Introduction to C++
M2 MMMEF

Fayçal DRISSI
Maison des Sciences Economiques

November 26, 2021

Université paris 1 - Sorbonne

Outline of the course

1. Introduction to C++
2. Objects, Types, and Values
3. Computations and functions in C++
4. Errors, Exceptions, Debugging and Testing
5. C++ functions and technicalities
6. Object-Oriented-Programming and Classes
7. Review
8. Last Homework

Introduction to C++

Introduction to C++

What is C++

Introduction to C++

What is C++

- Improved version of the C language
- Support data abstraction
- Supports Object Oriented Programming
- Supports Generic programming

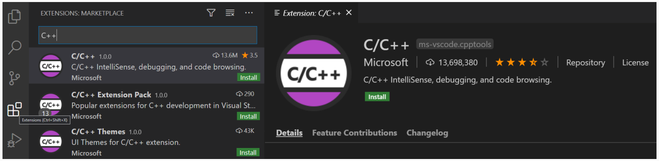
Introduction to C++

Installing C++

Introduction to C++

Installing C++

- Install an editor: Visual Studio Code
- Create a C++ development folder
- Install C/C++ Extension



Introduction to C++

Installing a C++ compiler

- Install a compiler for Windows: g++ with MinGw-x64
  `https://github.com/msys2/msys2-installer/releases/download/2021-06-04/msys2-x86_64-20210604.exe`
- Installation should be in C:/msys64/
- Add the path to the Mingw-x64 bin folder to the windows PATH:
  Windows Settings ->Edit environment variables for your account ->Choose Path ->Edit ->Select New ->add the path C:/msys64/mingw64/bin ->close and re-open vsual studio code.

Introduction to C++

Msys configuration

- Run "MSYS2 MSYS" from Start menu.
- On the msys2 command prompt and type:

  ```
  pacman -Syu
  ```
- Run "MSYS2 MSYS" again from Start menu.
- Update the rest of the base packages with

  ```
  pacman -Su
  ```
- Install some tools and the mingw-w64 GCC to start compiling:

  ```
  pacman -S --needed base-devel
  mingw-w64-x86_64-toolchain
  ```
- close this window and run "MSYS MinGW 64-bit" from Start menu. Now you can call make or gcc to build software

Introduction to C++

Hello, World!

Hello, World!

Writing the first program

- Create a file: helloworld.cpp, in your development folder, using Visual Studio Code.
- write the following code:

```
// This program outputs the message \Hello, World!"
// to the monitor

#include <iostream>
using namespace std;
int main() // C++ programs start by executing the function main
{
    cout << "Hello, World!" << "\n"; // output "Hello, World!"
    return 0;
}
```

Hello, World!

Building the first program

- Build Hello World: Terminal ->Run build task, or Ctrl+Shift+B
- Select "g++" in the choices.

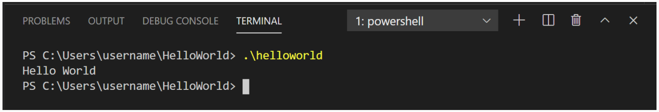
Executing the first program

- In a terminal (VS Code Integrated Terminal or normal terminal)
- Go to the dev folder and type:

  `./helloworld`

  `or`

  `./helloworld.exe`



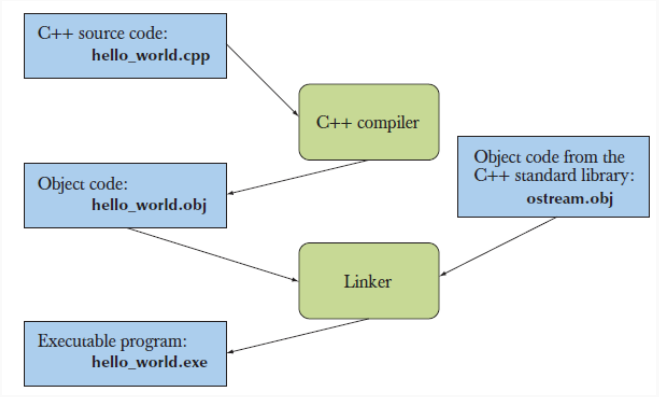
```
PS C:\Users\username\HelloWorld> .\helloworld
Hello World
PS C:\Users\username\HelloWorld> 
```

Hello, World!

Closer look to our first C++ program

```
// This program outputs the message \Hello, World!"
// to the monitor
#include <iostream>
using namespace std;
int main() // C++ programs start by executing the function main
{
    cout << "Hello, World!"; // output \Hello, World!"
    return 0;
}
```

Hello, World!



C++ source code:
hello_world.cpp
C++ compiler
Object code:
hello_world.obj
Object code from the C++ standard library:
ostream.obj
Linker
Executable program:
hello_world.exe

Hello, World!

The compiler

- Grammatical verification: does every word has a defined meaning ? is anything obviously wrong ?
- create object files: .o (Linux) or .obj (Windows)
- object files not portable between systems
- produces compile-time errors

Hello, World!

The compiler

- Grammatical verification: does every word has a defined meaning ? is anything obviously wrong ?
- create object files: .o (Linux) or .obj (Windows)
- object files not portable between systems
- produces compile-time errors

The Linker

- Links object files between them.
- produces link-time errors

Hello, World!

The compiler

- Grammatical verification: does every word has a defined meaning ? is anything obviously wrong ?
- create object files: .o (Linux) or .obj (Windows)
- object files not portable between systems
- produces compile-time errors

The Linker

- Links object files between them.
- produces link-time errors

The Execution

- produces run-time errors

Objects, Types, and Values

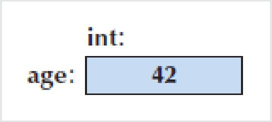
Objects, Types, and Values

Objects

Objects, Types, and Values

Simple definition of an object

- an object is a place, or a box in computer memory
- a *region* of *memory* with a *type* that specifies what kind of information can be placed in it.
- a named object is called a *variable*
- an object named "age", of type *int*, containing the integer 42:



Objects, Types, and Values

Input

Objects, Types, and Values

character input

```
// This program outputs the message \Hello, World!"
// to the monitor
#include <iostream>
using namespace std;
int main() // C++ programs start by executing the function main
{
    cout << "Please enter your name (followed by 'enter'):\n";
    string first_name; // first_name is a variable of type string
    cin >> first_name; // read characters into first_name
    cout << "Hello, " << first_name << "\n";
}
```

Objects, Types, and Values

Variables

Objects, Types, and Values

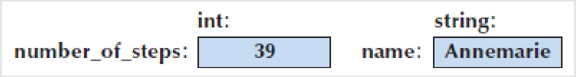
Variables and C++ jargon

- A place where we store data is called "object".
- To access an object, we need a "name"
- A named object is called a "variable"
- A variable always has a specific "type", such as *int* or *string*.
- The type of variable determines what we can put into the object.
- The type of variable determines which "operations" we can do with the object.
- Data items put into variable are called "values".
- A "statement" defining a variable is called a "definition".
- A "definition" usually provides an initial value.

Objects, Types, and Values

Variables example

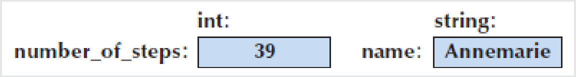
```
string name = "Annemarie";
int number_of_steps = 39;
```



Objects, Types, and Values

Variables example

```
string name = "Annemarie";
int number_of_steps = 39;
```



What you can't do

- You cannot put values of the wrong type into a variable

```
string name2 = 39; // error: 39 isn't a string
int number_of_steps = "Annemarie"; // error: "Annemarie"
                                   // is not an int
```

Objects, Types, and Values

Variable types

C++ provides a large number of types. Basic types:

```
int number_of_steps = 39; // int for integers
double flying_time = 3.5; // double for floating-point numbers
char decimal_point = '.'; // char for individual characters
string name = "Annemarie"; // string for character strings
bool tap_on = true; // bool for logical variables
```

Objects, Types, and Values

Variable types

C++ provides a large number of types. Basic types:

```
int number_of_steps = 39; // int for integers
double flying_time = 3.5; // double for floating-point numbers
char decimal_point = '.'; // char for individual characters
string name = "Annemarie"; // string for character strings
bool tap_on = true; // bool for logical variables
```

Variable values

```
39 // int: an integer
3.5 // double: a floating-point number
'.' // char: an individual character enclosed in single quotes
"Annemarie" // string: a sequence of characters
                  // delimited by double quotes
true // bool: either true or false
```

Objects, Types, and Values

Input and type

Objects, Types, and Values

>>and <<are sensitive to type

```
int main()
{
    cout << "Please enter your first name and age \n";
    string first_name = "???"; // string variable
    int age = 0; // integer variable (0 means "don't know the age")
    cin >> first_name >> age; // read a string followed by an integer
    cout << "Hello, " << first_name << " (age " << age << ")";

}
```

Objects, Types, and Values

>>and <<are sensitive to type

```
int main()
{
    cout << "Please enter your first name and age \n";
    string first_name = "???"; // string variable
    int age = 0; // integer variable (0 means "don't know the age")
    cin >> first_name >> age; // read a string followed by an integer
    cout << "Hello, " << first_name << " (age " << age << ")";

}
```

Rules:

- Reading of strings is terminated by what is called *whitespace*: space, newlines and tab
- Otherwise, whitespace by default is ignored by >>.
- we can read several values in a single input statement, just as we can write several values in a single output statement.

Objects, Types, and Values

Outputs

Input = "Bastien 22", Output =

Objects, Types, and Values

Outputs

Input = "Bastien 22", Output = "Hello, Bastien (age 22)"

Objects, Types, and Values

Outputs

Input = "Bastien 22", Output =
"Hello, Bastien (age 22)"
Input = "22 Bastien", Output =
"Hello, 22 (age 0)"

Objects, Types, and Values

Outputs

Input = "Bastien 22", Output =
"Hello, Bastien (age 22)"
Input = "22 Bastien", Output =
"Hello, 22 (age 0)"
Input = " Bastien", Output =

Objects, Types, and Values

Outputs

Input = "Bastien 22", Output =
"Hello, Bastien (age 22)"
Input = "22 Bastien", Output =
"Hello, 22 (age 0)"
Input = " Bastien", Output =
"Hello, Bastien (age 0)"

Objects, Types, and Values

Operations and Operators

Objects, Types, and Values

Addition and Substraction for Int and String

```
int main()
{
    int count;
    cin >> count; // >> reads an integer into count
    string name;
    cin >> name; // >> reads a string into name
    int c2 = count+2; // + adds integers
    string s2 = name + " Jr. "; // + appends characters
    int c3 = count-2; // - subtracts integers
    string s3 = name - " Jr. "; // error: - isn't defined for strings
}
```

Error of operation

Compile-time error ? Run-time error ? Link-time error ?

Objects, Types, and Values

Addition and Substraction for Int and String

```
int main()
{
    int count;
    cin >> count; // >> reads an integer into count
    string name;
    cin >> name; // >> reads a string into name
    int c2 = count+2; // + adds integers
    string s2 = name + " Jr. "; // + appends characters
    int c3 = count-2; // - subtracts integers
    string s3 = name - " Jr. "; // error: - isn't defined for strings
}
```

Error of operation

Compile-time error ? Run-time error ? Link-time error ?
The compiler knows exactly which operations can be applied to each variable and can therefore prevent many mistakes.

Objects, Types, and Values

Table of useful operators

Operation	bool	char	int	double	string
assignment	=	=	=	=	=
addition			+	+	
concatenation					+
substraction			-	-	
multiplication			*	*	
division			/	/	
remainder			%		
increment by 1			++	++	
decrement by 1			- -	- -	
increment by n			+= n	+= n	
decrement by n			-= n	-= n	

Objects, Types, and Values

Table of useful operators

Operation	bool	char	int	double	string
add to end					+=
multiply and assign			*=	*=	
divide and assign			/=	/=	
remainder and assign			%=		
read from s into x	s >>x	s >>x	s >>x	s >>x	s >>x
write x to s	s <<x	s <<x	s <<x	s <<x	s <<x
equals	==	==	==	==	==
not equal	!=	!=	!=	!=	!=
greater than	>	>	>	>	>
greater than or equal	≥	≥	≥	≥	≥
less than	<	<	<	<	<
less than or equal	≤	≤	≤	≤	≤

Objects, Types, and Values

Table of useful operators

`https://drive.google.com/drive/folders/1DY1Bl234ti2AzlDRHRICfosChfMSizlA?usp=sharing`

Objects, Types, and Values

Assignment and initialization

Objects, Types, and Values

Definitions

- Initialization: giving a variable its initial value
- Assignment: giving a variable a new value (destroys the old value)

```
int main()
{
    string a = "alpha"; // a starts out with the value "alpha"
    a = "beta"; // a gets the value "beta" (becomes "beta")
    string b = a; // b starts out with a copy of a’s value (that is, "beta")
    b = a+"gamma";
    a = a+"delta";

}
```

- Outputs for a and b ?

Objects, Types, and Values

Definitions

- Initialization: giving a variable its initial value
- Assignment: giving a variable a new value (destroys the old value)

```
int main()
{
    string a = "alpha"; // a starts out with the value "alpha"
    a = "beta"; // a gets the value "beta" (becomes "beta")
    string b = a; // b starts out with a copy of a’s value (that is, "beta")
    b = a+"gamma";
    a = a+"delta";

}
```

- Outputs for a and b ?

  a: betedelta

  b: betagamma

Objects, Types, and Values

Composite assignment operators

Objects, Types, and Values

Definition

changing the value of a variable based on its current value.

```
++counter; // means counter = counter + 1
a += 7; // means a = a+7
b -= 9; // means b = b-9
c *= 2; // means c = c*2
```

Objects, Types, and Values

Names (of variables)

Objects, Types, and Values

Names

- contain only letters, digits, and underscores
- are case sensitive
- choose meaningful names; not too long, not too short
- C++ style: underscores between words in an indentifier
- don't use all-capital-letter names, reserved for macros (conventionally)
- never use C++ keywords (about 85): if, string, while, ..
- never give names with starting underscores: clash with system entities or machine generated code

Objects, Types, and Values

Good names

`x, number_of_elements, Fourier_transform, z2, Polygon`

Objects, Types, and Values

Good names

```
x, number_of_elements, Fourier_transform, z2, Polygon
```

Bad names

```
2x // a name must start with a letter
time$to$market // $ is not a letter, digit, or underscore
Start menu // space is not a letter, digit, or underscore
```

Objects, Types, and Values

Good names

```
x, number_of_elements, Fourier_transform, z2, Polygon
```

Bad names

```
2x // a name must start with a letter
time$to$market // $ is not a letter, digit, or underscore
Start menu // space is not a letter, digit, or underscore
```

Find the 4 errors

```
int Main()
{
    STRING s = "Goodbye, cruel world! ";
    cOut << S << 'n';
}
```

Objects, Types, and Values

Types and objects

Objects, Types, and Values

Definitions

- a *type* defines a set of possible values and a set of operations (for an object).
- an *object* is some memory that holds a value of a given type.
- a *value* is a set of bits in memory interpreted according to a type.
- a *variable* is a named object.
- a *declaration* is a statement that gives a name to an object.
- a *definition* is a declaration that sets aside memory for an object.

Objects, Types, and Values

Memory

- Every *int* is of the same size. The compiler sets aside the same amount of memory: 4 bytes, or 32 bits.
- *bool*, *char* and *double* are also fixed size: 1, 1 and 8 bytes .
- different types of objects take up different amounts of space.
- meaning of bits in memory depends completely on the type:

```
00000000   00000000   00000000   011111000
```

can be "x" if it is read as a *char*, or 120 if it is read as *int*.

Objects, Types, and Values

Type safety

Objects, Types, and Values

Definition

- A program is *type-safe* when objects are used only according to the rules of their type.
- Complete type safety is the ideal and therefore the general rule for C++

Objects, Types, and Values

Definition

- A program is *type-safe* when objects are used only according to the rules of their type.
- Complete type safety is the ideal and therefore the general rule for C++

Type-safe examples

- Using a variable before it has been initialized is not considered type-safe
- Conversions with no information loss :
  - *bool* to *char*, *int* or *double*
  - *char* to *int* or *double*
  - *int* to *double*

```
char c = 'x';
int i1 = c;
int i2 = 'x';
```

Objects, Types, and Values

Type-unsafe examples

- Narrowing conversions:
    - *double* to *int*, *char* or *bool*
    - *int* to *char* or *bool*
    - *char* to *bool*

Objects, Types, and Values

Type-unsafe examples

- Narrowing conversions:
  - *double* to *int*, *char* or *bool*
  - *int* to *char* or *bool*
  - *char* to *bool*
- Example

```
int main()
{
    int a = 20000;
    char c = a; // try to squeeze a large int into a small char
    int b = c;
    if (a != b) // != means \not equal"
        cout << "oops!: " << a << "!=" << b << '';
    else
        cout << "Wow! We have large characters";
}
```

Objects, Types, and Values

Initialization in C++ 11

- C++11 introduced an initialization notation that outlaws narrowing conversions.
- Use {} instead of =

```
double x {2.7}; // OK
int y {x}; // error: double -> int might narrow
int a {1000}; // OK
char b {a}; // error: int -> char might narrow
```

- When the initializer is an integer, the compiler checks the size.

```
char b1 {1000}; // error: narrowing (assuming 8-bit chars)
char b2 {48}; // OK
```

Objects, Types, and Values

Exercises

Objects, Types, and Values

Exercise 1

Write a program that prompts the user to enter two integer values. Store these values in int variables. Write your program to determine the sum, difference, product, and ratio of these values and report them to the user. Make sure your output is clear and complete.

