



# HAE916E – THE C++ LANGUAGE

Master 2 EEA/2024

Mikhaël MYARA

[mikhael.myara@umontpellier.fr](mailto:mikhael.myara@umontpellier.fr)



DEPARTEMENT D'ENSEIGNEMENT

**EEA**  
ELECTRONIQUE - ELECTROTECHNIQUE - AUTOMATIQUE



**fds**  
FACULTÉ DES SCIENCES  
MONTPELLIER



*« Most computer problems come from the  
keyboard-to-chair interface. »*

Klaus Klages

**The C++ Language - Master 2 EEA/2024**

6th September 2024

Object-oriented programming and specificities of C++.  
by Mikhaël Myara, for EEA Department, University of Montpellier, France



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0  
Unported License.*



## Preface

This document is written for Master EEA students from the university of Montpellier, in order to get a first approach of the C++ language. It assumes the students have already an experience of the programming in an imperative language, best would be C. The knowledge of other languages like Python or Matlab should help too.

This course aims at helping the students to write **safe code in C++**, safer than they would have written in pure C and with far less efforts, while being able to use the awesome standard libraries that come with it. It is not a "very conceptual" object-oriented programming course, but is wanted to be a pragmatic one, that concerns the everyday life of the C++ coder, and targeting EEA needs.

This document is thought to be used in the following context : **the students should have their own computer with a C++ compiler installed inside the classroom**, and try and follow, during the course itself, the examples given, in order to be able to react to the problems they identify in real time. Depending on the network availability, they can download or get the code samples attached to this document, both are available with this document. It is identified along the document with this kind of visual element :

  [code/code-0-1.cpp]

The "paperclip" icon  allows to get the sample code embedded with the PDF document, with no requirement for any internet connection. The "globe" icon  allows to get exactly the same code but through the internet. Both are given because :

- › some PDF browsers do not work properly with the "paperclip" icon, but most work well. For example, with Adobe Acrobat, you may double click on the "paperclip" icon to get the code and that's all. With others it's ok with a single click.
- › But if your PDF browser does not work well with the "paperclip" icon and if you have a working internet connexion, then you can get the file over the internet with the "globe" icon.

So: if you have a "good" PDF reader software installed on your computer, you should simply download this document before the first course, store it somewhere on your hard disk, and everything should work at the university. If not, you can hope a WiFi connection will be available at the university (but it can depend on the classroom). However you'd better think about installing a PDF reader compatible with "attached documents", like Adobe Acrobat Reader for example.

## Install software

If you work with Linux, you should easily install a compiler using the terminal.

If you work with a Mac, things are a bit complex to describe, since I do not have access to a recent Mac, and Apple changed "everything" in a recent past (since M1 chips).

Most of you may work with Windows. You can install a Linux-like shell with C++ compiler. There are many possibilities :

- You can try to install a Linux under Windows. This can be done by two ways, with a virtual machine (such as Virtualbox) or with WSL2. These solutions are not described here.
- Install a limited Linux sub-os. That's what we show below.

## Install chocolatey, MSYS2 (and Eclipse)

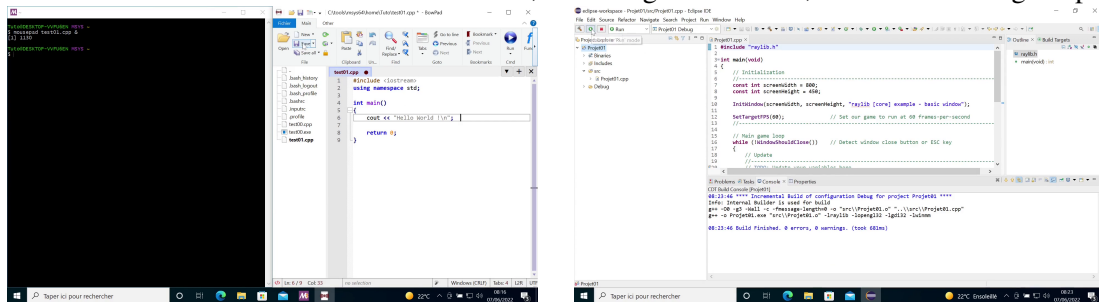
All the installation steps are described in the video tutorial:

<http://dl.eea-fds.umontpellier.fr/CppInstall/tuto-install-Cpp.mp4>

To follow this tutorial, you will need the package below :

<http://dl.eea-fds.umontpellier.fr/CppInstall/package.zip>

You will obtain 2 kinds of work environments, one using the terminal, another one using eclipse :



Once the install is successful, follow the next tutorial to check everything is ok and use the compiler :

<http://dl.eea-fds.umontpellier.fr/CppInstall/tuto-utilisation-Cpp.mp4>

If the installation was not successful, you should uninstall all cleanly and try to install again. Here is a small tutorial that shows how to cleanly uninstall our C++ package. :

<http://dl.eea-fds.umontpellier.fr/CppInstall/tuto-supprimer-Cpp.mp4>

## Contents

<b>Chapter 1 – Introduction</b>	<b>8</b>
1.1 Context	8
1.2 Popularity and usage of programming languages	9
1.3 What is C++ ?	10
1.4 Why studying C++ in EEA ?	10
1.5 Towards Object-Oriented programming	11
 <b>PART I – Basics of classes and objects</b>	
<b>Chapter 2 – Mixing code using namespace</b>	<b>14</b>
2.1 Creating namespace sections in your code	14
2.2 Using elements defined inside a namespace	16
2.3 namespace and libraries	17
2.4 Put namespace contents into the global scope	18
<b>Chapter 3 – A first glance to objects and class</b>	<b>20</b>
3.1 What are classes ?	20
3.2 Implementation of the display() function	22
3.3 Implementation of the byModulePhase function	24
3.4 Implementation of calcModule	25
3.5 Implementation of both sum and product	26
<b>Chapter 4 – class constructors and destructors</b>	<b>30</b>
4.1 Constructors	30
4.2 Constructor with parameters	32
4.3 Destructor	34
<b>Chapter 5 – Class members rights</b>	<b>38</b>
5.1 Demystifying the "public" word we already used	38
5.2 private access	40
5.3 Other things to know	41

**PART II – Some C++ concepts for better code**

<b>Chapter 6 – Stream operators (and strings) .....</b>	<b>45</b>
6.1 Replacing <code>printf</code>	45
6.2 Managing strings	45
6.3 Replacing <code>scanf</code>	47
6.4 Managing files with <code>&lt;&lt;</code> and <code>&gt;&gt;</code>	47
<b>Chapter 7 – Some "+" of C++ .....</b>	<b>50</b>
7.1 References on variables	50
7.2 Polymorphic functions	51
7.3 Overriding operators	51
7.4 Polymorphic functions	53
7.5 Automatic type specification	53
<b>Chapter 8 – The <code>const</code> Keyword .....</b>	<b>56</b>
8.1 Constant value	56
8.2 Constant parameter for a function	56
8.3 "Constant method" of a class	57
<b>Chapter 9 – "Template"-based C++ libraries .....</b>	<b>60</b>
9.1 Overview	60
9.2 Limits of pure C approach with types	60
9.3 How C++ Templates solve this kind of problem : example with <code>vector</code>	61
9.4 Iterators	63
9.5 The <code>list</code> type in C++ standard library	64
9.6 Lighter iterator syntax with <code>auto</code>	66
9.7 Easier loops range based loops	67
9.8 Other data organizations coming from C++ libraries	67
9.9 Template linear algebra libray : Eigen	67
<b>Chapter 10 – Errors management : try-throw-catch .....</b>	<b>71</b>
10.1 Classical error management in a C/C++ code	71

**PART III – Fundamental concepts for Object-Oriented programming**

<b>Chapter 11 – Dynamic allocation of objects .....</b>	<b>75</b>
11.1 Dynamically create objects : <code>new</code>	75
11.2 Destroy objects : <code>delete</code>	75
<b>Chapter 12 – A powerful concept : inheritance .....</b>	<b>77</b>
12.1 Inheritance : why ?	77
12.2 Inheritance syntax in C++	78
12.3 Customizing code	79

12.4	Ouch, pointers are back ...	81
12.5	Virtuality ? Polymorphism of pointed elements ?	82
12.6	public, protected and private inheritance	83

## PART IV – Practical considerations

<b>Chapter 13 – Compilation</b> .....	<b>86</b>
13.1 Interpreted language vs compiled language	86
13.2 Multi-file compilation in a basic case	87
13.3 Illustration of the overall toolchain	88
13.4 Multi-file compilation in more advanced cases	89
<b>Chapter 14 – Code style</b> .....	<b>90</b>

## PART V – More advanced topics

<b>Chapter 15 – Templates : how to create yours ? .....</b>	<b>94</b>
15.1 Writing templates !	94
15.2 Ouch ... that syntax hurts !	95
15.3 What if I want matrix of complex numbers ?	96
15.4 About compilation	96

## PART VI – Practical Works

<b>Chapter 16 – Practical works</b>	<b>98</b>
-------------------------------------	-----------

# Chapter 1 – Introduction

## Motivations and objectives

This chapter aims at situating C++ :

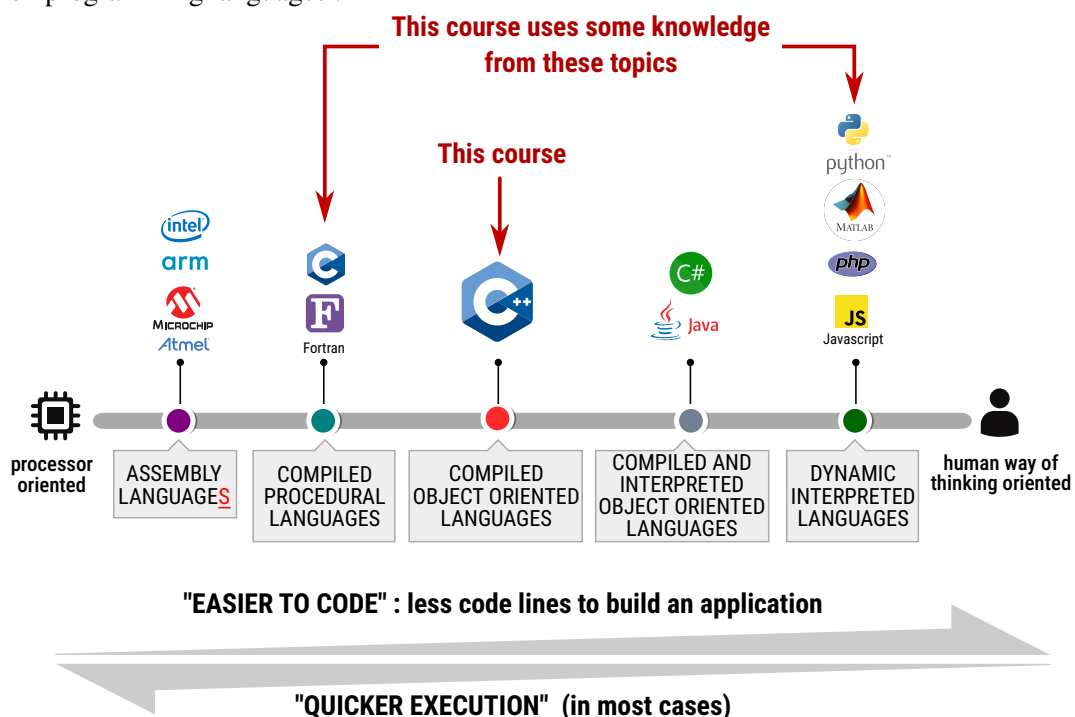
- › in the context of past and today's programming languages : Assembly, C, Matlab, Python and others
- › as well as in the various "paradigms" (= ways of thinking) in which languages are categorized : procedural, object-oriented, compiled, interpreted, etc.

## 1.1 Context

As an electrical engineering, "EEA" student, your academic background about computer science may contain, at least :

- › A "high level" language, usually a scripting language, that you used for scientific purposes (like simulations, signal or data analysis, etc.). These should have been Matlab or Python for example.
- › A "low level" language, usually a compiled language, that you used for more technical purposes. It should be typically the most standard one, the C language, and perhaps a few assembly language.

These languages can be situated on the graph below, that summarizes a (very) incomplete picture of common programming languages :





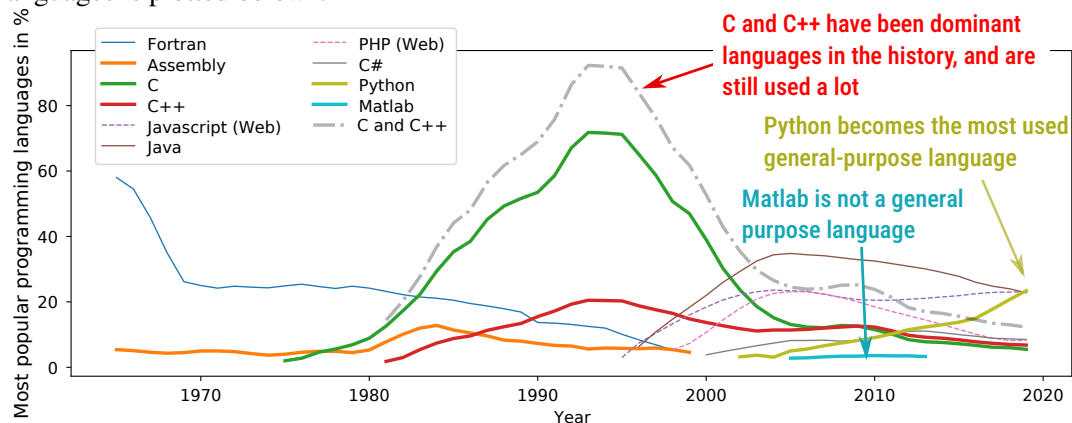
Somme comments :

- **compiled languages** are languages that are converted into assembly language, then to processor-runnable byte-code (often code "machine language") in a preliminary step before running, called "compilation". That makes the code **execution quick** because the transformation of the code, from the language to byte-code, is already performed. As a consequence, we can say that these languages do not perform too much supplementary, not-written operations, during run time.
- **interpreted languages** (sometimes called "scripting languages") create processor-runnable code while they perform the execution. For that reason, these languages lead to **much slower execution**. However, because of the fact that runnable code is generated during run time, they are more flexible. An obvious example : with such a language, you can generate code as a string built at run time, depending on the current context, and then "evaluate" (= execute) it. This is not easily available with compiled languages, as they rely on a compiler, which is external.

These differences are the main reasons for which microcontrollers require assembly or compiled languages (C or C++), in most cases : indeed, the available computing power is not enough to work with interpreted languages, whereas it is sufficient on a computer.

## 1.2 Popularity and usage of programming languages

About the usage of these languages, the evolution in computer science history of some of the most used languages is plotted below<sup>1</sup>:



What we can understand from this plot :

- Due to history, today, lots of Fortran, C and C++ code is already written and available. For example, most part of the matrix-oriented computation in Matlab comes from optimized opensource Fortran code, and lots of high performance libraries in python come from Fortran, C and C++ as well.
- because C and C++ are compiled, they are the only acceptable languages for optimized (high speed) code with a large spread. Also (for the same reason), they are massively used on microcontrollers.
- Assembly language is today mainly used in ultimate optimization. For example, parts of softwares like Photoshop, or gamer video cards drivers are written in assembly language.
- Since 2000, more than half the code written in the world is dedicated to web or smartphone applications, that uses specific languages, mainly to speed up the development time (not the execution time). That explains a part of the reduction of the C and C++ parts in these years, as the graph vertical unit is in percent.

<sup>1</sup> made from partial data found in "Most Popular programming Languages 1965-2019" in the "Data is Beautiful" youtube channel, which URL is : <https://www.youtube.com/watch?v=0g847HVwRSI>

- › Matlab is dedicated to scientific computing, and is more and more replaced by python for this purpose, in lots of scientific communities. It cannot be considered like a general purpose language, even if it can fit requirements beyond scientific computing.
- › Python is a very general purpose language and can fit lots of applications with a high level of performance : desktop software, web and smartphone applications, data mining, scientific computing, and even (at this time) hobbyist microcontroller code targeting basic applications.

### Remark

You may remember that **Matlab and Python are intrinsically slow execution and memory consuming languages**, because they are efficient regarding the (lower) effort necessary for the coder to build its software. In most cases, **when Matlab or Python perform computations quickly, it is only because the code delegates the intensive calculus part to a Fortran, C or C++ library.**

So at the modern time, a "good" program (targeting desktop computers for example), uses a language similar to Matlab or Python to build the application skeleton (for example its user interface), while relying on optimized libraries or, when necessary, writing custom C/C++ code for the parts that should work quickly. Even if it does not seem that natural, most modern softwares rely on various languages.

However in other contexts, like microcontrollers or real-time applications, C and C++ are mandatory in most cases.

## 1.3 What is C++ ?

The main point is that C++ is essentially the C language to which "Object-Oriented programming" feature has been added. This suggests to discuss about two points :

- › **What's the purpose of Object-oriented programming ?** Some people believe that graphic programming like in Labview or Simulink is what object-oriented programming. No : Object-oriented programming concerns new programming concepts, that lead to a far better organization of the code, thus more reliable code. You must understand that C++ does not allow to make things that C could not do, and the same level of performance can be reached with both if the programmer is sufficiently talented. However, if well written, C++ is much more easy to maintain and to read, and more things can be done with far less code. As a consequence, in practice a higher level of performance is possible with far lower development time and higher level of reliability, even with beginner programmers. Object-oriented programming indeed allows you to use with ease high performance already written code, and to **customize it easily**.
- › **What's the state of mind of Object-oriented programming in C ++ ?** C++ is not similar to most other object oriented programming languages, because it is made for high performance more than for being "purely" object-oriented. For example, writing the same program is usually more difficult in C++ than in Python (and sometimes by far). However, the code written in C++ will be executed much more quicker, usually by decades. **So C++ is a specific object-oriented language, because it is compiled and contains elements that do not exist in most other languages, that allow to reach quite easily a high level of performance.**

## 1.4 Why studying C++ in EEA ?

Now let's take the point of view of Electrical Engineering. In Electrical Engineering, we typically have two specificities that desktop application software do not meet :

- › we often work on chips with limited computation power and memory, as compared to desktop computers. If we don't want the applications to be too slow or simply to take more memory than the chip embeds, we have to choose a language that does not use too much resources (computation power and/or memory).
- › we also work for application requiring "real-time" operation, or requiring "immediate" response. In this case, building the quickest code is mandatory.

These two reasons lead to work with assembly, C or C++ language :

- › **assembly** leads to the quickest execution and less memory amount, but is very specific to each kind of chip. For that reason, when it can be avoided (in most cases) it is avoided, as it is a problem for flexibility : once a first version of the code is written, future versions of the same software may require to work on similar chip architecture. In other words the "abstraction" level, regarding the hardware, is not enough in most practical cases. Moreover, assembly requires a lot of lines of code, even for small projects.
- › **C language** is a usual very good choice, for moderate size projects. However, building bug-free code in C requires some expertise, and even basic techniques require a high level of skill related to this language. For large size projects, it becomes even more complex. Compared to assembly, the execution time and memory requirements is close to 1 to 3 times for an average coder, which is acceptable (in python or matlab, a simple for loop may take ... tens or hundreds of time longer as compared to C ! )
- › **C++ language** is today's best choice. Indeed, C++ does not require that much more resources than C (and even sometimes allows to easily reduce the computation time), allows to embed pure C code without effort, gives access to C and C++ libraries, offers object-oriented code organization, and lots of other benefits. It is thus compatible with a huge amount of code, that can be reused quite easily.

## 1.5 Towards Object-Oriented programming

Let's compare Object-oriented programming with traditional procedural programming.

**In procedural programming languages, like C**, we can organise code and data by means of two principal tools

- › **the code** can be organized by putting sub-parts of code inside **functions** : once made, a given part of code has a name (the name of the function), and can work on input data and give back output data, called "results". Once a function is written, it can be considered like a kind of "black box" that has inputs and brings output. That's how work all libraries, and well organized C code too.
- › **the data** can be organized by putting them in data structures, by combining memory blocks and the **typedef structs** of the C language.

