Roxana Dumitrescu

C++ in Financial Mathematics

What have we learnt?

- General structure of a C++ program
- Basic types of data, casting
- Variables, constants: need to be declared before they are used

```
int x;
x=5;
```

A const can't be changed!

```
const double y=5;
y=7; // ERROR
```

- Boolean expressions and operators (assignment, arithmetic operators, compound operators, logic operators (AND, OR, NOT)), conditional ternary operator.

Plan

- Flow of controls
- Functions
- Arrays

Flow of control

- If/switch statement
- Loops : for, while, do-while.
- Jump statements

Flow of control

If statement

The syntax is the following:

```
if (expression1) {
statements1
} else if (expression2) {
statements2
} else if (expression3) {
statements3
} else {
statements4
}
```

Flow of control

If statement

The syntax is the following:

```
if (expression1) {
statements1
} else if (expression2) {
statements2
} else if (expression3) {
statements3
} else {
statements4
}
```

Flow of control

If statement

The *expression* is allowed to be any basic data type. The value 0 is interpreted as **false** and other values are interpreted as true. For example, the following code prints out that the test passed.

```
if (-1.743) {
cout<<"Test passed \n";
}
```

Flow of control

If statement

A problem occurs when you use $=$ in tests by accident instead of $==$.

```
int i=1;
int j=3;
if (i=j) {
cout<<"i is equal to j \n";
}
```

In the if statement, the code assigns a value of 3 to the variable *i*, and then, observing that 3 is non-zero, the above code prints out the false claim that *i* is equal to *j*.

Flow of control

Switch statement

A more complex alternative to an **if** statement is a **switch** statement.

```
switch (expression){
case constant1:
statementA1
statementA2
...
break;
case constant2:
statement B1
statement B2
...
break;
...
```

Flow of control

Switch statement

```

default:
statementZ1
statementZ2
...
}
```

- The switch evaluates expression and, if expression is equal to constant1, then the statements beneath case constant 1: are executed until a break is encountered.
- If expression is not equal to constant1, then it is compared to constant2. If these are equal, then the statements beneath case constant 2: are executed until a break is encountered.

Flow of control

Switch statement

- If not, then the same process repeats for each of the constants, in turn. If none of the constants match, then the statements beneath default: are executed.
- Due to the peculiar behavior of switch-cases, curly braces are not necessary for cases where there is more than one statement (but they are necessary to enclose the entire switch-case).
- switch-cases generally have if-else equivalents but can often be a cleaner way of expressing the same behavior.

Flow of control

```
int main(){
int x=7;
switch(x){
    case 1: cout<<"x is 1 \n";
break;
    case 2: cout<<"x is 2 \n";
break;
    case 3: cout<<"x is 3 \n";
break;
    default:
cout<<"x is not 1,2,3";
    }
return 0;}
```

Flow of control

The program will print

```
x is not 1,2, or 3.
```

If we replace

```
int x=7;
```

with

```
int x=2;
```

then the program will print "x is 2".

Flow of control

Programming languages (in particular C++) provide various control structures that allow for the execution of a statement or group of statements multiple times. These control structures are called **loops**.

Flow of control

While loop

A **while** loop statement repeatedly executes a target statement as long as a given condition is true.

The syntax of a while loop in C++ is:

```
while(condition){
    statement(s);
}
```

Statement may be a single statement or a block of statements. The loop iterates while the condition is true. When the condition becomes false, the control is passed to the line following the loop.

Flow of control

```
int main()
{
    int a=10;
    while(a<20)
    {
        cout<<"Value of a:"<<a<<endl;
        a++;
    }
    return 0;
}
```

Values printed on the screen: 10, 11, ...19.

Flow of control

```
void loopForever()
{
    while (true) {
        cout<<"Still looping\n" ;
    }
}
```

This program will loop forever. If you run the program, you can stop it by typing **CTRL+C** (on Windows for e.g.).