Exercise 2

Write a program that gives the ASCII integer number of a *char*.

Computations and functions in C++

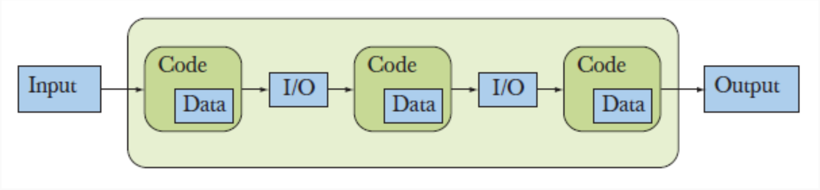
Computations and functions in C++

Computation

Computations and functions in C++

Definition

- The act of producing some outputs based on some inputs
- How do we express a program as a set of cooperating parts and how can they share and exchange data



Computations and functions in C++

Code structure in C++

- Should be as simple as possible to perform its task.
- Correctness, simplicity, and efficiency
- Structure and "quality of code" less time debugging

Computations and functions in C++

Code structure in C++

- Should be as simple as possible to perform its task.
- Correctness, simplicity, and efficiency
- Structure and "quality of code" less time debugging

Organizing a program: C++ tools

- Abstraction:
    - Hide details that we don't need
    - Convenient and general interface
- Divide and Conquer:
    - Divide a large problem into little ones.
    - C++ offers many tools to express the parts and their communications

Computations and functions in C++

Expressions

Computations and functions in C++

Expressions

- basic building block in C++.

Computations and functions in C++

Expressions

- basic building block in C++.

Constant expressions

- Symbolic constant, a named object to which you can't give a new value after it has been initialized

```
constexpr double pi = 3.14159;
pi = 7; // error: assignment to constant
double c = 2*pi*r; // OK: we just read pi; we don't try to change it
```

- Constants are better than *literals*

Computations and functions in C++

Constant expressions

- A *constexpr* symbolic constant must be given a value that is known at compile time.

```
void use(int n)
{
    constexpr int c1 = max+7; // OK: c1 is 107
    constexpr int c2 = n+7; // error: we don’t know the value of c2

}
```

Computations and functions in C++

Constant expressions, a second type: const

- Handles cases where the value of a "variable" that is initialized with a value that is not known at compile time but never changes after initialization.

```
constexpr int max = 100;
void use(int n)
{
    constexpr int c1 = max+7; // OK: c1 is 107
    const int c2 = n+7; // OK, but don't try to change the value of c2
    // ... some great and long code
    c2 = 7; // error: c2 is a const

}
```

Computations and functions in C++

Other operators for computations

Computations and functions in C++

Table of other useful operators

Operator	Name	Comment
f(a)	function call	pass a to f as an argument
!a	not	result is bool
-a	unary minus	
a<b	less than	result is bool
a $\leq b$	less than or equal	result is bool
a >b	greater than	result is bool
a $\geq b$	greater than or equal	result is bool
a==b	equal	result is bool
a!=b	not equal	result is bool
a && b	logical and	result is bool
a $\|b$	logical or	result is bool

Computations and functions in C++

Conversions: mixing different types in one expression

Computations and functions in C++

int and double conversion

- If an operator has an operand of type double, then C++ uses float arithmetics, otherwise int.
  - 5/2 is 2 (not 2.5)
  - 2.5/2 means 2.5/double(2) = 1.25
  - 'a'+1 means int{'a'} + 1
- type(value): converts "value" into "type".
- type{value}: converts "value" into "type" preventing narrowing.

Computations and functions in C++

int and double conversion

- If an operator has an operand of type double, then C++ uses float arithmetics, otherwise int.
  - 5/2 is 2 (not 2.5)
  - 2.5/2 means 2.5/double(2) = 1.25
  - 'a'+1 means int{'a'} + 1
- type(value): converts "value" into "type".
- type{value}: converts "value" into "type" preventing narrowing.

Examples

```
double d = 2.5;
int i = 2;
double d2 = d/i;
```

Computations and functions in C++

int and double conversion

- If an operator has an operand of type double, then C++ uses float arithmetics, otherwise int.
  - 5/2 is 2 (not 2.5)
  - 2.5/2 means 2.5/double(2) = 1.25
  - 'a'+1 means int{'a'} + 1
- type(value): converts "value" into "type".
- type{value}: converts "value" into "type" preventing narrowing.

Examples

```
double d = 2.5;
int i = 2;
double d2 = d/i; // d2 == 1.25
int i2 = d/i; //
```

Computations and functions in C++

int and double conversion

- If an operator has an operand of type double, then C++ uses float arithmetics, otherwise int.
  - 5/2 is 2 (not 2.5)
  - 2.5/2 means 2.5/double(2) = 1.25
  - 'a'+1 means int{'a'} + 1
- type(value): converts "value" into "type".
- type{value}: converts "value" into "type" preventing narrowing.

Examples

```
double d = 2.5;
int i = 2;
double d2 = d/i; // d2 == 1.25
int i2 = d/i; //  i2 == 1
int i3 {d/i};
```

Computations and functions in C++

int and double conversion

- If an operator has an operand of type double, then C++ uses float arithmetics, otherwise int.
  - 5/2 is 2 (not 2.5)
  - 2.5/2 means 2.5/double(2) = 1.25
  - 'a'+1 means int{'a'} + 1
- type(value): converts "value" into "type".
- type{value}: converts "value" into "type" preventing narrowing.

Examples

```
double d = 2.5;
int i = 2;
double d2 = d/i; // d2 == 1.25
int i2 = d/i; //  i2 == 1
int i3 {d/i};  // error: double -> int conversion may narrow
```

Computations and functions in C++

Conversions - Examples

```
double dc;
cin >> dc;
double df = 9/5*dc+32; // beware!
```

How to solve the problem ?

Computations and functions in C++

Conversions - Examples

```
double dc;
cin >> dc;
double df = 9/5*dc+32; // beware!
```

How to solve the problem ?

```
double dc;
cin >> dc;
double df = 9.0/5*dc+32; // much better!
```

Computations and functions in C++

statements

Computations and functions in C++

Selection: if-statement

```
int main()
{
    int a = 0;
    int b = 0;
    cout << "Please enter two integers \n";
    cin >> a >> b;
    if (a<b) // condition. 1st alternative (taken if condition is true):
        cout << "max(" << a << "," << b <<") is " << b <<"\n";
    else  // 2nd alternative (taken if condition is false):
        cout << "max(" << a << "," << b <<") is " << a << "\n";
}
```

Computations and functions in C++

Selection: switch-statement

```
constexpr double cm_per_inch = 2.54; // number of cms in an inch
double length = 1; // length in inches or centimeters
char unit = 'a';
cout<< "Please enter a length followed by a unit (c or i):\n";
cin >> length >> unit;

switch (unit) {
case 'i':
    cout << length << "in == " << cm_per_inch*length << "cm\n";
    break;
case 'c':
    cout << length << "cm == " << length/cm_per_inch << "in\n";
    break;
default:
    cout << "Sorry, I don't know a unit called '" << unit << "'\n";
    break;
}
```

Computations and functions in C++

Selection: switch-statement

- The value on which we switch must be of an integer, char, or enumeration type. In particular, you cannot switch on a string.
- The values in the case labels must be constant expressions. You cannot use a variable in a case label.
- You cannot use the same value for two case labels.
- Don't forget to end each case with a break.
- To select based on a string you have to use an if-statement or a map (later).
- You can use many cases for one action:

```
case ’0’: case ’2’: case ’4’: case ’6’: case ’8’:
    cout << "something";
    break;
```

Computations and functions in C++

Iteration: while-statement

```
// calculate and print a table of squares 0-99
int main()
{
    int i = 0; // start from 0
    while (i<100) {
        cout << i << '\t' << square(i) << '\n';
        ++i; // increment i (that is, i becomes i+1)
    }
}
```

- the code between the curly braces of the while statement is called *block* or *compound statement*

Computations and functions in C++

Iteration: for-statement

```
// calculate and print a table of squares 0-99
int main()
{
    for (int i = 0; i<100; ++i)
        cout << i << '\t' << square(i) << '\n';
}
```

Computations and functions in C++

Functions

Computations and functions in C++

Functions and voids

```
int square(int x) // return the square of x
{
    return x*x;
}
```

- Compilers always check the number and types of argument.
- If we don't want to return a result, we use a *void*:

```
void write_sorry() // take no argument; return no value
{
    cout << "Sorry";
}
```

Computations and functions in C++

Why functions

- Makes the computation logically separate
- Makes the program text clearer (by naming the computation)
- Makes it possible to use the function in more than one place in the program

Computations and functions in C++

Why functions

- Makes the computation logically separate
- Makes the program text clearer (by naming the computation)
- Makes it possible to use the function in more than one place in the program

Function declarations

- C++ makes it possible to seperate a declaration of a function from its real definition, or implementation.

```
int square(int); // declaration of square
double sqrt(double); // declaration of sqrt
```

- The function definition can be elsewhere.
- This distinction is very important in large programs.

Computations and functions in C++

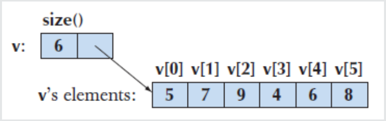
vectors

Computations and functions in C++

Why vectors ?

- Vectors are the simplest of data containers.
- To define a list of typed objects.
- A vector is simply a sequence of elements that you can access by an index.

```
vector<int> v = {5, 7, 9, 4, 6, 8}; // vector of 6 ints
```



- A vector knows its size, the type of its elements, and the initial set of elements.

Computations and functions in C++

vectors

- Vector of a given size without specifying the element values:

```
vector<int> vi(6); // vector of 6 ints initialized to 0
vector<string> vs(4); // vector of 4 strings initialized to ""
```

Computations and functions in C++

vectors

- Vector of a given size without specifying the element values:

```
vector<int> vi(6); // vector of 6 ints initialized to 0
vector<string> vs(4); // vector of 4 strings initialized to ""
```

- Errors:

```
v[2] = "Hume"; // error: trying to assign a string to an int
vi[20000] = 44; // run-time error
```

Computations and functions in C++

Going through a vector: with size

```
vector<int> v = 5, 7, 9, 4, 6, 8;
for (int i=0; i<v.size(); ++i)
    cout << v[i] << '';
```

Going through a vector: range-for-loop

```
vector<int> v = 5, 7, 9, 4, 6, 8;
for (int x : v) // for each x in v
    cout << x << '';
```

Computations and functions in C++

Modify the size: growing

```
vector<double> v; // start off empty; that is, v has no elements
```

Computations and functions in C++

Modify the size: growing

```
vector<double> v; // start off empty; that is, v has no elements

v.push_back(2.7); // add an element with the value 2.7 at end
// ("the back") of v, v now has one element and v[0]==2.7
```

Computations and functions in C++

Modify the size: growing

```
vector<double> v; // start off empty; that is, v has no elements

v.push_back(2.7); // add an element with the value 2.7 at end
// ("the back") of v, v now has one element and v[0]==2.7

v.push_back(5.6); // add an element with the value 5.6 at end of v
// v now has two elements and v[1]==5.6
```

- v.push_back: member function call

Computations and functions in C++

Modify the size: growing

Another example:

```
// read some temperatures into a vector
int main()
{
    vector<double> temps; // temperatures
    for (double temp; cin>>temp; ) // read into temp
           temps.push_back(temp); // put temp into vector
    // . . . do something . . .
}
```

Computations and functions in C++

Exercise 1:

Write a function which prompts a list of temperatures (floats) from a user, and gives back the sum, the mean and the median using the two member functions of vector: size and sort:

```
#include <algorithms>
#include <vector>
//........
sort(myVector.begin(), myVector.end()); // sort myVector
```

Exercise 2:

Write a function which prompts a list of words (with some repetitions), and gives back the words in alphabetical order, eliminating repetition. Ctrl+Z terminates an input stream under Windows and Ctrl+D under Unix.

Computations and functions in C++

Exercise 3:

Create a program to find all the prime numbers between 1 and 100. One way to do this is to write a function that will check if a number is prime (i.e., see if the number can be divided by a prime number smaller than itself) using a vector of primes in order (so that if the vector is called primes, primes[0]==2, primes[1]==3, primes[2]==5, etc.). Then write a loop that goes from 1 to 100, checks each number to see if it is a prime, and stores each prime found in a vector. Write another loop that lists the primes you found. You might check your result by comparing your vector of prime numbers with primes. Consider 2 the first prime.

Errors, Exceptions, Debugging and Testing

Errors, Exceptions, Debugging and Testing

Introduction

Errors, Exceptions, Debugging and Testing

Classification of errors

- *Compile-time errors*: Errors found by the compiler. Always C++ language rules violation, for example:
    - Syntax errors
    - Type safety errors
- *Link-time errors*: Errors found by the linker when it is trying to combine object files into an executable program.
- *Run-time errors*: Errors found by checks in a running program. We can further classify run-time errors as
    - Errors detected by the computer (hardware and/or OS).
    - Errors detected by a library.
    - Errors detected by user code.
- *Logic errors*: Errors found because the program gives wrong results.

Errors, Exceptions, Debugging and Testing

A good program

- Produces the desired results for good input
- rejects bad input and gives error messages
- allowed to terminate after finding an error

How ?

- Organize software (Code design) to minimize errors.
- Eliminate errors we make through debugging and testing.
- Make sure the remaining errors are not serious.

Errors, Exceptions, Debugging and Testing

Compile-time errors

Errors, Exceptions, Debugging and Testing

Find the errors

let's consider the following function, and function calls.

```
int area(int length, int width); // calculate area of a rectangle

int s2 = area(7)
```

Errors, Exceptions, Debugging and Testing

Find the errors

let's consider the following function, and function calls.

```
int area(int length, int width); // calculate area of a rectangle

int s2 = area(7)  // syntax error: ; missing
Int s3 = area(7);
```

Errors, Exceptions, Debugging and Testing

Find the errors

let's consider the following function, and function calls.

```
int area(int length, int width); // calculate area of a rectangle

int s2 = area(7)  // syntax error: ; missing
Int s3 = area(7);  // syntax error: Int is not a type
int s4 = area('7);
```

Errors, Exceptions, Debugging and Testing

Find the errors

let's consider the following function, and function calls.

```
int area(int length, int width); // calculate area of a rectangle

int s2 = area(7)  // syntax error: ; missing
Int s3 = area(7);  // syntax error: Int is not a type
int s4 = area('7);  // syntax error: non-terminated character
int x0 = arena(7);
```

Errors, Exceptions, Debugging and Testing

Find the errors

let's consider the following function, and function calls.

```
int area(int length, int width); // calculate area of a rectangle

int s2 = area(7)  // syntax error: ; missing
Int s3 = area(7);  // syntax error: Int is not a type
int s4 = area('7);  // syntax error: non-terminated character
int x0 = arena(7);   // type error: undeclared function
int x1 = area(7);
```

Find the errors

let's consider the following function, and function calls.

```
int area(int length, int width); // calculate area of a rectangle

int s2 = area(7)  // syntax error: ; missing
Int s3 = area(7);  // syntax error: Int is not a type
int s4 = area('7);  // syntax error: non-terminated character
int x0 = arena(7);   // type error: undeclared function
int x1 = area(7);   // type error: wrong number of arguments
int x2 = area("seven",2);
```

Errors, Exceptions, Debugging and Testing

Find the errors

let's consider the following function, and function calls.

```
int area(int length, int width); // calculate area of a rectangle

int s2 = area(7)  // syntax error: ; missing
Int s3 = area(7);  // syntax error: Int is not a type
int s4 = area('7);  // syntax error: non-terminated character
int x0 = arena(7);   // type error: undeclared function
int x1 = area(7);   // type error: wrong number of arguments
int x2 = area("seven",2);   // type error: 1st argument has a wrong type
int x4 = area(10,-7);
```

Errors, Exceptions, Debugging and Testing

Find the errors

let's consider the following function, and function calls.

```
int area(int length, int width); // calculate area of a rectangle

int s2 = area(7)  // syntax error: ; missing
Int s3 = area(7);  // syntax error: Int is not a type
int s4 = area('7);  // syntax error: non-terminated character
int x0 = arena(7);   // type error: undeclared function
int x1 = area(7);   // type error: wrong number of arguments
int x2 = area("seven",2);   // type error: 1st argument has a wrong type
int x4 = area(10,-7);   // OK: but rectangles do not have a negative width
int x5 = area(10.7,9.3);
```

Errors, Exceptions, Debugging and Testing

Find the errors

let's consider the following function, and function calls.

```
int area(int length, int width); // calculate area of a rectangle

int s2 = area(7)  // syntax error: ; missing
Int s3 = area(7);  // syntax error: Int is not a type
int s4 = area('7);  // syntax error: non-terminated character
int x0 = arena(7);   // type error: undeclared function
int x1 = area(7);   // type error: wrong number of arguments
int x2 = area("seven",2);   // type error: 1st argument has a wrong type
int x4 = area(10,-7);   // OK: but rectangles do not have a negative width
int x5 = area(10.7,9.3);  // OK: but calls area(10,9)
char x6 = area(100,9999);
```

Errors, Exceptions, Debugging and Testing

Find the errors

let's consider the following function, and function calls.

```
int area(int length, int width); // calculate area of a rectangle

int s2 = area(7)  // syntax error: ; missing
Int s3 = area(7);  // syntax error: Int is not a type
int s4 = area('7);  // syntax error: non-terminated character
int x0 = arena(7);   // type error: undeclared function
int x1 = area(7);   // type error: wrong number of arguments
int x2 = area("seven",2);   // type error: 1st argument has a wrong type
int x4 = area(10,-7);   // OK: but rectangles do not have a negative width
int x5 = area(10.7,9.3);  // OK: but calls area(10,9)
char x6 = area(100,9999);  // OK: but truncates the result
```

Errors, Exceptions, Debugging and Testing

Link-time errors

Errors, Exceptions, Debugging and Testing

Linking errors

- A program consists of several separately compiled parts: *translation units*
- Every function must have the same type in every translation unit.
- Every function is defined only once: Ensured by header files (later).