The definitive "tool" to help **organizing both code and data in Object-Oriented programming** is called a **class**. On the path to understanding the concept of **class**, we have to speak about a simple tool available in C++ to start organizing code : the **namespace**. This is the purpose of the next chapter.

### ✧ Summary

C and C++ are very important languages since a very long time. They both have influenced a lot the other languages that have been created after them. They were developed for high speed execution and low memory requirements, contrary to Python or Matlab that were more developed to build code easier.

C++ is a C language augmented with Object-Oriented programming, a concept to be described in

the following chapters.

The Object-Oriented approach allows to build smarter code, with a far better organization than procedural languages like C.

C++ is a specific Object-Oriented language as it is also a compiled language, contrary to most other Object-Oriented languages. This allows a level of performance (speed at execution) that is not available with other languages, such as Matlab or Python. It allows to take benefit from Object-Oriented concepts (and more) while allowing to generate optimal executables. For those reasons, it is the language of choice for electrical engineering (EEA) applications, as these applications rely on embeddable processors (thus with lower computation power), and/or require real-time or reactive software.

What will be new in C++ as compared to C is the concept of "class of objects" simply named class. ■

## **PART I**

---

### **Basics of classes and objects**

## Chapter 2 – Mixing code using namespace

### Motivations and objectives

This chapter introduces the use of **namespace**, and use it to mix two distinct codes that should exhibit conflicts without this feature of C++.

### 2.1 Creating namespace sections in your code

A "namespace" is simply a way to store a part code in its own context, in order to avoid names conflict (typically function names conflicts) between various parts of code. Let's start with a C example. You have written two codes, one for matrixes  $2 \times 2$  and one for complex numbers. So you should have written :

complexes.c

```
1 typedef struct {
2     float Re, Im;
3 }complex;
4
5 complex product(complex z1,
6     complex z2){
7     complex z3;
8     z3.Re = z1.Re * z2.Re - z1
9         .Im * z2.Im;
10    z3.Im = z1.Re * z2.Im + z1
11        .Im * z2.Re;
12
13    return z3;
14 }
15 // ..... etc.
```

  [code/code-II-1-2.cpp]

matrix22.c

```
1 typedef struct {
2     float a,b,c,d;
3 }matrix22;
4
5 matrix22 product(matrix22 M1,
6     matrix22 M2){
7     matrix22 M3;
8     M3.a = M1.a*M2.a + M1
9         .c*M2.b;
10    M3.b = M1.a*M2.b + M1
11        .b*M2.d;
12    M3.c = M1.c*M2.a + M1
13        .d*M2.c;
14    M3.d = M1.c*M2.b + M1
15        .d*M2.d;
16
17    return M3;
18 }
19 // ..... etc.
```

  [code/code-II-1-3.cpp]

### Quick Application

Imagin that you now need to use the code of both "complexes.c" and "matrix22.c", in the same program.

➤ For that, first download the two files :

 [\[code/code-II-1-4.cpp\]](#)

 [\[code/code-II-1-5.cpp\]](#)

➤ Then, a simple way is to copy/paste the contents of both in a single new file. Call this file "fusion.c".

➤ Now, let's try to compile it :

```
gcc fusion.c
```

Unfortunately this "fusion.c" contains a conflict : two functions named "product" exist now in your "fusion.c" code, and so the compiler drops a compilation error, because it is ambiguous, like so :

```
./chapters/2/code/fusion.c:24:10: error: conflicting types for 'product'
 24 | matrix22 product(matrix22 M1,matrix22 M2);
    |           ^~~~~~
./chapters/2/code/fusion.c:10:9: note: previous definition of 'product' was
here
 10 | complex product(complex z1,complex z2){
    |           ^~~~~~
./chapters/2/code/fusion.c:27:10: error: conflicting types for 'product'
 27 | matrix22 product(matrix22 M1,matrix22 M2){
    |           ^~~~~~
./chapters/2/code/fusion.c:10:9: note: previous definition of 'product' was
here
 10 | complex product(complex z1,complex z2){
    |           ^~~~~~
```

You could rename for example the complex's product function as `cplx_product`, but it is not a great solution : if this product is called somewhere else in the "complexes.c" file, you have to update along all the file for that reason. This can become so weird if many functions or types are concerned, and finally you have to check and update the whole code : this is not practical at all.

**namespace** can solve this : in C++, you can declare that the structures and functions of a portion of code belong to something called a "namespace" to avoid these conflicts.

## ? Quick Application

Here is how it works, if we take again the code from "fusion.c" below :

 [\[code/code-II-1-6.cpp\]](#)

For now, rearrange this code by yourself to reach the code given below. For that, you should simply copy/paste parts from the "fusion.c" code inside of the braces associated to each namespace. That will let you understand that all we do is rearranging the code, not writing anything new :

```
1 namespace cplx {
2     typedef struct {
3         float Re, Im;
4     }complex;
5
6     complex product(complex z1,complex z2){
7         complex z3;
8         z3.Re = z1.Re * z2.Re - z1.Im * z2.Im;
9         z3.Im = z1.Re * z2.Im + z1.Im * z2.Re;
10
11         return z3;
12     }
```

```

13 |
14 | }
15 |
16 |
17 | namespace m22 {
18 |     typedef struct {
19 |         float a,b,c,d;
20 |     }matrix22;
21 |
22 |     matrix22 product(matrix22 M1,matrix22 M2){
23 |         matrix22 M3;
24 |         M3.a = M1.a*M2.a + M1.c*M2.b;
25 |         M3.b = M1.a*M2.b + M1.b*M2.d;
26 |         M3.c = M1.c*M2.a + M1.d*M2.c;
27 |         M3.d = M1.c*M2.b + M1.d*M2.d;
28 |
29 |         return M3;
30 |     }
31 | }
32 |
33 | int main()
34 | {
35 | }

```

 [\[code/code-II-1-7.cpp\]](#)

and save that new file as **fusionWithNamespace.cpp**. Then compile it using :

```
g++ fusion_with_namespace.cpp
```

We get no more compilation error. Some comments :

- please notice that we now no more use **gcc** but use **g++** as compiler. It is because the word **namespace** is no part of pure C language and then the compiler may drop a compilation error,
- moreover, C++ files might have the .cpp extension. With some compilers, writing C++ code in a file with a .c or .C extension leads to compilation errors.
- please notice that, in the code, we simply embraced the previous existing code in two distinct **namespace** , one called **cplx** for the complex numbers part, and one called **m22** for  $2 \times 2$  matrixes. This is what makes the "names" separation between the two codes.
- Notice that the **main** has been added (it is empty) to avoid an additionnal compilation error.
- at this time, if you try to run it by invoking `./a.out`, it will not display anything as the **main()** function is empty.

## 2.2 Using elements defined inside a namespace

Now that it is done, how do we use the "things" defined inside of each namespace ?

Quick Answer : By using the `::` operator of the C++. So let's now replace the **main** (lines 33–35 from previous code) by :

```

1 | int main()
2 | {
3 |     // we use the "complex" element from the inside
4 |     // of the "cplx" namespace :
5 |     cplx::complex Za,Zb,Zc;
6 |     Za.Re = 2; Za.Im = 3;
7 |     Zb.Re = 5; Zb.Im = -8;

```



```

8
9 // and now call the function "product", again inside
10 // of the "cplx" namespace :
11 Zc = cplx::product(Za,Zb)
12
13
14 // and now let's try something similar with matrixes :
15 m22::matrix22 = M,N,P;
16 M.a = 1; M.b = 0; M.c = 0; M.d = 1;
17 N.a = 1; N.b = 2; N.c = 3; N.d = 4;
18 P = m22::product(M,N);
19
20 // this works too. You can try to print the results if you want.
21
22 return 0;
23 }

```

 [\[code/code-II-2-8.cpp\]](#)

You notice the `cplx::` and `m22::` (lines 5, 11, 15 and 18) that allow to access to the elements defined inside each namespace. For anything else in this code, everything is similar to what we would have done in C : we simply declare variables and call functions.

## ? Quick Application

Do the following :

- > Download "fusion\_with\_namespace.cpp" if necessary :

 [\[code/code-II-2-9.cpp\]](#)

- > Add the main from the example above at the end of the `fusionWithNamespace.cpp`, like so :

```
g++ fusionWithNamespace.cpp
```

- > test it :

```
./a.out
```

## ! Important

Please let's remember this syntax :

environmentName::elementName

i.e. the use of the `::` operator, it is used a lot in C++ and not only for namespace. ■

## 2.3 namespace and libraries

In C++, we should not use anymore the `stdio.h`, `stdlib.h` or `math.h` libraries, even if it is still possible. Instead of that, C++ defines new libraries called `cstdio`, `cstdlib` or `cmath`, that work exactly the same way as pure C libraries, except that they are use a namespace. They do so in order to avoid name conflicts with other libraries. They contain all the usual functions of C libraries, but all are inside the `std::` namespace.

## ? Quick Application

As you can see on a simple example :

```

1  #include <cmath>
2  #include <cstdio>
3
4  int main()
5  {
6      double t,y;
7
8      for(t=0;t<3;t=t+0.05) {
9          y = std::cos(2*M_PI*3*t);
10         std::printf("%f\t%f\n",t,y);
11     }
12     return 0;
13 }
```

 [\[code/code-II-3-10.cpp\]](#)

You can see that lines 9 and 10 are exactly similar to usual C calls to C libraries, except that we have to specify that these are located into the **std::** namespace.

Try to write this code, add and remove the **std::** to see how the compiler behaves.

## 2.4 Put namespace contents into the global scope

If you are sure that will no conflict may occur, you can tell C++ to import the contents of a given namespace in the global scope. To do this, you need the keyword **using namespace**. For example, the previous code can be written again as follows :

```

1  #include <cmath>
2  #include <cstdio>
3  using namespace std;
4
5  int main()
6  {
7      double t,y;
8
9      for(t=0;t<3;t=t+0.05) {
10         y = cos(2*M_PI*3*t);
11         printf("%f\t%f\n",t,y);
12     }
13     return 0;
14 }
```

 [\[code/code-II-4-11.cpp\]](#)

You may notice the line 3 that tells that all that is inside the namespace **std** is now accessible from the global scope. As a consequence, you see that lines 10 and 11 do not require anymore the mention of **std::**. Of course this ability is not limited to the **std** namespace and is possible with any namespace, as long as this does not lead to conflicts.

### Summary

We have seen that C++ allows to isolate portions of code concerning the used names (and only the used names), for example the names of functions or of new types. This allows to mix portions of code that have not been initially thought to work together.

About the syntax of namespace, a short example follows :

```

1  namespace test {
2      int sum(int a,int b) {
3          return a+b;
4      }
```


```
4     }
5 }
6
7 int main() {
8     int k;
9     k = test::sum(5,8);
10    return 0;
11 }
```

  [code/code-II-4-12.cpp]

A reverse way of thinking can lead to want the contents of a namespace in the global scope of the program, to make the code writing lighter. It is usual with libraries, but not limited to. For example :

```
1 #include <cstdio>
2 using namespace std;
3
4 int main() {
5     printf("hello world !\n");
6     // instead of std::printf("hello world !\n");
7     // without the "using namespace std;"
8 }
```

  [code/code-II-4-13.cpp]

We also have seen that compiling C++ code requires **g++** instead of **gcc** 

## Chapter 3 – A first glance to objects and `class`

### Motivations and objectives

This chapter introduces the most fundamental knowledge about the object-oriented programming paradigm. We show the most immediate consequences of that by means of an example, using the the word `class` of the C++.

### 3.1 What are classes ?

A first definition of what is a `class` can be done from the knowledge you have of the C language.

What do we always do in C ? We create `typedef struct` to say how are data arranged, and create functions that are specific for this structure : the data structures together with the related functions easily define a concept. This was the case, in the previous chapter, for complex numbers as well as for matrixes  $2 \times 2$ .

Now how does Object-programming work ? Object-programming takes the point of view that the data structures and the related functions should not be separated : they should work together, and the syntax must be really explicit about this.

This is the base of object-programming : doing this association data+functions consists in creating "a class". So a class is an object-oriented concept that assembles, in a single entity, both data and code. Let's take as starting point the complex numbers example with which we played in the previous chapter. Here is the pure C basic code, and we will work to convert into object-oriented C++ in a second step :

```
1  #include <stdio.h>
2  #include <math.h>
3
4  //== declarations ==
5  typedef struct {
6      float Re, Im;
7  }complex;
8
9
10
11 void display(complex z);
12 complex byModulePhase(float rho, float theta);
13 float calcModule(complex z);
14 complex sum(complex z1, complex z2);
15 complex product(complex z1, complex z2);
16
17
18
19 //== definitions ==
20 void display(complex z) {
21     printf("%f + i%f", z.Re, z.Im);
22 }
23
24 complex byModulePhase(float rho, float theta) {
```

```

25         complex z;
26         z.Re = rho * cos(theta);
27         z.Im = rho * sin(theta);
28
29         return z;
30     }
31
32     float calcModule(complex z)
33     {
34         return sqrt(z.Re*z.Re + z.Im*z.Im);
35     }
36
37     complex sum(complex z1, complex z2) {
38         complex z3;
39         z3.Re = z1.Re + z2.Re;
40         z3.Im = z1.Im + z2.Im;
41         return z3;
42     }
43
44     complex product(complex z1, complex z2){
45         complex z3;
46         z3.Re = z1.Re * z2.Re - z1.Im * z2.Im;
47         z3.Im = z1.Re * z2.Im + z1.Im * z2.Re;
48
49         return z3;
50     }
51
52     int main()
53     {
54         complex z1,z2,z3;
55
56         z1 = byModulePhase(10,0.1);
57         z2 = byModulePhase(1,-1.2);
58
59         display(z1);
60         display(z2);
61
62         z3 = sum(z1,z2);
63         display(z3);
64
65         return 0;
66     }
67 }

```

 [\[code/code-III-1-14.cpp\]](#)

Somme comments :

- › Lines 4–14 contain the **declaration** of the new **complex** type, as well as the prototype of all the functions that we will define later.
- › Lines 18–44 contain the **definition** of all the functions related to the type **complex**,
- › The end of the code is an example **main** function.

## ? Quick Application

Download (below), compile, run and analyze the code. Understand what the the code structure and what functions do in order to prepare the next step.

  [code/code-III-1-15.cpp]

Then we will want to start the conversion into object-oriented C++. As we said a class contains both data and functions, starting writing the complex class will lead to :

```
class complex {
    public :
        float Re, Im;
};

// a little main to try
int main() {
    complex z1, z2;

    z1.Re = 3;    z1.Im = 5;
    z2.Re = -8;   z2.Im = 2;

    return 0;
}
```

  [code/code-III-1-16.cpp]

It is that simple : we have written an incomplete class that contains, for now, only data. The fields `Re` and `Im` are called **member data**. Please notice the "public :" word that will be justified later. Of course the class is not finished for now, but at this time it is close to be a pure synonym of the `typedef struct` of the C.

Now we have to add the functions to the class, and it is a little bit subtle. When we work with classes, we have to change the point of view : because the functions will now be part of the class, the functions of the class work as if the member data were global variables. So, if we write a class inside the function that takes even no parameters, it already sees the member data, so here the `Re` and `Im` floats of the class.

## 3.2 Implementation of the `display()` function

A good example should be to re-write the `display` function. We do it like so :

```
1 #include <stdio>
2
3 // declaration of the class
4 class complex {
5     public :
6         float Re, Im;
7
8         void display() ;
9 }; // end of the class declaration
10
11
12 // definition of member functions
13 void complex::display() {
14     std::printf("%f + i%f", Re, Im);
15 }
16
17 // a little main to try
```

```

18 | int main() {
19 |     complex z1,z2;
20 |
21 |     z1.Re = 3;    z1.Im = 5;
22 |     z2.Re = -8;   z2.Im = 2;
23 |
24 |     z1.display();
25 |     z2.display();
26 |
27 |     return 0;
28 | }

```

 [\[code/code-III-2-17.cpp\]](#)

Some comments :

- › lines 3–9 concern the declaration of the class, that is : what data it contains, and what are the names (in fact the prototypes) of all its member functions.
- › lines 12–15 concern the definition of the function `display` of the class.
- › Lines 17–28 shows a main function example to use it.

## ? Quick Application

- › Please download this little code, compile and run it.

 [\[code/code-III-2-18.cpp\]](#)

- › Compare the prototype, line 13, of the `display()` member function, with the declaration line 8. You should remark that, on a syntactic point of view, creating a function in a class uses a similar syntax the one of namespace.
- › Compare the prototype, line 13, of the `display()` member function, with the definition at line 19 of the `complexes.c` file:

From " <b>complexes.c</b> "	From " <b>complexesA.cpp</b> "
<pre> <b>void</b> display(<b>complex</b> z) {     printf("%f + i%f",z.Re,            z.Im); } </pre>	<pre> <b>void</b> <b>complex</b>::display() {     std::printf("%f + i%f",Re,Im); } </pre>

Compare both and see how the context of the class changes the syntax for the C++ version.

- › Now in the main, you may see that the declaration of variables from the `complex` class is the same as with a structure in pure C. However, when a variable is created from a class, the associated vocabulary is "**instance**" : on this example, we say that `z1` and `z2` are "instances" of the class `complex`.
- › now please notice the lines 24–25 : they correspond to calling the `display()` member function of the `complex` class, within the object (or instance) `z1` as data, and then with `z2` as data.

The functions of a class (called "member function") takes the point-of-view of the inside of the class, and work as if the variables of the class (called "data members") were global variables. However : they seem to be global only from the point of view of the member functions, **not** from any other point of view (for example other classes or even the main function). ■

### 3.3 Implementation of the `byModulePhase` function

As you can read in "complex.c", this function converts a number described by (modulus,phase) into a Re,Im description, with as a result a filled `complex` structure. We will write it as a member function of the `complex` class. But, we have to think of it in terms of object-oriented programming : the Re and Im are the ones of the class, and not a complex outside of the class. So, the class is now written like :

```
// declaration of the class
class complex {
public :
    float Re, Im;

    void display() ;
    void byModulePhase(float rho, float theta) ;
}; // end of the class declaration
```

 [\[code/code-III-3-21.cpp\]](#)

Please compare the two prototypes :

> from "complexes.c" :

```
|complex byModulePhase(float rho, float theta);
```

> for the new version :

```
|void complex::byModulePhase(float rho, float theta);
```

Specifically, we notice the return type, that is now `void` instead of `complex`, because it is the Re and Im **inside the class** that will be set.

### ? Quick Application

We will create the "complexesB.cpp" code :

> if necessary, download a clean version of the previous code :

 [\[code/code-III-3-24.cpp\]](#)

> add, as described above, the `byModulePhase` declaration in the `complex` class

> add the `cmath` library to the includes.

> outside the class declaration, convert the `byModulePhase` code from `complex.c` to this new object-oriented implementation.

```
void complex::byModulePhase(float module, float phase) {
    Re = rho * std::cos(theta)
    Im = rho * std::sin(theta)
}
```

> explain why, compared to the function from `complexes.c`, there is no return in this new function.

> now write a `main()` function to test this new member function.



### 3.4 Implementation of `calcModule`

The prototype of the C version of `calcModulus` is :

```
float calcModule(complex z);
```

This is a function that returns a float and takes in parameter a complex `z`. We will do the same in C++, but the parameter `z` will be the contextual complex. The code becomes as follows :

```

1  #include <stdio>
2  #include <cmath>
3
4  // declaration of the class
5  class complex {
6      public :
7          float Re,Im;
8
9          void display() ;
10         void byModulePhase(float rho, float theta);
11         float calcModule();
12 }; // end of the class declaration
13
14
15 // definition of member functions
16 void complex::display() {
17     std::printf("%f + i%f",Re,Im);
18 }
19
20 void complex::byModulePhase(float rho, float theta)
21 {
22     Re = rho * std::cos(theta);
23     Im = rho * std::sin(theta);
24 }
25
26 float complex::calcModule()
27 {
28     return std::sqrt(Re*Re + Im*Im);
29 }
30 // a little main to try
31 int main() {
32     complex z1,z2;
33
34     z1.Re = 3;    z1.Im = 5;
35     z2.Re = -8;   z2.Im = 2;
36
37     z1.display();
38     z2.display();
39
40     float theModulusOfZ1;
41     theModulusOfZ1 = z1.calcModule();
42
43     printf("the Modulus is : %f\n",theModulusOfZ1);
44
45     return 0;
46 }
```

 [code/code-III-4-27.cpp]

### 3.5 Implementation of both `sum` and `product`

Let's examine the prototype of the pure C version of `sum` for example :

```
|complex sum(complex z1, complex z2);
```

This function is a bit different from the two ones that we already wrote : it concerns 3 complex at the same time, and not a single one. Indeed, `display` only printed a single complex, and `byModulePhase` works with a single complex too. And so, the only "point of view" that we can have taken was the one of this single complex number.