Flow of control

2. The do-while loop

```
do {statement} while (condition);
```

It behaves like a while-loop, except that *condition* is evaluated after the execution of *statement* instead of before, guaranteeing at least one execution of *statement*, even if *condition* is never fulfilled.

Flow of control

```
int main()
{
     int n=5;
     do { cout<<n<<" ";
          n--;
     } while (n!=0);
     return 0;
}
```

Values printed on the screen: 5, 4, 3, 2, 1.

Flow of control

3. The for loop

```
for(init; condition; increment){
statement(s);}
```

Like that while-loop, this loop repeats statement while condition is true. In addition, the for loop provides an *initialization* expression (executed before the loop begins the first time) and an *increase* expression (after each iteration).

Flow of control

```
int main() {
for (int i=0; i<20; i++) { cout<<"Value of
   i:"<<i<<endl;}
return 0;
}
```

Remark: C++ programmers start counting at 0. This is a metter of convention. All the standard data structures in C++ are labeled so that they start with 0. So it is strongly recommended that when programming in C++ to start counting from 0 and to use $<$ signs rather than $\leq$ to compensate.

Flow of control

Jump statements

Break

break leaves a loop, even if the condition for its end is not fulfilled. It can be used to end an infinite loop, or to force it to end before its natural end.

```
int main() {
     for (int n=10; n>0; n--)
     {      cout<<n<<",";
          if (n==3)
          {
               cout<<"countdown aborted!";
               break;}
      }
}
```

Flow of control

Continue

The *continue* statement causes the program to skip the rest of the loop in the curent iteration and go to the next statement. Thus *continue* means "continue looping" whereas *break* means "break out of the loop".

```
int main() {
    for (int n=10; n>0; n--)
    {    if (n==5) continue;
        cout<<n<<",";
    }
    cout<< "End program";
}
```

Functions

Functions

- Build in functions
- User-defined functions: void and value returning
- Overloading
- Scope of the variables
- Static variables

Functions

- A function in C++ is a piece of code that you can call to perform some task
- They allow complicated programs to be divided into manageable pieces
- Some advantages of functions:
    - A programmer can focus on just that part of the program and construct it, debug it, and perfect it
    - Different people can work on different functions simultaneously
    - Can be re-used (even in different programs)
    - Enhance program readability

Functions

- Functions
  - Called modules
  - Like miniature programs
  - Can be put together to form a larger program

Functions

Predefined (Build-in) Functions

- In algebra, a function is defined as a rule or correspondance between values, called the function's arguments, and the unique value of the function associated with the arguments
  - If $f(x) = 2x + 5$, then $f(1) = 7$, $f(2) = 9$, and $f(3) = 11$
    - 1,2, and 3 are arguments
    - 7,9, and 11 are the corresponding values

Functions

Predefined (Build-in) Functions

- Some of the predefined mathematical functions are:

  sqrt(x)
  pow(x,y)
  floor(x)

- Predefined functions are organized into separate libraries
- I/O (input/output) functions are in **iostream** header
- Math functions are in **cmath** header (you have to use # include<cmath>).

Functions

Predefined Functions

- pow(x,y) calculates x^y
  - pow(2.,3)=8.0
  - Returns a value of type **double**
  - x and y are the parameters (or arguments)
    - The function has two parameters
- sqrt(x) calculate the nonnegative square root of x, for x>=0.0
  - sqrt(2.25) is 1.5
  - Type **double**

Functions

Predefined Functions

- The floor function floor(x) calculates the largest whole number not greater than x
  - floor(48.79) is 48
  - Type **double**
  - Has only one parameter

Functions

User-Defined Functions

- **Value-returning functions:** have a return type
  - Return a value of a specific data type using the **return** statement
- **Void functions:** do not have a return type
- Do **not** use a **return** statement to return a value (you can only use the keyword return (without a value!) in order to return the execution to the main program - we'll see an example).