Linking errors: an example

```
double area(double x, double y) { /* . . . */ }
int area(int x, int y, char unit) { /* . . . */ }
int area(int length, int width);
int main()
{
     int x = area(2,3); //Link time error
}
```

Errors, Exceptions, Debugging and Testing

Run-time errors

Errors, Exceptions, Debugging and Testing

An example

```
int area(int length, int width) // calculate area of a rectangle
{
    return length * width;
}
int framed_area(int x, int y) // calculate area within frame
{
    return area(x-2,y-2);
}
int main()
{
     int x = -1;
     int y = 2;
     int z = 4;
     // . . .
     int area1 = area(1,x);
     int area2 = framed_area(y,z);
     double ratio = double(area1)/area2;
}
```

Errors, Exceptions, Debugging and Testing

Exceptions

Errors, Exceptions, Debugging and Testing

Definition

- C++ mechanism to help deal with errors.
- Seperation between error detection, and error handling.
- Ensure that a detected error can't be ignored.

Mechansim

- If a function finds an error it cannot handle: it does not *return*, it *throws* an *exception*.
- A direct (or indirect) caller can *catch* the exception.
- A function expresses interest in handling error with a *try-block*.

Errors, Exceptions, Debugging and Testing

Bad argument error

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

class Bad_area  ; // a type specifically for reporting errors from area()

int area(int length, int width)
{
    if (length<=0 || width<=0) throw Bad_area();
    return length*width;
}
```

Bad argument error

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

class Bad_area  ; // a type specifically for reporting errors from area()

int area(int length, int width)
{
    if (length<=0 || width<=0) throw Bad_area();
    return length*width;
}

int main()
{
    try {
        int area1 = area(-1,2);
    } catch (Bad_area)
        cout << "Oops! bad arguments to area()";
    }
    return 0;
}
```

Errors, Exceptions, Debugging and Testing

Range errors

```
#include <iostream>
#include <vector>

using namespace std;

int main()
try {
    vector<int> v; // a vector of ints
    for (int x; cin>>x; )
        v.push_back(x); // set values
    for (int i = 0; i<=v.size(); ++i) // print values
        cout << "v[" << i <<"] == " << v.at(i) << '';
    } catch (out_of_range) {
        cerr << "Oops! Range error";
        cout << "waw";
        return 1;
    } catch (...) { // catch all other exceptions
        cerr << "Exception: something went wrong";
        return 2;
    }
}
```

Errors, Exceptions, Debugging and Testing

Bad input

```
#include <iostream>

using namespace std;

class Bad_input { };

double some_function()
{
    double d = 0;
    cin >> d;
    if (!cin) throw Bad_input();
        // do something useful
}
```

Errors, Exceptions, Debugging and Testing

Some standard library types of exceptions

- *out_of_range*: when a container tests if you are accessing beyond its size.
- *runtime_error*: Holds a string that can be used by an error handler :
- The two types are different, but come from a common base (supertype): *exception*.

Errors, Exceptions, Debugging and Testing

Some standard library types of exceptions

- *out_of_range*: when a container tests if you are accessing beyond its size.
- *runtime_error*: Holds a string that can be used by an error handler :
- The two types are different, but come from a common base (supertype): *exception*.

Runtime Error

```
int myErrorFunc(int a) {
    throw runtime_error("I found a mistake !");
}
int main()
try {
    int i = myErrorFunc(3);
    return 0; // 0 indicates success
}
catch (runtime_error& e) {
    cerr << "runtime error: " << e.what() << '\n';
    return 1; // 1 indicates failure
}
```

Errors, Exceptions, Debugging and Testing

handle all exceptions

```
class unknown_exc { };
int myErrorFunc(int i) {
    if (i == 0) {
        throw unknown_exc();
    } else {
        throw runtime_error("I found a mistake !");
    }
}

int main()
try {
    int i = 0;
    cout << "Enter 0 for unknown exception, 1 for runtime";
    cin >> i;
    int j = myErrorFunc(i);
    return 0; // 0 indicates success
} catch (exception& e) {
    cerr << "error: " << e.what() << '';
    return 1; // 1 indicates failure
} catch (...) {
    cerr << "Oops: unknown exception!\n";
    return 2; // 2 indicates failure
}
```

Errors, Exceptions, Debugging and Testing

Debugging

Errors, Exceptions, Debugging and Testing

Before debugging

- Report all possible errors.
- Comment your code well: name, purpose, who wrote the code, design idea...
- Meaningful variable names.
- Break code into small functions.
- external libraries rather than own code..
- Pre-conditions: always add tests on inputs: logical, numerical, etc..
- Post-conditons: alway add tests on intermediate and final outputs: logical, numerical, etc..

Debugging with Visual Studio Code

Errors, Exceptions, Debugging and Testing

Testing

Errors, Exceptions, Debugging and Testing

Other errors: testing

- Testing: executing a program with systematically selected set of inputs and comparing the results to what is expected.
- Realistic programs can require millions of test cases.
- Unitary tests: programs that test every unit of the program.
    - Unitary testing for program crashes: test all possible inputs, even wrong input.
    - Unitary testing for logical errors, when we know the expected output for an input.

```
int main()
try {
    int result = testfunc(1);
    if (result==1) {
        cout << "test1: OK";
    } else {
        cout << "test1: NOK";
    }
} catch (someexception) {
    cout << "test1: NOK";
}
```

C++ functions and technicalities

C++ functions and technicalities

Declarations and definitions

C++ functions and technicalities

Definition

- A *declaration* is a statement that introduces a name into a scope
- Every name must be declared: explicitely, or in an #include statement.
- A declaration is just an interface of how to use a name: function, class, variable..
- A *definition* specifies exactly what a name refers to, specifies a function's body, allocates memory if needed...
- You can't define something twice.

```
double sqrt(double); // declaration
double sqrt(double d) { /* . . . */ } // definition
double sqrt(double); // another declaration of sqrt
double sqrt(double); // yet another declaration of sqrt
int sqrt(double); // error: inconsistent declarations of sqrt
```

C++ functions and technicalities

Definition

- A *declaration* is a statement that introduces a name into a scope
- Every name must be declared: explicitely, or in an #include statement.
- A declaration is just an interface of how to use a name: function, class, variable..
- A *definition* specifies exactly what a name refers to, specifies a function's body, allocates memory if needed...
- You can't define something twice.

```
double sqrt(double); // declaration
double sqrt(double d) { /* . . . */ } // definition
double sqrt(double); // another declaration of sqrt
double sqrt(double); // yet another declaration of sqrt
int sqrt(double); // error: inconsistent declarations of sqrt
```

- Two functions with the same signature can't have two different output types.

C++ functions and technicalities

Why declarations and definitions

- Reflects the fundamental distinction between an interface and an implementation.
- Allows to separate a program into many parts that can be compiled separately.
- Consistency: every name is known in every part of the seperate compiled parts.

C++ functions and technicalities

Example

- How to make this work ?

```
double primary()
{
    expression();
}
double term()
{
    primary();
}
double expression()
{
    term();
}
```

C++ functions and technicalities

Example

- Use a *forward* declaration, before definition.

```
double expression(); // just a declaration, not a definition
double primary()
{
    expression();
}
double term()
{
    primary();
}
double expression()
{
    term();
}
```

C++ functions and technicalities

Kinds of declarations

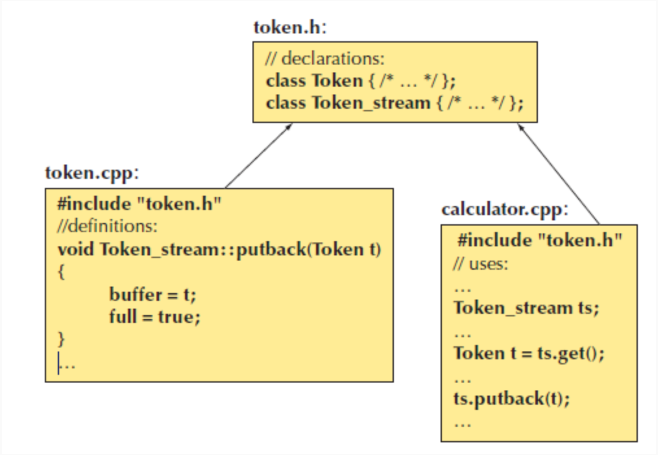
- Variables
- Constants
- Functions
- Namespaces
- Types: Classes and enumerations
- Templates

C++ functions and technicalities

Headers

- A *header* is a collection of declarations, defined in a file, so a header is also called a *header file*.
- Main reason: organization of the code.
  - To separate block of codes in meaningful code units.
  - the *include* statement is part of "*preprocessing*".

C++ functions and technicalities



```
// declarations:
class Token { /* ... */ };
class Token_stream { /* ... */ };
```

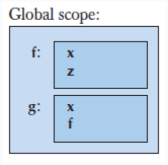
```
#include "token.h"
//definitions:
void Token_stream::putback(Token t)
{
        buffer = t;
        full = true;
}
...
```

```
#include "token.h"
// uses:
...
Token_stream ts;
...
Token t = ts.get();
...
ts.putback(t);
...
```

C++ functions and technicalities

Scope

- A *scope* is a region of program text.
- A name is valid within the scope, from the time it is declared, until the end of the scope.
- Purpose: to keep names local, and not to interfere with other names.



Example

```
void f(int x) // f is global; x is local to f
{
    int z = x+7; // z is local
}

int g(int x) // g is global; x is local to g
{
    int f = x+2; // f is local
    return 2*f;
}
```

C++ functions and technicalities

Scope

- Kinds of scopes to control the use of names:
    - The global scope: outside any other scope
    - The namespace scope: a named scope.
    - The class scope: the area of text within a class.
    - The local scope: between { and } of a block of code.
    - A statement score: for example, in a *for* statement.

C++ functions and technicalities

Functions: call and return

C++ functions and technicalities

Declaring arguments types

```
double fct(int a, double d); // declaration of fct (no body)
```

C++ functions and technicalities

Declaring arguments types

```
double fct(int a, double d); // declaration of fct (no body)

double fct(int, double); // declaration of fct (no naming)
```

C++ functions and technicalities

Declaring arguments types

```
double fct(int a, double d); // declaration of fct (no body)

double fct(int, double); // declaration of fct (no naming)

double fct(int a, double d) { return a*d; } // definition of fct
```

C++ functions and technicalities

Declaring arguments types

```
double fct(int a, double d); // declaration of fct (no body)

double fct(int, double); // declaration of fct (no naming)

double fct(int a, double d) { return a*d; } // definition of fct

int f_no_arg(); // function with no argument
```

C++ functions and technicalities

Declaring arguments types

```
double fct(int a, double d); // declaration of fct (no body)

double fct(int, double); // declaration of fct (no naming)

double fct(int a, double d) { return a*d; } // definition of fct

int f_no_arg(); // function with no argument

void f_no_return(int a); // function with no return
```

C++ functions and technicalities

Declaring return types

- A function declared to return a value must return a value.

```
double my_abs(int x)
{
    if (x < 0)
        return -x;
    else if (x > 0)
        return x;
} // error: no value returned if x is 0
```

- we can use return without a value, in a *void* function:

```
void print_until_s(vector<string> v, string quit)
{
    for(string s : v) {
        if (s==quit) return;
        cout << s << '';
    }
}
```

C++ functions and technicalities

Pass-by-value

- Passing an argument to a function by giving the function a copy of the value you use as the argument

```
// pass-by-value (give the function a copy of the value passed)
int f(int x)
{
    x = x+1; // give the local x a new value
    return x;
}
int main()
{
    int xx = 0;
    cout << f(xx) << '\n'; // write: 1
    cout << xx << '\n'; // write: 0; f() doesn't change xx
}
```

C++ functions and technicalities

Pass-by-const-reference

- Passing by value is efficient when we pass small values: *int*, *char*...
- When a value is large: copying is costly.

```
void print(vector<double> v) // pass-by-value; appropriate?
{
    for (int i = 0; i<v.size(); ++i) {
        cout << v[i];
}
int main() {
    vector<double> vd1(10); // small vector
    vector<double> vd2(1000000); // large vector
    print(vd1);
    print(vd2);

}
```

C++ functions and technicalities

Pass-by-const-reference

- We give the function *print* a memory *address* of the vector.
- Such an address is called: a *reference*.

```
void print(vector<double>& v) // pass-by-const-reference
{
    for (int i = 0; i<v.size(); ++i) {
        cout << v[i];
}
```

- & means reference.
- A *const* reference has the property that we can't accidentally modify the object passed.

C++ functions and technicalities

Pass-by-reference

- When we want a function to modify its arguments.

```
void init(vector<double>& v) // pass-by-reference
{
    for (int i = 0; i<v.size(); ++i) v[i] = i;
}
void g(int x)
{
    vector<double> vd2(1000000); // large vector
    init(vd1);
}
```

C++ functions and technicalities

Pass-by-reference

- A reference allows a user to declare a new name for an object.

```
int i = 7;
int& r = i; // r is a reference to i 7
r = 9;      // i becomes ?
```

C++ functions and technicalities

Pass-by-reference

- A reference allows a user to declare a new name for an object.

```
int i = 7;
int& r = i; // r is a reference to i 7
r = 9;      // i becomes ?  9
i = 10;
cout << r << ' ' << i << '\n'; // write ?:
```

C++ functions and technicalities

Pass-by-reference

- A reference allows a user to declare a new name for an object.

```
int i = 7;
int& r = i; // r is a reference to i 7
r = 9;      // i becomes ?  9
i = 10;
cout << r << ' ' << i << '\n'; // write ?:  10 10
```

- A reference allows to read and write in a memory address.
- Makes the code easier to write.

```
vector< vector<double> > v; // vector of vector of double
/* ... */
double val = v[f(x)][g(y)]; // val is the value of v[f(x)][g(y)]
// How to write ?
double& var = v[f(x)][g(y)]; // var is a reference to v[f(x)][g(y)]
var = var/2+sqrt(var);
```

C++ functions and technicalities

Pass-by-reference VS Pass-by-value

```
void g(int a, int& r, const int& cr)
{
    ++a; // change the local a
    ++r; // change the object referred to by r
    int x = cr; // read the object referred to by cr
}
int main()
{
   int x = 0;
   int y = 0;
   int z = 0;
   g(x,y,z); // output ?
```

C++ functions and technicalities

Pass-by-reference VS Pass-by-value

```
void g(int a, int& r, const int& cr)
{
    ++a; // change the local a
    ++r; // change the object referred to by r
    int x = cr; // read the object referred to by cr
}
int main()
{
   int x = 0;
   int y = 0;
   int z = 0;
   g(x,y,z); // output ?  x==0; y==1; z==0
   g(1,2,3); // output ?
```

C++ functions and technicalities

Pass-by-reference VS Pass-by-value

```
void g(int a, int& r, const int& cr)
{
    ++a; // change the local a
    ++r; // change the object referred to by r
    int x = cr; // read the object referred to by cr
}
int main()
{
   int x = 0;
   int y = 0;
   int z = 0;
   g(x,y,z); // output ?  x==0; y==1; z==0
   g(1,2,3); // output ?  error: reference argument r needs a variable to refer to
   g(1,y,3); // output ?
```

C++ functions and technicalities

Pass-by-reference VS Pass-by-value

```
void g(int a, int& r, const int& cr)
{
    ++a; // change the local a
    ++r; // change the object referred to by r
    int x = cr; // read the object referred to by cr
}
int main()
{
   int x = 0;
   int y = 0;
   int z = 0;
   g(x,y,z); // output ?  x==0; y==1; z==0
   g(1,2,3); // output ?  error: reference argument r needs a variable to refer to
   g(1,y,3); // output ?  OK: since cr is const we can pass a literal
}
```

C++ functions and technicalities

Pass-by-reference VS Pass-by-value

- When to pass by value, reference, or const reference ?
  - Use pass-by-value to pass very small objects.
  - Use pass-by-const-reference to pass large objects that you don't need to modify.
  - Return a result rather than modifying an object through a reference argument.
  - Use pass-by-reference only when you have to.