But now, with 3 complex numbers, which one will be our point of view ? We can imagine various scenarios. Let's speak of two of them.

#### Scenario #1 :

What we want is to write the code for something like :

```
|complex z1, z2, z3;  
|// ... fill them with values, and then ...  
|z3 = z1.sum(z2);
```

That means that the `sum` function is called from `z1`, and thus `z1` is the point of view from which the code is launched. This way of writing should be the mathematical equivalent of :

$$z_3 = z_1 + z_2$$

Please notice that :

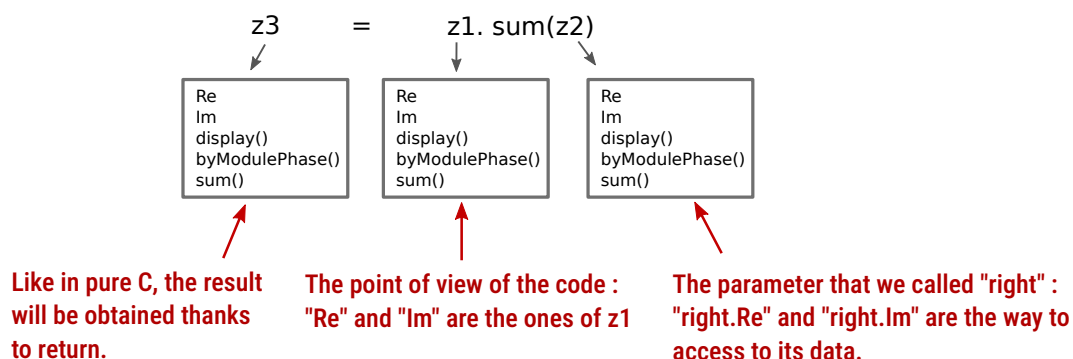
- `z1` is the complex at the **left** of the `+` sign, which means that `z1` is the object from which the `sum` function is executed.
- `z2` is the complex at the **right** of the `+` sign. Concerning the code, it is **an external complex from the point of view of the `sum` function**, and will arrive to the `sum` function as a parameter.
- `z3` takes the **result** of the operation, and will be caught by the return mechanism, like in pure C.

This way of writing also means that the prototype of the `sum` function is :

```
|complex complex::sum(complex right);
```

 [\[code/code-III-5-30.cpp\]](#)

you can identify that what was `z2` at the moment of the call is now a parameter called `"right"` for the `sum` function. **Simply, this scenario assumes that we chose the complex number at the "left" of the `+` sign.** If we do so, we can understand it as follows :



### ? Quick Application

We will now write the code of this function in a new file called **"complexesC.cpp"** :

- If necessary, download the clean code from **"complexesB.cpp"** below

 [\[code/code-III-5-31.cpp\]](#)

- › Declare the new `sum` function in the class,
- › Write the code of the `sum` function.
- › If you did not succeed, here is what you should have written :

```
complex complex::sum(complex right) {
    complex result;
    result.Re = Re + right.Re;
    result.Im = Im + right.Im;

    return result;
}
```

 [\[code/code-III-5-32.cpp\]](#)

- › Now a little main function :

```
int main() {
    complex z1,z2,z3;
    // here fill z1 and z2 with data, and then :
    z3 = z1.sum(z2);
    z3.display();
    return 0;
}
```

 [\[code/code-III-5-33.cpp\]](#)

### Scenario #2:

What we want is to write the code for something like :

```
complex z1,z2;
// ... fill them with values, and then ...
z1.addToMe(z2);
```

to be equivalent to :

$$z_1 = z_1 + z_2$$

Please notice that we changed the name of the function to avoid ambiguity.

## ? Quick Application

In other words, this time we want to directly modify the contents of `z1`, by adding `z2`. We will write that in a new file called "**complexesD.cpp**". You should now be more self-directed :

- › If necessary, download the clean code from "**complexesC.cpp**" below

 [\[code/code-III-5-35.cpp\]](#)

- › Declare the `addToMe` function into the class. It is very important that you think about an adequate prototype.
- › Now write this function and a little main function to test that.

The code for the multiplication should be similar and should be made as homework (look at the end of this chapter).

## ✔ Summary

We have seen the very basic operation of the **class** in C++. A class contains :

- data, called "**member data**", like we would do with `typedef struct` in pure C,
- functions, called "**member functions**" or "**methods**" of the class, similar to regular functions in pure C.
- Here is a sample code for a basic **class declaration** :

```
class myExample {
public:

    int k;
    float m;

    void f1();
    float f2(float p, int n);
};
```

The word "public:" will be discussed in a next chapter.

- The functions `f1` and `f2` are not already defined, and we have to do this with the same `::` operator that we introduced in namespace. This is called "**methods definition**" and can be done as follows:

```
void myExample::f1(){
    // some code here
}

float myExample::f2(float p, int n) {
    // some other code here
}
```

- It is very important to understand that the **class methods see the class data members as if they were global variables** (whereas they are not global, they do not exist outside the class). So for example:

```
float myExample::f2(float p, int n) {
    k = 2*n+3; // changes the value of the k data member
    m = 82.5*p + n; // same comment for m data member

    return m*5.2;
}
```

- And then, once the class is defined, we can create an **instance** or an **object**, which is similar to say, in the world of pure C, that we create a new **variable** based on this class. Once this variable is created, **the access to all the members** (data or functions) **works similarly to what we do with the fields of the structures** in pure C. A quick example:

```
int main()
{
    myExample A,B,C; // we create 3 variables A, B and C
                    //based on the definition of the class myExample

    B.k = 3;
    B.m = 2;

    B.f1();
}
```

```

C.f2();

return 0;
}

```

So C++ brings to us the concept of **class** to incite us to reach a better organization of the code. ■

## Homework

Write by yourself all the necessary code to write the **product** function for the **complex** class that works to obtain:

$$z_3 = z_1 \times z_2$$

and the function **multiplyWithMe** that works to obtain :

$$z_1 = z_1 \times z_2$$

If necessary, download the clean code from "**complexesD.cpp**" below

  [\[code/code-III-5-40.cpp\]](#)

## Homework

Write all that we have written here for complexes to convert the **matrix22** code in object-oriented C++. The original C code to be converted can be downloaded here:

  [\[code/code-III-5-41.cpp\]](#)

One exception: the `byModulePhase` that makes no sense in this context. As a substitution, you may write a function called `buildHomothety` that takes a single float called  $k$  as a parameter, and that sets the matrix to be:

$$\begin{pmatrix} k & 0 \\ 0 & k \end{pmatrix}$$

## Chapter 4 – class constructors and destructors

### Motivations and objectives

An easier subject about classes. C++ allows us to define automated initialization and clean-up when creating variables coming from classes. These mechanisms are called constructors and destructors.

### 4.1 Constructors

You certainly know that, in pure C, creating a variable leads to random contents. For example :

```
int main(){
float k;

printf("Value of k : %f",k);
return 0;
}
```

 [code/code-IV-1-42.cpp]

leads to compilation warning and displays a random value. The C does not force a default value because it has no idea about the value you want : if it would place, for example, a 0 by default in the variable, nothing says that you need this 0. And so, initializing with arbitrary value such as 0 is simply ... time lost, and C does not want to use execution time for nothing usefull. For a single variable of course this time lost should be no problem in lots of cases. But if you work with large arrays or structures, or if this variable is created inside a loop that runs very often, this may impact the execution time. So, even if it is not a secured path for a beginner coder, it is the most optimal choice regarding the computation time consumed, and it this what C always targets.

However in other situations, for things more complex than simply a float, it should be great to have a controlled initialisation. That's what C++ allows to do through the idea of constructors.

Let's place a simple context for a basic example. Imagine we want to define a class to manage points in the 3d space, with 3 coordinates :  $(x,y,z)$  and a name, given by a single letter.

### Quick Application

So let's start. Here is the example :

```
1  #include <stdio>
2  class point {
3      public:
4          float x,y,z;
5          char name;
6
7          void printMe();
8  };
9
10 // members definitions
11 void point::printMe() {
```

```

12     std::printf("(c): %f\t%f\t%f\t", name, x, y, z);
13 }
14
15
16 int main(){
17     point P1,P2;
18     P1.printMe(); // prints awful things
19     return 0;
20 }

```

> Download the code :

 [\[code/code-IV-1-44.cpp\]](#)

> And try it : compile and run.

By default, this point has no given coordinate and also no known name : simply performing the declaration of the variables does not bring to it any reliable values. However, if we need to, we can tell C++ to give reliable values, by default. To do this, we have to define a constructor, and it can be done as follows.

## ? Quick Application

We want now to add a constructor to the class point.

> We use the code just above as starting point. This code is available as "**pointsA.cpp**", that you can download there :

 [\[code/code-IV-1-45.cpp\]](#)

And we want now to build a new code file named "**pointsB.cpp**", that is, at this time, only a copy of "**pointsA.cpp**".

> Please add what misses from the code below in the code you are working on, namely line 7 as well as lines 12–16 :

```

1  #include <stdio>
2  class point {
3      public:
4          float x,y,z;
5          char name;
6
7          point();
8          void printMe();
9  };
10
11 // members definitions
12 point::point() {
13     x=0; y=0; z=0;
14     name = 'A';
15     std::printf("point initialized !\n");
16 }
17
18 void point::printMe() {
19     std::printf("(c): %f\t%f\t%f\t", name, x, y, z);
20 }
21
22 // main function

```

```

23 | int main(){
24 |     point P1,P2;
25 |
26 |     return 0;
27 | }

```

 [\[code/code-IV-1-46.cpp\]](#)

As you can see, the constructor has two specificities :

- It is a function that has the same name as the class,
- This function has **no return type** : not even **void** ! just nothing, and it is the only kind of function that works like this.

➤ Now run the code. You see that whereas we did not call explicitly any function, the terminal displays "points initialized" twice, one for each point. This is the actual evidence that the code coming from `point::point` has been automatically executed.

➤ Now let's add, line 26 in the main, the following two lines :

```

| P1.printMe();
| P2.printMe();

```

 [\[code/code-IV-1-47.cpp\]](#)

And run the code again. You can see the effect of the constructor : we do not have any random data anymore.

However, everything is not so beautiful ... this code displays "A" for the name for each point : it is normal because it is what the constructor says. But not so good. We will do better in the following.

## Remark

There is an alternative syntax to initialise members of a class. The previous constructor can be rewritten using the "constructor initializer list", as follows :

### classical version

```

point::point() {
    x=0; y=0; z=0;
    name = 'A';
    std::printf("point
                initialized !\n");
}

```

### constructor initializer list version

```

point::point()
: x(0),y(0),z(0),name('A')
{
    std::printf("point
                initialized !\n");
}

```

## 4.2 Constructor with parameters

What we have just made in the previous example is writing a "default constructor" : it is a constructor without parameter. But we can also want to give parameters to a constructor, to help it building the object in a more useful way.

## Quick Application



For example, we would like to give a name to the point during the declaration of the variable. Here is how to do this.

- Download the last code called "**pointsB.cpp**" if required :

 [code/code-IV-2-50.cpp]

- Just after line 7 :

- declare a constructor that takes the name of the point as a only parameter,
- declare another constructor that takes the name of the point as well as the 3 coordinates.

Then the class becomes :

```
class point {
    public:
        float x,y,z;
        char name;

        point();
        point(char inName);
        point(char inName, float inx, float iny, float inz);
        void printMe();
};
```

 [code/code-IV-2-51.cpp]

- now we can define the new constructors, for example just above the definition of printMe :

```
point::point() { // HERE is the constructor DEFINITION
    x = 0; y = 0; z = 0;
    name = 'A';
}

point::point(char inName) {
    name = inName;
    x = 0; y = 0; z = 0;
}

point::point(char inName, float inx, float iny, float inz) {
    name = inName;
    x = inx;
    y = iny;
    z = inz;
}
```

 [code/code-IV-2-52.cpp]

- Now change the main function to the following :

```
int main() {
    // please notice the calls to constructors
    point P1('B'), P2('C', 1, 2, 3);

    P1.printMe();
    P2.printMe();
    return 0;
}
```

 [code/code-IV-2-53.cpp]

As you can see, now point P1 calls the one-parameter version of the constructor and P2 calls the 4 parameter version.

- now compile and run the code. Observe the results given by the printMe, and correlate them to the constructor calls.

### Remark

You can see that in this code, there are 3 functions named `point::point`. As you know, this is not possible in pure C to have various prototypes for a single function name. However, in C++, as long as it is not ambiguous, you can create several functions with the same name : C++ uses the parameters to solve a possible ambiguity. This behaviour is thus not specific to constructors, but general with C++ functions or class methods, and will be discussed later. This feature is called "**function polymorphism**".

### Remark

You can see the code is not very optimal : all the constructors we defined contain quite similar code, and this is not a careful way to build code. To make better code, we can call a given constructor from another. That is not a so hot topic of C++ as if you don't know this you can write a regular member function and call it from various constructors. But if you want you can use the constructor call, as follows, by using the ":" operator :

```

1 point::point()
2     : point('A',0,0,0) {
3     // Nothing useful here for this example
4 }
5
6 point::point(char inName)
7     : point(inName,0,0,0) {
8     // Nothing useful here for this example
9 }
10
11 point::point(char inName, float inx, float iny, float inz) {
12     name = inName;
13     x = inx;
14     y = iny;
15     z = inz;
16 }
```

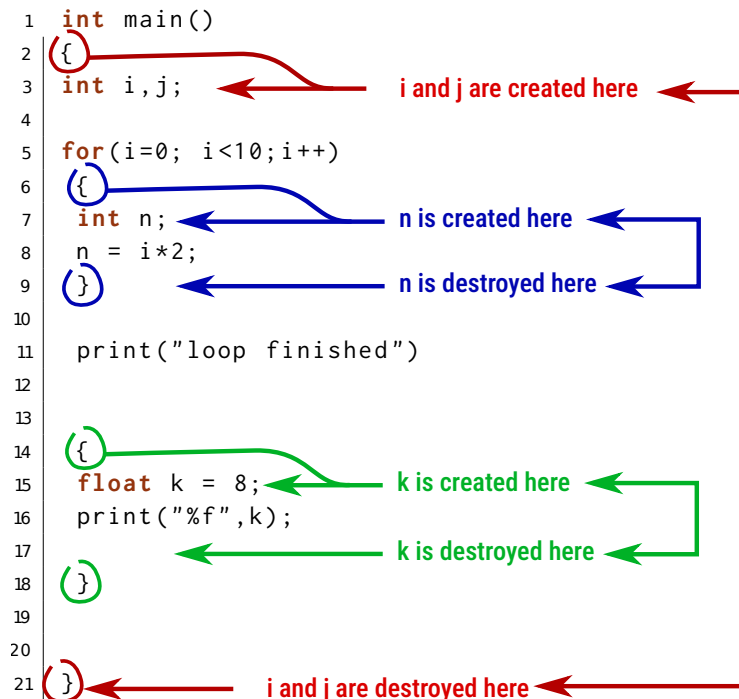
 [\[code/code-IV-2-54.cpp\]](#)

Please notice the call to the 4 parameters constructor thanks to the ":" operator at lines n° 2 and 6.

## 4.3 Destructor

The destructor is symmetrical as compared to the constructor : it is a function that is invoked automatically by C++ when a variable is destroyed. So : when does C++ destroy a variable ? The answer is simple : when the program execution arrives at the `}` that closes the `{` at which the variable has been created. This true whatever the reason that made you create a pair of `{...}` in your code : new function, for, if, while, or even a simple delimited block of code.

For example :



What does "destroyed" variable means ? in pure C, simply "free the memory required for this variable". In C++ : first call the "destructor" function, then free the memory required. So we can define a function that will be called just before the moment at which the object vanishes from memory. This is very usefull : it is to understand like the opportunity to perform some "clean-up". For example, if the object has performed memory allocation, it can perform the deallocation at the level of the destructor, it ensures the memory will be freed at a given step of the global program. Or if the object connects to another computer or device, the destructor can force the connection to close, etc.

## ? Quick Application

We will take an example about files. For the example, we will create an object that "logs" operations in a file. The constructor will be used to open the file, a function will be used to write a single line in the log file, and the destructor will close the file. So let's go : create a file "logFile.cpp". Please try each part of the code below.

> Here is the class itself :

```

#include <stdio>
#include <cstring>
using namespace std;
class logFile {
public :
    // data member
    FILE* f;

    // methods
    logFile(char* name);
    ~logFile();

    void addline(char* line);
};

```

 [code/code-IV-3-55.cpp]

You see the declaration of the destructor line , that is a "tilda" symbol followed by the name of the class.

> We now can write the constructor :

```
logfile::logfile(char* name) {
    print("Opening log file ...");
    f = fopen(name, "w");
    if(f != NULL) {
        printf("file successfully opened.\n")
    }
    else {
        printf("file NOT opened.");
    }
}
```

 [\[code/code-IV-3-56.cpp\]](#)

> Let's write the addline() method :

```
void logfile::addline(char* line) {
    if (f != NULL) {
        fwrite(line, sizeof(char), strlen(line), f);
        printf("one line written\n");
    }
}
```

 [\[code/code-IV-3-57.cpp\]](#)

> now the destructor :

```
void logfile::~~logfile() {
    if(f!=NULL) {
        fclose(f);
        printf("file closed\n");
    }
}
```

 [\[code/code-IV-3-58.cpp\]](#)

> and finally a main function to test :

```
int main() {
    logfile myLog("test.log"); // creates the test.log file on disk

    myLog.addline("program started ...")
    int k = 5+8;
    myLog.addline("intense computation terminated !");

    return 0;
}
```

 [\[code/code-IV-3-59.cpp\]](#)

Please run that code, observe in the terminal that the constructor has been called at the end of the program (thus at the closing "}" of the main). Finally open the file "test.log" on your hard disk. See if everything is consistent.

We have seen the idea of constructor and destructor in C++. These are simply functions that are automatically called by the C++ when necessary :

- the **constructor** is called **automatically** when the **variable is created**. It is a function that has the name of the class. It can also take parameters, and you can have multiple constructor with various parameters for a single class.
- the **destructor** is called **automatically** when the **variable is destroyed**. It is a function that has the name of the class with a tilde ~ added in front. It can **not** take any parameter.

A basic example :

```
class myClass {
public :
    int myNum;

    myClass(); // declare "default" constructor
    myClass(int inValue); // declare constructor with one parameter
    ~myClass(); // declare destructor
    // + other methods
}

// define 2 constructors + the destructor
myClass::myClass() {
    print("creating object ...\n");
    myNum = 0;
}
myClass::myClass(int inValue) {
    print("creating object ...\n");
    myNum = inValue;
}
myClass::~~myClass() {
    print("object being destroyed ...\n")
}

// main function example
int main() {
    myClass C1;
    myClass C2(8);

    return 0;
}
```

 [\[code/code-IV-3-60.cpp\]](#)

We also discussed about the operator ":" that allows to call a constructor from another one (look at remarks in the main text of this chapter), or to initialize the class data members without writing it with an affectation into the constructor code. ■

## Homework

Take back the C++ code from homework 3.5 (page 29). Add a default constructor that takes an int k as parameter in order to set the matrix like :

$$\begin{pmatrix} k & 0 \\ 0 & k \end{pmatrix}$$

## Chapter 5 – Class members rights

### Motivations and objectives

We have seen that object-oriented programming allows to :

- mix code and data in a single element called "object" in order to make functions and data interact in a more natural way, making code easier to write and read,
- define automated initialization and destruction mechanisms for objects,

We now will see how well written object-oriented programs also makes data member of classes safer.