Functions

User-Defined Functions - Value returning functions

Let us now consider the following mathematical function:

$$\textbf{compoundInterest}\colon \mathbb{R} \times \mathbb{R} \times \mathbb{Z} \to \mathbb{R}$$

given by

$$\textbf{compoundInterest}(P, i, n) = P\left(1 + \frac{i}{100}\right)^{n} - P.$$

Here P is the principal, i is the annual percentage rate, and n is the number of years.

Functions

User-Defined Functions

In C++ , one would write the **compoundInterest** function as follows:

```
double compoundInterest(double P, double i,
   int n)
{
    double interest = P*pow(1+0.01*i,n)-P;
    return interest;
}
```

Functions

Value returning Functions

The first line contains the following items (from left to right)

- Data type of the value returned: called the type
- Name of the function
- Number of parameters
- Data type of each parameter of the function

Functions

Value returning Functions

```
ReturnType functionName (ParameterType1
   parameterName1, ParameterType2
   parameterName2, .... ){
... statements...}
```

Functions

Value returning Functions - example of program

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

double compoundInterest(double P, double i,
   int n){
    double interest=P*pow(1+0.01*i, n)-P;
    return interest;
}

int main(){
    int principal;
    double interestRate;
    int numberOfYears;
```

Functions

```
    cout<<"How much are you investing?\n"
    cin>>principal;
    cout<<"What's the annual ineterest rate?\n"
    cin>>interestRate;
    cout<<"How long for (years)?\n ";
    cin>>numberOfYears;
    double interest=
   compoundInterest(principal, interestRate,
   numberOfYears);
    cout<<"You will earn";
    cout<<interest;
    cout<<"\n";
    return 0;
}
```

Functions

Remarks

- The definitions of the functions are written sequentially in the file. We first define compoundInterest completely, then we define the main completely.
- The variable names used when we call the function can be completely different from those used in the definition of the function. When we choose the names of parameters and variables in a function, those names have no meaning outside of the function.
- There is a **return** statement at the end of each function. When a function has computed the desired value, it sends it back to the caller using the **return** keyword.

Functions

Remarks

- **The declaration and definition of the functions:**
  - Notice that we defined the function **compoundInterest** before the **main** function. We did this because in **main** the **compoundInterest** function is called.
  - Alternative: C++ allows **to declare a function and then to define it.**

Functions

Value returning Functions - example of program

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

double compoundInterest(double P, double i,
   int n); // declaration

int main(){
    int principal;
    double interestRate;
    int numberOfYears;
    cout<<"How much are you investing?\n"
    cin>>principal;
    cout<<"What's the annual ineterest rate?\n"
```

Functions

```
     cin>>interestRate;
     cout<<"How long for (years)?\n ";
     cin>>numberOfYears;
     double interest=
    compoundInterest(principal, interestRate,
    numberOfYears);
     cout<<"You will earn";
     cout<<interest;
     cout<<"\n";
     return 0;
}
```

Functions

```
double compoundInterest(double P, double i,
   int n) // definition
{
double interest=P*pow(1+0.01*i, n)-P;
return interest;
}
```

Functions

Functions that do not return a value

Very often you want a function to perform a task and don't actually want to compute a value. To do this, you use the special keyword **void** to describe the return type.

```
void printHello(){
    cout<<"Hello\n";
}
```

Functions

Special case of using return in a function that do not return a value

In a functions that do not return a value, you can use *a return* statement in the middle of a loop. This stops all looping and returns the execution to the point where the function was called.

```
void countdown() {
    int i=10;
    while (true) {
        if (i==0) {
            return;
        }
    cout<<i<<"\n";
    i--;
    }
}
```

Functions

Recursion

Function can call other functions. For example our **compoundInterest** function calls the function **pow**. An interesting feature is that functions can call themselves. This programming *technique* is called *recursion*.

Consider the following recursively defined sequence (which gives $n!$):

$$x_n = nx_{n-1}, \quad n \geq 1$$

with $x_0 = 1$.