C++ functions and technicalities

Argument checking

- Consider

```
void f(T x);
T2 y = val;
f(y);
```

- When is it legal ?

C++ functions and technicalities

Argument checking

- Consider

```
void f(T x);
T2 y = val;
f(y);
```

- When is it legal ?
- If we can write, without error:

```
T x = y;
```

C++ functions and technicalities

Evaluation (Execution)

C++ functions and technicalities

Variables: Initialization and Destruction

- When the variable goes out of scope, the variable is destroyed

```
string program_name = "OLIVIER";
vector<string> v; // v is global
void f()
{
    string s; // s is local to f
    while (cin>>s && s!="quit") {
        string stripped; // stri is local to the loop
        string notletters;
        for (int i=0; i<s.size(); ++i) // i has statement scope
            if (isalpha(s[i]))
                stripped += s[i];
            else
                notletters += s[i];
        v.push_back(stripped);
    }
}
```

C++ functions and technicalities

Namespaces

C++ functions and technicalities

Definition

- A way to organize classes, functions, data, and types into an identifiable and named part of a program.

```
namespace TextLib {
    class Text { /* . . . */ };
    class Glyph { /* . . . */ };
    class Line { /* . . . */ };
// . . .
}
```

C++ functions and technicalities

Definition

- A way to organize classes, functions, data, and types into an identifiable and named part of a program.

```
namespace TextLib {
    class Text { /* . . . */ };
    class Glyph { /* . . . */ };
    class Line { /* . . . */ };
// . . .
}
```

- How to call a function from a specific Namespace

```
TextLib::Text(/* . . . */);
TextLib::Line(/* . . . */);
// . . .
using namespace TextLib;

}
```

C++ functions and technicalities

Exercise

C++ functions and technicalities

Exercise

Write a function that takes a *vector<string>* argument and returns a *vector<int>* containing the number of characters in each string. Write another function that finds the longest and the shortest string. Write another function that funds the lexicographically first and last string.

- Declare the functions in a *header* file: *functions.h*
- Write your *header*

  ```
  #ifndef FUNCTIONS_H
  #define FUNCTIONS_H
  /* ... beautiful code ...*/
  #endif
  ```

- Define the functions in a *.cpp* file: *functions.cpp*
- The functions should handle all possible errors.
- Write the *main* in a file *main.cpp*.
- The main should do unitary tests: error handling, sanity checks.

C++ functions and technicalities

Exercise: Functions.h

```
#ifndef FUNCTION_H
#define FUNCTION_H

#include <vector>
#include <iostream>

using namespace std;

namespace Myfunctions
{
    vector<int> stringslengths(const vector<string>&);
    vector<string> Shortest_Longest_String(const  vector<string>&);
    vector<string> First_Last_Strings(const vector<string>&);
}

#endif
```

C++ functions and technicalities

Exercise: Functions.cpp: stringslengths

```
#include "Functions.h"
#include <vector>
#include <algorithm>
using namespace std;

vector<int> Myfunctions::stringslengths(const vector<string>& l_Strings)
{
    if (l_Strings.size() == 0)
        throw runtime_error("Bad input !");
    vector<int> l_nbCharacters;
    for (string s: l_Strings) {
        l_nbCharacters.push_back(s.length());
    }
    return l_nbCharacters;
}
```

C++ functions and technicalities

Exercise: Functions.cpp: Shortest_Longest_String

```
// Assumes l_strings has a nonnull size
vector<string> Myfunctions::Shortest_Longest_String(const
                                        vector<string>& l_Strings)
{
    if (l_Strings.size() == 0)
        throw runtime_error("Bad input !");
    vector<string> res;
    string longestString = "";

    string shortestString = l_Strings[0];

    for (string s: l_Strings) {
        if (s.length() <  shortestString.length())
            shortestString = s;
        if (s.length() >  longestString.length())
            longestString = s;
    }
    res.push_back(shortestString);
    res.push_back(longestString);
    return res;
}
```

C++ functions and technicalities

Exercise: Functions.cpp: First_Last_Strings

```
vector<string> Myfunctions::First_Last_Strings(const
                                  vector<string>& l_Strings)
{
    if (l_Strings.size() == 0)
        throw runtime_error("Bad input !");
    vector<string> loc_l_Strings{l_Strings};
    vector<string> res;
    sort(loc_l_Strings.begin(), loc_l_Strings.end());

    res.push_back(loc_l_Strings[0]);
    res.push_back(loc_l_Strings[loc_l_Strings.size()-1]);

    return res;
}
```

C++ functions and technicalities

Exercise: main.cpp

```
#include "Functions.h"
#include <vector>
#include <iostream>

using namespace std;

int main()

    vector<string> l_test_list1 = {"oneword";
    vector<string> l_test_list2 = {"oneword", "twowords"};
    vector<string> l_test_list3 = {"a", "b", "c", "c", "aa"};
    vector<string> l_test_list4 = {" "};
    vector<string> l_test_list5;
    vector<vector<string>> all_test_lists = {l_test_list1, l_test_list2,
                   l_test_list3, l_test_list4, l_test_list5};

    try {
        for (vector<string> l_test_list: all_test_lists) {
            vector<int> result1 = stringslengths(l_test_list);
            vector<string> result2 = Shortest_Longest_String(l_test_list);
            vector<string> result3 = First_Last_Strings(l_test_list);
```

C++ functions and technicalities

Exercise: main.cpp

```
            cout << " ****************  \{n} input : ";
            for (string s: l_test_list)  cout << s << " ";
            cout << " \n String lengths : ";
            for (int s: result1)  cout << s << " ";
            cout << " \n shortest & longest : ";
            for (string s: result2)  cout << s << " ";
            cout << " \n First & last strings : ";
            for (string s: result3)  cout << s << " ";
            cout << ' \n';
        }
    } catch (runtime_error& e) {
        cerr << "run time error : " << e.what() << '';
    } catch (...) { //catch other exceptions
        cerr << "Something went wrong !";
    }
    return 0;
}
```

Object-Oriented-Programming and Classes

Object-Oriented-Programming and Classes

Monte-Carlo Pricing

Object-Oriented-Programming and Classes

The theory

- We suppose a stock price, having log-normal dynamics

$$dS_t = \mu S_t dt + \sigma S_t dW_t \tag{1}$$

- We suppose a riskless bond, growing at a continuously compounding rate r.
- We suppose a *Vanilla* option, with expiry T and a payoff f, a function of S_T.
- For example, a *Call* option with strike K, has the following payoff:

$$f(S_T) = Max\left(S_T - K, 0\right) \tag{2}$$

- The buyer of a *Call* option has the right, but not the obligation, to buy an agreed quantity of a particular stock (the underlying) from the seller of the option at a certain time T (maturity) for a certain price (K).

Object-Oriented-Programming and Classes

The theory

- The Black & Scholes theory:
    - The price of a vanilla option, with maturity T, and *payoff* function f, is equal to
$$e^{-rT}\mathbb{E}\left[f\left(S_T\right)\right] \tag{3}$$
    - The expectation is taken under the risk-neutral process:
$$dS_t = rS_t dt + \sigma S_t dW_t \tag{4}$$
- We solve equation (4) (pass to log and use Ito):
$$S_T = S_0 \exp\left[\left(r - \frac{\sigma^2}{2}\right)T + \sigma\sqrt{T}\mathcal{N}(0,1)\right] \tag{5}$$
- Finally, the price of a *Vanilla* option is
$$P = e^{-rT}\mathbb{E}\left[f\left(S_0 \exp\left[\left(r - \frac{\sigma^2}{2}\right)T\right] + \sigma\sqrt{T}\mathcal{N}(0,1)\right)\right] \tag{6}$$

Object-Oriented-Programming and Classes

Monte Carlo Pricing

- Approximate this expectation by using the law of large numbers.
- if Y_j are i.i.d, then $\frac{1}{N}\sum_{n=1}^{N} Y_j$ converges to $\mathbb{E}\left[Y_1\right]$.
- Algorithm to price a Vanilla option by Monte Carlo:
    1. Draw a random variable x from the $\mathcal{N}(0,1)$ distribution.
    2. Compute $f\left(S_0 \exp\left[\left(r - \frac{\sigma^2}{2}\right)T\right] + \sigma\sqrt{T}x\right)$.
    3. Repeat many times
    4. Take the average and multiply by e^{-rT}

Object-Oriented-Programming and Classes

From theory to C++

Object-Oriented-Programming and Classes

Building the first pricer: SimpleMonteCarlo.cpp

```cpp
#include "Random1.h"
#include <iostream>
#include <cmath>

using namespace std;

double SimpleMonteCarlo(double T, double Strike, double S0,
                        double Sigma, double r, unsigned long NumberOfPaths)
{
    double movedSpot = S0*exp( (r - 0.5*Sigma*Sigma) * T);
    double runningSum = 0;
    for (unsigned long i=0; i < NumberOfPaths; i++)
    {
        double OneGaussian = GetOneGaussianByBoxMuller();
        double ST = movedSpot*exp( Sigma*OneGaussian*sqrt(T));
        double CallPayoff = (ST - Strike) >0 ? (ST - Strike) : 0;
        runningSum += CallPayoff;
    }
    double avg = runningSum / NumberOfPaths;
    return avg * exp(-r*T);
}
```

Object-Oriented-Programming and Classes

Some criticism

- No adaptability, no reusability:
  - What if we want to price puts, digital options, asian options ?
  - What if we want to price options with two strikes, two maturities?
  - What if we want to add a standard error as output?
  - What if we want to price by aiming for a run time, or precision error?
- Solutions:
  - Solution 1: We copy/paste our program and specialize for every combination of these cases.
  - Solution 2: We use *if*-statements. Everytime we add a new payoff, we have to modify every place of the program where payoffs are used.
- C++ solution: use classes.
  - We *encapsulate* the real-world concept of a pay-off.

Object-Oriented-Programming and Classes

Identifying Classes

Object-Oriented-Programming and Classes

Introduction

- Classes, are the heart of C++.
- Classes are the key to many of the most effective design techniques.
- A class — or a set of classes — is the mechanism through which we represent our concepts in code.
- OOP: how to elegantly express useful ideas as classes.

C++ in Finance

- OOP will encapsulate natural financial concepts: Yield curves, volatility surfaces, derivatives..
- Reusability for an always evolving field.
- We can quickly write prototypes of functionalities, and improve at no cost.
- Separation of interface from implementation: Large quant teams can work together!

Object-Oriented-Programming and Classes

Encapsulation

- We encapsulate the notion of a *Pay-off*. Header file (*Payoff1.h*):

```
#ifndef PAYOFF_H
#define PAYOFF_H
class PayOff
{
public:
    enum OptionType {call, put};
    PayOff(double Strike_, OptionType TheOptionsType_);
    double operator()(double Spot) const;
private:
    double Strike;
    OptionType TheOptionsType;
};
#endif
```

Object-Oriented-Programming and Classes

Enumerations

Object-Oriented-Programming and Classes

scoped enumerations: Definition

- Very simple user-defined type.
- Specification of a list of possible values: *enumerators*

```
enum class Month {
    jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
};
```

- Enumerations are named integer constants
- You can specify a value, or leave it to the compiler.
- To refer to *mar*, we write:

  `Month m = Month::mar;`
- The type *Month* is seperate from type *int*

Object-Oriented-Programming and Classes

plain enumerations: Definition

- Enum names can be used in the scope they are defined in.

```
enum Month {  // note: no \class"
    jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
};

Month m = feb; // OK: feb in scope
Month m2 = Month::feb; // also OK
m = 7; // error: can’t assign an int to a Month
```

- Plain enums are sources of pollution.
- Old-school C++.

Object-Oriented-Programming and Classes

Classes

Object-Oriented-Programming and Classes

Types

- A type is called built-in if the compiler knows without being told by the developer.
- Types that are not built-in are called *user-defined types*
- Why do we define types:
    - Representation: A type “knows” how to represent the data needed in an object.
    - Operations: A type “knows” what operations can be applied to objects.

Object-Oriented-Programming and Classes

Classes and members

- A class is a user-defined type.
- A class has (0 ore more) *members*: data from other types, and functions to handle this data.

```
class X {
public:
    int m; // data member
    int mf(int v) { return v + m; } // function member
};
```

- Accessing a class member:

```
X var; // var is a variable of type X
var.m = 7; // assign to var’s data member m
int x = var.mf(9); // call var’s member function mf()
```

Object-Oriented-Programming and Classes

Interface and implementation

- Interface: part of the class's declaration that its users access directly: *public*
- Implementation: part of the class's declaration that its users access indirectly: *private*
- A general class squeleton

```
class X {
public:
    // public members: the interface to users (accessible by all)
    // functions
    // types
    // data (often best kept private)
private:
    // private members: the implementation (used by members of class only)
    // functions
    // types
    // data
};
```

Object-Oriented-Programming and Classes

Interface and implementation

- A member is declared private by default.
- We can't access a private member, but we can create a public function that uses them, or a public *accessor*:

```
class X {
    int m;
    int private_F(int);
public:
    int public_F(int i) { m=i; return private_F(i); }
    int public_m() { return m; } //Accessor
};
X x;
int y = x.publicF(2);
```

Object-Oriented-Programming and Classes

Interface and implementation

- For classes with only *public* members: use *struct*.

```
struct X {
    int m;
    // . . .
};
```

Object-Oriented-Programming and Classes

Constructor

Object-Oriented-Programming and Classes

Construction

- A class member with the same name as the class, is called a constructor.
- A constructor is used for initialization.
- If a class has a constructor with arguments, it is an (compile-time) error not to initialize.

Object-Oriented-Programming and Classes

Construction

- Consider

```
struct Date {
    int y, m, d; // year, month, day
    Date(int y, int m, int d); // check for valid date and initialize
    void add_day(int n); // increase the Date by n days
};
```

- Ways to call a constructor:

```
Date my_birthday; // error: my_birthday not initialized
Date today 10, 18, 2021; // C++95
Date last (2000,12,31); // OK (old colloquial style)
Date last {2000,12,31}; // OK (old colloquial style)
Date next = {2014,2,14}; // also OK (slightly verbose)
Date christmas76 = Date{1976,12,24}; // also OK (verbose style)
```

Object-Oriented-Programming and Classes

Default construction

- A constructor guarantees that every object of a class is initialized.
- What if we want a default value, like:

```
string s1; // default value: the empty string " "
vector<string> v1; // default value: the empty vector; no elements
```

- Default constructor: gives meaning to the creation of an object without an explicit initializer.

```
class Date {
public: Date(); // default constructor
private:
    int y;
    Month m;
    int d;
};
Date::Date()
:y{2011}, m{Month::jan}, d{1}
{}
```

Object-Oriented-Programming and Classes

Default construction - 2

- We can initialize attributes: *in-class initializer*

```
class Date {
public:
    Date(); // default constructor
    Date(int y, Month m, int d);
    Date(int y); // January 1 of year y
private:
    int y {2001};
    Month m {Month::jan};
    int d {1};
};
```

Object-Oriented-Programming and Classes

Default construction - 3

- We can use a constant for construction.

```
const Date& default_date()
{
    static Date dd {2001,Month::jan,1};
    return dd;
}
```

- We define the default constructor as

```
Date::Date() :  y{default_date().year()},
                   m{default_date().month()},
                   d{default_date().day()}
{ }
```

Object-Oriented-Programming and Classes

Default construction - 4

- Default constructors are used implicitely to initialize containers of objects.

```
vector<Date> birthdays(10); // ten elements with the default Date value Date{}
```

- Without default constructors :

```
vector<Date> birthdays(10,default_date()); // ten default Dates
vector<Date> birthdays2 = {default_date(), default_date(), ..., default_date()};
```

Object-Oriented-Programming and Classes

Defining class members

Object-Oriented-Programming and Classes

Defining class members

- We can define a member outside of the class.

```
// simple Date (some people prefer implementation details last)
class Date {
public:
    Date(int y, int m, int d); // constructor: check for valid date
    void add_day(int n); // increase the Date by n days
private:
    int y, m, d; // year, month, day
};
```

- Define the member functions

```
Date::Date(int yy, int mm, int dd) // constructor
:y{yy}, m{mm}, d{dd} // note: member initializers
{
}
void Date::add_day(int n) { /* . . . */ }
```

Object-Oriented-Programming and Classes

Defining class members

- initializer list avoids default initialization before value assignment.
- The rule that a name must be declared before it is used is relaxed within the limited scope of a class.