### 5.1 Demystifying the "public" word we already used

In most object-oriented languages, access to class members can be allowed or forbidden, mainly to have control about things that should not be modified in an uncontrolled way. In the previous examples, the code we wrote deactivated this control. Indeed, you have seen this in all the classes that we have written, for example in the following code (*simplified after a previous example*) :

```
1 class point {
2     public:
3         float x,y,z;
4         char name;
5
6         point();
7         void printMe();
8 };
```

  [code/code-V-1-61.cpp]

You notice the "public :" word at line 2, that has never been discussed at this time. Using "public :" that way, we work without any control, that is like structures in pure C language. In the point class, there is no big deal in controlling something : changing x, y or z will not produce any inconsistency in the object. But in other cases, such a control may be very useful. Let's use a new sample code, still from geometry : define a vector class, like in maths :

```
1 class vector {
2     public:
3         float x,y,z;
4         char name;
5
6         vector();
7         float norm();
8         void printMe();
9 };
```

  [code/code-V-1-62.cpp]

So very similar to the point class, except that we want to be able to compute the norm of the vector (*see method declared line 7*). Building this method can be made as follows :

```

1 #include <cmath> // at top of the .cpp file
2 // ...
3
4 float vector::norm()
5 {
6     float n;
7     n = std::sqrt(x*x + y*y + z*z);
8     return n;
9 }

```

 [\[code/code-V-1-63.cpp\]](#)

Okay good. But imagine that the norm function is called very often in the code that uses the class. We would like to avoid to call the function because it uses mathematical expressions that uses CPU. So, if the vector coordinate do not change too often, we would prefer to store the result in the class, and rewrite all like that. That's what we want to do in the following.

## ? Quick Application

Because when the vector changes the norm changes too, we have, this time, to write a function that sets the vector coordinates, and that, at the same time, computes the norm and stores it into the class as normvalue. Then, no need for a norm() function, we only have to look at the field normvalue. So we start with the following code that does all that :

> Download and try this :

```

1 #include <cmath>
2 #include <cstdio>
3 class vector {
4     public:
5         float x,y,z;
6         float normValue;
7         char name;
8
9         vector();
10        void set(float inx,float iny, float inz);
11        void printMe();
12 };
13
14 void vector::set(float inx,float iny, float inz)
15 {
16     // set class data
17     x = inx; y = iny; z = inz;
18     // compute and set norm inside of class
19     normValue = std::sqrt(x*x + y*y + z*z);
20 }
21
22 int main(){
23     vector V1;
24     V1.set(1,2,3);
25     std::printf("norm : %f",V1.normValue)
26
27     return 0;
28 }

```

 [\[code/code-V-1-64.cpp\]](#)

We see that the method "set", defined line 14, changes the vector coordinates and computes the norm and stores it in the object. This allows easy further acces to the norm without any

computation. Kind of "cache" mechanism thus.

- Okay good. So we can use now the value of the norm at 0 computation cost. Nice, that's what we wanted ! But what if I write the following main :

```

1  int main(){
2  vector V1;
3  V1.set(1,2,3);
4  V1.x = 8; // dangerous !
5
6  std::print("norm : %f",V1.normValue)
7
8  return 0;
9  }
```

  [code/code-V-1-65.cpp]

The norm is now no more the good one as the vector coordinates have changed while not making it through the "set" function.

We could think that this problem is the one of the programmer : it is his work to make things work well. On a so simple example, of course it is not critical and the coder can manage it. But imagine a program made by tens of coders, during a long time, or if coders need to make the V. 2 of a software some years later after the first version. This code behaviour is so dangerous (whereas it would have been accepted in pure C, but it's kind of "another age" of coding). C++ can help you avoiding this kind of problem : you can forbid a direct modification of the class "from the outside".

What do we want to say by "from the outside" ? In the code above :

- line 4 changes the value of x whereas it is not made by a method of the class, it is a direct access to it : we say it is made from the outside.
- however, the function "set", called at line 3, makes the changes from the context of the class, as its code shown line 14 demonstrates it : once the "set" function called, everything is made "from the inside of the class".

So in the following, what we want to do is allow the call from line 3, because it's safe regarding the norm, while forbidding the possibility to use something like line 4. And C++ permits this, as we see below.

## 5.2 private access

What we would like would be to forbid the free (public) access to x, y and z and to the norm, because these 4 members are related. That's what we can do with the private word. This is done as follows :

```

class vector {

    private:
        float x,y,z;
        float normValue;

    public:
        char name;

        vector();

        void set(float inx, float iny, float inz);
        void printMe();

};
```



 [\[code/code-V-2-66.cpp\]](#)

And that's all, nothing to change somewhere else in the code. We just say that the access, **from the outside** of the class, is not possible for *x*, *y*, *z* and *normValue*, and this is due to the fact that all these values are related between them.

## ? Quick Application

Now do the following :

- Change the class description as above, try to compile and observe that the compiler does not want to compile this.
- From the main, remove or comment-out the line :  
`V1.x = 8;`  
 And try again. This should not even work because the main still accesses the "normValue" field. What we can do is a read-only access method for this field.
- To solve this, please write a **public** method called :

```
|float vector::getNorm();
```

That simply returns the norm, and then update the main function consequently. Check everything works fine.

So we implemented an interesting mechanism to protect the programmer against bad ideas. This mechanism is not, at first glance, to understand like something that allows to take control about what the coder that uses the object will do, like "punishing" him by forbidding some operations. It is more a way to help the programmer building safer programs, and help him understand, without reading too much code, what is dangerous to change and what is not dangerous. This should be your state of mind with this protection system when programming.

## ! Important

It is **very important** to notice that we always will speak about **allowed or forbidden access as seen from the outside of the class** ! The member functions of the class always have access to all members, whatever their protection status !

*Please notice that this is rigorous "at this time", and can become false when we will speak, later, about a complex topic called "inheritance". But at this time and given what we know, we can think things like this.*

## 5.3 Other things to know

We have seen the most important things. However, you should notice that :

- without knowing it, you wrote something called an **accessor** function : it is a function which only purpose is to permit the access we want to specific data members, to overcome a too strict protection mechanism with private.

```
|float vector::getNorm();
```

- You should **never create a private constructor or destructor** : this will forbid the object to build or destroy itself, and thus will make it completely unusefull.

- C++ also defines a **protected** access. This access is, at this time of the course, exactly similar to private access. The difference will make sense only while speaking, later, about "inheritance". So at this time both are synonyms.
- Last remark : In some cases, it is usefull to make private or protected methods too. The syntax is the same, we do not give any example for that here.

## ✔ Summary

We have seen that C++ allows to manage acces to class fields by two main ways :

- the word **public**: written inside the declaration of the class will make anything declared below freely accessible, without any protection : all can be accessed at anytime, like in C.
- the word **private**: written inside the declaration of the class will make everything **not accessible, if accessed from the outside of the class**. However, methods of the class can freely access these elements, as usual.

We mentionned a third one, the **protected**: way, that is, at this time, exactly the same as private:. Both will be different if we speak about inheritance, and only in this context.

Moreover, we also have seen that in lots of situations, managing protection often requires to define **accessors** methods, that allow to read and/or write the private members, depending on what is required.

Finally, designing a class by taking care of the members that should be protected or not has a name: it is called **encapsulation**. ■

## 💬 Remark

In lots of situations, managing this kind of security in the objects that you build by yourself is not mandatory, as long as you code alone and on small pieces of code. However, when object oriented programmation will become more natural to you, you should manage it even on simple examples in order to develop good habits.

Moreover, you should understand that you will be very quickly concerned by the security embedded in objects made by others, for example coming from libraries. That's why you have to know what it means.

## 🏠 Homework

We give the code from the "Quick Application" from page 35 about the logfile :

🔗 [code/code-V-3-69.cpp]

- First check it works.
- at line 44, add :

```
|         myLog.f = NULL ;
```

🔗 [code/code-V-3-70.cpp]

and recompile and run. See it does not work, perhaps crashes.

- add the protections that do not allow this behaviour by managing the rights on the fields of the `logFile` object. The result should be that the code like that should not compile anymore, while removing this line will let the code work just fine.

## **PART II**

---

### **Some C++ concepts for better code**

## Chapter 6 – Stream operators (and strings)

### Motivations and objectives

Here are discussed mainly two new operators of C++, as compared to pure C++ : stream operators. No big deal here : mainly a pragmatic way to replace printf, scanf and make file operations easier. These operators can be, in lots of cases, related to strings, and C++ gives library to make strings management far easier than pure C did. So it is discussed here too.

### 6.1 Replacing printf

C++ allows to get rid of printf, by using a new object called cout. Here is an example :

```
#include <iostream> // is the library that contains cout
using namespace std;

int main()
{
    cout << "Hello World !\n";
    int i,j;

    i=3;
    j=2*i;

    cout << i << " times 2 is " << j << "\n";

    return 0;
}
```

  [\[code/code-VI-1-71.cpp\]](#)

That displays :

```
Hello World !
3 times 2 is 6
```

As you can see, it is no more necessary to have the knowledge of the "%" formatted strings (like "%s" or "%d" for example) : cout can detect the type of the object to display. It also uses a new operator (in fact it exists in pure C but is related to binary operations only), the "data stream operator" written "<<". This is far better for the programmer and quicker while executing, and so we should not use anymore the pure C printf function.

### 6.2 Managing strings

C++ has a library that allows to manage strings in a more object-oriented way. Here is a quick example:

```
1 #include <string>
2 #include <iostream>
3
```

```

4  using namespace std;
5
6  int main()
7  {
8  string myHello;
9  myHello = "Hello ";
10 cout << myHello << "\n";
11
12 // concatenate
13 myHello += " World !";
14 cout << myHello << "\n";
15
16 // get size (= strlen from pure C)
17 cout << "the string '" << myHello << "' size is : " << myHello.size() << "\n";
18
19 // test strings equality (= strcmp from pure C)
20 if (myHello == "Hello World !") { cout << "the two strings are the same\n"; }
21
22 // update string
23 myHello[4] = ' ';
24 myHello[5] = 'i';
25 myHello[6] = 'n';
26 cout << myHello << '\n';
27
28 return 0;
29 }

```

 [\[code/code-VI-2-72.cpp\]](#)

That displays :

```

1 Hello
2 Hello World !
3 the string 'Hello World !' size is : 17
4 the two strings are the same !Hell in World !

```

## ? Quick Application

Please get the code :

 [\[code/code-VI-2-73.cpp\]](#)

Then test, and play with it.

We see that :

- › line 8 create a string objects, while line 9 sets the contents of the string object thanks to a pure C style string.
- › line 10 simply demonstrates that cout object recognizes string objects, not only pure C strings.
- › lines 13–14 demonstrate the easy concatenation feature simply with operator +=,
- › lines 16–17 shows how to get the size of a string,
- › line 20 shows string test : no more traps with pointer comparizon like in C++, no more need to rely on extra function to perform this test.
- › 23–26 demonstrate the [ . . . ] operator that works like on usual pure C strings

Other goodies exist, please have a look on the internet searching things like "std::string C++" in a search engine.

## 6.3 Replacing scanf

Similarly, scanf can be replaced by the cin object. Here is an example :

```

1 #include <iostream> // is the library that contains cin
2 using namespace std;
3
4 int main()
5 {
6     cout << "Good morning sir ! " << "\n" << "What's your name ? \n";
7
8     string myName;
9     cin >> myName;
10
11    cout << "Hello sir " << myName << "! \n" << "How old are you ? \n";
12    int age;
13    cin >> age;
14    cout << "So you are " << age << " years old \n";
15
16    return 0;
17 }
```

 [code/code-VI-3-74.cpp]

### ? Quick Application

Download the corresponding code, analyse and test it. Nothing difficult by here.

As you can see, it is nice as cout : no more requirement for "%-formatted" syntax, and no more use of address. Great and easy compared to pure C !

## 6.4 Managing files with << and >>

The same syntax as with cin and cout can be used to read or write in files. This can be made with the fstream library.

### ? Quick Application

Please test the following code :

```

1 #include <iostream>
2 #include <fstream>
3 using namespace std;
4
5 int main() {
6     // Create and open a text file to be written.
7     ofstream MyFile("filename.txt");
8
9     // Write to the file
10    MyFile << "Most computer problems come from the keyboard-to-chair
        interface \n Klaus Klages \n";
11
12    // Close the file
```

```

13 |   MyFile.close();
14 | }

```

 [\[code/code-VI-4-75.cpp\]](#)

Run this code and check that it created a file called "filename.txt" in the same path as your .cpp file, then open it with the notepad and see what's inside.

So its very easy :

- line 7 : All you have to do is to instanciate the ofstream object from the standard C++ library while giving the filename to the constructor of the class. It's a bit similar to the logFile class that we wrote at page 35.
- linr 10 : Subsequent uses of the ofstream instance (here called myFile) are simply made with the << stream operator, exactly the same way cout works.
- line 13 : simply closes the file to synchronize pending write operations and permits its use by other processes.

To read from a file, use either the ifstream or fstream class, and the name of the file.

Note that we also use a while loop together with the getline() function (which belongs to the ifstream library) to read the file line by line, and to print the content of the file:

```

// Create a text string, which is used to output the text file
string myText;

// Read from the text file
ifstream MyReadFile("filename.txt");

// Use a while loop together with the getline() function to read the file
// line by line
while (getline (MyReadFile, myText)) {
    // Output the text from the file
    cout << myText;
}

// Close the file
MyReadFile.close();

```

 [\[code/code-VI-4-76.cpp\]](#)

This time, we do not use the >> like with cin, because reading a file requires to understand its structure : in most cases, we do not read a file in a single block but analyze short amounts of data. This is not really adaptated to text-files parsing, but can work for binary files, as we can rely on the data type size.

## Remark

It is clear that this description, about "reading file contents", is a lot simplified, not accurate and not very meaningful. But : understanding deeply that topic assumes everything is clear for you concerning binary and text files, and is not really related to object-oriented programming. In other words, it may require a certain background that, I guess, cannot really be assumed. This could be discussed if necessary, for example during the course or in another context, but it can be a tricky subject depending on your experience of programmation. So the description given here concerns mainly simple text files, a quite usual case.

## Summary




We have introduced the so nice **stream** operators of C++. Even if there is a conceptual background, to us it will simply be an easy and efficient replacement for pure C functions like `printf`, `scanf`, as well as file related operations. These operators are `<<` and `>>`, depending on the direction of the stream relatively to the object on which it is applied. For us, the syntax of the stream operators is nice as it avoids to use the %-formatted strings that come from pure C. A basic example that may replace `printf`:

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

int main()
{
    double a=3.5,b=8;

    cout << a << " at the power of " << b << " is " << pow(a,b);
    // equivalent in pure C to :
    // printf("%f at the power of %f is %f",a,b,pow(a,b));

    return 0;
}
```

  [code/code-VI-4-77.cpp]

## Homework

Rewrite the `logFile` object from page 35 by using the stream operators instead of the functions from `cstdio`. You will have to understand, by documenting yourself on the internet, how to check if the file was opened or not.

## Chapter 7 – Some "+" of C++

### Motivations and objectives

Here are discussed some so nice tools that have been added to C++ as compared to C. These are not really related to object-oriented programming (but you are encouraged to use them in the frame of object-oriented programming too). However they are very nice improvements of the language as compared to pure C.

### 7.1 References on variables

Let's begin with so good news : if you suffered so much about **pointers** in pure C, you will love references. For a single reason : in lots of cases (however not all cases), references are a way to avoid using pointers. Yes, nothing less !

A traditionnal and difficult usage of pointers for early coders in pure C is about functions. Indeed, you certainly learned that a function has as many inputs as you want using the "parameters" part of the function, and a single "official" output (using the return mechanism). To have more than one output, you have to use pointers in the "parameters" part of the function. Then, outputs are said to be "passed by address", whereas inputs are said to be "passed by value" in pure C. This is not an easy task for most beginners. A little pure C example follows: imagine we write a trivial C function that computes the sum and product of two values :

```
1 void myFunctionPureC(float a, float b, float* sum, float* prod) {
2     *sum = a+b;
3     *prod = a*b;
4 }
5
6 int main() {
7     float i,j;
8     myFunctionPureC(5,8,&i,&j);
9 }
```

  [\[code/code-VII-1-78.cpp\]](#)

In this code, in the main, the variable i will take the sum as computed by myFunction, and j the product. "Good"...

This is complex because you have to understand that the sum variable from myFunctionPureC points on i of the main, and respectively prod points j. Then, you use the star operator (which is ambiguous for you because you think it means "pointer" in this context, whereas it means "dereferencing"), that dereferences the pointers to set the pointed values. And when you perform the call, line 8, you have to remember that you must give the address of an existing variable, and not a simply declared pointer. All that lead, for early coders, to head-akes as well as awfully buggy and crashing code.

C++ proposes a so elegant solution to avoid such tricky (but usual) situations : instead of declaring sum and prod as pointers, it allows to define them as **references**, using the symbol & that, in the considered context, does not mean anymore "address", but "reference". Let's have a look at the same code converted to C++ and using references :

```

1 void myFunctionCpp(float a, float b, float& sum, float& prod) {
2     sum = a+b;
3     prod = a*b;
4 }
5
6 int main() {
7     float i,j;
8     myFunctionCpp(5,8,i,j);
9 }

```

 [\[code/code-VII-1-79.cpp\]](#)

So to define that a parameter is an output for a function, for now, all you have to do is to declare it as a reference ! isn't that so nice ?

## ? Quick Application

- Write down in a cpp file the myFunctionPureC function and the associated main, and test that it work. Please do not copy the code in a "dumb" way, but please remember the underlying mechanism of pure C, as described above.
- Then change the declaration of the two pointers into references, compile to see errors, then update the code so that everything works fine, using now references.

## 7.2 Polymorphic functions

In C++, and contrary to pure C, you can write functions with the same name but with different parameters types. This is possible as long as the call is not ambiguous. A quick example :

```

// multiply and accumulate : used in digital filters
void mac(int& a, int b,int c) {
a += b*c;
}

void mac(float& a, float b,float c) {
a += b*c;
}

// not used with complex numbers in the real world,
// but it's just to show
// assumes the complex class we wrote exists.
void mac(complex& a, complex b,complex c) {
a = a.sum(b.product(c));
}

```

 [\[code/code-VII-2-80.cpp\]](#)

So here we defined 3 times the "same" function (same name), that makes the "same" operation, but depending on the type. Whereas not allowed in pure C, it is common in C++. However, be careful as some values can be automatically converted by C++ (for example int to float) in some cases. This should lead to calling the version of the function that you do not expect if the syntax seems to be, at first glance, ambiguous (it is in fact never ambiguous for the compiler, or if it is it drops a compilation warning or error). The fact to define various functions with the same name is called **polymorphism** : i.e. the function takes various (poly) "forms" (morph).

## 7.3 Overriding operators

It's not a fundamental subject, but this section may help you to understand why "lots of things" are that intuitive in C++ as compared to pure C. It is, in a way, something in the same state of mind to polymorphic

functions. For example the usage of operators like += or == with strings (which is not possible with pure C and leads to sometimes a little bit complex considerations about pointers), or with the >> and << stream operators, or the ++ – and \* operators on the iterator from the template libraries.

In fact, C++ allows the programmer to redefine all the operators, depending on the type of the objects on which they are used.

For example, in the very beginning, we wrote the complex class, with the sum member function for example (page 26), as follows :

```
class complex {
    public :
        float Re, Im;

        void display();
        // other stuff ....

        complex sum(complex right);
};
```

This code makes that if we want to perform a sum of complex numbers, we will have to write :

```
int main()
{ complex z1,z2,z3;
  // some stuff
  z3 = z1.sum(z2);
  return 0;
}
```

And we would really want to write :

```
int main()
{ complex z1,z2,z3;
  // some stuff
  z3 = z1 + z2;
  return 0;
}
```

 [\[code/code-VII-3-83.cpp\]](#)

And this can be done by writing :

```
class complex {
    public :
        float Re, Im;

        void display();
        // other stuff ....

