Advanced Programming flash-cards

C++ introduced features of the Object-Oriented paradigm, including classes, objects, inheritance and polymorphism; it also introduced templates, that allow for generic programming, enabling the creation of data structures and algorithms that could work with different data types.

Compiled VS interpreted languages:
A compiled language has a compiler that compiles the source file producing an executable which run gives an output.
An interpreted language has an interpreter that interprets the source file producing the output.
A compiled language (like C) is harder to write but its execution is faster, while for an interpreted language (like R, Python, MatLab) is the other way round.

Build process:
The user gives a source file to the preprocessor, that gives a preprocess source file to the compiler. The compiler generates an object file which is given to the linker. The linker links all the object files generated from different source files with libraries; the loader loads shared libraries and executes the executable.

Preprocessor:
A preprocessor is a program or set of directives which is part of the build process and performs preliminary operations on the source code before passing it to the compiler. It has the role of handling directives (like including header files with `#include`, conditional compilation `#ifdef #ifndef #else #endif`, compiler-specific directives `#pragma`) and macros (defining them with `#define`). It's used for code reusability and organization.

Compiler:
Translates the source code into assembly/machine code.

Linker:
Combines object files into an executable. A static linking is larger in dimensions and includes all libraries; a dynamic linking is smaller and links the libraries at runtime.

Loader:
Loads the executables for execution. Before doing so, it reserves memory, adjusts addresses and sets up the environment.

Shell:
A shell is a program that provides traditional, text-only user interface for Linux and other UNIX-like operating systems. Its primary function is to read commands typed into a console and execute them.

It can be a login or a non-login shell; respectively if you login into a virtual terminal or just open a graphic terminal.
It can be interactive or non-interactive shell; the last one is usually run from an automated process.

Difference between single pipe and double pipe:
The single pipe is like the composition of a function: the output of the command1 gets forwarded as input for the command2. For example `cut -f 3 -d ',' file.csv | sort | wc -l` uses cut to subdividing a file (with some flags for how to do it), then its output gets sorted, lastly it applies a word count (actually a line count due to the flag) to the sorted values.
The double pipe is the logical "or" operation. For example to verify if a file exists and create it if it does not: `[ -e file.txt ] || touch file.txt`.

Version control:
aka source control, is the practice of tracking and managing changes to software code. Version control systems are software tools that help software teams manage changes to source code over time. Git is a free and open-source version control system.

Stack VS heap memory:
Stack is a region of memory for function call frames. Variables stored here, have a fixed size and scope, it's well suited for small and short-lived variables. Pointers to stack variables can be used safely within their scope. Memory is allocated and deallocated automatically.
Heap is a region of dynamic memory for data with varying lifetimes. Variables stored here, have a dynamic size and longer lifetimes. Heap-allocated variables require pointers for access, which must be managed carefully and deleted after. Memory allocation and deallocation are explicit i.e. manual.

Dynamic memory management:
It's the allocation and deallocation of memory at runtime using pointers. It's more flexible than static memory allocation (i.e. at compile time) because the program can adapt to varying data sizes and structures during execution.

Differences new/delete and malloc/free:
`new`/`delete` is type safe, while `malloc`/`free` is not and it doesn't even invoke constructors and destructors. The first automatically determines the size, while the second requires that information. For this reason `malloc`/`free` is more appropriate when dealing with simple memory allocations where constructors are not invoked. When it isn't so, it's better to use `new`/`delete`, or even better smart pointers.

Undefined behaviour in C++:
There is a situation for which the language doesn't prescribe a specific outcome, generating any result.

Some examples can be:
```bash
// Trying to access memory beyond the allocated one
std::vector<int> v;
v.resize(2, 5);
std::cout << v[6];

// Using uninitialized variables
int x;
int y = x + 2;

// Modifying a const object
const int pi = 3.1415926;
pi = 3.14

// Type mismatch
int x = 5;
std::string saluto = "ciao";
double sum = x + saluto;
```

References:
References provide an alias for an existing variable.

Pass and return by value VS pointer VS reference:
Pass and return by value for small, non-mutable data because it does a copy.
Pass by pointer for modifying values or working with arrays.
Return by pointer for dynamically allocated data (it points to that value).
Pass/return by reference for efficiency and direct modification of values/data (it returns the reference to that value).

const correctness:
It prevents unintended modifications, self-documents the code indicating the intent of data usage, allows the compiler to perform certain optimizations since it knows the const data won't change.

Pre VS post increment:
The pre-increment increases the variable's value before using it, so the updated value is immediately reflected; the post-increment uses the current variable's value before incrementing it.

Function overloading:

Feature of C++ that allows you to define multiple functions with same name but different parameters.

Struct:
In C they are a special type of class, a proper name would be Plain Old Data structs. They are classes with simple data members and no user-defined constructors or destructors. For default the members of struct are private, while in class they are public.

