is called.

- Note the difference with respect to:

```
Complex b;
Complex q;
q=b;
```

  where first the **default constructors** are called and then the **assignement operator** is used.

Complex class

Copy constructor

- First implementation :

```
Complex::Complex (const Complex& c)
{re=c.re; im=c.im; }
```

- Implementation in terms of assignement:

```
Complex::Complex (const Complex& c)
{*this=c; }
```

- Recall that **this** is a pointer to "this object", ***this** is the present object, so ***this=c** means setting the present object equal to *c*, i.e. this $\rightarrow$ operator=(c)

Complex class

Copy constructor

- The copy constructor defines the way in which the copy is done. This also includes the argument. That's why the following statement

```
Complex::Complex (const Complex c):
    re(c.re), im(c.im){}
```

  represents an ERROR. In this case, this call would imply an infinite recurrence.

RULE: The correct declaration of the copy constructor is

```
Complex (const Complex& c);
```

Dont' forget the & symbol!

Complex class

Overloading the output operator

- Output format of a complex number: (re,im), i.e. $(1.4, -1)$
- Desired user syntax:

```
cout<<c;
```

- The effect of « for a Complex object is defined in

```
ostream& operator<< (ostream& o, Const
    Complex& c)
{o<< "(" <<c.Re()<< ","<<c.Im()<<")";
    return o;}
```

Complex class

Some comments on the overloaded operator $<<$

- The **operator** $<<$ is defined as a non-member function.
- The **operator** $<<$ always takes an ostream in its first input. This is because we always have a stream on the left of $<<$ (ostream is a class and **cout** is an object of "type" ostream).
- The second parameter is, in this case, a Complex. This is because this is the type of data we wish to print out.
- The function **operator** $<<$ returns a reference to the **ostream**. This will in practice always be the same **ostream** that we pass in as the parameter out.

Complex class

Why returning by reference?

- Recall that return by reference is acceptable so long as **you don't return a reference to a local variable .** Return by reference is more efficient than return by value, since it avoids copying (recall that when a function returns by value, the copy constructor is called).
- One effect of returning a reference is that whoever receives the reference can use that reference to modify whatever it points to. See an example on the following slide.

Complex class

Why returning by reference?

Consider the code:

```
cout<<"To be"<<"or not to be";
```

This code is equivalent to the following:

```
(cout<<"To be")<<"or not to be";
```

This shows why the fact the the operator $<<$ returns a stream by reference is useful. **We can apply the** $<<$ **operator again**!

Working with different files

Working with different files

When writing programs, we try to split the program into independent pieces or modules. In general, we create three files:

- *Header file* describing the class members (data and functions). The header file has the extension .h
- *The implementation of the class* goes into the .cpp file
- File containing the program that uses your class (which has the extension .cpp).

Remark: In the case when we don't have classes, only functions: Function declarations must be done in the header file and the definitions go into the .cpp file.

Working with different files

File: **Complex.h**

```
# pragma once
class Complex
{
private:
double re;
double im;
public:
Complex();
Complex(double x, double y);
Complex(const Complex& c);
~Complex(){};
double Get_Re();
Complex& operator=(const Complex&); // and all
    the functions and operators
};
```

Working with different files

File: **Complex.cpp**

```
# include "Complex.h"
Complex::Complex(): re(0.0), im(0.0){};
Complex::Complex(double x, double y){re=x;
    im=y;};
double Complex::Get_Re()
{
return re;
};

// and the other definitions
```

Working with different files

File: **main.cpp**

```
#include <iostream>
# include "Complex.h"

using namespace std;

int main()
{
Complex z1; // default constructor
cout<<z1.Get_Re()<<endl;
return 0;
}
```

Working with different files

Some rules:

- **Pragma once**
  Every header file has to start with **pragma once.** The reason you should start every file with pragma once is that it stops the same file being include twice.
- Don't include definitions of functions in the header file, **except** for the inline functions!
- Don't use **using namespace std** in a header file.
- Another rule you should follow is to never have circular dependencies through include. For example, two header files should not include each other.
- Each .cpp file has to include the header file.

Summing up

- Classes
    - Constructor/destructor
    - This pointer
    - Classes with static memebrs
- Overloading operators
- Study case: Complex class
- Working with different files