Common information to all classes

- How to define shared information and make sure that there is just one copy of the value in the program: *static*

```
class Year { // year in [min:max) range
    static const int min = 1800;
    static const int max = 2200;
public:
    class Invalid { };
    Year(int x) : y{x} { if (x<min || max<=x) throw Invalid{}; }
    int year() { return y; }
private:
    int y;
};
```

Object-Oriented-Programming and Classes

Const member functions

- *const* variables: immutable values.
- When we declare an object *const*, we can't access its members

```
void some_function(const Date& d)
{
    int b = d.day(); // should be OK (why?)
    d.add_day(3); // error
}
```

- The compiler doesn't know which methods do not change the object.
- Solution: we classify operations in a class as modifying or nonmodifying:

Object-Oriented-Programming and Classes

Const member functions - 2

```
class Date {
public:
    int day() const; // const member: can’t modify the object
    Month month() const; // const member: can’t modify the object
    int year() const; // const member: can’t modify the object

    void add_day(int n); // non-const member: can modify the object
    void add_month(int n); // non-const member: can modify the object
    void add_year(int n); // non-const member: can modify the object
private:
    int y; // year
    Month m;
    int d; // day of month
};
```

Object-Oriented-Programming and Classes

Const member functions - 3

It is a (compile-time) error to modify an attribute in a const member:

```
int Date::day() const
{
    ++d; // error: attempt to change object from const member function
    return d;
}
```

Object-Oriented-Programming and Classes

Operator Overloading

Object-Oriented-Programming and Classes

Operators

- You can define almost all C++ operators for class or enumeration operands.

```
enum class Month {
    Jan=1, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec
};
Month operator++(Month& m) // prefix increment operator
{
    m = (m==Month::Dec) ? Month::Jan : Month(int(m)+1); // \wrap around"
    return m;
}
```

- Then we use the operator for our type.

```
Month m::Sep;
++m; // m becomes Oct
```

Object-Oriented-Programming and Classes

Output operator

```
ostream& operator<<(ostream& os, Month m)
{
    return os << int(m);
}
```

Few rules

- You can only overload existing operators, with the same number of operands.
- An overloaded operator must have at least one user-defined type as operand.

Object-Oriented-Programming and Classes

Back to Monte-Carlo Pricing

Object-Oriented-Programming and Classes

Payoff1.h

```
#ifndef PAYOFF_H
#define PAYOFF_H
class PayOff
{
public:
    enum OptionType {call, put};
    PayOff(double Strike_, OptionType TheOptionsType_);
    double operator()(double Spot) const;
private:
    double Strike; // Why private ?
    OptionType TheOptionsType;
};
#endif
```

Object-Oriented-Programming and Classes

Payoff1.cpp

```
#include "PayOff1.h"
#include <iostream>

PayOff::PayOff(double Strike_, OptionType TheOptionsType_):
          Strike(Strike_), TheOptionsType(TheOptionsType_)
{ }
double PayOff::operator ()(double spot) const
{
    switch (TheOptionsType) {
    case call :
        return (spot - Strike) >0 ? (spot - Strike) : 0;
    case put:
        return (Strike -spot) >0 ? (Strike -spot) : 0;
    default:
        throw std::runtime_error("unknown option type found.");
    }
}
```

Object-Oriented-Programming and Classes

Building the second pricer: Pricer1.h

```
#ifndef PRICER1_H
#define PRICER1_H
#include "PayOff1.h"
double SimpleMonteCarlo1(const PayOff& thePayOff,
                         double T,
                         double S0,
                         double Sigma,
                         double r,
                         unsigned long NumberOfPaths);
#endif
```

Object-Oriented-Programming and Classes

Building the second pricer: Pricer1.cpp

```
#include "Random1.h"
#include "Pricer1.h"
#include <iostream>
#include <cmath>

using namespace std;

double SimpleMonteCarlo1(const PayOff& thePayOff, double T, double S0,
                        double Sigma, double r, unsigned long NumberOfPaths)
{
    double movedSpot = S0*exp( (r - 0.5*Sigma*Sigma) * T);
    double runningSum = 0;
    for (unsigned long i=0; i < NumberOfPaths; i++)
    {
        double OneGaussian = GetOneGaussianByBoxMuller();
        double ST = movedSpot*exp( Sigma*OneGaussian*sqrt(T));
        double CallPayoff = thePayOff(ST);
        runningSum += CallPayoff;
    }
    double avg = runningSum / NumberOfPaths;
    return avg * exp(-r*T);
}
```

Object-Oriented-Programming and Classes

main.cpp

```
#include "Pricer1.h"
#include <iostream>
using namespace std;
int main()
{
    double Expiry, Strike, Spot, Vol, r;
    unsigned long NumberOfPaths;

    cout << "\nEnter expiry \n";
    cin >> Expiry;
    cout << "Enter strike";
    cin >> Strike;
    cout << "\nEnter spot\n";
    cin >> Spot;
    cout << "\nEnter vol\n";
    cin >> Vol;
    cout << "\nr\n";
    cin >> r;
    cout << "\nNumber of paths\n";
    cin >> NumberOfPaths;
```

Object-Oriented-Programming and Classes

main.cpp

```
    PayOff callPayOff(Strike, PayOff::call);
    PayOff putPayOff(Strike, PayOff::put);
    double resultCall = SimpleMonteCarlo1(callPayOff, Expiry, Spot, Vol, r,
                                          NumberOfPaths);
    double resultPut = SimpleMonteCarlo1(putPayOff, Expiry, Spot, Vol, r,
                                         NumberOfPaths);
    cout <<"the prices are "
         << resultCall << " for the call and "
         << resultPut << " for the put\n";
    double tmp;
    cin >> tmp;
    return 0;

}
```

Object-Oriented-Programming and Classes

Class exercise

- Create a class *PayOffDoubleStrike* for payoffs depending on two strikes.
- The class should have the same structure as *Payoff*.
- The class has two data members: *strike1* and *strike2*.
- The option types are: *doubleDigital* and *Strangle*.
- Double digital options:
    - The *doubleDigital* option pays 1 if the price is between the two strikes, 0 otherwise.

Object-Oriented-Programming and Classes

Class exercise

- Create a class *PayOffDoubleStrike* for payoffs depending on two strikes.
- The class should have the same structure as *Payoff*.
- The class has two data members: *strike1* and *strike2*.
- The option types are: *doubleDigital* and *Strangle*.
- Double digital options:
    - The *doubleDigital* option pays 1 if the price is between the two strikes, 0 otherwise.
- Strangle options:
    - The *Strangle* option is the payoff of a call option of strike *strike1* and a put option of strike *strike2*. Use the previous class *PayOff*
- Define a new pricer function for this new type of payoff, called "SimpleMonteCarlo2".

Object-Oriented-Programming and Classes

Solution: PayOffDoubleStrike.h

```
#ifndef PAYOFF_DS_H
#define PAYOFF_DS_H
class PayOffDoubleStrike
{
public:
    enum DoubleStrikeOptionType {doubleDigital, Strangle};
    PayOffDoubleStrike(double Strike1, double Strike2,
                       DoubleStrikeOptionType TheOptionsType);
    double operator()(double Spot) const;
private:
    double Strike1;
    double Strike2;
    DoubleStrikeOptionType TheDoubleStrikeOptionsType;
};
#endif
```

Object-Oriented-Programming and Classes

Solution: PayOffDoubleStrike.cpp, constructor

```
#include "payoffDoubleStrike.h"
#include "Payoff1.h"
#include <iostream>

PayOffDoubleStrike::PayOffDoubleStrike(double Strike1_, double Strike2_,
                                   DoubleStrikeOptionType TheOptionsType_):
    Strike1(Strike1_), Strike2(Strike2_),
    TheDoubleStrikeOptionsType(TheOptionsType_)
{
    if (Strike1 > Strike2) {
        double tmp = Strike2;
        Strike2 = Strike1;
        Strike1 = tmp;
    }
}
#endif
```

Object-Oriented-Programming and Classes

Solution: PayOffDoubleStrike.cpp, Operator ()

```
double PayOffDoubleStrike::operator ()(double spot) const
{
    switch (TheDoubleStrikeOptionsType) {
    case doubleDigital:
        if ((spot>=Strike1) && (spot<=Strike2))
            return 1;
        return 0;
    case Strangle:
    {
        PayOff callPayOff(Strike1, PayOff::call);
        PayOff putPayOff(Strike2, PayOff::put);
        return callPayOff(spot) + putPayOff(spot);
    }
    default:
        throw std::runtime_error("unknown option type found.");
    }
}
```

Object-Oriented-Programming and Classes

Solution: Pricer.h

```
#ifndef PRICER1_H
#define PRICER1_H
#include "Payoff1.h"
#include "PayoffDoubleStrike.h"

double SimpleMonteCarlo2(const PayOffDoubleStrike& thePayOff,
                        double T,
                        double S0,
                        double Sigma,
                        double r,
                        unsigned long NumberOfPaths);
#endif
```

Object-Oriented-Programming and Classes

Solution: Pricer.cpp

```
double SimpleMonteCarlo2(const PayOffDoubleStrike& thePayOff,
                        double T,
                        double S0,
                        double Sigma,
                        double r,
                        unsigned long NumberOfPaths)
{
    double movedSpot = S0*exp( (r - 0.5*Sigma*Sigma) * T);
    double runningSum = 0;
    for (unsigned long i=0; i < NumberOfPaths; i++)
    {
        double OneGaussian = GetOneGaussianByBoxMuller();
        double ST = movedSpot*exp( Sigma*OneGaussian*sqrt(T));
        double CallPayoff = thePayOff(ST);
        runningSum += CallPayoff;
    }
    double avg = runningSum / NumberOfPaths;
    return avg * exp(-r*T);
}
```

Object-Oriented-Programming and Classes

Further defects

- How to add a new pay-off: extend the *enum*-statement, and the *switch*-statement.
- Defects:
  - Adding functionality changes the interface.
  - Recompile everywhere where *Payoff1.h* is *incuded*.
  - No elegance, no clarity.
- C++ *open-close* principle :
  - "open": code should always be open for extension (new pay-offs)
  - "close": Existing files are closed for modification. Extension without modifying any existing code.
- C++ solution: Interface design: inheritance and virtual functions.

Object-Oriented-Programming and Classes

Inheritance and virtual functions

Object-Oriented-Programming and Classes

What we try to achieve

- Ideal program: We encapsulate the concepts of the application domain directly in code.
- If we understand the domain, we understand the code.
- Pay-offs:
    - a Pay-off is a general notion of a final financial (cash) flow.
    - the payoff of a Call "is a kind of" pay-off.

Object-Oriented-Programming and Classes

Payoff.h

```
#ifndef PAYOFF_H
#define PAYOFF_H
class PayOff {
public:
    PayOff(){};
    virtual double operator()(double Spot) const=0;
    virtual ~PayOff(){}
private: };
#endif
```

Payoffcall.h

```
#ifndef PAYOFFCALL_H
#define PAYOFFCALL_H
#include "Payoff.h"
class PayOffCall : public PayOff
{
public:
    PayOffCall(double Strike_);
    virtual double operator()(double Spot) const;
    virtual ~PayOffCall(){}
private:
   double Strike;
};
#endif
```

Object-Oriented-Programming and Classes

Payoffcall.cpp

```
#include "Payoffcall.h"
#include <minmax.h>
PayOffCall::PayOffCall(double Strike_) : Strike(Strike_)
{ }

double PayOffCall::operator () (double Spot) const
{
    return max(Spot-Strike,0.0);
}
```

Object-Oriented-Programming and Classes

Base and derived class

- *Derivation / Inheritance*
    - a way to build one class from another so that the new class can be used in place of the original.
    - The derived class (*Call*) *inherits* all of the members of its base (*Payoff*) in addition to its own.
    - A derived class is called a *subclass*.
    - A base class is called a *superclass*.

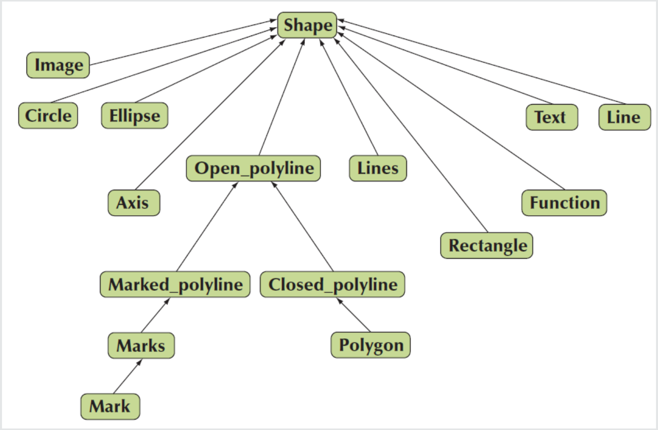
Object-Oriented-Programming and Classes

Base and derived class

- *Virtual functions*:
    - The ability to define a function in a base class and have a function of the same name and type in a derived class.
    - When *Payoff* calls the operator *()* for a payoff that is a *PayOffCall*, it is the implementation in *PayOffCall* that is executed.
    - *runtime Polymorphism*: The function called is determined at run-time based on the type.

Object-Oriented-Programming and Classes

Inheritance diagram



Object-Oriented-Programming and Classes

Deriving classes

```
class PayoffCall: Public Payoff { /* .. */}
```

- *Public*: Public members of *Payoff* are public, and private members are private.

```
class PayoffCall: Payoff { /* .. */}
class PayoffCall: private Payoff { /* .. */}
```

- *Payoff* is a private base for *PayoffCall*: *Payoff*'s public functions are inaccessible.

