



# C++ - Module 06

## C++ casts

*Summary:*

*This document contains the exercises of Module 06 from C++ modules.*

*Version: 6.2*

# Contents

<b>I</b>	<b>Introduction</b>	<b>2</b>
<b>II</b>	<b>General rules</b>	<b>3</b>
<b>III</b>	<b>Additional rule</b>	<b>6</b>
<b>IV</b>	<b>Exercise 00: Conversion of scalar types</b>	<b>7</b>
<b>V</b>	<b>Exercise 01: Serialization</b>	<b>10</b>
<b>VI</b>	<b>Exercise 02: Identify real type</b>	<b>11</b>
<b>VII</b>	<b>Submission and peer-evaluation</b>	<b>12</b>

# Chapter I

## Introduction

*C++ is a general-purpose programming language created by Bjarne Stroustrup as an extension of the C programming language, or "C with Classes" (source: [Wikipedia](#)).*

The goal of these modules is to introduce you to **Object-Oriented Programming**. This will be the starting point of your C++ journey. Many languages are recommended to learn OOP. We decided to choose C++ since it's derived from your old friend C. Because this is a complex language, and in order to keep things simple, your code will comply with the C++98 standard.

We are aware modern C++ is way different in a lot of aspects. So if you want to become a proficient C++ developer, it's up to you to go further after the 42 Common Core!

# Chapter II

## General rules

### Compiling

- Compile your code with `c++` and the flags `-Wall -Wextra -Werror`
- Your code should still compile if you add the flag `-std=c++98`

### Formatting and naming conventions

- The exercise directories will be named this way: `ex00`, `ex01`, ... , `exn`
- Name your files, classes, functions, member functions and attributes as required in the guidelines.
- Write class names in **UpperCamelCase** format. Files containing class code will always be named according to the class name. For instance:  
`ClassName.hpp/ClassName.h`, `ClassName.cpp`, or `ClassName.hpp`. Then, if you have a header file containing the definition of a class "BrickWall" standing for a brick wall, its name will be `BrickWall.hpp`.
- Unless specified otherwise, every output messages must be ended by a new-line character and displayed to the standard output.
- *Goodbye Norminette!* No coding style is enforced in the C++ modules. You can follow your favorite one. But keep in mind that a code your peer-evaluators can't understand is a code they can't grade. Do your best to write a clean and readable code.

### Allowed/Forbidden

You are not coding in C anymore. Time to C++! Therefore:

- You are allowed to use almost everything from the standard library. Thus, instead of sticking to what you already know, it would be smart to use as much as possible the C++-ish versions of the C functions you are used to.
- However, you can't use any other external library. It means C++11 (and derived forms) and Boost libraries are forbidden. The following functions are forbidden too: `*printf()`, `*alloc()` and `free()`. If you use them, your grade will be 0 and that's it.

- Note that unless explicitly stated otherwise, the `using namespace <ns_name>` and `friend` keywords are forbidden. Otherwise, your grade will be -42.
- **You are allowed to use the STL in the Module 08 and 09 only.** That means: no **Containers** (vector/list/map/and so forth) and no **Algorithms** (anything that requires to include the `<algorithm>` header) until then. Otherwise, your grade will be -42.

### A few design requirements

- Memory leakage occurs in C++ too. When you allocate memory (by using the `new` keyword), you must avoid **memory leaks**.
- From Module 02 to Module 09, your classes must be designed in the **Orthodox Canonical Form, except when explicitly stated otherwise**.
- Any function implementation put in a header file (except for function templates) means 0 to the exercise.
- You should be able to use each of your headers independently from others. Thus, they must include all the dependencies they need. However, you must avoid the problem of double inclusion by adding **include guards**. Otherwise, your grade will be 0.

### Read me

- You can add some additional files if you need to (i.e., to split your code). As these assignments are not verified by a program, feel free to do so as long as you turn in the mandatory files.
- Sometimes, the guidelines of an exercise look short but the examples can show requirements that are not explicitly written in the instructions.
- Read each module completely before starting! Really, do it.
- By Odin, by Thor! Use your brain!!!



Regarding the Makefile for C++ projects, the same rules as in C apply (see the Norm chapter about the Makefile).



You will have to implement a lot of classes. This can seem tedious, unless you're able to script your favorite text editor.



You are given a certain amount of freedom to complete the exercises. However, follow the mandatory rules and don't be lazy. You would miss a lot of useful information! Do not hesitate to read about theoretical concepts.