        complex operator+(complex right);
};

complex complex::operator+(complex right) {
    complex result;
    result.Re = Re + right.Re;
    result.Im = Im + right.Im;
    return result;
}
```

 [\[code/code-VII-3-84.cpp\]](#)

So it's the same code as sum, just it's inside the "operator+" function, and it now defines the meaning of syntax like `z3 = z1 + z2`; Great, isn't it ?

We only show this so that you know it exists and why lots of objects related operations can be written in a so intuitive way. It is not targeted that you know this as it can sometimes behave surprisingly for a beginner. You of course can play with writing your own operators redefinition, but you should have in mind that it can sometimes lead to ambiguous situations. So even if it seems simply nice, in some contexts it can become a tricky subject. However, lots of high performance libraries define this kind of operator, to enable more intuitive code.

## 7.4 Polymorphic functions

In C++, and contrary to pure C, you can write functions with the same name but with different parameters types. This is possible as long as the call is not ambiguous. A quick example :

```
// multiply and accumulate : used in digital filters
void mac(int& a, int b,int c) {
    a += b*c;
}

void mac(float& a, float b,float c) {
    a += b*c;
}

// not used with complex numbers in the real world,
// but it's just to show
// assumes the complex class we wrote exists.
void mac(complex& a, complex b,complex c) {
    a = a.sum(b.product(c));
}
```

 [\[code/code-VII-4-85.cpp\]](#)

So here we defined 3 times the "same" function (same name), that makes the "same" operation, but depending on the type. Whereas not allowed in pure C, it is common in C++. However, be careful as some values can be automatically converted by C++ (for example int to float) in some cases. This should lead to calling the version of the function that you do not expect if the syntax seems to be, at first glance, ambiguous (it is in fact never ambiguous for the compiler, or if it is it drops a compilation warning or error). The fact to define various functions with the same name is called **polymorphism** : i.e. the function takes various (poly) "forms" (morph).

## 7.5 Automatic type specification

In various situations, the declaration of the type of a variable can become automatic. Let's take a simple example :

```
#include <iostream>

int main()
{
    int i = 5;

    std::cout << i;
}
```

 [\[code/code-VII-5-86.cpp\]](#)

In this simple code, the type of i is int because it is declared. But it has to be compatible with the type of the value 5, which is an int too. And so C++ could have "guessed" that we wanted i to be an int. We can ask C++ to guess this, with the keyword auto :

```
#include <iostream>

int main()
```

```
{
    auto i = 5;

    std::cout << i;
}
```

 [\[code/code-VII-5-87.cpp\]](#)

In this code, `int` was replaced by `auto`, and `i` is automatically an `int` because it is implicit, due to the type of the value 5.

We can ask "why is it interesting?". At this time not a lot. We can mention that if we write :

```
#include <iostream>

int main()
{
    auto i = 5.0;

    std::cout << i;
}
```

 [\[code/code-VII-5-88.cpp\]](#)

then `i` will become a **double**. But it is not a big deal. We will speak again of the `auto` feature later, while speaking of templates.

## Summary

## Homework

We give the pure C code, that decomposes a total time in seconds into days, hours, minutes and remaining seconds, as follows :

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int totalTime, days, hours, min, sec, r1, r2;
6
7      days = totalTime / (24*60*60);
8      r1 = totalTime % (24*60*60);
9      hours = r1 / (60*60);
10     r2 = r1 % (60*60);
11     min = r2 / 60;
12     sec = r2 % 60;
13
14     printf("%d seconds is %d day, %d hours, %d minutes and %d seconds\n",
15           totalTime, days, hours, min, sec);
16
17     return 0;
18 }
```

 [\[code/code-VII-5-89.cpp\]](#)

We want to transform this pure C code into C++ style. To do so :

- From this code, build a **function** that takes as input the total time in seconds, and uses references to pass the 4 results (days, hours, minutes and remaining seconds) to a main function, that you will have to write too.

- of course, make the main function display the results by means of `cout`.

## Chapter 8 – The `const` Keyword

### Motivations and objectives

C++ syntax can help programmers to identify, in an already written text, values that are modifiable or not. This relies on the `const` keyword. This identification in the code is to understand as a good practice when creating programs, as it helps a lot the code readability, thus its reliability. But things are a bit subtle sometimes. That's what we want to mention here.

### 8.1 Constant value

This topic is not a very big deal but you have to know because it's very common. C++ can protect some variables against writing, even while not using the `private` or `protected` statements in a class. Indeed, you can do that for example in a simple variable in the `main`. It targets at defining "constants". It's simple, it simply uses the modifier `const`, as follows :

```
int main()
{
    int k=8;
    const int j=8;

    k=3; // ok allowed
    j=2; // not allowed, compilation error.

    return 0;
}
```

  [code/code-VIII-1-90.cpp]

### Quick Application

Try the code above in a single example, notice that the compiler refuses the compilation, then remove line 7 and compile again.

This kind of thing, like `private` and `protected` do, are made to encourage a coder that reads the code to have good practices regarding some variables. Indeed, a coder that reads a code cannot easily know if it is dangerous to change the value of a given variable. So this is mainly made for code readability, to help coders to better understand existing code, and use it in a cleaner way.

### 8.2 Constant parameter for a function

A function can have parameters passed by value, by address or by reference. In C++, in most cases, parameters should not be passed by address, since using references is easier and thus leads to less bugs.

So most function parameters may be values or references. But there are various arguments to pass a reference to a function :

- › We want the parameter to be an output



- › We want to speed up the function call, by avoiding a copy
- › We don't want the constructor to be called once again. Indeed, copying a parameter leads to side effects as the constructor is called again. For example, imagine a constructor that opens a file, copying the object may try to open again the same file, and this is not wanted

So, in various scenarios, parameters passed by reference are not modified by the function. We can use the keyword `const` to show this in the function prototype. Let's take a small example :

```

1 #include<iostream>
2 using namespace std;
3 class color {
4 public :
5     int R,G,B;
6 };
7
8 void mix(color& first, color& second, color& result) {
9     result.R = (first.R + second.R) /2;
10    result.G = (first.G + second.G) /2;
11    result.B = (first.B + second.B) /2;
12 }
13
14 int main()
15 {
16     color C1,C2,C3;
17     C1= {255,0,255};
18     C2= {0,255,255};
19     mix(C1,C2,C3);
20
21     cout << C3.R << " " << C3.G << " " << C3.B << endl;
22
23     return 0;
24 }
```

 [\[code/code-VIII-2-91.cpp\]](#)

If we read the code of the function `mix`, we can see that only the `result` parameter is modified : the two others are not. And so we can inform the compiler that `first` and `second` are not modified at the level of the prototype, by adding `const` at the level of their declaration. And so the function can be rewritten as follows :

```

void mix(const color& first, const color& second, color& result) {
    result.R = (first.R + second.R) /2;
    result.G = (first.G + second.G) /2;
    result.B = (first.B + second.B) /2;
}
```

Of course, if we add a `const` to `result`, we get a compilation error, as the `result` variable is modified by the code.

But why is it useful ? It is something like : the code is self-documented. You don't need extra comments to see, simply with the prototype, that `C1` and `C2` of the `main` are not modified. Without `const`, you can think that if you use the function, can modify `C1` and `C2`, and to preserve them you can think about making a copy of them before invoking the function.

So using `const` is a good practice as it helps documenting the code, while the compiler check the consistency of this documentation.

## 8.3 "Constant method" of a class

There is a last usage of `const` : methods that do not modify a class. Let's take a small example, in which we add a `display` function to the previous class `color` :

```

#include<iostream>
using namespace std;
class color {
public :
    int R,G,B;

    void display();
};

void
color::display() {
    cout << R << " "<<G<<" " << B<<endl;
}

int main()
{
    color C1;
    C1= {255,0,255};
    C1.display();

    return 0;
}

```

 [\[code/code-VIII-3-93.cpp\]](#)

Once again, we can notice the user of the class that the **display** function does not change the members of the class (it is normal, they are only displayed), by adding **const**, as follows :

```

#include<iostream>
using namespace std;
class color {
public :
    int R,G,B;

    void display() const;
};

void
color::display() const {
    cout << R << " "<<G<<" " << B<<endl;
}

int main()
{
    color C1;
    C1= {255,0,255};
    C1.display();

    return 0;
}

```

 [\[code/code-VIII-3-94.cpp\]](#)

## ✔ Summary

The **const** keyword means, in a way "constant", but it is a little bit more subtle than that. The typical applications of this keyword are (see the main text for the syntax) :

- > variables (the easy case),

- references for non-modified parameters,
- methods of a class that do not modify the class members.

## Chapter 9 – "Template"-based C++ libraries

### Motivations and objectives

It is an incredible strength of C++ : its standard libraries. These libraries contain classes to help data organization, which are probably the quickest and most efficient that you could find in any programming language. For most of them, it is due to the fact that they are based on a concept that exists almost only in C++ : the template.

In this chapter, we will not learn how to create "template classes" by ourselves, because it is a little bit too technical and the syntax is a few ugly for that. However, we will learn how to use classes based on templates, which is easy concerning the syntax, and enables to use these wonderful libraries.

### 9.1 Overview

C++ embeds lots of high quality library that give access to lots of features : Memory management (RAM), error handling, strings handling, containers, files, multitask, input/output, etc. There are so many that we will not enumerate them here, all the necessary information is easily available on the internet.

Lots of these libraries use the C++ concept of "template". Templates are nearly a specificity of the C++ language (most other languages do not implement this), and this is heavily powerfull to write quick and efficient code easily.

### 9.2 Limits of pure C approach with types

Let's start with a pure C example. We want to build a function that creates an array of float. We will write the trivial following function :

```
#include <stdlib.h>
float* createTabFloat( int n){
    return malloc(n*sizeof(float));
}
```

 [\[code/code-IX-2-95.cpp\]](#)

Ok good. But imagine, now we need to perform the same operation, exactly the same, but with int instead of float. In pure C, you have no other solution than writing the same function with a new prototype and name, as follows :

```
#include <stdlib.h>
int* createTabInt( int n){
    return malloc(n*sizeof(int));
}
```

 [\[code/code-IX-2-96.cpp\]](#)

All you did is a copy/paste (that we don't want to do !) and replace all float by int ... Ugly, isn't it ?

And now imagine we want to do it with chars, shorts, etc, and imagine you have to do this for various functions/algorithms. This makes a huge amount a functions to build, to copy/paste, and thus lots of code to maintain for bad reasons.

In C++, there are tools to generate automatically the variations that you need for a given function, generically for any type or class : this is what templates enable.

## 9.3 How C++ Templates solve this kind of problem : example with vector

At this point, we will not learn how to create templates : in most cases it is not that useful in "everyday C++ coder life". Indeed C++ contains beautiful libraries that contain most common useful things that you may want to template, and thus simply learning to use them is enough. Moreover, the quality of these libraries is that ultimate that you will never be able to write it more efficiently or smartly : it is simply the "top of the top" of the high quality code, easily available, and enforces C++ in its position of efficient language. Nearly nothing is more efficient than the template classes of C++ for the purpose they have been designed for. And moreover it's easy to use. So ... let's try it !

### ? Quick Application

We will start with an example. In the template-based libraries of interest, there is the vector library. A vector is simply a general-purpose, template-based dynamic arrays, which are called **vector**. They replace, with lots of advantages, the dynamic allocation of **malloc** in most usual cases. So it makes the code much easier to read and write, and also more reliable.

Here is a first sample code with C++ vectors. You should download and test it :

```

1  #include <vector>
2  #include <iostream>
3  #include <cstdlib>
4  using namespace std;
5
6  int main() {
7
8  // create two empty dynamical arrays of int
9  vector<int> tab1, tab2;
10 // create a single dynamical array of float
11 vector<float> tab3;
12
13
14 // work with tab3 : make it a table of 50 float
15 tab3.resize(50);
16 // similar work with other tab :
17 tab1.resize(10);
18 tab2.resize(20);
19
20
21 // check its size with the "size" method
22 cout << "size of the table : " << tab3.size() << endl;
23
24
25 // fill the array : same syntax as pure C !
26 int k;
27 for(k=0; k<50; k++) {
28     tab3[k] = float(rand())/RAND_MAX;
29 }
30
31 // check contents
32 for(k=0; k<50; k++) {
33     cout << tab3[k] << "\n";
34 }
35
36

```

```

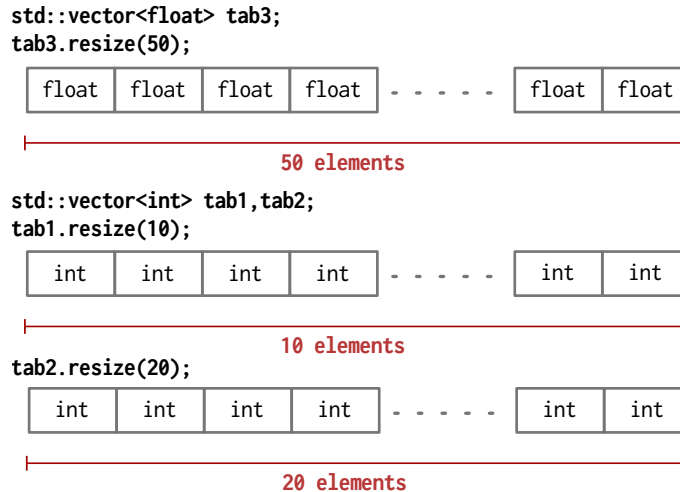
37 |return 0;
38 |}

```

  [code/code-IX-3-97.cpp]

Some comments :

- line 1 performs the inclusion of the vector library.
- at lines 10 and 12 you see typical usage of template class. The typical syntax is :  
`classname<typename> variablename;`  
 The fact that the vector class itself is a template is identified by this `<...>` syntax after the class name.
- Line 10, it is written in C++ that "tab1 and tab2 are vectors of int", in other words, dynamic arrays for which each element is an int.
- Similarly, line 11 says "tab3 is a vector of float", thus a dynamic array of float.
- At this level the dynamic array size is not mentionned and thus these arrays are arrays of 0 elements. In fact, contrary to pure C, template dynamic arrays (vectors) can easily be resized (but you should avoid to do that too many times, other C++ data organizations are better for that). And creating an array of 50 elements consists, as written in line 16, in resizing it with the amount of elements as parameter. For that purpose, we use the `resize()` member function of the vector class. Similar job is performed lines 18 and 19. You can imagine what does the code up to this line as follows :



- Line 23, we use the member function `size()` to check the vector size (= number of elements).
- All the code after line 23 is exactly what would be written in pure C : indeed, the `[...]` operator is available on vectors, it is part of the C++ "magic", not discussed in this course.

As we can see, C++ gives to us a beautiful tool through templates : it gives to us the possibility to use, with a single template-class called "vector" all the management of dynamic arrays, whatever the type of the elements inside. This is a major strength of C++ as the resulting code we wrote is close to be as quick as an array managed by ourselves with pure C. And this is only an example, lots of other data organizations are possible with that, as we demonstrate in the following.

 **Remark**

A word about performance : vectors are a really quick and pragmatic way to manage arrays of memory. Accessing elements is as quick as accessing to a "malloc'ed" array in pure C. And it is the same thing : the vector template is a way to perform dynamic memory allocation (thus **heap** allocation), but the C++ way. But you also have advanced functionalities, like resizing, easily available.

Since 2011, C++ also gives another type to manage static arrays (thus **stack** allocation) : the array type. This one is as quick as pure C static array whereas relying on the practical programming interface given by templates. Here is a basic sample code to create a C++ template array:

```
#include <array>

int main()
{
    // creates an array of 50 float called tab3, like in previous example
    array<float,50> tab3;
}
```

 [\[code/code-IX-3-98.cpp\]](#)

Once this variable is created, all that we have written for the tab3 vector in the previous example, except `resize()`, works the same way, with the same level of performance as static arrays in pure C. Because they are static, they can be created even without operating system (it is not true for **vector**). However please notice that they **cannot be resized**, and they should not be used in case of large memory amount or you may experience "stack overflow" crash of the code.

## 9.4 Iterators

With **vectors** and **arrays**, we have seen that it is possible to use the usual `[...]` syntax to access to the elements of the vector. This is good and easy in the case of vectors. However, this is not a systematic way to use the template-classes libraries : with some other types, this `[...]` will not be available, for good reasons that will be discussed later, and we will have to work with another concept called **iterator**. Iterators are a way to create something like the counter for the index on the elements, like the  $k$  variable in the previous code, but in a more abstract way. Here is, on an example, how it works.

### ? Quick Application

Let's rewrite the previous code with an iterator. Download and try the code.

```
1 #include <vector>
2 #include <cstdlib>
3 #include <cstdlib>
4
5 using namespace std;
6
7 int main() {
8     // create two empty arrays of int
9     vector<int> tab1,tab2;
10    // create a single array of float
11    vector<float> tab3;
12
13    // work with tab3 : make it a table of 50 float
14    tab3.resize(50);
15    // similar work with other tab :
16    tab1.resize(10);
17    tab2.resize(20);
18
19    // check its size with the "size" method
20    cout << "size of the table : "<< tab3.size() << endl;
```

```

21
22 // fill the array : same syntax as pure C !
23 vector<float>::iterator k;
24 for( k=tab3.begin() ; k != tab3.end() ; k++) {
25     *k = float(rand())/RAND_MAX;
26 }
27 // check contents
28 for( k=tab3.begin() ; k != tab3.end() ; k++) {
29     cout << *k << endl;
30 }
31 return 0;
32 }

```

 [\[code/code-IX-4-99.cpp\]](#)

Somme comments :

› up to line 21 everything is the same. So the creation of the vectors is made exactly the same way.

› Now have look at line 23 :

```
|vector<float>::iterator k;
```

This declares an "iterator", that is something able to look at elements of a template class from the C++ library, and this iterator will be called *k*. This iterator is specific as it is made to look at `vector<float>` arrays. That's why the syntax seems to be that complex.

*Remark: Of course, at this time using iterators seems to be only unusefull and complex as compared to the int counter that we used above. But this more general way will be usefull later, and is easier to understand on something as simple as a vector.*

› Then, line 24, the for loop :

```
|for( k=tab3.begin() ; k != tab3.end() ; k++) {
```

As with any loop over an array, you need a beginning, an end and a step, like for any counter. That's exactly what we look at, we use the `begin()` function to make the iterator *k* look at the first element of the array when the loop starts, we test if the iterator does not reach the end of the vector with the `end()` function, and increase the step by one at each step using `k++`.

*Remark : Please remember that it is not because we use the `k++` that *k* is an int ! The operator `++` is available on other types and classes than simply on int in C++ : it has been redefined (as we have seen §7.3) for **iterators** to behave the good way.*

› Finally, the access to the elements, line 25 :

```
|*k = float(rand())/RAND_MAX;
```

It is here that we really see, in the syntax, how should iterators be understood : they should be considered like "kind-of" pointers on the elements of the vector<sup>1</sup>. For that reason, we use, line 25, the star-syntax of pointer on *k*, whereas *k* is not really a pointer but an iterator. Even if *k* is not a real pointer, what is important is that the star syntax `*k` gives access to the element we are working on.

## 9.5 The `list` type in C++ standard library

We have seen that what makes vector interesting is that we do not have anymore to really think about the memory management : we only define a vector, choose the size and it's done.

<sup>1</sup>Again, in depth, the deferencing operator `*` has been redefined in the **iterator** class



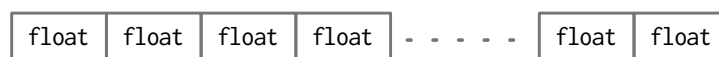
But arrays are not always the good data model for a given problem. In various circumstances, when insertions of elements are frequently required typically, we should better choose the so-called **linked-lists**.

## Remark

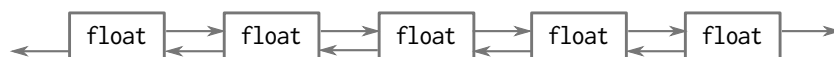
The important differences are :

- as vectors use successive elements in memory, it is easy to locate any of them, and thus what is called "random-access" (i.e. accessing any case at any location at any moment) is quick. However, because elements are successive in memory again, increasing its size is not always possible at low cost (in time or memory) and can lead to a full recopy of the vector. This should thus be avoided.
- lists are made for insertion. It is completely due to the fact that elements are not consecutive in memory, but referenced successively, forming a chain of elements, each being anywhere in memory. This is the reason that makes that elements should be accessed sequentially and not randomly, and thus justifies the fact that the `[...]` operator is not available.

### case of vectors



### case of lists



People here that tried to implement linked-lists in C know it's not an easy thing. Moreover, they should know that if we change the type of the element, all the code of the linked-list should be re-written. Standard template classes of the C++ solve this in a very smart and efficient (quick code generated) way.

