Roxana Dumitrescu

C++ in Financial Mathematics

What have we learnt?

Classes

Study case: Complex class

- Default Constructor:

```
Complex();
```

- Constructor with parameters

```
Complex(double re, double im);
```

- Copy constructor

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

- Destructor

```
~Complex(){};
```

What have we learnt?

- **Pointer this**: pointer to the current instance of the class.

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

or

```
Complex::Complex(double re, double im)
{
(*this).re=re;
(*this).im=im;
}
```

What have we learnt?

- **Static members**: Static member variables and static functions are associated with the class, not with a particular instance. For example, if you want to number the instances of a class, you could use a static member variable to keep track of this number.

What have we learnt?

- 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;}
```

  Const objects can only call **const** functions!

What have we learnt?

- **Overloading operators**
  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.
  We have seen several examples. It is suggested to:
  - Declare the operators $=$, $+=$,$*=$, $/=$ and the unary operators as member functions of the class.
  - Declare the operators $+$, $*$, $/$, $==$ as non-member functions of the class.
  - Declare the output operator $<<$ as a non-member function of the class.

What have we learnt?

- **Working with several files**
  - A header file (e.g. Complex.h) which starts with pragma once ;
  - 2 source files (e.g. Complex.pp and main.cpp) which starts with include"Complex.h";
  - Don’t put using namespace std in the header file;
  - Don’t define non-inline functions in the header file.

Plan

- Classes with Pointer Data Member
- Study case: YourVector class
- Some useful C++ classes

Classes with Pointer Data Member

Classes with Pointer Data members

Classes with Pointer Data Member

Every class that has a pointer data member should include the following member functions:

- a destructor
- a copy constructor
- operator= (assignement)

Classes with Pointer Data Member

The rule of three: Whenever you write a destructor (other than an empty destructor) you must:

- overload the assignment operator (=)
- write a copy constructor

In fact, if you write any one of these three things:

- a non-trivial destructor;
- a copy constructor;
- an assignment operator =;

then **you should write all three**.

Classes with Pointer Data Member

Consider the following example:

```
class Table{
int* Name;
int size;

public:
Table ();
Table (int size);
~Table();
}
```

Classes with Pointer Data Member

Constructor by default

```
Table::Table ()
{ this->size=0;
Name=NULL;} ;
```

Constructor with parameters

```
Table::Table (int size)
{ this->size=size;
Name=new int[size];}
}
```

Classes with Pointer Data Member

Non-trivial Destructor Function

```
Table::~Table(){ delete [] Name;}
```

The purpose of the destructor is to free any dynamically allocated storage.
Note that an object's destructor is called when that object is about to "go away"; i.e. when:

- A class object (a value parameter or a local variable) goes out of scope
- A pointer to a class object is deleted (the dynamically allocated storage pointed to by the pointer is freed by the programmer using the **delete** operator).

Classes with Pointer Data Member

Example

```
void f(Table T){
Table *p= new Table;
while(...){
Table T1;
...
}
delete p;
}
```

In this example, the scope of value parameter *T* is the whole function; *T* goes out of scope at the end of the function. So when function *f* ends, *T*′s destructor function is called.

Classes with Pointer Data Member

The scope of variable *T*1 is the body of the while loop.

- **T1's constructor** function is called at the beginning of every iteration of the loop
- **The destructor function** is called at the end of every iteration of the loop.

Variable *p* is a pointer to a *Table*. When a *Table* is allocated using **new**, that object's constructor is called. When the storage is freed, the object's destructor function is called (and then the memory for the Table itself is freed).

Classes with Pointer Data Member

The copy constructor

In the case of a class a **non-trivial destructor**, if you don't write a copy constructor, you may have problems!

Explanation

If you don't write a copy constructor, the compiler will provide one that just copies the value of each data member. If some data member is a pointer this causes *aliasing* (both the original pointer and the copy point to the same location), and may lead to trouble.

Classes with Pointer Data Member

Example

```
void h()
{     Table t1;
      Table t2=t1;
}
```

The problem

- The **constructor by default** of Table is called once, for **t1**.
- **The object t2 is created by copy**, using the copy constructor provided by the compiler.
- The destructor is called two times, for t1 and t2. As t1.Name and t2.Name point to the same memory location, we try to free twice the same storage! $\Rightarrow$ **Problem!**

Classes with Pointer Data Member

Conclusion: You have to define your own copy constructor in order to avoid the previous problem!

How should we define the copy constructor?

The copy constructor should perform the following tasks:

- Initialize the *size* field to have the same values as the one in *T.size* (where *T* is the copy constructor's Table parameter)
- Allocate a new array of int of size *T*.*size* ; we set *Name* to point to this new array
- Copy the values in the array pointed to by T.Name to the new array

Classes with Pointer Data Member

Overloading the operator =

As we have seen, in C++ you can assign from one class object to another (of the same type). For example,