Object-Oriented-Programming and Classes

Virtual functions

```
virtual int OneVirtualFunc(int someInput) const;
```

- A virtual function must be declared virtual in its class declaration.
- Overriding a virtual function: use exactly the same name and type as in the base class.

  ```
  int OneVirtualFunc(double someInput) const; // ERROR
                                               //(compiler warning)
  int OneVirtualFunc(int someInput); // ERROR (compiler warning)
  ```
- We can explicitly declare that a function is meant to override.

  ```
  int OneVirtualFunc(int someInput) const override;
  ```

Object-Oriented-Programming and Classes

Virtual functions overriding: technical example

```
struct B {
    virtual void f() const { cout << "B::f "; }
    void g() const { cout << "B::g "; } // not virtual
};
struct D : B {
    void f() const { cout << "D::f "; } // overrides B::f
    void g() { cout << "D::g "; }
};
struct DD : D {
    void f() { cout << "DD::f "; } // doesn’t override D::f
                                                   (not const)
    void g() const { cout << "DD::g "; }
};
```

Object-Oriented-Programming and Classes

Virtual functions overriding: technical example

```
void call(const B& b) // a D is a kind of B, so call() can accept a D
// a DD is a kind of D and a D is a kind of B, so call() can accept a DD
{
    b.f();
    b.g();
}
int main() {
    B b;
    D d;
    DD dd;

    call(b); //output:
```

Object-Oriented-Programming and Classes

Virtual functions overriding: technical example

```
void call(const B& b) // a D is a kind of B, so call() can accept a D
// a DD is a kind of D and a D is a kind of B, so call() can accept a DD
{
    b.f();
    b.g();
}
int main() {
    B b;
    D d;
    DD dd;

    call(b); //output:  B::f B::g
    call(d); //output:
```

Object-Oriented-Programming and Classes

Virtual functions overriding: technical example

```
void call(const B& b) // a D is a kind of B, so call() can accept a D
// a DD is a kind of D and a D is a kind of B, so call() can accept a DD
{
    b.f();
    b.g();
}
int main() {
    B b;
    D d;
    DD dd;

    call(b); //output:  B::f B::g
    call(d); //output:  D::f B::g
    call(dd); //output:
```

Object-Oriented-Programming and Classes

Virtual functions overriding: technical example

```
void call(const B& b) // a D is a kind of B, so call() can accept a D
// a DD is a kind of D and a D is a kind of B, so call() can accept a DD
{
    b.f();
    b.g();
}
int main() {
    B b;
    D d;
    DD dd;

    call(b); //output:  B::f B::g
    call(d); //output:  D::f B::g
    call(dd); //output:  D::f B::g

    b.f(); //output:
```

Object-Oriented-Programming and Classes

Virtual functions overriding: technical example

```
void call(const B& b) // a D is a kind of B, so call() can accept a D
// a DD is a kind of D and a D is a kind of B, so call() can accept a DD
{
    b.f();
    b.g();
}
int main() {
    B b;
    D d;
    DD dd;

    call(b); //output:  B::f B::g
    call(d); //output:  D::f B::g
    call(dd); //output:  D::f B::g

    b.f(); //output:  B::f
    b.g(); //output:
```

Object-Oriented-Programming and Classes

Virtual functions overriding: technical example

```
void call(const B& b) // a D is a kind of B, so call() can accept a D
// a DD is a kind of D and a D is a kind of B, so call() can accept a DD
{
    b.f();
    b.g();
}
int main() {
    B b;
    D d;
    DD dd;

    call(b); //output:  B::f B::g
    call(d); //output:  D::f B::g
    call(dd); //output:  D::f B::g

    b.f(); //output:  B::f
    b.g(); //output:  B::g
    d.f(); //output:
```

Object-Oriented-Programming and Classes

Virtual functions overriding: technical example

```
void call(const B& b) // a D is a kind of B, so call() can accept a D
// a DD is a kind of D and a D is a kind of B, so call() can accept a DD
{
    b.f();
    b.g();
}
int main() {
    B b;
    D d;
    DD dd;

    call(b); //output:  B::f B::g
    call(d); //output:  D::f B::g
    call(dd); //output:  D::f B::g

    b.f(); //output:  B::f
    b.g(); //output:  B::g
    d.f(); //output:  D::f

    d.g(); //output:
```

Object-Oriented-Programming and Classes

Virtual functions overriding: technical example

```
void call(const B& b) // a D is a kind of B, so call() can accept a D
// a DD is a kind of D and a D is a kind of B, so call() can accept a DD
{
    b.f();
    b.g();
}
int main() {
    B b;
    D d;
    DD dd;

    call(b); //output:  B::f B::g
    call(d); //output:  D::f B::g
    call(dd); //output:  D::f B::g

    b.f(); //output:  B::f
    b.g(); //output:  B::g
    d.f(); //output:  D::f

    d.g(); //output:  D::g
    dd.f(); //output:
```

Object-Oriented-Programming and Classes

Virtual functions overriding: technical example

```
void call(const B& b) // a D is a kind of B, so call() can accept a D
// a DD is a kind of D and a D is a kind of B, so call() can accept a DD
{
    b.f();
    b.g();
}
int main() {
    B b;
    D d;
    DD dd;

    call(b); //output:  B::f B::g
    call(d); //output:  D::f B::g
    call(dd); //output:  D::f B::g

    b.f(); //output:  B::f
    b.g(); //output:  B::g
    d.f(); //output:  D::f

    d.g(); //output:  D::g
    dd.f(); //output:  DD:f
    dd.g(); //output:
```

Object-Oriented-Programming and Classes

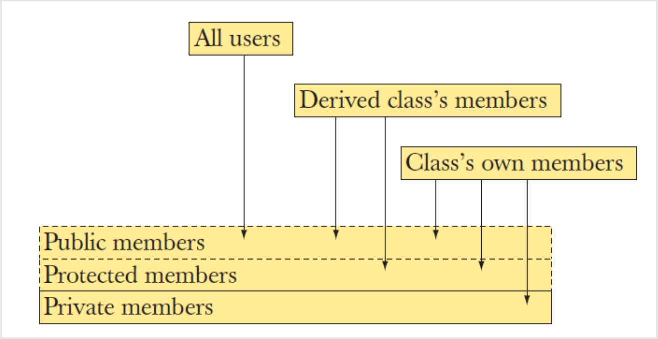
Virtual functions overriding: technical example

```
void call(const B& b) // a D is a kind of B, so call() can accept a D
// a DD is a kind of D and a D is a kind of B, so call() can accept a DD
{
    b.f();
    b.g();
}
int main() {
    B b;
    D d;
    DD dd;

    call(b); //output:  B::f B::g
    call(d); //output:  D::f B::g
    call(dd); //output:  D::f B::g

    b.f(); //output:  B::f
    b.g(); //output:  B::g
    d.f(); //output:  D::f

    d.g(); //output:  D::g
    dd.f(); //output:  DD:f
    dd.g(); //output:  DD:g
}
```

Object-Oriented-Programming and Classes

Inheritance and access

- Three access types for members
  - *private*: can be used only by members of the class in which it is declared.
  - *public*: can be used by all functions.
  - *protected*: can be used only by members of the class in which it is declared and members of classes derived from that.



Object-Oriented-Programming and Classes

Pure Virtual functions and abstract classes

```
virtual void f() =0; // pure virtual function
```

- An abstract class is a class that can be used only as a base class.
- $= 0$ notation says that the virtual function *f()* is "pure".
- If a class has pure virtual functions, we cannot create an object of the class.
- A *pure virtual function* must be overridden in some derived class.
- When inheriting, unless all pure virtual functions are overridden, the resulting class is still abstract.

Object-Oriented-Programming and Classes

Back to Pricing, SimpleMonteCarlo2: Payoff2.h

```
#ifndef PAYOFF2_H
#define PAYOFF2_H
class PayOff {
public:
    PayOff(){};
    virtual double operator()(double Spot) const=0;
private:
};

#endif
```

SimpleMonteCarlo2: PayoffCall2.h

```
#ifndef PAYOFFCALL2_H
#define PAYOFFCALL2_H
#include "Payoff2.h"
class PayOffCall : public PayOff {
public:
    PayOffCall(double Strike);
    virtual double operator()(double Spot) const;
private:
    double Strike;
};

#endif
```

Object-Oriented-Programming and Classes

Back to Pricing, SimpleMonteCarlo2: PayoffCall2.cpp

```
#include "PayoffCall2.h"

PayOffCall::PayOffCall(double Strike_) : Strike(Strike_)
{ }

double PayOffCall::operator () (double Spot) const
{
    return (Spot - Strike) >0 ? (Spot - Strike) : 0;
}
```

Object-Oriented-Programming and Classes

Some criticism

- Our *polymorphic* class *Payoff* determines the pay-off of a Vanilla option.
- OOP goal is to *encapsulate* real-world concepts: Options.
- An option= An expiry + A Pay-off.
- Solutions:
    - Copy all the pay-off and add a data member "Expiry" ?

Object-Oriented-Programming and Classes

Some criticism

- Our *polymorphic* class *Payoff* determines the pay-off of a Vanilla option.
- OOP goal is to *encapsulate* real-world concepts: Options.
- An option= An expiry + A Pay-off.
- Solutions:
  - Copy all the pay-off and add a data member "Expiry" ? no code re-use and very time consuming to add pay-offs (new option type for each pay-off).
  - Create a *VanillaOption* class which has a *Payoff* and a *double* as data members ?

Object-Oriented-Programming and Classes

Some criticism

- Our *polymorphic* class *Payoff* determines the pay-off of a Vanilla option.
- OOP goal is to *encapsulate* real-world concepts: Options.
- An option= An expiry + A Pay-off.
- Solutions:
  - Copy all the pay-off and add a data member "Expiry" ? no code re-use and very time consuming to add pay-offs (new option type for each pay-off).
  - Create a *VanillaOption* class which has a *Payoff* and a *double* as data members ? *Payoff* is abstract so no possible instantiation of an object.
  - Make the class non-abstract, by defining *operator()* ?

Object-Oriented-Programming and Classes

Some criticism

- Our *polymorphic* class *Payoff* determines the pay-off of a Vanilla option.
- OOP goal is to *encapsulate* real-world concepts: Options.
- An option= An expiry + A Pay-off.
- Solutions:
  - Copy all the pay-off and add a data member "Expiry" ? no code re-use and very time consuming to add pay-offs (new option type for each pay-off).
  - Create a *VanillaOption* class which has a *Payoff* and a *double* as data members ? *Payoff* is abstract so no possible instantiation of an object.
  - Make the class non-abstract, by defining *operator()* ? Data member will be copied and truncated into a base class..
- What we want: a *Vanilla Option* object to be able to contain an object from an unknown class.
- Our problem: The unknown class object will not be constant !

Object-Oriented-Programming and Classes

A solution

- Refer to extra data outside the object using pointers or references.

```
class VanillaOption
{
public:
    VanillaOption(PayOff& ThePayOff_, double Expiry_);
    double GetExpiry() const;
    double OptionPayOff(double Spot) const;
private:
    double Expiry;
    PayOff& ThePayOff;
};
```

Object-Oriented-Programming and Classes

Some issues

- The *VanillaOption* stores a reference to a *Payoff* object defined outside the class.
- The *VanillaOption* will not exist as an independent object.
- Major crashes if:
    - If the *PayOff* object is "deleted" before the option ceased to exist: call of non-existent object methods.
    - If the *VanillaOption* is "deleted" before the *PayOff* object: memory leak.
- Solution:

Object-Oriented-Programming and Classes

Some issues

- The *VanillaOption* stores a reference to a *Payoff* object defined outside the class.
- The *VanillaOption* will not exist as an independent object.
- Major crashes if:
  - If the *PayOff* object is "deleted" before the option ceased to exist: call of non-existent object methods.
  - If the *VanillaOption* is "deleted" before the *PayOff* object: memory leak.
- Solution: the Payoff object must return a copy of itself for the vanillaOption to use.

Object-Oriented-Programming and Classes

Solution: virtual construction, virtual copy construction

```
class PayOff
{
public:
    PayOff(){};
    virtual double operator()(double Spot) const=0;
    virtual ~PayOff(){}
    virtual PayOff* clone() const=0;
private:
};
```

Object-Oriented-Programming and Classes

C++ and OOP features: Pointers

Object-Oriented-Programming and Classes

Why deal with memory ?

- Implement programs that deal with data in an optimized way.
- Implement containers (Matrix operations).
- Understand how a program maps onto memory gives a grasp of higher level topics: Data structures and algorithms.

Our own class vector

- How to represent a set of elements where the number of elements can vary ?

```
vector<double> age(2); // a vector with 2 elements of type double
age[0]=33.1;
age[1]=22.0;
```

Object-Oriented-Programming and Classes

First attempt

- We need the address of the first element and the length of the vector.
- A data member that points to the set of elements so that we can make it point to a different set of elements when we need more space.

```
// a very simplified vector of doubles
class vector {
    int sz; // the size
    double* elem; // pointer to the first element (type double)
public:
    vector(int s); // constructor: allocate s doubles
                   // let elem point to them, store s in sz
    int size() const { return sz; } // the current size
};
```

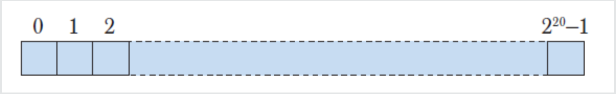
Object-Oriented-Programming and Classes

Pointers

- Pointers are closely related to the notion of "array", and is key to C++'s notion of "memory".
- In C++, a data type that can hold an address is called a pointer and is syntactically distinguished by the suffix *.
- *double** means **pointer** to *double*

Memory, addresses, and pointers

- Computer memory: Sequence of bytes, numbered from 0 to N.
- Every one of these numbers indicates a location in memory: and *address*.
- One mega-byte of memory: 2^{20} bytes

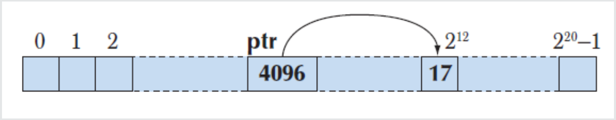


Object-Oriented-Programming and Classes

Memory, addresses, and pointers

- An object that holds an address value is called a *pointer*.
- The type needed to hold a pointer to an int is called a "int pointer".

```
int var = 17;
int* ptr = &var;
```



- *&* is the "address of" operator.

Object-Oriented-Programming and Classes

Memory, addresses, and pointers

```
double e = 2.71828;
double* pd = &e; // pointer to double
```

- To see the value of the object pointed to: use the "**content of**" operator: *.

```
cout << "pd==" << pd << "; contents of pd==" << *pd << "";
```

- * is often called *dereference* operator.
- * can be used as lvalue and rvalue.

```
*pi = 27; // OK: assign 27 to the int pointed to by pi
*pd = 3.14159; // OK: assign 3.14159 to the double pointed to by pd
*pd = *pi; // OK: assign an int (*pi) to a double (*pd)
```

Object-Oriented-Programming and Classes

Memory, addresses, and pointers

- Pointers do not mix: a char* is not int*

  ```
  char* pc = pi; // error: can’t assign an int* to a char*
  pi = pc; // error: can’t assign a char* to an int*
  ```

- Why ?

Object-Oriented-Programming and Classes

Memory, addresses, and pointers

- Pointers do not mix: a char* is not int*

```
char* pc = pi; // error: can’t assign an int* to a char*
pi = pc; // error: can’t assign a char* to an int*
```

- Why ? Data types have different allocated memory !

sizeof operator

- How to know the size of a type ?

```
void sizes(char ch, int i, int* p)
{
    cout << "the size of char is " << sizeof(char) << ' ' << sizeof (ch) << 'n';
    cout << "the size of int is " << sizeof(int) << ' ' << sizeof (i) << 'n';
    cout << "the size of int* is " << sizeof(int*) << ' ' << sizeof (p) << 'n';

}
```

Object-Oriented-Programming and Classes

sizeof operator

- How much memory used by a *vector* ?

```
vector<int> v(1000); // vector with 1000 elements of type int
cout << "the size of vector<int>(1000) is " << sizeof (v) << 'n';
```

Object-Oriented-Programming and Classes

sizeof operator

- How much memory used by a *vector* ?

```
vector<int> v(1000); // vector with 1000 elements of type int
cout << "the size of vector<int>(1000) is " << sizeof (v) << 'n';
```

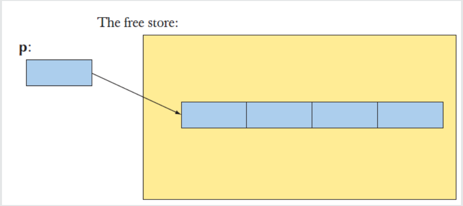
Pointers and Free store

- Four kind of memories set by the compiler
  - *Code storage*
  - *Static storage*: Global variables.
  - Stack storage: Local variables of functions
  - *Free store* or *heap*: Free memory.
- The *heap* is available to every program, through the operator **new**.

Object-Oriented-Programming and Classes

Pointers and Free store

```
double* p = new double[4]; // allocate 4 doubles on the free
                           // store and returns a pointer to us
```



- **new** returns a pointer to the object.
- If **new** created several objects (vector), it returns a pointer to the first object.
- **new** does **not** know how many elements it points to.

Object-Oriented-Programming and Classes

Pointers and Free store

- The **new** operator can allocate individual elements or sequences.

```
double* pd = new double; // allocate one double
double* qd = new double[n]; // allocate an array of n doubles
```

Object-Oriented-Programming and Classes

Pointers and Free store

- The **new** operator can allocate individual elements or sequences.

```
double* pd = new double; // allocate one double
double* qd = new double[n]; // allocate an array of n doubles
```

Access through pointers

- We can use the *subscript* operator **[]**

```
double* p = new double[4]; // allocate 4 doubles on the free store
double x = *p; // read the (first) object pointed to by p
double y = p[2]; // read the 3rd object pointed to by p
```

- **p* means the same as *p[0]*.
- The operators *** and *[]* can also be used for writing.

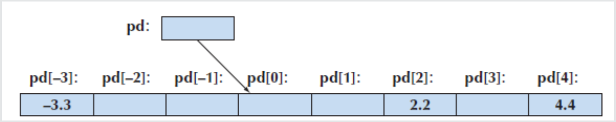
```
*p = 7.7; // write to the (first) object pointed to by p
p[2] = 9.9; // write to the 3rd object pointed to by p
```

Object-Oriented-Programming and Classes

Ranges

- Pointers' major problem: do not know how many elements it points to.
- Danger: Out-of-range access

```
double* pd = new double[3];
pd[4] = 4.4;
pd[-3] = -3.3;
```



- Why do Pointer do not know their size ?
- To give the developer all the tools he needs for memory optimization.
- Quick solutions: STL vectors, "smart pointers".

