*Roxana Dumitrescu*

# C++ in Financial Mathematics

## What have we learnt?

- Pointers to variables
- References
- Passing parameters to functions: by value, pointers and references
- Returning pointers/references

## Arrays

**Arrays**

## Arrays

An array is a series of elements of the same type placed in contigous memory locations that can be invidually referenced by adding an index to a unique identifier.
For example, five values of type int can be declared as an array without having to declare 5 different variables (each with its own identifier). Instead, using an array, the five int values are stored in contiguous memory locations, and all five can be accessed using the same identifier, with the proper index.

Like any variable, an array must be declared before it is used.

```
type name[number_elements];
```

## Arrays

```
    // Create an unintialized of length 5
    int myArray[5];
    for (int i=0; i<5; i++) {
        cout<<"Entry "<<i<<"=";
        cout<<myArray[i];
        cout<<"\n";}
```

- Create an array of 5 integers, without initialising it.
- Run through the entries and print them out.
- The entries start at 0.
- We use [] to access entries.
- There is no size function.

## Arrays

```
// Create an initialised array
int myArray[] = {1, 1, 2, 3, 5};
for (int i=0; i<5; i++) {
    cout<<"Entry "<<i<<"=";
    cout<<myArray[i];
    cout<<"\n";
}
```

- We can initialise an array by specifying the values.
- Simply place the values in a comma separated list between curly brackets.
- Notice that we no longer have to specify the length of the array when we create it.

## Arrays

```
// Create an initialised array
int myArray[] = {1, 1, 2, 3, 5};
for (int i=0; i<5; i++) {
    cout<<"Entry "<<i<<"=";
    cout<<myArray[i];
    cout<<"\n";
}
```

- We can initialise an array by specifying the values.
- Simply place the values in a comma separated list between curly brackets.
- Notice that we no longer have to specify the length of the array when we create it.

## Arrays

```
// Create an initialised array to 0
int myArray[5] = {0};
for (int i=0; i<5; i++) {
    cout<<"Entry "<<i<<"=";
    cout<<myArray[i];
    cout<<"\n";
}
```

- We specify the size of the array.
- We assign it the value {0}.
- This gives an array of the desired length full of zeros.

## Arrays

```
    // Create a general initialised array
    int myArray[5] = {1,2,3};
    for (int i=0; i<5; i++) {
        cout<<"Entry "<<i<<"=";
        cout<<myArray[i];
        cout<<"\n";
    }
```

- This prints out the values 1, 2, 3, 0, 0.
- The length of the array is specified.
- Some of the values are specified; the rest is padded with zero.

## Arrays

### Passing arrays to functions

```
    int sumArray( int toSum[], int length ) {
    int sum = 0;
    for (int i=0; i<length; i++) {
        sum+=toSum[i];
    }
    return sum;
}
```

- One problem with arrays is that because we don’t have a function which gives automatically their **size**, we must pass their length to a function. So, the functions receive as parameters the array and its length.

## Arrays

**How to call sumArray function in the main program?**

```
int main()
{  int n=5;
int a[5]={1,2,3,4,5};
cout<<sumArray(a,n);
}
```

- The function call is **sumArray(a,n).**

## Arrays

**Don’t return arrays!**

- Do NOT return arrays from functions.
- The caller receives a pointer to *where the array used to be*. The computer may have reused that memory for almost anything.
- If you attempt to return an array, the behaviour is undefined.

## Arrays

```
int* thisFunctionReturnsAnArray(int &length) {
    /* This produces a compiler warning */
    length=5;
    int array[5] = {1,2,3,4,5};
    return array;
}

void testDontReturnArrays() {
    int length=0;
    int* b =
   thisFunctionReturnsAnArray(length);
    for (int i=0;i<length;i++)
    cout << b[i]<<" ";
    cout << "\n";}
}
```

## Arrays

**Don't return arrays!**

```
int main()
{ testDontReturnArrays();
return 0;
}
```

I have a warning message: "Address of stack memory associated with local variable array returned".

Execute the program $\mapsto$ strange values!

## Arrays

**You can't vary the length of an array!**

- You cannot change the length of an array.
- You cannot insert a new item or add some at the end.
- In fact the size is fixed AT COMPILE TIME!