```
Table T1, T2;
T1=T2; // this assignment is OK
```

By default, class assignment is field-by-field assignment. This assignment is equivalent to:

```
T1.size=T2.size;
T1.Name=T2.Name;
```

Classes with Pointer Data Member

Overloading the operator =

Problem: the same as in the case of a "default" copy constructor: the destructors of $T1$ and $T2$ are called; we try to free twice the same memory location.

Solution: Write in the class your own assignment operator!

Classes with Pointer Data Member

Some differences between the operator = and the copy constructor

- The object being assigned to has already been initialized.
- It is possible for a programmer to assign from a variable into *itself*, for example $T1 = T1$. The operator $=$ must check for this and do nothing.
- The operator $=$ function returns a reference to an object of type *Table* (this is important in order to write $(T1 = T2) = T3$). The copy constructor has no return value (or reference).

Case Study: YourVector Class

Case Study: Array Class

```
class YourVector{
private:
    int size; // the size
    int* Name; // pointer to the first element
   of pointer-based array
```

Case Study: YourVector Class

```
public:
    YourVector(); //default constructor
    YourVector(int size); // constructor with
   parameter
    YourVector(const YourVector&); //copy
   constructor;
    ~YourVector(); // destructor
    int getsize() const; //return size;
    YourVector& operator= (const YourVector
   &);// assignment operator
    // subscript operator for non-const
   objects return modifiable lvalue
    int& operator [] (int);
    // subscript operator for const objects
   returns rvalue;
    int operator [] (int) const;
```

Case Study: YourVector Class

```
// Non-member operators
// equality operator
    bool operator== (const YourVector& left,
   const YourVector& right );

//inequality operator; returns opposite of ==
    bool operator!=(const YourVector& left,
   const YourVector& right);

// output operator
    std::ostream& operator<< (std::ostream&
   out, const YourVector& v);

};
```

Case Study: YourVector Class

1. Constructor by copy

```
  YourVector::YourVector(const YourVector&
   VectorToCopy):size(VectorToCopy.size){
    Name=new int[size];
    for (int i=0;i<size;i++)
        Name[i]=VectorToCopy.Name[i];
}
```

Case Study: YourVector Class

2. Non-trivial Destructor

```
 YourVector::~YourVector()
{ delete [] Name; // release pointer-based
    array space
}
```

Case Study: YourVector Class

Before presenting the overloaded operators, we explain two concepts from programming language called *lvalues* and *rvalues*.

- *Lvalue* is a value that resides in memory and is addressable (you can take its address). It stands for "left value" and it is a value that can be on the left-hand side of an assignment.
- *Rvalue* is a value that' not lvalue. It stands for "right value" and it is a value that can only be on the right-hand side of an assignement (the rvalue is intended to be non-modifiable).

Case Study: YourVector Class

For example, in the statement

```
x=5;
```

x is an lvalue and 5 is an rvalue.

In the statement

```
x=y+z;
```

x, y, z are lvalues , but $y + z$ is an rvalue (result of operator+).

Case Study: YourVector Class

3. Overloading the assignment operator

```
 YourVector& YourVector::operator=(const
   YourVector& right)
{ if (&right!=this)  //avoid self-assignment
    {  // for arrays of different sizes,
   deallocate original left-side array, then
   allocate new left-side array
        if (size!=size.n)
        {  delete [] Name; //release space;
            size=right.size;
            Name=new int[size];
        }
```

Case Study: YourVector Class

3. Overloading the assignment operator

```
        for (int i=0; i<size; i++)
        {
            Name[i]=right.Name[i]; // copy
   array into object
        }
    }
    return *this;
}
```

Case Study: YourVector Class

Notice that **operator=** returns a reference to *this.
Which is the benefit?

```
Table t1(3);
Table t2(3);
Table t3(3);
```

Assume that we initialize the values of of t1.Name with 1, the elements of t2.Name with 2 and the elements of t3.Name with 3. If we write

```
(t1=t2)=t3,
```

then *t*1.*Name* holds the values of *t*3.*Name*.

Case Study: YourVector Class

Remark: We have respected the rule of three: we wrote a copy constructor, the destructor and the assignment operator.

Case Study: YourVector Class

4. Operator == which determines if two vectors are equal

```
bool operator== (const YourVector& left, const
    YourVector& right)
{
     if (left.getsize!=right.getsize())
          return false;

     for (int i=0; i<size; i++)
          if (left.getName(i)!=right.getName(i))
               return false;
     return true;
}
```

Case Study: YourVector Class

5. Inequality operator; returns opposite of ==