Object-Oriented-Programming and Classes

Pointer initialization

- ALWAYS initialize !!!!

```
double* p0; // uninitialized: likely trouble
*p0 = 7.0; // Potential bad crash !
double* p1 = new double; // get (allocate) an uninitialized double
double* p2 = new double{5.5}; // get a double initialized to 5.5

double* p3 = new double[5]; // get (allocate) 5 uninitialized doubles
```

Object-Oriented-Programming and Classes

Pointer initialization

- ALWAYS initialize !!!!

```
double* p0; // uninitialized: likely trouble
*p0 = 7.0; // Potential bad crash !
double* p1 = new double; // get (allocate) an uninitialized double
double* p2 = new double{5.5}; // get a double initialized to 5.5

double* p3 = new double[5]; // get (allocate) 5 uninitialized doubles
```

- For built-in types, no need to initialize elements, but possible to do:

```
double* p4 = new double[5] {0,1,2,3,4};

double* p5 = new double[] {0,1,2,3,4};
```

Object-Oriented-Programming and Classes

Pointer initialization

- ALWAYS initialize !!!!

```
double* p0; // uninitialized: likely trouble
*p0 = 7.0; // Potential bad crash !
double* p1 = new double; // get (allocate) an uninitialized double
double* p2 = new double{5.5}; // get a double initialized to 5.5

double* p3 = new double[5]; // get (allocate) 5 uninitialized doubles
```

- For built-in types, no need to initialize elements, but possible to do:

```
double* p4 = new double[5] {0,1,2,3,4};

double* p5 = new double[] {0,1,2,3,4};
```

- For other types
  - If a class **X** has a *default* constructor:

    ```
    X* px1 = new X; // one default-initialized X

    X* px2 = new X[17]; // 17 default-initialized Xs
    ```

  - Otherwise, explicitly initialize !

    ```
    X* px1 = new X; // error: no default constructor
    X* py2 = new X{13}; // OK: initialized to X{13}
    X* py3 = new X[17]; // error: no default constructor

    X* py4 = new X[3] {X{0},X{-5},X{9}};
    ```

Object-Oriented-Programming and Classes

Null pointers

- If (big if) you have no other pointer for initialization, use the null pointer

  ```
  double* p0 = nullptr;
  ```

- To avoid mistakes, you can test if a pointer points to something, before dereferencing:

  ```
  if (p0 != nullptr) ...
  if (p0)..
  ```

Object-Oriented-Programming and Classes

C++ and OOP features: Deallocation and Destruction

Object-Oriented-Programming and Classes

Heap de-allocation

- Return memory to the free store once we are finished using it.
- Avoid memory leakage

```
double* calc(int res_size, int max) // leaks memory
{
    double* p = new double[max];
    double* res = new double[res_size];
    // use p to calculate results to be put in res
    return res;
}
double* r = calc(100,1000);
```

- Each call "**leaks**" the doubles allocated for *p*.

Object-Oriented-Programming and Classes

Heap de-allocation

- We use the operator *delete* to return memory.

```
double* calc(int res_size, int max)
// the caller is responsible for the memory allocated for res
{
    double* p = new double[max];
    double* res = new double[res_size];
    /* use p to calculate results to be put in res */
    delete[] p; // we don’t need that memory anymore: free it
    return res;
}
double* r = calc(100,1000);
/* ... use r ... */
delete[] r; // we don’t need that memory anymore: free it
```

- We can create objects in a function and pass them to the caller.

Object-Oriented-Programming and Classes

Heap de-allocation

- We can create objects in a function and pass them to the caller.
- *delete p* frees memory for an individual object.
- *delete[] p* frees memory for an array of objects.

Garbage collection

- Automatic memory de-allocation.
- Is not cost-free: Not optimal for optimization routines for example.

Object-Oriented-Programming and Classes

vector Class

```
class vector {
    int sz; // the size
    double* elem; // a pointer to the elements
public:
    vector(int s) : //constructor
        sz{s}, // initialize sz
        elem{new double[s]} // initialize elem
    {
        for (int i=0; i<s; ++i) elem[i]=0; // initialize elements
    }
    int size() const { return sz; } // the current size
};
```

- Memory leakage ?

```
void f(int n) {
    vector v(n); // allocate n doubles
    /* ... stuff ... */
}
```

Object-Oriented-Programming and Classes

Destructors

- Automatic memory de-allocation: implicitely called when an object goes out of scope.
- A constructor makes sure an object is properly created and initialized, a destructor makes sure an object is properly cleaned up.

Object-Oriented-Programming and Classes

Destructors

- Automatic memory de-allocation: implicitely called when an object goes out of scope.
- A constructor makes sure an object is properly created and initialized, a destructor makes sure an object is properly cleaned up.

```
class vector {
    int sz; // the size
    double* elem; // a pointer to the elements
public:
    vector(int s) // constructor
        :szs, elemnew double[s] // allocate memory
    {
        for (int i=0; i<s; ++i) elem[i]=0; // initialize elements
    }
    ~vector() // destructor
        { delete[] elem; } // free memory
    /* . . . */ other members

};
```

Object-Oriented-Programming and Classes

Destructors

- Now we can write

```
void f(int n)
{
    double* p = new double[n]; // allocate n doubles
    vector v(n); // the vector allocates n doubles
    // . . . use p and v . . .
    delete[ ] p; // deallocate p’s doubles

} // vector automatically cleans up after v
```

Object-Oriented-Programming and Classes

Destructors

- Now we can write

```
void f(int n)
{
    double* p = new double[n]; // allocate n doubles
    vector v(n); // the vector allocates n doubles
    // . . . use p and v . . .
    delete[ ] p; // deallocate p's doubles

} // vector automatically cleans up after v
```

- If a class member has a destructor, it is called when the object containing it is destroyed.

```
struct Eleve{
    string name;
    vector<string> addresses;
    // . . .
};
void some_fct() {
    Customer Corentin;

}
```

- Automatic destruction of name and addresses: *Compiler generated destructor*

Object-Oriented-Programming and Classes

Virtual destruction

```
class PayOff
{
public:
    PayOff(){};
    virtual double operator()(double Spot) const;
    ~PayOff(){}
};

class DerivedPayOff: public PayOff
{
public:
    DerivedPayOff(){};
    virtual double operator()(double Spot) const;
    ~DerivedPayOff(){}
private:
    double* someMemory;
};

void myFunc(PayOff* PO) {
    ...;
}
```

- Which destructor is called ?

Object-Oriented-Programming and Classes

Virtual destruction

```
class PayOff
{
public:
...
    virtual ~PayOff(){}
};

class DerivedPayOff: public PayOff
{
....
    ~DerivedPayOff(){}
};
```

- If a class has a pure virtual functions then it should have a virtual destructor.
- If a class has a virtual function, and inherited classes can have new memory, it should have a virtual destructor.

Object-Oriented-Programming and Classes

Pointers to class objects

Object-Oriented-Programming and Classes

Pointers to class objects

- The notion of "pointer" is general. we can point to objects from our classes.

```
vector* p = new vector(s); //allocate a vector on free store
delete p; // deallocate
```

- Steps of allocation ?

Object-Oriented-Programming and Classes

Pointers to class objects

- The notion of "pointer" is general. we can point to objects from our classes.

  ```
  vector* p = new vector(s); //allocate a vector on free store
  delete p; // deallocate
  ```

- Steps of allocation ?
  - allocate memory for a *vector*

Object-Oriented-Programming and Classes

Pointers to class objects

- The notion of "pointer" is general. we can point to objects from our classes.

```
vector* p = new vector(s); //allocate a vector on free store
delete p; // deallocate
```

- Steps of allocation ?
    - allocate memory for a *vector*
    - invoke the *vector* constructor to initialize it.

Object-Oriented-Programming and Classes

Pointers to class objects

- The notion of "pointer" is general. we can point to objects from our classes.

  ```
  vector* p = new vector(s); //allocate a vector on free store
  delete p; // deallocate
  ```

- Steps of allocation ?
  - allocate memory for a *vector*
  - invoke the *vector* constructor to initialize it.
  - the constructor allocates memory for the vector's elements, and initialize them.

Object-Oriented-Programming and Classes

Pointers to class objects

- The notion of "pointer" is general. we can point to objects from our classes.

```
vector* p = new vector(s); //allocate a vector on free store
delete p; // deallocate
```

- Steps of allocation ?
  - allocate memory for a *vector*
  - invoke the *vector* constructor to initialize it.
  - the constructor allocates memory for the vector's elements, and initialize them.
- Steps of deallocation

Object-Oriented-Programming and Classes

Pointers to class objects

- The notion of "pointer" is general. we can point to objects from our classes.

  ```
  vector* p = new vector(s); //allocate a vector on free store
  delete p; // deallocate
  ```

- Steps of allocation ?
  - allocate memory for a *vector*
  - invoke the *vector* constructor to initialize it.
  - the constructor allocates memory for the vector's elements, and initialize them.
- Steps of deallocation
  - invokes the vector's destructor.

Object-Oriented-Programming and Classes

Pointers to class objects

- The notion of "pointer" is general. we can point to objects from our classes.

  ```
  vector* p = new vector(s); //allocate a vector on free store
  delete p; // deallocate
  ```

- Steps of allocation ?
  - allocate memory for a *vector*
  - invoke the *vector* constructor to initialize it.
  - the constructor allocates memory for the vector's elements, and initialize them.
- Steps of deallocation
  - invokes the vector's destructor.
  - the destructor invokes first the elements' destructors (if they have).

Object-Oriented-Programming and Classes

Pointers to class objects

- The notion of "pointer" is general. we can point to objects from our classes.

  ```
  vector* p = new vector(s); //allocate a vector on free store
  delete p; // deallocate
  ```

- Steps of allocation ?
  - allocate memory for a *vector*
  - invoke the *vector* constructor to initialize it.
  - the constructor allocates memory for the vector's elements, and initialize them.
- Steps of deallocation
  - invokes the vector's destructor.
  - the destructor invokes first the elements' destructors (if they have).
  - deallocate memory used for vector.
- it works recursively:

  ```
  vector<vector<double>>* p = new vector<vector<double>>(10);
  delete p;
  ```

Object-Oriented-Programming and Classes

Pointers to class objects

- How to access the members of our object of type *vector*.
- all classes support the operator . for accessing members:

  ```
  vector v(4);
  int x = v.size();
  ```

- Similarly, all classes classes support the operator -> to access members from a pointer.

  ```
  vector* p = new vector(4);
  int x = p->size();
  double d = p->get(3);
  ```

Object-Oriented-Programming and Classes

C++ and OOP features: Pointers VS References

Object-Oriented-Programming and Classes

Difference between pointers and references

- Reference: automatically dereferenced immutable pointer, or an alternative name for an object.
- Difference between pointers and references:
    - Assignment to a pointer changes the pointer's value.
    - Assignment to a reference changes the value of the object.

Object-Oriented-Programming and Classes

Difference between pointers and references

- Reference: automatically dereferenced immutable pointer, or an alternative name for an object.
- Difference between pointers and references:
  - Assignment to a pointer changes the pointer's value.
  - Assignment to a reference changes the value of the object.
  - To get a pointer, you use

Object-Oriented-Programming and Classes

Difference between pointers and references

- Reference: automatically dereferenced immutable pointer, or an alternative name for an object.
- Difference between pointers and references:
  - Assignment to a pointer changes the pointer's value.
  - Assignment to a reference changes the value of the object.
  - To get a pointer, you use **new** or **&**.

Object-Oriented-Programming and Classes

Difference between pointers and references

- Reference: automatically dereferenced immutable pointer, or an alternative name for an object.
- Difference between pointers and references:
  - Assignment to a pointer changes the pointer's value.
  - Assignment to a reference changes the value of the object.
  - To get a pointer, you use **new** or **&**.
  - To access a value pointed to by a pointer, you dereference: with

Object-Oriented-Programming and Classes

Difference between pointers and references

- Reference: automatically dereferenced immutable pointer, or an alternative name for an object.
- Difference between pointers and references:
  - Assignment to a pointer changes the pointer's value.
  - Assignment to a reference changes the value of the object.
  - To get a pointer, you use **new** or **&**.
  - To access a value pointed to by a pointer, you dereference: with * or *[ ]*.

Object-Oriented-Programming and Classes

Difference between pointers and references

- Reference: automatically dereferenced immutable pointer, or an alternative name for an object.
- Difference between pointers and references:
  - Assignment to a pointer changes the pointer's value.
  - Assignment to a reference changes the value of the object.
  - To get a pointer, you use **new** or **&**.
  - To access a value pointed to by a pointer, you dereference: with * or *[ ]*.
  - A reference cannot refer to another object after initialization.
  - Assignment of references does deep copy; assignment of pointers does not.

Object-Oriented-Programming and Classes

Pass by pointer and reference

- When you want to change the value of variabble passed as argument of a function.

```
void incr_r(int& r) { ++r; } // pass a reference
void incr_p(int* p)
// pass a pointer, dereference it and increment the result
```

Object-Oriented-Programming and Classes

Pass by pointer and reference

- When you want to change the value of variabble passed as argument of a function.

```
void incr_r(int& r) { ++r; } // pass a reference
void incr_p(int* p)
// pass a pointer, dereference it and increment the result
{ ++*p; }
```

- How to choose ?
    - For small objects: by value.
    - Use pointers when ”no object” can be an argument.
    - Otherwise, use references.

Object-Oriented-Programming and Classes

Pointers, references, and inheritance

- Pointers can be implicitely converted to an inherited class.

```
void MyFunc(PayOff * r);//
PayOff* PO1 = new CallPayOff{...};
CallPayOff PO2{...};

MyFunc(PO1);
MyFunc(&PO2);
```

- References can also be implicitely converted to an inherited class.

```
void MyFunc(PayOff& r);//
PayOff* PO1 = new CallPayOff{...};
CallPayOff PO2{...};

MyFunc(*PO1);
MyFunc(PO2);
```

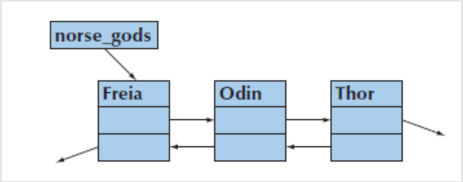
Object-Oriented-Programming and Classes

Exercise: Doubly-linked lists

Object-Oriented-Programming and Classes

Doubly-linked lists

- Lists are the most common and useful data structures.
- A list is made out of "links", holding information and pointers to next links.
- *doubly-linked* list :



- Douby-linked list class called *Link*.
    - A null pointer indicates that a Link doesn't have a successor or predecessor.
    - Given a Link object, we should get to its successor using a pointer and to its predecessor using another pointer.

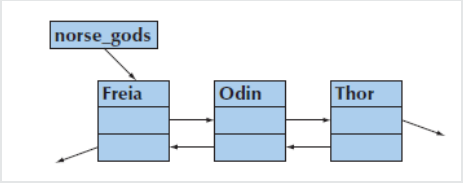
Object-Oriented-Programming and Classes

Link: the class

```
class Link {
public:
    Link(const string& v, Link* p = nullptr, Link* s = nullptr)
        : value{v}, prev{p}, succ{s} { }
    Link* next() const { return succ; }
    Link* previous() const { return prev; }
    string val() const { return value; }

    Link* previous;
    Link* next;
private:
    string value;
};
```

Object-Oriented-Programming and Classes

Doubly-linked lists



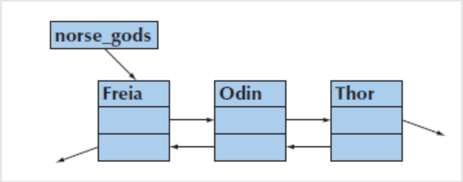
- Create the list :

```
cout << "Test of basic creation:";
Link* norse_gods = new Link{"Thor",nullptr,nullptr};
norse_gods = new Link{"Odin",nullptr,norse_gods};
norse_gods->next->previous = norse_gods;
norse_gods = new Link{"Freia",nullptr,norse_gods};
norse_gods->next->previous = norse_gods;
cout << norse_gods;
```

Object-Oriented-Programming and Classes

Exercise

- Create an *insert* method
    - Takes one argument: *Link* n*.
    - Inserts *n* before *this*
    - returns the pointer n.
    - Must handle all possiblre cases (this or n are null, this has no predecessor ..)
- create the list with your new *insert* function.



```
Link* norse_gods2 = new Link{"Thor",nullptr,nullptr};
norse_gods2 = norse_gods2->insert(new Link{"Odin"});
norse_gods2 = norse_gods2->insert(new Link{"Freia"});
cout << norse_gods2 <<"";
```

Object-Oriented-Programming and Classes

Insert function

```
Link* Link::insert(Link* n) // insert n before p; return n
{
    Link* p = this; // pointer to this object
    if (n==nullptr) return p; // nothing to insert
    if (p==nullptr) return n; // nothing to insert into
    n->succ = p; // p comes after n
    if (p->prev) p->prev->succ = n;
    n->prev = p->prev; // p's predecessor becomes n's predecessor
    p->prev = n; // n becomes p's predecessor
    return n;
}
```

Main

```
Link* norse_gods = new Link{"Thor"};
norse_gods = insert(norse_gods,new Link{"Odin"});
norse_gods = insert(norse_gods,new Link{"Freia"});
```

Object-Oriented-Programming and Classes

Exercises

- Create the following list operations for our class
- Test them by printing, with real lists and with nullptrs.
    - add: insert after an elements, insert n after p, return n.
    - erase: remove an element, return the successor.
    - find: find a value in a list. return nullptr if not found.
    - advance: move n positions in list, n can be negative. return nullptr if list too short.