## Arrays

### Multi-dimensional arrays

```
// Create an initialised 3x5 array
int myArray[][5] = {{1, 2, 3, 4, 5},
                    {2, 0, 0, 0, 0},
                    {3, 0, 0, 0, 0}};
for (int i=0; i<3; i++) {
    for (int j=0; j<5; j++) {
        cout<<"Entry ("<<i<<","<<j<<")=";
        cout<<myArray[i][j];
        cout<<"\n";
    }
}
```

- RULE: You have to write explicitly the last dimension!

## Arrays

**Remark:** In addition to accessing array elements using subscripts, array elements can also be accessed using pointers. Because an array name returns the starting address of the array (the address of the first element of the array), *an array name can also be used as a pointer to the array.*

```
    int array[5]={0,1,2,3,4}
```

**How to access the elements?**

- Array-indexing:

```
    array[0] // first element
    array[1] // second element
    array[2] // third element
...
```

## Arrays

- Pointer notation:

```
    *array; // first element
    *(array+1) // second element
    *(array+2) // third element
...
```

## Arrays

### Arrays and pointers

Because arrays are not flexible enough, we can work with pointers! This allows to work with sequences of data of varying lengths.

```
int n = 5;
int* myArray = new int[n];
for (int i=0; i<n; i++) {
    cout<<"Entry "<<i<<"=";
    cout << myArray[i];
    cout << "\n";
}
delete[] myArray;
```

## Arrays

### Arrays and pointers

- **int * myArray** contains the memory address where the array starts.
- We use the **new ...[]** operator to allocate a chunk of memory. We are creating a sequence of int data types data in memory, but you can use other types of data instead.
- You can choose the size at runtime.
- The memory created will **NOT** be automatically deleted when the function exits.

## Arrays

### Arrays and pointers

- You must use **delete []** operator to manually delete everything you create with the new[] operator. As we'll see, this is good and bad.
  - With arrays we couldn't return arrays because the memory was deleted automatically, but we don't have to remember to call **delete[]**.
  - With memory created using **new[]** we have to remember to delete the memory by hand, but you can safely return the data.

## Arrays

### Arrays and pointers

```
int sumUsingPointer( int* toSum, int length ) {
    int sum = 0;
    for (int i=0; i<length; i++) {
        sum+=toSum[i];
    }
    return sum;
}
```

- We specify the type of the parameter as `int *`.
- The code here is identical to that with arrays except that we declare the type using `*` rather than `[]`.
- Note that you have to pass the number of elements as well as the pointer.

## Arrays

### Arrays and pointers

#### Returning arrays dynamically allocated

- We have seen that you should never return arrays from functions! If you do it, the code will behave unpredictably. It probably will print some junk if you run it.
- You are allowed to return a pointer created with `new []`, but then you'll have to make sure the caller knows whether or not they will be expected to call `delete[]` at some point.
- By convention in C and C++, if a function returns a pointer, the caller is **NOT** expected to call `delete[]`.

## Arrays

```
int* thisFunctionReturnsAPointer(int& n) {
    int* ret = new int[n];
    for (int i=0;i<n;i++) ret[i]=i;
    return ret;
}

void usingReturnPointerFunction() {
    int n=5;
    int* b= thisFunctionReturnsAPointer(n);
    for (int i=0;i<n;i++) cout<<b[i];
   // free the memory
    delete[] text;
}
```

This violates the convention on NOT deleting the return value of a function, so it is considered to be confusing code.

## Arrays

### Looping with pointers

```
int sumUsingForAndPlusPlus( int* begin, int n)
    {
     int sum = 0;
     int* end = begin + n;
     for (int* ptr=begin; ptr!=end; ptr++) {
          sum += *ptr;
     }
     return sum;
}
```

- You can use `++` to move a pointer on to the next item.
- You can use `==` to compare pointers.
- This code is equivalent to the last one, we just use `++` instead of arithmetic.

## Arrays

### Looping with pointers

The previous code is equivalent to:

```
int sumUsingForAndPlusPlus( int* toSum, int n)
    {
     int sum = 0;
     for (int i=0; i<n; i++)
{sum+=toSum[i];}
     return sum;
}
```

The above code is identical to the previous one, excepting that we declare the type using * rather than [].

## C Style Strings

**C Style Strings**

## C Style Strings

The *C*-style character string originated within the *C* language and continues to be supported within $C++$. The string is actually a one-dimensional array of characters which is terminated by a **null** character '\0'.

The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more then the number of characters in the word "Hello".