Functions

Recursion

How to implement this recursive function in C++ ?

```
int factorial (int n)
{
    if(n==0){ return 1;}
    return n*factorial(n-1);
}
```

Functions

Overloading C++ allows *overloading* the function name, i.e. it allows more than one function to have the same name, provided all functions are either distinguishable by the typing or the number of their parameters.

Functions

Overloading - Example 1

```
int average(int first_number, int
   second_number, int third_number);

int average(int first_number, int
   second_number);
int main()
{     int number_A = 5, number_B = 3, number_C
   = 10;

    cout << "The integer average of " <<
   number_A << " and ";
    cout << number_B << " is ";
    cout << average(number_A, number_B) <<
   ".\n\n";
```

Functions

Overloading - Example 1

```
    cout << "The integer average of " <<
   number_A << ", ";
    cout << number_B << " and " << number_C <<
   " is ";
    cout << average(number_A, number_B,
   number_C) << ".\n";

    return 0;
}
```

Functions

Overloading - Example 1

```
int average(int first_number, int
   second_number, int third_number)
{
    return ((first_number + second_number +
   third_number) / 3);
}
int average(int first_number, int
   second_number)
{
    return ((first_number + second_number) /
   2);
}
```

Functions

Overloading - Example 2

```
int max(int a, int b)
{
    if (a>b) return a;
    return b;
}

double max(double a, double b)
{
    if (a>b) return a;
    return b;
}
```

Functions

The C++ compiler can work out which function we are calling. The code $max(1, 2)$ would call the first version, whereas the code $max(1.0, 2.0)$ would call the second version. This is desirable because we are avoiding unnecessary conversions from int variables to double variables.

The identity of a function is determined by both its name and the types of its parameters, this combination being called *signature* of the function. Two functions are the same if they have the same signature.

If when you call a function there isn't a version with just the right signature available, C++ will perform automatic casting if necessary. For example, if you type $max(1, 2.0)$, it will call the version of the code which treats all parameters as doubles.

Functions

Global and local variables

Consider the following code:

```
const double PI =  3.141592653589793;

double computeArea (int r){
    double answer= 0.5*PI*r*r;
    return answer;
}

double computeCircumference (int r){
    double answer= 2.0*PI*r;
    return answer;
}
```

Functions

Global and local variables

The *scope* of a variable refers to the parts of code where that variable can be used. We have two kinds of variables:

- *Global variables*:declared outside any function (Example: the constant PI)
- *Local variables*: the variable *answer*

The names for local variables within a function have no relationship with the names you use in another function. For example, we reused the variable name "answer" in two different functions to refer to different quantities.

Functions

Static variables (with local scope)

A local variable is initialized at each function call and each function call generates a copy of the variable. If a local variable is declared **static**, an unique object representing this variable will be created for all function calls. The static variable is initialized at the first execution of its definition.

We give now an example of a function which contains a loop. Inside the loop, we define two variables: *n* (which is of static type) and *x* (which is a normal variable). Compare the results!

Functions

Example

```
void f(int a)
{   while (a--)
    {   static int n=0;
        int x=0;
        cout<<"n== "<<n++<<",x=="<<x++<<'\n';
    }
}

int main()
{ f(3);}
```

Results:

```
n==0, x==0;  n==1, x==0; n==2, x==0.
```

Summing up

- Flow of control
  - Conditional flow of control: if, switch
  - Loops: for, while, do-while

Summing up

- Functions
  - Predefined (built-in) functions - examples: pow, sqrt, floor (in order to use them, you have to include **cmath**)
  - User defined functions:

    - Value returning function

```
double max (double a, double b);
```

  - Void function

```
void print (double a);
```

Summing up

- Functions
    - Signature of a function: name+number of parameters+type of parameters.
    - Oveloading functions - you can write functions with the same name, but different signatures!

```
double max (double a, double b);
int max (int a, int b);
```

    - Scope of the variable: global and local
    - Static variables (with local scope)