```
 bool operator!=(const YourVector& left,
const YourVector& right)
 {return !(left==right);};
```

Case Study: YourVector Class

We now overload the subscript operator (as **member of the class**) in two ways.

Case Study: YourVector Class

6. Overloading subscript operator 1

```
int & YourVector:: operator [](int subscript){
    // check for subscript out-of-range error
     if (subscript<0 || subscript >=size)
     {   std::cerr<<"\nError: Subscript
   "<<subscript<<"out of range"<<std::endl;
         exit(1); // terminate program;
   subscript out of range
     } // end if
     return Name[subscript]; // reference
   return;
 };
```

Case Study: YourVector Class

6. Overloading subscript operator 1

Remark:

The reference creates a modifiable lvalue. More precisely, one can write

```
Table t1(3);
t1[2]=5;
```

Case Study: YourVector Class

7. Overloading subscript operator 2

```
int YourVector:: operator [](int subscript)
    const {
     // check for subscript out-of-range error
     if (subscript<0 || subscript >=n)
      { std::cerr<<"\nError: Subscript
    "<<subscript<<"out of range"<<std::endl;
          exit(1); // terminate program;
    subscript out of range
     } // end if

     return Name[subscript];
};
```

Case Study: YourVector Class

7. Overloading subscript operator 2

Remark:

- Using this subscript operator you can not write:

```
YourVector t1(3);
t1[2]=5;
```

  You obtain the following error: "Expression is not assignable".

- This subscript operator is used with const objects (*const YourVector t*).

Case Study: YourVector Class

8. Overloading the output operator <<

```
std::ostream & operator<< (std::ostream &
   output, const YourVector& v)
{
    for (int i=0; i<v.getsize();i++)
        std::cout<< v[i];
    return output; // enables cout<<x<<y
}
```

Case Study: YourVector Class

How to use YourVectorClass?

Case Study: YourVector Class

```
YourVector integers1();
YourVector integers2(10);
YourVector integers3=integers2;
```

- We instiantize two objects of class YourVector: integers1 with 7 elements, integers2 with 10 elements.

Case Study: YourVector Class

```
    for (int i=0;i<=integers1.getsize();i++)
integers1[i]=i;
for (int i=0;i<=integers2.getsize();i++)
integers2[i]=2*i;
```

- We use the member function getsize() in order to obtain the size of the vector.
- In order to initialize the values of the vector *integers1* and *integers2*, it is used the first overloaded subscript operator.

Case Study: YourVector Class

```
    if ( integers1 != integers2 )
        cout << "integers1 and integers2 are
  not equal" << endl;
```

- We test the overloaded inequality operator by evaluating the condition.

Case Study: YourVector Class

```
Vector integers3( integers1 );
```

- This line instantiates a third Vector called integers3 and initializes it with a copy of YourVector integers1. This invokes the YourVector copy constructor to copy the elements of integers1 into integers3. The copy constructor can also be invoked by writing

```
YourVector integers3=integers1;
```

- The equal sign in the preceeding statement is **not** the assignment operator. When an equal sign appears in the declaration of an object, it invokes a constructor for that object!

Case Study: YourVector Class

Some final remarks on the copy constructor

- We declare a copy constructor that initializes YourVector by making a copy of an existing YourVector object. As we have said, *such copying must be done carefully to avoid the pitfall of leaving both Vector objects pointing to same dynamically allocated memory. This is exactly the problem that occur with default memberwise copying, if the compiler is allowed to define a default copy constructor for this class*.

Case Study: YourVector Class

Some final remarks on the copy constructor

- *Note that a copy constructor must receive its argument by reference, not by value*. Otherwise, the copy constructor call results in infinite recursion (a fatal logic error) because receiving an object by value requires the copy constructor to make a copy of the argument object. Recall that any time a copy of an object is required, the class's copy constructor is called. If the copy constructor received its argument by value, the copy constructor would call itself recursively to make a copy of its argument!

Using some useful C++ classes

- C++ Standard Library provides some classes, which can be used including the corresponding header files (we have already included <iostream> in our programs in order to use *cin* and *cout*).

Using some useful C++ classes

- include <vector> to work with vectors.

  In your programs, you should use this class instead of creating your own vector class. We have learnt to write YourVector class in order to learn how to manipulate classes with non-trivial destructors!
- include <string> to use strings.
- include <sstream> to use strings efficiently.
- include <fstream> to work with files.

Using some useful C++ classes

- Matrices? Sorry, you have to write your own! The design of the class is very similar to *YourVector* class (you have to deal with the same non-trivial destructor and the rule of three!)

Using some useful C++ classes

Class Vector from the C++ Standard Library

```
// create a vector
vector<double> myVector;