## ? Quick Application

Again, let's start with a small code, download and test the following :

```

1  #include <list>
2  #include <iostream>
3  #include <cstdlib>
4
5  using namespace std;
6
7  int main() {
8  // create a linked-list of float
9  list<float> myList;
10
11 // and add some data randomly
12 int m;
13 for (m=0; m<10; m++) {
14   myList.push_back(float(rand())/RAND_MAX);
15 }
16
17 // check its size with the "size" method
18 cout << "size of the list : " << myList.size() << endl;
19
20
21 //check its contents : same syntax as vector !

```

```

22 list<float>::iterator k;
23 for( k=myList.begin() ; k != myList.end() ; k++) {
24     cout << *k << endl;
25 }
26
27 return 0;
28 }

```

  [code/code-IX-5-103.cpp]

Some comments :

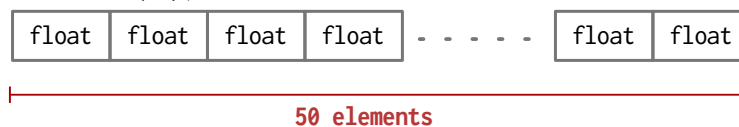
- › Whereas the code is not that different, the data organization of lists is really different, as compared to vectors :

#### case of vectors

```

std::vector<float> tab3;
tab3.resize(50);

```

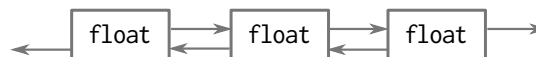


#### case of lists

```

std::list<float> myList;
myList.push_back(5.1);
myList.push_back(2);
myList.push_back(3);

```



- › in both cases, we simply invoke the template class by a similar way. In the case of the vector, we resized to define the array. With the list, we use `push_back()` to add dynamically an element at the end. This is done at lines , 12–15 in the code, that adds random values in the list, 10 times. You can have also `push_front()` to add at the beginning of the list, and there are also ways (not described here) to add an element anywhere.

*Remark: `push_back()` also exists with vector but was not mentionned. However, even if `push_back()` works with vectors, the syntax `[...]` is not available for lists.*

- › Then, line 18 you can get the size in the same way as with vectors.
- › And lines 22–25 a loop which is the same as with arrays, except that the iterator is an iterator for lists and not for vectors (line 22).

And that's all. At this level, you understand that this C++ library is very powerfull and easy to use, as the code is very similar from a kind of data organization to another, whereas the underlying mechanisms are totally different. That is a part of the efficiency of object-oriented programming, combined with templates-programmation, and with the very clean concept of iterator too.

## 9.6 Lighter iterator syntax with `auto`

Whereas working with iterators is very smart, the `iterator` syntax is quite heavy. We have seen in §7.5 that we can ask C++ to determine by itself the type of a variable if it is not ambiguous in the context, typically because we make an affectation to it while created. That's what we can do here. Indeed, lines 22—25 of the previous code can be rewritten like so :

```

auto k = myList.begin();
for( ; k != myList.end() ; k++) {
    cout << *k << endl;
}

```

In this syntax, as `tab1.begin()` returns a `list<int>::iterator`, the type of `k` is automatically `list<int>::iterator`. It is more interesting than above as this type is more complex to write. Moreover, if we update the code so that `myList` becomes for example a `list<float>`, the type of the iterator will adapt itself without rewriting it. It is thus a good and reliable way to write code, not only a way to shorten the syntax.

Far better isn't it ? But there is even better, as we see below.

## 9.7 Easier loops range based loops

The syntax of iterators is a few complex and requires to write a lot. We can make much better with "range based loops" of C++ 2011. Let's see that on a simple example with vectors :

```

std::vector<float> v;
v.push_back(10);
v.push_back(20);
v.push_back(30);
for (float x : v) {
    std::cout << x << std::endl;
}

```

 [\[code/code-IX-7-105.cpp\]](#)

please notice that the loop :

```

for (float x : v) {
    std::cout << x << std::endl;
}

```

Is equivalent to :

```

float x;
for (vector<float>::iterator it=v.begin(); it!=v.end(); it++) {
    x = *it;
    std::cout << x << std::endl;
}

```

So nice isn't it ! ? And we can do the same with all "iterable" types from `std`, for example with lists :

```

std::list<float> myList ;
myList.push_back(10);
myList.push_back(20);
myList.push_back(30);

for(float i: myList) {
    std::cout << i << " ";
}

```

 [\[code/code-IX-7-108.cpp\]](#)

## 9.8 Other data organizations coming from C++ libraries

There are lots of other template-based data organizations available with similar way of working that come either from standard C++ library or from third-party ones. These are not described in this documents, but you may find various kinds of trees, maps, stacks, etc.

## 9.9 Template linear algebra library: Eigen

There is also a library that you may use in the frame of various scientific / technical works and typically in robotics, for linear algebra : **Eigen**, which is a emplate library too. We show below a basic example of

usage of Eigen :

```

1 #include <iostream>
2 #include <eigen3/Eigen/Dense>
3 using namespace std;
4
5 int main()
6 {
7   Eigen::Matrix<double, 3, 3> m = Eigen::Matrix<double, 3, 3>::Random();
8   Eigen::Matrix<double, 3,1> v= {1,0,0};
9   cout << m << endl;
10  cout << v << endl;
11  cout <<m*v <<endl;
12  return 0;
13 }
```

 [\[code/code-IX-9-109.cpp\]](#)

You can see in this code various points :

- › The **Matrix** class requires the specification of the type of the values, the amount of lines and the amount of columns in the template <.....> syntax. That means that the code for m and v will be written in a very "static" way at compilation time, as if we would have written specific code for the specific case of 3x3 matrixes of **doubles** (which is not the case : we only used a template !). This makes computation performance very very quick as compared to a more generic code.
- › Please notice the **\*** operator between the 2 matrixes at line 11, as well as the **<<** operator at the level of the **cout** with matrixes again. That means that the class **Matrix** has redefined **operator\*** and **operator<<** to allow writing a more natural code.

## ❖ Summary

We have shown some aspects of C++ standard libraries, the ones that rely on the template concept. This concept allows C++ to generate classes, here used to manage data structures, specialized for a given type (or class), which boosts a lot performances compared to other object-oriented languages. In practical terms, C++ libraries give to us high performance and flexible data-organization systems, for everyday life. We have seen :

- › **array** : fixed size array with similar performances as pure C allocated RAM areas,
- › **vector** : flexible, resizable array with good level of performance
- › **list** : highly flexible linked-list (so quick for this kind of data organization)

All these data-organization related classes are called **containers**.

Using array or vector replaces easily and nicely all the allocation/deallocation features of pure C like calling malloc and free, or code that would work with static allocation. Using list gives an incredibly easy access to linked-lists, as compared to pure C.

These containers are, in depth, very different. However, C++ libraries give a unique interface to run through these structures, called **iterator**. You should think of an iterator like a kind of abstract pointer, that supports the same operators as pointers (**++** - and **\***) to reach a similar functionality, but with a systematic syntax whatever the data structure on which they operate.

A basic example about vector :

```

#include <vector>
using namespace std;

int main()
```

```

{
    vector<float> myVector;
    myVector.resize(35);

    vector<float>::i;
    for(i=myVector.begin(); i != myVector.end(); i++) {
        *i = 15*i+3;
    }
    return 0;
}

```

  [code/code-IX-9-110.cpp]

## Homework

- > Write a code that stores in a vector 20 random values.
- > Then, "reverse" these values in another vector, that is this new vector should contain exactly the same values as the original one, but in reverse order.
- > Of course print all the values of the two vectors.
- > rewrite these 3 questions with list as type. Notice that if you think a little bit about how linked-lists are different from vectors, this version could be much easier.

## Homework

A little bit more complex, but very doable. We start with the following code, that intends to manage 2-dimension signals, with equal amount of points in  $x$  and  $y$  directions :

```

int main()
{
    vector< vector<double> > signal2d;
    int dimension = 10;

    return 0;
}

```

  [code/code-IX-9-111.cpp]

- > write the usefull code to make this signal2d variable a  $dimension \times dimension$  signal.
- > We want to put the values of a 2d gaussian centered at the  $(x_0 = dimension/2, y_0 = dimension/2)$  coordinates, and with a  $\sigma = 3$  for both axes. The formula of a 2d gaussian is :

$$f(x,y) = \exp \left[ - \left( \frac{(x-x_0)^2}{2\sigma_x^2} + \frac{(y-y_0)^2}{2\sigma_y^2} \right) \right].$$

So let's do it.

- > Then display the data with printf so that each line is a line of the signal. The result is probably good if you see values close to 1 at the center of the data, while decaying when reaching left, right, up and down.

- Copy/paste from terminal to excel or libreOffice calc and display the result. Or : Copy/paste in a text file and import it with Matlab or Python, then display it as an image.

## Chapter 10 – Errors management : try-throw-catch

### Motivations and objectives

When we manage errors in a code, it makes the readability of the error-free path more difficult. The **try-throw-catch** mechanism allows to solve this problem.

### 10.1 Classical error management in a C/C++ code

Imagine you have to read the contents of 2 files because working on the data inside. You will write a code like so :

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

int main()
{
    ifstream File1("myFile1.txt");
    if(File1.is_open())
    {
        // then work with the file
        //
        //      .....

        ifstream File2("myFile2.txt");
        if(File2.is_open())
        {
            //then work with the file
            //
            //      .....
        }
        else {
            cout << "can't open myFile2.txt !\n";
            cout << "exiting with error !\n";
            return -1;
        }
    }
    else {
        cout << "can't open myFile1.txt !\n";
        cout << "exiting with error !\n";
        return -1;
    }
    return 0;
}
```

 [code/code-X-1-112.cpp]

We had to add a lot of **if ... else** and this makes the code complex. With the **try-throw-catch** mechanism, you can write the code as if there were no error, and then, in a separate part of the code,

manage these errors. This mechanism is to use with this kind of error, which are kind of "fatal" error, that is the program cannot continue if these errors occur. The code can be rewritten as follows :

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

int main()
{
    try {

        ifstream File1("myFile1.txt");
        if( ! File1.is_open()) {
            throw -1;
        }
        // then work with the file
        //
        //

        ifstream File2("myFile2.txt");
        if( ! File2.is_open()) {
            throw -2;
        }
        // then work with the file
        //
        //

    }
    catch(int error)
    {
        if (error == -1) {
            cout << "can't open myFile1.txt !\n";
        }
        if (error == -2) {
            cout << "can't open myFile2.txt !\n";
        }
        cout << "exiting with error !\n";
    }

    return 0;
}
```

 [\[code/code-X-1-113.cpp\]](#)

You can notice the **try** block, that contains the code that works without managing errors : it only does the **throw** when an error occurs. Here, what is "thrown" are the values -1 and -2, depending on which file has not been properly opened. If a **throw** is run, then the **try** block is stopped at the location of the **throw**, and the program jumps automatically at the **catch** block. In the **catch** block, we have decided to get the error as an **int** (because this is what we have thrown), and so we will only catch errors that were thrown as an **int**. Then, we check which one it was and manage it. The result is, again, a more readable code, that separates the "normal" operation on one side, and the error management in another block.

You must also notice that functions or libraries can throw errors. An example with vectors :

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

int main()
{
    vector<float> V;
    V.resize(-1);
    // some work here
}
```



```
    return 0;
}
```

 [\[code/code-X-1-114.cpp\]](#)

Of course, resizing with -1 elements is not possible, and the way for vector library to manage it is to throw an error. But because there is no **try--catch** block, the program will finish with an error, like so :

```
terminate called after throwing an instance of 'std::length_error'
what():  vector::_M_default_append
Abandon (core dumped)
```

But we can also manage this error by ourselves with a **try--catch** block block :

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

int main()
{
    try {

        vector<float> V;
        V.resize(-1);
        // some work here

    }
    catch(length_error error)
    {
        cout << "Memory error !\n";
    }

    return 0;
}
```

 [\[code/code-X-1-115.cpp\]](#)

You notice that, this time, we catch a **length\_error** type error, and not an **int** type error. This time, the program displays :

```
Memory error !
```

## **PART III**

---

# **Fundamental concepts for Object-Oriented programming**

## Chapter 11 – Dynamic allocation of objects

### 11.1 Dynamically create objects : new

From here, we always simply instantiate objects like simple variables. In pure C we would say it is "the static way". In most cases and as long as you use the template classes of C++ you don't need more, because the dynamic aspects of allocation are naturally managed by these classes, for you.

But sometimes there is no choice and you may have to dynamically create objects by yourself. In C++, it is made with the word **new**:

```
class myClass {
public :
    int k;
    myClass();
    myClass(int i);
};
int main()
{
    myClass* p;
    p = new myClass(30);

    return 0;
}
```

 [\[code/code-XI-1-116.cpp\]](#)

As you can see, it is the first time that we mention pointers in C++. Modern C++ is made so that we can avoid in most cases speaking about pointers. But when we mention dynamic creation of objects, it is mandatory. You should note that the `malloc` cannot be used, like in pure C, to create a class, mainly because `malloc` has no way to call the constructor.

### 11.2 Destroy objects : delete

Symmetrically, you can destroy an object, with another C++ word `delete`. An example:

```
class myClass {
public :
    int k;
    myClass();
    myClass(int i);
};
int main()
{
    myClass* p;
    p = new myClass(30);

    delete p;
    return 0;
}
```

 [\[code/code-XI-2-117.cpp\]](#)

That's all. `new` and `delete` are simply new words of the C++ to replace pure C `malloc()` and `free()`, mainly because pure C can't run constructor and destructor of objects, it only can allocate memory.

## Chapter 12 – A powerful concept : inheritance

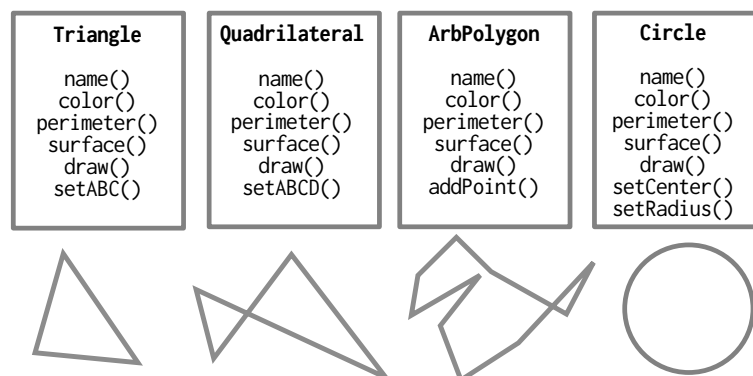
### Motivations and objectives

This is optional for this course, because it is very oriented in the thinking of writing quick and efficient code. However it is deeply what makes object oriented programming so powerful, whatever the object oriented language you consider, as it is the most conceptual element of all these languages. Here is a quick overview.

### 12.1 Inheritance : why ?

The main idea behind inheritance is to reuse and customise existing code, but without modifying the original one. That way, the two can co-exist in the same program. In other words, you can have a basic class, and various versions of itself, customized for various purposes, in the same program. So all the variations, called "**derived classes**" or "inheriters", will share some common code from the mother class, called "**base class**", and will be able to redefine what should be.

Let's start with a simple example: imagine we want to write a software that manages geometrical shapes. So we may have various classes defined, one per kind of shape :



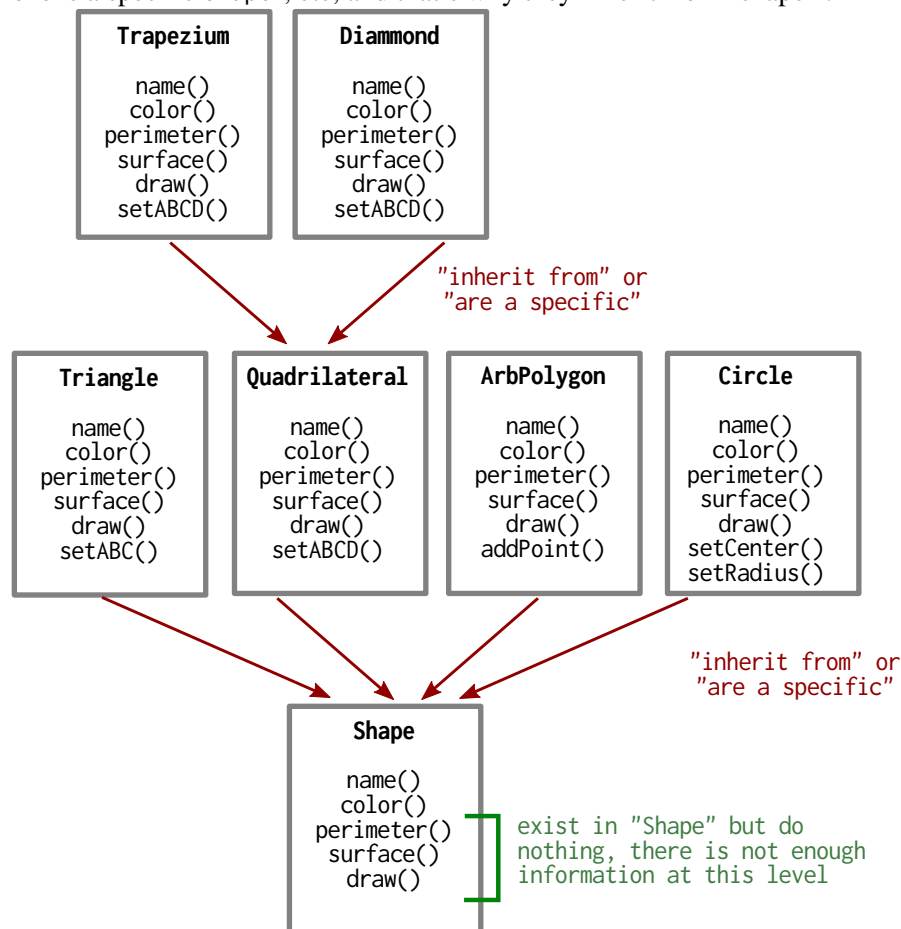
It is clear that the requirements, for all these classes, are similar :

- › all of them need to set and get a name and a color, and this should be similar for all,
- › all need to compute area and perimeter, but this will not be done the same way for all,
- › all need to be drawn, each by a specific way.
- › However all should not contain the same data members. For example, a triangle is 3 points whereas a circle has a point for the center and a float for the radius. Thus the functions that set the data of each kind of figure cannot be the same (see `setABC`, `setCenter`, `addPoint` ...)

So what we understand is that we would like to share some code between these classes, and define, when necessary, custom code, depending on the kind of figure.

A simple idea is to say that, finally, all these classes are "shapes" and that's the deep feature they have in common. So, we can write some common code in a new class, called "shape", and then say that

subsequent classes inherit from this. And so, we can think about inheritance like "a triangle is a specific shape", "a circle is a specific shape", etc, and that's why they inherit from "shape" :



In a similar way, we also defined specific quadrilaterals, to show that inheritance can be imagined among all the levels we want.

## 12.2 Inheritance syntax in C++

We will not write a whole example, this would be too long. We will take an example with the Shape class, and show how to inherit with Triangle and Circle. Moreover, we need a class Point to make things more fluid. So let's start. Here is the base class code.

```

1 // define Point class
2 class Point {
3     public:
4         float x,y;
5         float distanceTo(Point other);
6 };
7 float point::distanceTo(Point other) {
8     float dx = other.x-x;
9     float dy = other.y-y;
10    return sqrt(dx*dx + dy*dy);
11 }
12
13 // define Shape class
14 class Shape {
15     public :
16         string myName;
17         int R,G,B; // coords of the color
  
```

```

18
19         float perimeter();
20         float surface();
21         void draw();
22     };
23     // define Shape methods
24     float Shape::perimeter() { return -1;}
25     float Shape::surface() { return -1;}
26     void Shape::draw() {}

```

 [\[code/code-XII-2-118.cpp\]](#)

Now let's write the Triangle inheritance.

```

class Triangle : public Shape { // says triangle inherits from Shape
};

```

 [\[code/code-XII-2-119.cpp\]](#)

This inheritance, here, says Triangle is exactly the same as Shape, the only thing that is different is the name, one is Triangle, the other one is Shape. It's because at this time, we did not write anything in the class : it has taken, whereas not so explicitly written, all the members (data and methods) from Shape, so that you can write :

```

int main()
{ Triangle T1;
  T1.myName = "T1";
  T1.R = 255; T1.G = 0; T1.B = 0;
}

```

 [\[code/code-XII-2-120.cpp\]](#)

## 12.3 Customizing code

As said before, the Triangle class, whereas "empty" inside its {...}, contains all that's inside Shape. Well.