## C Style Strings

```
char greeting[6]={'H', 'e', 'l', 'l', 'o',
    '\0'};
```

You can also write the above statement as follows:

```
char greeting[]="Hello";
```

## C Style Strings

```
#include <iostream>
using namespace std;
int main();
{
char greeting[6]={'H','e','l','l','o', '\0'};
cout<<"Greeting message:  ";
cout<<greeting<<endl;
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Greeting message: Hello
```

## Object-oriented programming
## Classes

## Classes

A class can be visualized as a three-compartment box, as follows:

- **Classname** (or identifier): identifies the class
- **Data members** or **Variables** (or *attributes*, *states*, *fields*): contains the *static attributes* of the class.
- **Member functions** (or *methods*, *behaviors*, *operations*): contains the *dynamic operations* of the class.

**Class Members:** The data members and member functions are collectively called class members.

## Classes

**Terminology**

We already know the following.

- Every *variable* in C++ has a type (for example, **double**).

We are learning a programming style called "object-oriented programming" in which we should use a slightly different vocabulary for the same ideas.

- Every *object* in C++ has a class (for example, **Myclass**).

## Classes

The term *object* in object-oriented programming refers to a data type which consistes of a bundle of data together with helpful functions to work with that data.

**double**, **int**, **char*** are not object-oriented types because they consist purely of data with no special helper functions.

The type of data and functions supported by an object depend only on its class. All instances of the same class have the same functions and the same types of data.

## Classes

**An intuitive example**

Table : Class Student

| | |
|---|---|
| **Classname** (Identifier) | **Student** |
| **Data Member** (Attributes) | name ; grade |
| **Member Functions** (Operations) | getName() ; printGrade() |

## Classes

### An intuitive example

Table : Instances class Student

| | |
|---|---|
| **Classname** | **Paul: Student** |
| **Data Member** | name="Paul Lee"; grade=3.5 |
| **Member Functions** | getName(); printGrade() |

| | |
|---|---|
| **Classname** | **Peter: Student** |
| **Data Member** | name="Peter Tan"; grade=3.9 |
| **Member Functions** | getName(); printGrade() |

## Classes

### A first example

**Class declaration:**

```
class Day
{ public:
int year;
int month;
int day;
void output();
};
```

## Classes

- The name of our class is **Day**.
- Our class **Day** contains 3 values called *year*, *month*, *day*. These represent the *members variables*. The class also contains a member function *void output()*.
- Users of our class are allowed to use the values of *year*, *month* and *day*, as well as the fonction *output()* in their own code, so these are marked **public**.

## Classes

### A first example

### Using the class

```
int main()
{
Day today, birthday;
cout<<"Enter today’s date: \n";

cin>>today.year;
cin>>today.month;
cin>>today.day;
cout<<"Enter your birthday:\n ";
cin>>birthday.year;
```

## Classes

### A first example

```
cin>>birthday.month;
cin>>birthday.day;
today.output();
cout<<"Your birthday is";
birthday.output();
if(today.year==birthday.year&&
   today.month==birthday.month &&
   today.day==birthday.day)
cout<<"Happy birthday!\n";
else cout<<"Happy Unbirthday! \n";
return 0;
}
```

## Classes

**A first example**

**Member function definition**

```
void Day::output()
{
cout<<"year= "<<year<<"month= "<<month<< ",day
    = "<<day<<endl;}
```

## Classes

### A first example

### Analysis of the code in the main function

- We first create two instances of the class Day: *today* and *birthday*.

```
Day today, birthday;
```

- We then initialize the objects *today* and *birthday*:

```
cin>>birthday.day;...
```

- When you want to access the data inside a class you use a dot . followed by the name of the member variable.

```
birthday.day
```

  In our case, you can access the variables because they have been marked as **public**. Remove the word **public** and see what compiler error you get.

## Classes

### A first example

### Methods

**Calling member functions:**

- The member function output is called with the object today as follows: **today.output()**
- The member function output is called with the object birthday as follows: **birthday.output()**

## Classes

**A first example**

**Defining member functions:**

- When a member function is defined, the definition must include the class name because there may be two or more classes that have member functions with the same name.

  **void DayofYear::output()**

- A member function is defined in the same way as any other function **except** that the *Class_name* and the scope resolution operator :: are given in the function heading.