Object-Oriented-Programming and Classes

Exercises: Link class

```
class Link {
public:
    string value;
    Link(const string& v, Link* p = nullptr, Link* s = nullptr)
        : value{v}, previous{p}, next{s} { }
    //Accessors & public data
    string val() const  return value;
    Link* previous;
    Link* next;
    // List oprations
    Link* insert(Link* n) ; // insert n before this object
    Link* add(Link* n) ; // insert n after this object
    Link* erase() ; // remove this object from list
    Link* find(const string& s); // find s in list
    Link* advance(int n); // move n positions in list
private:
    string value;
};
```

Object-Oriented-Programming and Classes

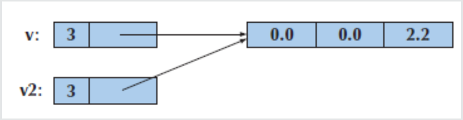
Copy constructor, assignment operator

Object-Oriented-Programming and Classes

Copy constructors

```
void f(int n) {
    vector v(3); // define a vector of 3 elements
    v.set(2,2.2); // set v[2] to 2.2
    vector v2 = v; // what happens here?
    // . . .
}
```

- The default meaning of copying for a class is :Copy all the data members.
- For pointer members, copying causes problems ! A copy of a pointer is another pointer to the same object.



- When going out of scope, the destructor is called two times: via the pointer v, and the pointer v2: potential crash.

Object-Oriented-Programming and Classes

Copy constructors

- Solution: provide a copy operation that copies the elements.
- Make sure that this copy operation gets called when we initialize:

```
vector v2 = v; // what happens here?
```

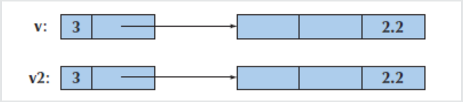
- Initialization of objects of a class is done by a constructor. We need a constructor that copies: *a copy constructor*.
- A copy constructor is
  - Defined to take as its argument a reference to the object from which to copy
  - Called when we try to initialize one object with another.

Object-Oriented-Programming and Classes

Copy constructors

- For our class vector, we allocate memory for the elements in the copy constructor:

```
vector:: vector(const vector& arg) // allocate elements,
                                   //then initialize them by copying
    :sz{arg.sz}, elem{new double[arg.sz]}
{
    for (int i=0; i<arg.sz; ++i) elem[i] = arg[i];
}

vector v2 = v; // what happens here?
```



- the two vectors are now independent: we can change the value of elements in v without affecting v2.

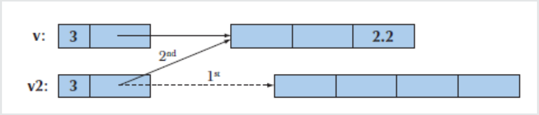
Object-Oriented-Programming and Classes

Copy assignment

- We handled copy construction (initialization), but we can also copy vectors by assignment.

```
void f2(int n)
{
    vector v(3); // define a vector
    vector v2(4);
    v2 = v; // assignment: what happens here?
}
```

- Default meaning of copy assignment is memberwise copy: problem for pointer members.



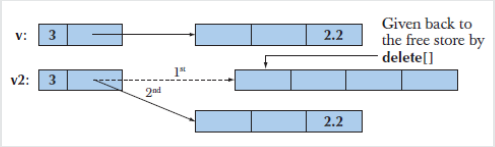
Object-Oriented-Programming and Classes

Copy assignment

- Solution: We define an assignment that copies properly.

```
vector& vector::operator=(const vector& a)
// make this vector a copy of a
{
    double* p = new double[a.sz]; // allocate new space
    for (int i=0; i<a.sz; ++i) elem[i] = arg[i]; // copy elements
    delete[] elem; // deallocate old space
    elem = p; // now we can reset elem
    sz = a.sz;
    return *this; // return a self-reference (see §17.10)
}
```

- Assignment is a bit more complicated than construction because we must delete the old elements.



Object-Oriented-Programming and Classes

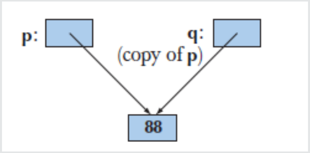
this pointer

- Get a pointer to the object itself.
- In every member function, the identifier *this* is a pointer to the object for which the member function was called.

Copy terminology

- *Shallow copy* copies only a pointer so that the two pointers now refer to the same object.

```
int* p = new int{77};
int* q = p; // copy the pointer p
```

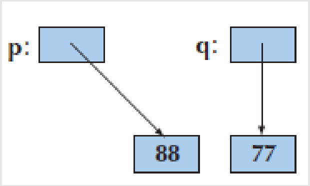


Object-Oriented-Programming and Classes

Copy terminology

- *Deep copy* copies what a pointer points to so that the two pointers now refer to distinct objects.

```
int* p = new int{77};
int* q = new int{*p};
```



Object-Oriented-Programming and Classes

Virtual copying

- We do not know what we are copying.
- Solution: create a virtual method for cloning : clone.

```
class A
{
    // ***
    virtual A* clone() const;
}

class B: public A
{
    // ****
}
A* B::clone() const
{
    return new B(*this); //call for the copy constructor
}
```

Object-Oriented-Programming and Classes

Back to pricing

Object-Oriented-Programming and Classes

Payoff class: what we need

- We want the vanilla option to store its own copy of the pay-off.
- We need a (virtual) copy assignment method!
- We need a virtual destructor so base classes can handle memory (new, delete).

Payoff class

```
class PayOff
{
public:
    PayOff(){};
    virtual PayOff* clone() const=0; // Virtual copy assignment
    virtual double operator()(double) const=0;
    virtual ~PayOff(){} // Virtual destructor
private:
};
```

Object-Oriented-Programming and Classes

Payoff class: PayoffCall3.h

```
class PayOffCall : public PayOff
{
public:
    PayOffCall(double Strike );
    PayOffCall(const PayOffCall&);
    double getStrike() const {return Strike;}
    void setStrike(const double s) {Strike = s;}

    virtual PayOff* clone() const; // Virtual copy assignment
    virtual double operator()(double) const;
    virtual ~PayOffCall(); // Virtual destructor
private:
    double Strike;
};
```

Object-Oriented-Programming and Classes

Payoff class: PayoffCall3.cpp

```
/*Constructors*/
PayOffCall::PayOffCall(double Strike_) : Strike(Strike_){ }

PayOffCall::PayOffCall(const PayOffCall& POC) // Copy constructor
{    Strike = POC.getStrike();  }

/* Virtual methods */
double PayOffCall::operator () (double Spot) const
{    return (Spot - Strike) >0 ? (Spot - Strike) : 0; }

PayOff* PayOffCall::clone() const // Virtual cloning
{    return new PayOffCall(*this); }

PayOffCall::~PayOffCall() // Virtual destructor
{}
```

Object-Oriented-Programming and Classes

Exercise: An exotic asian option

- Define the following class for exotic options, where the payoff is equal to $\sum Spot^i$

```
class PayOffExoticAsian : public PayOff {
public:
    // Constructors
    PayOffExoticAsian(double* weights_, int nbDates_);
    PayOffExoticAsian(const PayOffExoticAsian&);

    // Accessors
    double getnbDates() const {return nbDates;}
    void setnbDates(const double s) {nbDates = s;}
    double* getweights() const {return weights;}
    void setweights(double* s) {weights = s;}

    //Virtual functions
    virtual PayOff* clone() const; // Virtual copy assignment
    virtual double operator()(double) const;
    virtual ~PayOffExoticAsian(); // Virtual destructor
private:
    double* weights;
    int nbDates;

};
```

Object-Oriented-Programming and Classes

Exercise: An exotic asian option

- The usual methods :

```
/*Constructors*/
PayOffExoticAsian::PayOffExoticAsian(double* weights_, int nbDates_) :
                   weights(weights_), nbDates(nbDates_){ }

PayOffExoticAsian::PayOffExoticAsian(const PayOffExoticAsian& POC) // Copy constructor
{ TODO }

/*Virtual methods*/
double PayOffExoticAsian::operator () (double Spot) const {
    double res = 0;
    for (int i=0; i<nbDates; ++i) res += pow(Spot, weights[i]);
    return res;
}

PayOff* PayOffExoticAsian::clone() const // Virtual cloning
{TODO }

PayOffExoticAsian::~PayOffExoticAsian() // Virtual destructor

{ TODO }
```

Object-Oriented-Programming and Classes

Solution: the copy constructor

```
PayOffExoticAsian::PayOffExoticAsian(const PayOffExoticAsian& POC)
// Copy constructor
{
    // create new dates
    int size = POC.getnbDates();
    double* newWeights = POC.getweights();

    weights = new double[size];
    nbDates = size;
    for (int i=0; i<size; ++i)
        weights[i] = newWeights[i];
}
```

Object-Oriented-Programming and Classes

Solution: Virtual cloning & destruction

```
PayOff* PayOffExoticAsian::clone() const // Virtual cloning
{
    return new PayOffExoticAsian(*this);
}

PayOffExoticAsian::~PayOffExoticAsian() // Virtual destructor
{
    delete[] weights;
}
```

Object-Oriented-Programming and Classes

Solution: MAIN

```
//Tests for Exotic option
double* Weights = new double[4] 0.2, 0.2, 0.3, 0.3;
PayOffExoticAsian EOPayOff(Weights, 4);
VanillaOption EOOption(EOPayOff, Expiry);

 double resultEO = SimpleMonteCarlo3(EOOption,
                                Spot,
                                Vol,
                                r,
                                NumberOfPaths);

cout << "****** Results for the exotic option *******" << '';
cout << "the prices are "
     << resultEO << " for the exotic option";
```

Review

Review

Basics

- What is the difference between declaration and definition ?

Review

Basics

- What is the difference between declaration and definition ?
- What are the three types of errors in C++ ?

Review

Basics

- What is the difference between declaration and definition ?
- What are the three types of errors in C++ ?
- When is a programm “Type-safe” ?

Review

Basics

- What is the difference between declaration and definition ?
- What are the three types of errors in C++ ?
- When is a programm “Type-safe” ?
- What conversion of built-in types is type-safe, which is not ?

Review

Basics

- What is the difference between declaration and definition ?
- What are the three types of errors in C++ ?
- When is a programm “Type-safe” ?
- What conversion of built-in types is type-safe, which is not ?
- What is the difference between a ”constexpr” and a ”const” expression.

Review

Basics

- What is the difference between declaration and definition ?
- What are the three types of errors in C++ ?
- When is a programm “Type-safe” ?
- What conversion of built-in types is type-safe, which is not ?
- What is the difference between a ”constexpr” and a ”const” expression.
- How to initialize while enforcing type-safety ?

Review

Basics

- What is the difference between declaration and definition ?
- What are the three types of errors in C++ ?
- When is a programm “Type-safe” ?
- What conversion of built-in types is type-safe, which is not ?
- What is the difference between a ”constexpr” and a ”const” expression.
- How to initialize while enforcing type-safety ?
- what is a scope ? Give examples

Review

Exceptions

```
class unknown_exc { };
int myErrorFunc(int i) {
    if (i == 0) {
        throw unknown exc();
    } else {
         throw runtime_error("I found a mistake !");
     }
}
```

- How to handle all errors in the main ?

Review

Technicalities

- What is the difference between passing by reference, and by const reference ?

Review

Technicalities

- What is the difference between passing by reference, and by const reference ?
- How to use a method of an object passed by const reference ?

Review

Technicalities

- What is the difference between passing by reference, and by const reference ?
- How to use a method of an object passed by const reference ?
- What is a namespace?

Review

Technicalities

- What is the difference between passing by reference, and by const reference ?
- How to use a method of an object passed by const reference ?
- What is a namespace?
- What is encapsulation? polymorphism?

Review

Technicalities

- What is the difference between passing by reference, and by const reference ?
- How to use a method of an object passed by const reference ?
- What is a namespace?
- What is encapsulation? polymorphism?
- What is an enumeration ? What is the difference between a scoped and a plain enumeration ?

Review

OOP

- What is the difference between a "struct" and a "class" ?

Review

OOP

- What is the difference between a ”struct” and a ”class” ?
- What is a default constructor ?

Review

OOP

- What is the difference between a "struct" and a "class" ?
- What is a default constructor ?
- What is the meaning of "static" in :

```
class Year {
    static const int min = 1800;
public:
    Year(int x)...
};
```

Review

OOP

- What is the difference between a "struct" and a "class" ?
- What is a default constructor ?
- What is the meaning of "static" in :

```
class Year {
    static const int min = 1800;
public:
    Year(int x)...
};
```

- Is this OK ? a compile-time error ? a link-time eror ? a run-time error?

```
class Year {
    int y;
public:
    int f(int i) { ..}
};
void some function(const Year& y) {
    int r = d.f(3);
}
```

Review

OOP

- Is this OK ? a compile-time error ? a link-time eror ? a run-time error?

```
class Year {
    int y;
public:
    int f(int i) const {
        return ++i;
    }
};
```

Review

OOP

- Is this OK ? a compile-time error ? a link-time eror ? a run-time error?

```
class Year {
    int y;
public:
    int f(int i) const {
        return ++i;
    }
};
```

- Is this OK ? a compile-time error ? a link-time eror ? a run-time error?

```
enum class Month {
Jan=1, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec };
Month operator+(Month& m) {
    m = /* ... */
    return m;
}
```

Review

OOP

- What does the following line mean?

```
int OneVirtualFunc(int someInput) const override;
```

Review

OOP

- What does the following line mean?

```
int OneVirtualFunc(int someInput) const override;
```

- What is the difference between private, public, and protected class members, in the context of Inheritance ?

Review

OOP

- What does the following line mean?

```
int OneVirtualFunc(int someInput) const override;
```

- What is the difference between private, public, and protected class members, in the context of Inheritance ?
- What is an abstract class ?

Review

OOP

- What does the following line mean?

```
int OneVirtualFunc(int someInput) const override;
```

- What is the difference between private, public, and protected class members, in the context of Inheritance ?
- What is an abstract class ?
- Is this OK ? a compile-time error ? a link-time eror ? a run-time error?

```
class A {
    int a;
public:
    virtual void f() = 0;
    virtual void g() = 0;
};
class B: public A {
    int a;
public:
    void f(); };
```

Review

OOP

- Is this OK ? a compile-time error ? a link-time eror ? a run-time error?

```
class A {
    int a;
public:
    virtual void f() = 0;
    virtual void g() = 0;
};

class B: public A {
    int a;
public:
    void f();
};

int main() {
    B b;
}
```

Last Homework

Homework

Exercise

- Create the following methods for the VanillaOption class (use pointers !):
  - a clone method, such as the one for Payoff.
- Create a class Portfolio, which a doubly linked list, but with the following data members:
  - VanillaOption* theOptions: a pointer on the options.
  - double* IDs: a pointer on doubles which are the identifiers of the options.
  - double PFsize: the size of the portfolio (should be the size of "theOptions" and "IDs").
- Create the following methods for the Portfolio class (use pointers !)
  - A constructor / destructor (the constructor should copy every option, using the clone method).
  - A method to insert a VanillaOption to the Portfolio (every option has a different ID).

Homework

Exercise ...

- A method to delete an option from the Portfolio (using an ID).
- A copy constructor method for the Portfolio class.
- A copy assignment method for the Portfolio class.
- A method to give the price of the Portfolio.
- A method overloading of the operator<<, to print informations about the options (Price of every option): (bonus: you can create a method overloading the operator<<for the classes "Payoff" and "VanillaOption", make them virtual. Thanks to this, you can also print the type of the options in the overloading of operator<<of Portfolio).

- Test your Portfolio class, by testing every method.

The End

Thank You !