So now let's customize Triangle ! It simply consists in writing "things" into the {...}, as usual. Let's start :

```

class Triangle : public Shape { // says triangle inherits from Shape
public :
    float perimeter();
    float surface();
    void draw();
    void setABC(point& inA, point& inB, point& inC);
private:
    point A,B,C;
};

float Triangle::perimeter(){
    float p = A.distanceTo(B) + B.distanceTo(C) + C.distanceTo(A);
    return p;
}

void Triangle::setABC(point& inA, point& inB, point& inC) {
    A = inA; B = inB; C = inC;
}

float Triangle::surface () {
    // should be defined, a little long, not done here
}

void Triangle::draw () {
    // should be defined, out of scope for this course
}

```

 [\[code/code-XII-3-121.cpp\]](#)

This time, Triangle is a customized version of Shape : it has new data (the points), it has new definitions for various functions. Great !

Similar approach for circle :

```
class Circle : public Shape { // says triangle inherits from Shape
public :
    float perimeter();
    float surface();
    void draw();
    void setCenter(point& inCenter);
    void setRadius(float inR);
private:
    point Center;
    float R;
};
float Circle::perimeter(){
    return 2*3.14159*R;
}
void Circle::setCenter(point& inCenter) {
    Center = inCenter;
}
void Circle::setRadius(float inR) {
    R = inR;
}
float Circle::surface () {
    // should be defined, a little long, not done here
}
void Circle::draw () {
    // should be defined, out of scope for this course
}
```

  [code/code-XII-3-122.cpp]

## Remark

So, during the inheritance process, you define a new class that takes all the members from the so-called "base" class. In this process, you can :

- add new data members (you cannot remove however existing data members)
- add new methods, without specific link with the base class
- redefine methods with the same name and prototype as the base class. This technique is called **override**. It is important to have this idea in mind, as override is a very used feature of object-oriented languages (It is one of the main features of object-oriented programming). Moreover, in depth, the way override works in C++ is quite specific, as compared to other languages, and makes it more complex to understand than in other languages.

## Quick Application

- Create a class color that contains 3 int called R,G,B for Red, Green and Blue. They should be private.
- This class may contain a constructor that allows to the 3 fields.
- Add a public method called "display" that ... displays, with cout, the R,G and B fields.



- Add a public method called "lum" that computes the luminosity of the color (a very basic computation here) : this is obtained by calculating the average value of R,G and B.
- Create an "inheriter" (called "derived" class) from color, called tColor, in which you add a float called "alpha", that is here to speak about the transparency (its value is between 0 for totally transparent up to 1 for totally opaque).
- Update the constructor and the display function in the inheriter.
- Add a method called "tlum" that returns the luminosity multiplied by the alpha value.
- Build a main that instanciates color and tColor and that performs calls to show the overrides work well.

## 12.4 Ouch, pointers are back ...

Now imagine we want "plenty" of different shapes. Imagine it is a drawing software, and the user creates on the fly, various classes derived from Shape, like we did with Triangle and Circle above, but we added Diamond, Square, Polygon, etc. So we want to store all the instances created by the user "somewhere". In a list for example, it should be the best here but ... all the classes are not the same ... So let's try. If we simply write :

```
int main()
{
    list<Shape> myPainting;

    return 0;
}
```

 [\[code/code-XII-4-123.cpp\]](#)

it is inappropriate, as we can't store, for example, a Triangle in an element made to store a Shape. The C++ way to solve this is creating a list of pointers on Shape, as follows :

```
int main()
{
    list<Shape*> myPainting;

    return 0;
}
```

 [\[code/code-XII-4-124.cpp\]](#)

### Important

Indeed, object-oriented programming (and thus C++) allows **a base class pointer to point to an inherited class. The contrary is not possible.**

It is quite intuitive : all the inheritors (derived classes) know the properties of the base class, and so "the minimum" of "inheriters" (derived classes) knowledge is the one of the base class. The contrary is not true at all. ■

And so, we can write :

```
int main()
{
    list<Shape*> myPainting;

    Shape* myShape1;
    Shape* myShape2;
    Shape* myShape3;
```

```

myShape1 = new Triangle();
myShape2 = new Circle();
myShape3 = new Triangle();

myPainting.push_back(myShape1);
myPainting.push_back(myShape2);
myPainting.push_back(myShape3);

return 0;
}

```

 [\[code/code-XII-4-125.cpp\]](#)

Now play with the list :

```

int main()
{
    // same code as above but shorter to write
    list<Shape*> myPainting;

    myPainting.push_back(new Triangle());
    myPainting.push_back(new Circle());
    myPainting.push_back(new Triangle());

    // Here we play with the list
    list<Shape*>::iterator i;
    for (i = myPainting.begin() ; i != myPainting.end() ; i++) {
        Shape* theShape = *i; // please remember this * is due
                               // to the iterator, not to the pointer
        // here, i is a pointer on a Triangle or Circle, but its
        // type is Shape. What about ?
        cout << (*theShape).perimeter();
    }

    return 0;
}

```

 [\[code/code-XII-4-126.cpp\]](#)

**What follows is the big deal with C++, it is THE subtle thing to understand, and that is specific to C++.**

What's happens here ? In "standard" object-oriented programming, the perimeter function that should be called would be the one that correspond to the real type of the object : if theShape points on a Triangle, then it is the one of the triangle. If it is a circle, it should be the one of the circle. It is natural but you must have in mind that this assumes that the program determines, dynamically, which is the good object and method to call : this takes time while the program is running, and C++ dislikes that.

And so ... what does C++ ? ... The answer is : "because C++ doesn't like to spend time, with the code that we wrote, C++ would call the one of the Shape class, because it will work with the type of the pointer, which is known even during compilation. And so ... all our good code is ... just useless ...

So it's the end ? No it's not. Because C++ allows to use, if you decide so, the "normal" object-oriented programming way of working. It is made with the virtual keyword.

## 12.5 Virtuality ? Polymorphism of pointed elements ?

Virtuality is what allows C++ to work like a "real" object-oriented language, if the coder decides that. But it has to be mentioned for all the methods that we want to behave "the object-oriented programming way". For that we have to change the whole code, at the level of the declarations. Please look below :

```

class Shape {
public :
    string myName;
    int R,G,B; // coords of the color

    virtual float perimeter();
    virtual float surface();
    virtual void draw();
};

class Triangle : public Shape {
public :
    virtual float perimeter();
    virtual float surface();
    virtual void draw();
    virtual void setABC(point& inA, point& inB, point& inC);
private:
    point A,B,C;
};

class Circle : public Shape {
public :
    virtual float perimeter();
    virtual float surface();
    virtual void draw();
    virtual void setCenter(point& inCenter);
    virtual void setRadius(float inR);
private:
    point Center;
    float R;
};

```

 [code/code-XII-5-127.cpp]

## Important

This point, virtuality mixed to pointer, is the most difficult thing to understand, in depth, with C++. However, in most cases, if performance (high speed) is not required, you should always use virtual : this is the "good way of working" for object-oriented programming. In other words, "if you don't really know", if you hesitate, always use virtual, and have in mind removing it concerns mainly optimization, not object-oriented programming. Two exceptions : constructors and destructors, that are never virtual. That's all. ■

Concerning this code, some virtual are more or less useless. For example, it should be surprising that someone that inherits from Circle may change the way we set the center point or the radius, and thus we could remove the virtual for these two methods. But it is only assumed. However, for perimeter or draw, virtual is mandatory of course, as the code will always change from an "inheriter" (derived classes) to another.

## 12.6 public, protected and private inheritance

It's not a big deal here. Simply : the rights to members can be modified during inheritance, and that's the only reason for the protected status to exist (moreover this way to inherit more or less specific to C++, it does not exist in most other languages). Here is how all that is influenced :

		kind of inheritance		
		public	protected	private
rights in in the base class	public	public	protected	private
	protected	protected	protected	private
	private	no access	no access	no access

However, it should be terrific if you should have to do something else than public inheritance. These possibilities with protected and private are to consider for coders that work on high quality object-oriented software. For example you should find them in the creation of wide spread libraries such as the standard libraries of C++. But when developing a simple application it should not so often occur. So : if "you don't know", use public.

## Summary

This was a very short introduction to inheritance in C++. Inheritance is the most fundamental concept of object-oriented programming, and it's not that easy to understand "the first time". However once you know it you cannot think about programming without it.

You also have to understand that inheritance in C++ is complexified, through the `virtual` keyword, because C++ always wants to optimize execution time.

To sum up :

- › inheriting from a "base" class means creating a new class that contains all that the base class contains. It's done as follows :

```
class myBase {
    public :
        int k;
        virtual void f();
        virtual void g();
}
class myDerived : public myBase{
    public :
        float b;
        virtual void g();
}
```

 [\[code/code-XII-6-128.cpp\]](#)

In this code, `myDerived` is exactly `myBase` but augmented with the supplementary member `float b` and redefines function `g()`.

- › Please notice the `virtual` : Optionnal in C++ but mandatory if you want to think "the oriented-object programming way" (see main text). This is crucial if you want your program to call dynamically "the good method", i.e. the one of the instance instead the one of the reference (pointer) on the instance. This seems to be far-fetched but when we use libraries it is a very common situation. You can nearly avoid this problem if you always `virtual`, for all class methods, except constructor or destructor (not accepted by compiler, because does not make sense).
- › inheritance can be public, protected or private. However these choices concern advanced objects design. You can remember that if you have no explicit good reason you perfectly understand, always using public is ok in most cases.

## **PART IV**

---

# **Practical considerations**

## Chapter 13 – Compilation

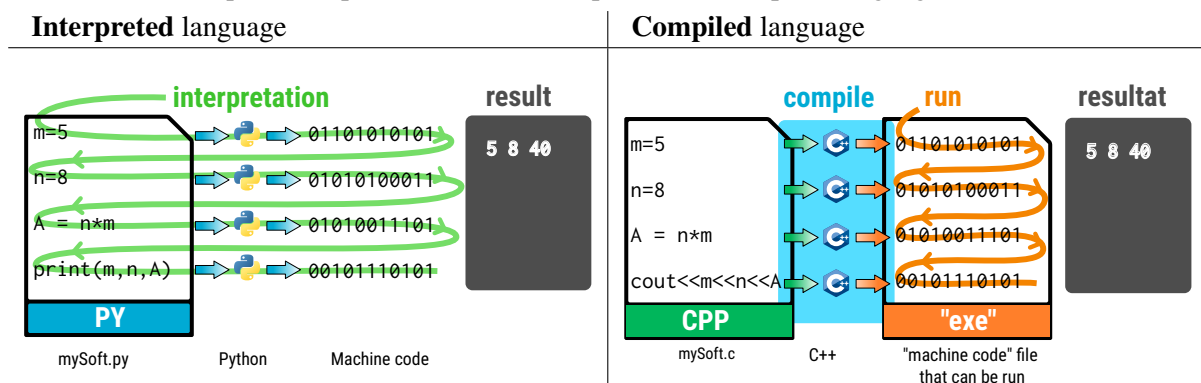
### Motivations and objectives

We first discuss about the differences between interpreted languages, such as Python, and compiled languages like C++.

Separately, we want to understand how to work with multiple C++ files in a single program. Indeed, a "big" program in C++ may be splitted in many files, to improve code readability. In fact, it is the concept that is beyond the concept of "libraries", except that the splitting is already done.

### 13.1 Interpreted language vs compiled language

We show below a quick comparison between interpreted and compiled languages :



Using interpreted language, the transformation of source code to executable code is made along the execution. Because of this process, **the resulting execution is slower**. Because the program is assumed to be slower, extra features are decided/checked during the execution. For example, the type of the variables, in python, is checked during the execution, and that's why we don't need to declare the type of the variables. This makes Python a more flexible language in the end, for example a function of a single parameter can get a numerical value or a string for this parameter depending on the call, which is not possible in C/C++. A ultimate example of this is the "generated code" : a Python program can create a string that contains Python code, then run it. This is impossible with compiled C++, because the compiler is not used while the program is running.

For a compiled language, the source code is transformed by the compiler in a first phase. Once it is done, the program can be run as many times as we want. During that, the compiler is not used anymore. Because the executable code has already been produced, the program obtained is **much faster**. Because the language is made to be faster, it is made so that **as many decisions as possible are taken while compiling**. That's why C++ language requires type declaration for variables : that allows to build executable that has not to check the type of variables, which makes the execution much more faster. Globally, C++ is optimized to take as many decisions as possible while compiling, so that there is less work to do while running, leading to quicker execution. However, things like "code generation" is no more possible, and the language is usually more tricky.

## 13.2 Multi-file compilation in a basic case

Let's start with a simple example :

```

1 #include <iostream>
2 #include <cmath>
3 using namespace std;
4
5 void howManyZeros(double a) {
6     cout << "\n" << "log(" <<a << ") = " << log10(a) ;
7     cout << " because there are " << (int) (log10(a) +1) << " digits\n";
8 }
9
10 int main()
11 {
12     howManyZeros(100);
13
14     return 0;
15 }
```

  [code/code-XIII-2-129.cpp]

qui affiche :

```
log(100) = 2 because there are 3 digits
```

We assume the code may be too long (which is not true here) and that we would want to split it into many files. We could create for example two files `.cpp`, for example `main.cpp` and `calculation.cpp`, which contents should be:

file `main.cpp`

```

1 int main()
2 {
3     combienDeZeros(100);
4
5     return 0;
6 }
```

  [code/code-XIII-2-130.cpp]

file `calculation.cpp`

```

1 #include <iostream>
2 #include <cmath>
3 using namespace std;
4
5 void combienDeZeros(double a) {
6     cout << "\n" << "log(" <<a << ") = " << log10(a);
7     cout << " parce qu'il y a " << (int) (log10(a)
8         +1) << " chiffres\n";
9 }
```

  [code/code-XIII-2-131.cpp]

We notice there is no more `#include` sequence in `main.cpp`. Indeed: `main.cpp` does not require `iostream` or `cmath` to work, as no line of its code uses any function of these libraries. For everything else, we only performed a copy/paste of parts of the code.

However, if we do compile :

```
g++ main.cpp calculation.cpp -o myProg
```

We obtain a compilation error :

```

main.cpp: In function 'int main()':
main.cpp:3:1: error: 'howManyZeros' was not declared in this scope
  3 | howManyZeros(100);
    | ^~~~~~
```

Indeed, in the `main` function of the `main.cpp` file, there is no information about a function called `howManyZeros`

However, the compiler requires this information : it has to check that the call to `howManyZeros` is well made, thus to know which are the parameters types and return type of the function. So it requires simply the prototype of the function, so we update the files as follows :

file **main.cpp**

```

1 void howManyZeros(double
    a);
2
3 int main()
4 {
5     howManyZeros(100);
6
7     return 0;
8 }

```

  [code/code-XIII-2-132.cpp]
file **calculation.cpp**

```

1 #include <iostream>
2 #include <cmath>
3 using namespace std;
4
5 void howManyZeros(double a) {
6     cout << "\n" << "log(" << a << ") = " << log10(a);
7     cout << " because there are " << (int)(log10(a)
        + 1) << " digits\n";
8 }

```

  [code/code-XIII-2-133.cpp]

Now, if we compile again, we get exactly the same program as before, when there were only a single **cpp** file : that works.

### Better "style"

In a big project, it is not practical that way, because we can make mistakes while rewriting the prototype of the functions that come from other **.cpp** files. In most cases, we create a supplementary file for each **cpp** file (usually except for the **main**) which extension is **.h** in pure C, and it has no extension in modern C++ libraries (however these are only conventions). These supplementary files are called "headers" and contain mainly the prototypes of the functions. And so, we may do the following :

file **main.cpp**

```

1 #include "calculation.h"
2
3 int main()
4 {
5     howManyZeros(100);
6
7     return 0;
8 }

```

  [code/code-XIII-2-134.cpp]
file **calculation.h**

```

1 #pragma once
2
3 void howManyZeros(double a);

```

  [code/code-XIII-2-135.cpp]
file **calculation.cpp**

```

1 #include <iostream>
2 #include <cmath>
3 using namespace std;
4
5 void howManyZeros(double a) {
6     cout << "\n" << "log(" << a << ") = " << log10(a);
7     cout << " parce qu'il y a " << (int)(log10(a)
        + 1) << " chiffres\n";
8 }

```

  [code/code-XIII-2-136.cpp]

We can see that the **calculation.h** file only contains the prototype of **howManyZeros**, and we added some other stuff to avoid infinite inclusions with **#pragma once**<sup>1</sup>. Please remember that you should use this kind of protection systematically.

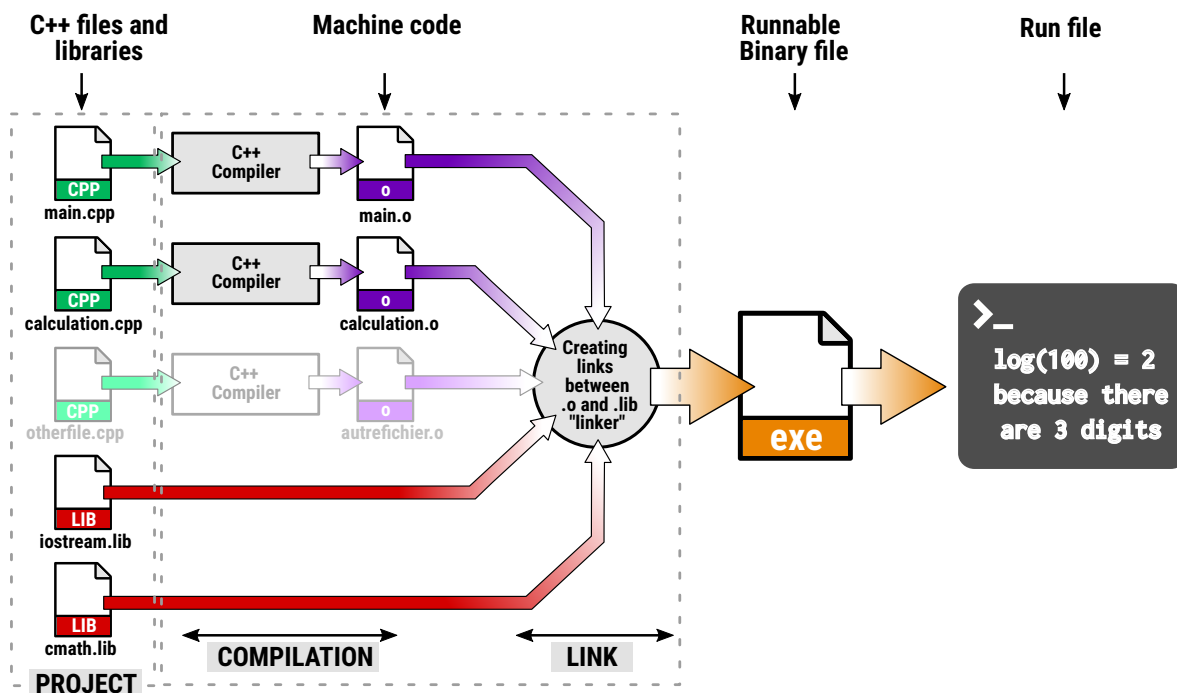
Next, **#include "calculation.h"** simply inserts prototypes in **main.cpp**, which allows again a successful compilation.

## 13.3 Illustration of the overall toolchain

We show below what happens when we compile a C++ project, made of various **.cpp** files with various libraries.

<sup>1</sup>imagine a header file that includes another one, and the other that includes the first one : you are in a case of circular inclusion that makes compiling impossible. Even if this example is too trivial to seem realistic, in more complex projects with much more header files, such a cycle could easily occur, and we want to avoid it





*Note :* This is a sum-up. Indeed, a preliminary phase, called "precompilation" is not shown here. During this phase, the commands starting with a `#` are evaluated. For example, the contents of files specified by `#include` are copied at the location where the command appears.