## Classes

**Defining member functions: SYNTAX**

```
Returned Type  Class name:: Function
   Name(Parameter List)
{ Function Body Statements}
```

## Classes

### public and private members

- **Data hiding** is one important feature of Object Oriented Programming which allows preventing the direct access to the internal representation of a class type. The access restriction to the class members is specified by the labels **public** and **private.** The keywords public and private are called **access specifiers**.
  A class can have multiple public, or private labeled sections.

## Classes

**public and private members**

(i) The **public** members are accessible from anywhere outside the class but within the program. You can set and get the value of public variables without any member function.

(ii) A **private** member variable or function cannot be accessed, or even viewed from outside the class **Only the class** can access private members (we'll see later the friend functions). By default all the members of a class would be private.

## Classes

**GENERAL SYNTAX**

```
class CLASS_NAME{
public:
DATA AND FUNCTION DECLARATIONS
private:
DATA AND FUNCTION DECLARATIONS

     };
```

## Encapsulation

The idea of hiding data using **private** keyword is often called **encapsulation**. This concept refers to two things:

- Combining a number of items - variables and functions - into a single package, such as an object of some class;
- Preventing direct messing with the internal data of an object.

**Encapsulation** is the first feature of Object Oriented Programming!

## Encapsulation

**An intuitive example from the "real" life: The design of a car**

- In a car all the lighting controls are put on the dashboard, they are separeted from the controls for the windows and the seats. This grouping of functionality in a car's controls corresponds to the grouping of functionality into different classes in software design.
- You can control the car through standard functions (turn left, turn right) but the internal workings of a car are hidden from the user completely $\Rightarrow$ **Software design principle**: keep most of the details private and make a small number of functions public (only those which are necessary to the user of the class).

## Encapsulation

**An intuitive example from the "real" life: The design of a car**

- Cars are much easier to use because of the use of encapsulation. You don’t need to be a trained mechanic in order to drive a car! Similarly, the concept of encapsulation in object oriented programming makes programs easy to use.

## Encapsulation

**Remark**

It is considered good programming style to make *all* member variables private on your class. Doing this allows you to guarantee that your object always remains in a consistent state. In addition it allows you to change your mind in the future about the implementation details (i.e. how data is stored), without users of your class being affected in anyway.

## Classes

**An example of class with private members**

```
class Day
{ public:
        void output();
        void set (int new_year, int new_month,
    int new_date);
        void input();
        int get_year();
        int get_month();
        int get_day();
    private: void check_date();
int year;
int month;
int day;};
```

## Classes

### An example of class with private members

```
void Day::input() { cout<<"Enter the year:";
   cin>>year;
cout<<"Enter the month:"; cin>>month;
cout<<"Enter the day";
cin>>day;
check_date();
};

void Day::checkdate()
{ if ((year<1)&&(year>2017)||(month<1) ||
   (month>12) || (day<1) || (day>31))
  {cout<<"Illegal date. Aborting program.\n";
        exit(1);}
}
```

## Classes

```
int Day::get_year()
{ return year;}

int Day::get_month()
{  return month;}

int Day::get_day()
{ return day;}

void Day::output(){...}

void  Day::set(int new_year, int new_month,
   int new_date){ year=new_year;
month=new_month;
day=new_date; check_date();}
```

## Classes

**An example of class with private members**

- today.year=1997; //NO.
- today.month=12; //NO.
- today.day=25; // NO.
- cout«today.month; // NO.
- cout«today.day; // NO.
- if (today.month==1) //NO.

You can access the member variables only by using the member function set.

- today.set(12,1, 1) // OK

The function **set** is called **accessor function**.

## 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** always points to the object being operated on. More precisely, "this" is a const pointer. 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 (see next lecture!).

## 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. 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.

## 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 of .h
- *The implementation of the class* goes into the .cpp file
- File containing the program that uses your class (which has the extension .cpp).

## Working with different files

File: **Num.h**

```
# pragma once
class Num
{
private:
int num;
public:
Num(int n);
int getNum();

}
```

## Working with different files

File: **Num.cpp**

```
# include "Num.h"
Num::Num(): num(0) {}
Num::Num(int n): num(n) {}
int Num::getNum()
{
return num;
};
```

## Working with different files

File: **main.cpp**

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

using namespace std;

int main()
{
Num n(35);
cout<<n.getNum()<<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.
- 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

- Arrays (static and dynamic allocation)
- Classes
  - The notions of object/instance
  - Member values and member functions
  - The concept of encapsulation
  - Constructor/destructor
  - This pointer
  - Classes with static memebrs
- Working with different files