// add three elements to the end
myVector.push_back( 12.0 );
myVector.push_back( 13.0 );
myVector.push_back( 14.0 );

// read the first, second and third elements
cout << myVector[0] <<"\n";
cout << myVector[1] <<"\n";
cout << myVector[2] <<"\n";
```

Using some useful C++ classes

Class Vector from the C++ Standard Library

```
// change the values of a vector
myVector[0] = 0.1;
myVector[1] = 0.2;
myVector[2] = 0.3;

// loop through a vector
int n = myVector.size();
for (int i=0; i<n; i++) {
    cout << myVector[i] <<"\n";
}
```

Using some useful C++ classes

Class Vector from the C++ Standard Library

```
// Create a vector of length 10
// consisting entirely of 3.0's
vector<double> ten3s(10, 3.0 );

// Create a vector which is a copy of another
vector<double> copy( ten3s );

// replace it with myVector
copy = myVector;
```

Using some useful C++ classes

Class Vector from the C++ Standard Library

```
// The function size gives the number of
   elements
cout<<ten3s.size();
```

Using some useful C++ classes

Class Vector from the C++ Standard Library

1. Some useful functions of the class Vector

```
push_back // add element at the end
size() // return size
resize() // change size
operator [] // access element
```

Using some useful C++ classes

Class Vector from the C++ Standard Library

Use of resize:

```
myVector.resize(5); // resize the vector
// at this point, the vector, the vector
   contains
// 0.1, 0.2, 0.3, 0,0

myVector.resize(2); // resize the vector
// at this point, the vector, the vector
   contains
// 0.1, 0.2
```

Using some useful C++ classes

Class Vector from the C++ Standard Library

2. Array/vector

Just **like** arrays, vectors use contiguous storage locations for their elements. Vector combines the advantages of both static and dynamic arrays: the size can change dynamically as in the case of dynamic arrays (internally, vectors use a dynamically allocated array to store their elements) and the used memory is automatically deleted as in the case of static arrays.

Using some useful C++ classes

Passing big objects around When you write a function that takes a vector parameter you should write it like this:

```
double sum( const vector<double>& v ) {
    double total = 0.0;
    int n = v.size();
    for (int i=0; i<n; i++) {
        total += v[i];
    }
    return total;
}
```

- **Rule:** Use const and & symbols. We need to pass the vector by **reference** because vectors are too big to keep copying all the time.

Remark

- For very small data types (double , int , bool), pass by value is quicker.
- For big data types, pass by reference is quicker.

Using some useful C++ classes

How to write to files?

```
// create an ofstream
ofstream out;

// choose where to write
out.open("myfile.txt");

out << "The first line\n";
out << "The second line\n";
out << "The third line\n";

// always close when you are finished
out.close();
```

Works just like std::cout **except** for the **open and closing**.

Using some useful C++ classes

Passing a stream as a parameter Pass a reference to an ostream .

```
void writeHaiku( ostream& out ) {
    out << "The wren\n";
    out << "Earns his living\n";
    out << "Noiselessly.\n";
}
```

Using some useful C++ classes

Passing a stream as a parameter

```
void testWriteHaiku() {
    // write a Haiku to cout
    writeHaiku( cout );
    // write a Haiku to a file
    ofstream out;
    out.open("haiku.txt");
    writeHaiku( out );
    out.close();
}
```

Why can we do this? Because an ofstream is an ostream .

Using some useful C++ classes

Working with strings

```
// Create a string
string s("Some text.");

// Write it to a stream
cout << s<< "\n";
cout << "Contains "
     << s.size() <<
     " characters \n";
```

Using some useful C++ classes

Working with strings

```
// Change it
s.insert( 5, "more ");
cout << s <<"\n";

// Append to it with +
s += " Yet more text.";
cout << s <<"\n";
// Test equality
if ( s=="Some more text. Yet more text.");
```

Using some useful C++ classes

Technical points about strings

- Using a **string** is better than using a **char*** (C style strings) because they have lots of helpful functions!

Using some useful C++ classes

Working with strings efficiently

Using + to build up strings is slow. Don't do this:

```
string s("");
for (int i=0; i<100; i++) {
    s+="blah ";
}
cout << s<<"\n";
```

Using some useful C++ classes

Working with strings efficiently Do this:

```
stringstream ss;
for (int i=0; i<100; i++) {
    ss<<"blah ";
}
string s1 =ss.str();
cout << s1 <<"\n";
```

You have to use *#include < sstream >*. A stringstream is an ostream .

Summing up

We have learnt about:
- Classes with pointer data member variables and the rule of three
- We have studied the class YourVector, which simulates the Vector of the Standard Library
- We have learnt about several useful classes of the Standard Library.