### Important

When you write `#include<iostream>` or `#include<cmath>`, you "think" that you "include a library". This is not a good way to think about it. As you have seen above, `#include` files contain only prototypes. That means that the compiled code of the library is "somewhere else", in another file, which is the binary part (or the source code) of the library. The header files are only used at compilation time, while the binary part is used at link time.

Please also notice that for standard libraries, most C++ compiler automatically include the good binary part when it detects some `#include`. That's why it seems sometimes to be a bit confusing. For example, when including `cmath`, some additional binary part is automatically added by the compiler.

## 13.4 Multi-file compilation in more advanced cases

On the same principle, we give an example of "more C++ code", containing classes : the class should be in the header file (like in pure C, structures definitions may be in the header file too), while the code may be in the corresponding `cpp` file.

## Chapter 14 – Code style

Juste a word about it. You must understand that **the readability of source code is very important**. To help readability, coders use some **conventions** when they code together. There are various possible conventions, each group of coder has its own ones. But it is important, when people work together, to have such a convention.

To make things more clear, we copy below some aspects of a given convention, used at Lirmm, the full version is available online at :

[https://gite.lirmm.fr/common-docs/doc-rob/-/wikis/coding\\_rules](https://gite.lirmm.fr/common-docs/doc-rob/-/wikis/coding_rules)

### C/C++ coding convention

#### General naming convention

- › Classes/structures/enums: their name is made of words, each word starting with a upper case character and other characters are lower case. Avoid too long names, e.g.:

```
|class FooBar //OK  
|class TheClassFooBarIsAreallyLongNameForAClass //NOT OK
```

- › namespaces are one word and lower case, e.g.:

```
|namespace boost
```

- › Names of variables and functions' arguments are alphanumeric lower case sequence of words using only underscores ('\_') as special characters between each word, e.g.:

```
|int my_variable_name;
```

- › Names of classes' attributes follows the same rule as variables but ends with an underscore, e.g.:

```
|int my_attribute_name_;
```

- › Constants are all capital, words are separated by underscores, e.g.:

```
|const int MY_CONSTANT
```

- › Names of functions/methods follow same rules as variables, e.g.:

```
|int this_is_the_name_of_my_function(...)
```

#### Coding guideline

These are general rules to follow when developing a C/C++ project:

- › **Indentation:** the indentation of the code should follow the bloc structure of the program. Put only one instruction by line.

- › **Comprehensive naming** : make your variables/functions/classes' names explicit and comprehensive as regard of the context. Use real life words, avoid meaningless random sequence of letters (e.g. one-letter names) except for integral iterator variables (i, j, k). An exception is tolerated if the notation come from a paper (e.g. matrix name).
- › **namespace scoping**: use namespaces to scope your code and gather all classes, functions and global variables (e.g. constants) that belong to a same project. Only closely related projects should share the same namespace.
- › **Class inheritance**: The basic usage is to subclass by using the public statement if there is no multiple inheritance in your hierarchy and by using virtual public otherwise (this corresponds to the standard specialization mechanism when using multiple inheritance in object oriented programming). Never use protected or private inheritance. Use non virtual inheritance only if you are sure that multiple inheritance will not take place in the hierarchy of the target class or if you wants somehow to forbids it.
- › **Class protection**: all attributes of a class must be private : define public or protected getters and/or setters only for attributes that can be accessed from outside of the class. Methods used only internally to a class must be private. Methods that are used in base class but that can be redefined in child classes must be protected. In a sub-class, never change the protection of elements defined in its base class.
- › **Friendship**: use friend statement with caution : friend keyword should be used only if a class as specific access to a "private interface" (set of private or protected methods) of another class (the accessed class then declares the accessing class as a friend). This is used when two classes in two distinct inheritance hierarchies are deeply bound to each other.
- › **Polymorphism**: by default all public and protected methods must be virtual, and private methods have to be not virtual. If a public or protected method cannot be redefine in sub-classes then make them not virtual (e.g. most of time the case for getters/setters).
- › **Templates**: templates classes and functions must be implemented in header files.
- › **Inlining**: never inline a virtual method ; inline only short methods that are to be widely used (getters and setters for example).
- › **Guarding headers**: the headers must be protected by multiple inclusion by using #pragma once guards :

```
#pragma once
... //code goes there
```

- › **Functions**: Always pass input "heavy objects" by const reference or by pointer to const objects. For instance:

```
int example_method(const Foo & in, Bar* out);
int other_method(const Foo * in, Bar* out);
```

- › **Functions arguments**: limit the number of arguments of a function to 5 at maximum.
- › **Be a const addict**: add const whenever it is possible: in the parameters of a function (input/output) and for methods that must not modify the class attributes. Return elements by const reference when you want to make readable an object attribute of a class or by pointer on const object when you want to make readable a pointer attribute. For instance:

```
const std::vector<double> get_attribute_value() const;  
const Bar * get_attribute_pointer() const;
```

- › **Preprocessor macros and constants:** should be limited as far as possible. Prefer inline functions, enums, and const variables.
- › **Preprocessor instructions:** usage of `#ifndef`, `#if`, etc. should be limited as far as possible, except when dealing with platform specific code (OS specific libraries) or when dealing with alternative third-party dependencies in the code.
- › **Exceptions:** exceptions are the preferred error-reporting mechanism for C++, as opposed to returning integer error codes. Always document what exceptions can be thrown by your package, on each function / method. Do not throw exceptions from destructors. Do not throw exceptions from callbacks that you do not invoke directly. When your code can be interrupted by exceptions, you must ensure that resources you hold will be deallocated when stack variables go out of scope. In particular, mutexes must be released, and heap-allocated memory must be freed. Accomplish this safety by using smart pointers.
- › **Documentation:** use *doxygen* annotations in header files to describe all methods (arguments, return value, usage, etc.), attributes, class (role and relationship with other classes), etc.
- › **Licensing:** systematically apply a license, authors and version information at the beginning of all source file (C/C++ headers and source). Input in you project a global file describing precisely the license that apply to the whole project.

## **PART V**

---

### **More advanced topics**





## Chapter 15 – Templates : how to create yours ?

### Motivations and objectives

Optionnal part for this course : a few words about how to create your own template class in C++.

### 15.1 Writing templates !

This is a very quick introduction to writing templates, just for the principle, a little example.  
Imagine you have written the matrix22, as we did page 14 :

complexe.c	matrix22.c
<pre>1 typedef struct { 2     float Re, Im; 3 }complex; 4 5 complex product(complex z1, 6     complex z2){ 7     z3.Re = z1.Re * z2.Re - z1 8         .Im * z2.Im; 9     z3.Im = z1.Re * z2.Im + z1 10        .Im * z2.Re; 11 12     return z3; 13 } 14 // ..... etc. 15   [code/code-XV-1-146.cpp]</pre>	<pre>1 typedef struct { 2     float a,b,c,d; 3 }matrix22; 4 5 matrix22 product(matrix22 M1, 6     matrix22 M2){ 7     matrix22 M3; 8     M3.a = M1.a*M2.a + M1 9         .c*M2.b; 10    M3.b = M1.a*M2.b + M1 11        .b*M2.d; 12    M3.c = M1.c*M2.a + M1 13        .d*M2.c; 14    M3.d = M1.c*M2.b + M1 15        .d*M2.d; 16 17     return M3; 18 } 19 // ..... etc. 20   [code/code-XV-1-147.cpp]</pre>

But this time : we want the matrix22 type work with float, int and ... even complex numbers ! So we have to rewrite the matrix22 type as a template class, so that we can choose the type. Let's start slowly :

```
1 template <typename T> class Matrix22 {
2     public:
3         T a,b,c,d;
4         void display();
5
6 };
```

In this simple code, you see that the class `Matrix22` is "templated". Between the `< ... >`, you mention the type that is, at this time, unknown, but will be `float`, `int`, or anything else later. Note that in more complex set-ups you use many types to template a class. We do not discuss about it.

Inside the class, then, we can use this `T` as any type. Here, compared to the pure C version, we declare the fields `a, b, c, d` to be variables with type `T`. We added a simple function to display, and it seems that's all. Now let's define the display function :

```
//requires iostream of course.
template <typename T> void Matrix22<T>::display() {
    cout << a << " " << b << "\n" << c << " " << d << "\n";
}
```

Well a little bit ... tricky ... Please remember member functions definition occur outside the class. And because of that, we have to give the full context for the display function. What's this context ? It is inside a template class, this is mentionned at the begining. That's ok. Then, everything is nearly like with any class : return type then classname then `::` and then function name. But you can see the `Matrix22<T>` whereas we could have thought `Matrix22` as usual. This supplementary `<T>` is here to mention that we are speaking of the `Matrix22` class when instanciated with a given type `T`. It seems tricky (and it is), but this is the syntax of templates, its like this.

Then, inside the function, everything is "normal", except that, if we want, we can declare variables with type `T`.

This simple example already works. You can try :

```
int main() {
    Matrix22<int> M1;
    Matrix22<float> M2;
    // here fill M1 and M2 fields, then :
    M1.display();
    M2.display();
}
```

Greate ! a little bit tricky but not that awful. And for a lot of work it is far enough. But it's only the begining ...

```
template <typename T> class Matrix22 {
public:
    T a,b,c,d;
    void display();
    Matrix22<T> prod(Matrix22<T>& right);
};
```

So the product should return a `Matrix22` and take as parameter another `Matrix22`. But not any random kind of `Matrix22` : specifically the ones that correspond to the `<T>` type and we have to mention it. Note that we could want to make operations, even in the template class, on any specific version of `Matrix22`, for example `Matrix22<float>`. So specifying this `<T>` is useful and necessary.

## 15.2 Ouch ... that syntax hurts !

And now it becomes really awfull : let's define the `prod` function. The scheme of the syntax is exactly the same as for `display`, just the elements are .... heavier :

```
template <typename T> Matrix22<T> Matrix22<T>::prod(Matrix22<T>& right)
{
    Matrix22<T> M;

    M.a = a*right.a + c*right.b;
    M.b = a*right.b + b*right.d;
    M.c = c*right.a + d*right.c;
    M.d = c*right.b + d*right.d;
```

```
    return M;  
}
```

Same thing as before : template, then return type (`Matrix22<T>`), then the class which is again `Matrix22<T>`, then `::`, then the function name and then the parameters. The inside of the function, however, is as simple as usual.

So what make templates frightening is the syntax for the declaration, not the usage.

### 15.3 What if I want matrix of complex numbers ?

It should simply work if we do :

```
int main(){  
    Matrix22<complex> M3;  
  
    return 0;  
}
```

However it does not work, for two equivalent reasons :

- the display function assumes the `<<` operator is available to print the `T` type. It is true for `int` and `float`, but we did not do this for complex numbers.
- similarly, the product function assumes the `*` and `+` operators exist for the `T` type, which is not true for complex.

So to make it work, you should redefine the operator`<<`, operator`+` and operator`*` for complex numbers. You must have in mind that it is not that difficult. But the declaration of these operators must be made by the good way. Earlier in this document, we did this for operator`+` and this should be similar for operator`*`. But for operator`<<` you must see, for example on the internet, how to declare it (not difficult but far out of the scope from this document).

### 15.4 About compilation



## **PART VI**

---

### **Practical Works**

## Chapter 16 – Practical works

### 🔗 Motivations and objectives

Here are exercises / practical works that are inspired from exams.

### ❓ Problem 16.1 : C to C++

#### – Convert the Code –

We give the following pure C code :

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

float* createData(int count, float dx) {
    float* y = malloc(count * sizeof(float));
    for(int k =0;k<count;k++) {
        float x = k*dx;
        y[k] = 5*exp(-x)*cos(10*x)+ 0.5*((float)rand())/RAND_MAX;
    }
    return y;
}

void average(float* data, int nb, float* result) {
    *result = 0;
    int k=0;
    while(k< nb) {
        *result += data[k];
        k++;
    }
}

int main()
{
    float* y1 = createData(100,0.05);
    int n = 6;

    float* y2 = malloc( (100-n)*sizeof(float));
    for (int k = n/2; k < 100 - n/2; k++) {
        average(y1+k,n,&y2[k-n/2]);
    }

    for(int k=0;k < 100-n;k++) {
        printf("%f\n",y1[k]);
    }

    return 0;
}
```

  [code/code-XVI-0-148.cpp]

**A.** Please convert the code above into a new version using as many C++ elements as possible. Mainly :

- › use C++ template types instead of memory allocation
- › use C++ STL's iterators when possible
- › use C++ references instead of pointer when possible
- › use C++ libraries instead of standard C libraries when possible
- › feel free to change the functions as you want, but the final code must still have 3 functions making the same work as they do now. However the prototypes can change a lot (and they should)

*Please notice that the `rand()` function is part of `stdlib.h` in pure C.*

**B.** What kind of operation does this code operate on the `y` variable of the main ? If you know it, please give the "official" name of this operation.

### - Transform the Code -

**C.** Please rewrite this code by replacing what were memory allocations in the initial C code by STL lists.

**D.** Explain why this transformation is a bad idea ■

## ② Problem 16.2 : Checkers Game

We want to write a Checkers game ("*jeu de dames*"). So we want to write a class called checkers.

### - Define the class checkers -

**A.** Write the declaration (*and only the declaration !!*) of a class called checkers that contains :

- › the board data. It will be defined as a 2D space of `int`, using the following syntax :

```
vector<vector<int>> > board;
```

  [code/code-XVI-0-149.cpp]

- › A constructor that has no parameter,
- › A destructor,
- › A function called `putSquare` that fills a square ("*une case*") with a given value. This function requires the `x` and `y` positions, as well as the color (given by an `int`) of the pawn ("*pion*").

All elements should be accessible from the outside of the class, except the board field of the class.

### - Write functions -

**B.** Write the constructor. It should set the board field so that it describes a  $8 \times 8$  squares board.

**C.** Write `putSquare`.

**D.** Write a main function that instantiates the checkers class and adds 2 white pawns and 2 black pawns (wherever you want). You may choose your own convention for the pawn color for that. ■

### ❓ Problem 16.3 : Geometrical Shapes

We want write code to implement "geometrical shapes" that are able to draw themselves.

#### - OpenGL -



We give the following code as startup code for OpenGL + GLUT :

```

1  #include <GL/glut.h>
2
3  void display(void)
4  {
5      glClear( GL_COLOR_BUFFER_BIT);
6      glColor3f(0.0, 1.0, 0.0);
7      glMatrixMode(GL_MODELVIEW);
8      glLoadIdentity();
9      glTranslatef(3,3, 0.0f);
10     glRotatef(30,0.0,0.0,1.0);
11     glBegin(GL_POLYGON);
12     glVertex3f(0, 0, 0.0);
13     glVertex3f(0, 4, 0.0);
14     glVertex3f(4, 4, 0.0);
15     glVertex3f(4, 0, 0.0);
16     glEnd();
17     glFlush();
18 }
19
20 int main(int argc, char **argv)
21 {
22     glutInit(&argc, argv);
23     glutInitDisplayMode ( GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
24
25     glutInitWindowPosition(100,100);
26     glutInitWindowSize(300,300);
27     glutCreateWindow ("square");
28     glClearColor(0.0, 0.0, 0.0, 0.0); // black background
29     glMatrixMode(GL_PROJECTION); // setup viewing projection
30     glLoadIdentity(); // start with identity matrix
31     glOrtho(0.0, 10.0, 0.0, 10.0, -1.0, 1.0); // setup a 10x10x2 viewing
        world
32
33     glutDisplayFunc(display);
34     glutMainLoop();
35
36     return 0;
37 }
```

 [\[code/code-XVI-0-150.cpp\]](#)

**A.** Download it, name the file **shapes.cpp**, and try to compile it as follows :

- > under  Linux : **g++ shapes.cpp -lglut -lGL -o shapes**
- > under  Windows/MinGW : **g++ shapes.cpp -lfreeglut -lopengl32 -lglu32 -o shapes**

Then run it : you should see a green square.

#### - The Shape class -

**B.** In this file, create the base **Shape** class that contains :

- › its name as a **string**,
- › its color, in an OpenGL compatible format, like 3 **float** for red, green and blue,
- › its position, like 2 **float** x0 and y0,
- › its orientation (an angle in degrees), like a **float** called **angle**

**C.** Add a constructor that simply sets the color to green, the position to (0,0), and the angle to 0.

**D.** Add a destructor that does nothing.

**E.** Add a method **void draw()**; that, again, does nothing.

### - The Square class -

**F.** Create a **Square** class that inherits from **Shape**

**G.** Add :

- › a new float the is the size of the square,
- › a constructor that takes as parameter the size of the square (and uses t to set the internal size of the square of course)
- › override the **void draw()**;. The contents of this function may be a copy/paste of lines 6 to 16 from the original code. Once the copy/paste made, you should adapt the code so that it uses the color that is inherited from **Shape**, the values of x0 and y0 in the **glTranslatef** function, and **angle** in the **glRotatef**.

**H.** Now, replace lines 6 to 16 from the **display** function by :

```
Square s(3);
s.x0 = 2;    s.y0 = 5;    s.angle = 75;    s.R=1;
s.draw();
```

 [\[code/code-XVI-0-151.cpp\]](#)

You should see a yellow square somewhere on the screen.

*Note :* For now, we only made by a more complex way what we made before. But you will see how more powerfull the code will become in a few.

### - The Triangle class -

**I.** In a similar way, create the Triangle class. If we consider the triangle is "ABC", we will fix the "A" point to (0,0) (it is not important, we also have the translation feature, so the whole triangle wil be able to move). Then, you will add 4 **float** for xB, yB, xC, yC. Create an adapted constructor, and adapt the contents of the **draw** function override. Try your triangle in the **display** function, like we did for the square.

### - Override does not work ? -

**J.** Now replace the **display** function by the following code :

```

void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    Square s(3);
    s.x0 = 2;    s.y0 = 5;    s.angle = 75;    s.R=1;
    Triangle t(3,8,2,5);
    t.x0 = 4; s.angle=10;    s.B=1;

    Shape* p1;
    Shape* p2;
    p1 = &s;
    p2 = &t;

    p1->show();
    p2->show();

    glFlush();
}

```

 [\[code/code-XVI-0-152.cpp\]](#)

Now run the code and ... it displays nothing, whereas we would have displayed a triangle and a square. Weird isn't it ?

**K.** This is because we did not say the **draw** function is virtual : thus, the decision, for C++, is to call the **draw** function that corresponds to the type of the pointer, thus the one of **Shape**, that does ... nothing.

Now, add the **virtual** keyword to all the **draw** functions. And the drawing is back.

### - Drawing all the shapes we want -

**L.** Now, create a **global** variable: **vector<Shape\*> allShapes**; Then, at the top of the **main** function, add the following code:

```

    Square s(3);
    s.x0 = 2;    s.y0 = 5;    s.angle = 75;    s.R=1;
    Triangle t(3,8,2,5);
    t.x0 = 4; s.angle=10;    s.B=1;
    allShapes.push_back(&s);
    allShapes.push_back(&t);

```

 [\[code/code-XVI-0-153.cpp\]](#)

And then update the **display** function so that it uses the **allShapes** variable. You should see the same image displayed.

**M.** Now, we want that when we press "C", a new squares is added, and when we press "T", a new triangle is added. For that, find in the internet how to catch the keyboard event. This concerns the function **glutKeyboardFunc** of glut.

### - Better design -

Something is wrong in this code : you can see that half the code of the **draw** functions is exactly the same, for the triangle as well as for the square. That's not good ! We will make better.

**N.** Add to the **Shape** class a new function **void draw\_shape()**, that does nothing.

**O.** Now, change the contents of the **draw** function of **Shape** like so :

```

    void Shape::draw() {
        glColor3f(R, G, B);
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
        glTranslatef(x0,y0, 0.0f);
        glRotatef(angle,0.0,0.0,1.0);
    }

```

```
draw_shape();  
}
```

  [\[code/code-XVI-0-154.cpp\]](#)

**P.** Now, rename the **draw** methods from **square** and **triangle** into **draw\_shape**, and remove from these functions all the code that sets the color, the translation and the rotation. This would now work as before, but more code is in common into the **Shape** class, so it is a better design. ■

Written with open-source software only