# Chapter III


## Additional rule

The following rule applies to the entire module and is not optional.

For each exercise, the type conversion must be solved using one specific type of casting. Your choice will be checked during defense.

# Chapter IV

## Exercise 00: Conversion of scalar types

	Exercise 00
Conversion of scalar types	
Turn-in directory : <i>ex00/</i>	
Files to turn in : Makefile, *.cpp, *.{h, hpp}	
Allowed functions : Any function to convert from a string to an int, a float or a double. This will help, but won't do the whole job.	

Write a class `ScalarConverter` that will contain only one `static` method "convert" that will takes as parameter a string representation of a C++ literal in its most common form and output its value in the following serie of scalar types :

- char
- int
- float
- double

As this class doesn't need to store anything at all, this class must not be instanciable by users.

Except for char parameters, only the decimal notation will be used.

Examples of char literals: `'c'`, `'a'`, ...

To make things simple, please note that non displayable characters shouldn't be used as inputs. If a conversion to char is not displayable, prints an informative message.

Examples of int literals: `0`, `-42`, `42...`

Examples of float literals: `0.0f`, `-4.2f`, `4.2f...`

You have to handle these pseudo literals as well (you know, for science): `-inff`, `+inff`



and `nanf`.

Examples of double literals: `0.0`, `-4.2`, `4.2...`

You have to handle these pseudo literals as well (you know, for fun): `-inf`, `+inf` and `nan`.

Write a program to test that your class works as expected.


You have to first detect the type of the literal passed as parameter, convert it from string to its actual type, then convert it **explicitly** to the three other data types. Lastly, display the results as shown below.

If a conversion does not make any sense or overflows, display a message to inform the user that the type conversion is impossible. Include any header you need in order to handle numeric limits and special values.

```
./convert 0
char: Non displayable
int: 0
float: 0.0f
double: 0.0
./convert nan
char: impossible
int: impossible
float: nanf
double: nan
./convert 42.0f
char: '*'
int: 42
float: 42.0f
double: 42.0
```

# Chapter V

## Exercise 01: Serialization

	Exercise : 01
Serialization	
Turn-in directory : <i>ex01/</i>	
Files to turn in : <b>Makefile</b> , <b>*.cpp</b> , <b>*.{h, hpp}</b>	
Forbidden functions : <b>None</b>	

Implement a class `Serializer`, who will not be initializable by the user by any way, with the following **static** methods:

```
uintptr_t serialize(Data* ptr);
```

It takes a pointer and converts it to the unsigned integer type `uintptr_t`.

```
Data* deserialize(uintptr_t raw);
```

It takes an unsigned integer parameter and converts it to a pointer to `Data`.

Write a program to test that your class works as expected.


You must create a non-empty (it means it has data members) `Data` structure.

Use `serialize()` on the address of the `Data` object and pass its return value to `deserialize()`. Then, ensure the return value of `deserialize()` compares equal to the original pointer.

Do not forget to turn in the files of your `Data` structure.

# Chapter VI

## Exercise 02: Identify real type

	Exercise : 02
Identify real type	
Turn-in directory : <i>ex02/</i>	
Files to turn in : <code>Makefile</code> , <code>*.cpp</code> , <code>*.{h, hpp}</code>	
Forbidden functions : <code>std::typeinfo</code>	

Implement a **Base** class that has a public virtual destructor only. Create three empty classes **A**, **B** and **C**, that publicly inherit from **Base**.



These four classes don't have to be designed in the Orthodox Canonical Form.

Implement the following functions:

```
Base * generate(void);
```

It randomly instantiates A, B or C and returns the instance as a Base pointer. Feel free to use anything you like for the random choice implementation.

```
void identify(Base* p);
```

It prints the actual type of the object pointed to by p: "A", "B" or "C".

```
void identify(Base& p);
```

It prints the actual type of the object pointed to by p: "A", "B" or "C". Using a pointer inside this function is forbidden.

Including the `typeinfo` header is forbidden.

Write a program to test that everything works as expected.

# Chapter VII

## Submission and peer-evaluation

Turn in your assignment in your `Git` repository as usual. Only the work inside your repository will be evaluated during the defense. Don't hesitate to double check the names of your folders and files to ensure they are correct.



16D85ACC441674FBA2DF65190663E136253996A5020347143B460E2CF3A3784D794B  
104265933C3BE5B62C4E062601EC8DD1F82FEB73CB17AC57D49054A7C29B5A5C1D8  
2027A997A3E24E387