Modular programming:
It divides code into separate modules that focus on a specific task or functionality. This improves code organization, readability and reusability; it's also easier to maintain and debug. The code modules are header files for declarations, source files for definitions, implementation files for non-template classes. Header files are included in source files to access declarations and define the interface to a module or class. Without header guards (as #ifndef + #define + #endif or #pragma once), if a header file is included multiple times it can lead to redefinition errors.

Namespaces:
Namespaces group related declarations to avoid naming collisions and better organize the code into logical units.

Object-Oriented Programming:
It's a programming paradigm that revolves around objects. Objects represent instances of classes, encapsulating data and behavior.
OOP is based on key principles: encapsulation, inheritance and polymorphism.
Encapsulation bundles data and functions that operate on the data into a single unit called an object, promoting data hiding and reducing the complexity of the code.
Inheritance allows to create new classes based on existing ones, enabling code reuse and the creation of class hierarchies.
Polymorphism allows objects of different classes to be treated as objects of a common base class, promoting code flexibility and working with objects at a higher level of abstraction.

RAII idiom:
Resource Acquisition Is Initialization: holding a resource is a class invariant, tightly bound to the object's lifetime. It encapsulates a resource within a class (constructor), utilizes the resource through a local instance of the class, automatically releases the resource when the object goes out of scope (destructor).

Static members:
Are shared among all instances of the class, are useful for maintaining shared data or functionality across objects. For this reason, if we call more times a function that, for

example, initialize a variable to 0 and increments it, the final value won't be 1 but the number of times it was called.

This pointer:
Represents a pointer to the current instance of a class, allowing to access member of an object from within its member functions and resolving ambiguity when local variables and member variables of a class have the same name. It's a hidden argument to all non-static member functions and it's automatically passed to those functions by the compiler.

Constructors:
Are special member functions that initialize objects when they are created. They have the same name of the class. Within it, a initializer list is used to initialize member variables before entering the constructor body. There are different types of constructor:
Default constructor: takes no arguments; if no constructor is provided, C++ generates a default one automatically using default values (such as empty for strings, zero for numbers).
Parametrized constructor: takes one or more parameters to initialize member variables based on the provided values.
Copy constructor: creates a new object as a copy of an existing object of the same class, taking a reference to an object of the same class as a parameter. It is invoked when objects are copied, passed by value or initialized with other objects.
Copy assignment operator: constructor used for copy assignment
Move constructor: it allows to transfer the resources from one object to another
Destructor: special member function used (and automatically called) to clean up resources held by an object before it goes out of scope or is explicitly deleted. If it's done inside an abstract class, it must be virtual.

Rule of three:
If a class defines or deletes one of destructor, copy constructor, copy assignment operator, then it should probably provide all three of them.

Inline directive:
Can be applied to free functions to suggest that the function should be inlined by the computer. It's most effective for small functions.

In VS out of class definition:
Member functions of a class can be defined either in-class (inline) or out of class. The first one is in the header file and it's common for short, simple functions in order to get a more compact and concise code, but may lead to code bloat and changes may necessitate recompilation of all translation units that include the header. On the contrary, for the second, functions are defined separately in a source file and it's common for larger implementations, even though it can be slightly more verbose in terms of code.

Data encapsulation:
Bundles data and methods that operate on that data into a single unit called an object, exposing only the necessary functionality through well-defined interfaces. The access specifiers can be public, private or protected; which means that the members declared in those ways respectively are or are not accessible or are accessible within the class and derived classes.

Getter and setter methods:
Getter methods allow reading the values of private variables; thus they are constant.
Setter methods enable modifying those values in a controlled manner.

Operator overloading:
It enables to extend the functionality of operators beyond their predefined meanings, to improve code readability and maintainability. Some operators (like ::, .*, ?:) can't be overloaded. An example is the overload of the [] operator to allow indexing and accessing individual data elements, both for read and write access:

```cpp
// Write access operator
double &operator[](const unsigned int &index) {
	assert(index >= 0 && index < size);
	return data[index];
}
// Read access operator
const double &operator[](const unsigned int &index) const {
	assert(index >= 0 && index < size);
	return data[index];
{
```

Relationships between classes:
Association: loose relationship where classes are related but one does not necessarily contain the other.
Dependency: one class depends on another class.
Aggregation: a "has-a" relationship where one class contains the other as a part, but the contained object can exist independently.
Views/proxy: another type of aggregations that enables access to the members of the aggregating object using a different, often more specialized, interface.
Composition: a stronger "whole-part" relationship where one class contains another as a part and the part can't exist independently.
Inheritance: a "is-a" relationship where the derived/subclass inherits properties and behaviors from the base/superclass.

Difference aggregation-inheritance:

Aggregation is a "has-a" relationship where one class contains the other as a part, but the contained object can exist independently; while inheritance is a "is-a" relationship where the derived class inherits properties and behaviours from the base class, therefore it can't exist without the base class and it's stronger.
An example of aggregation is the University class that may aggregate Department classes.

Polymorphism:
Is implemented through public inheritance; it allows objects belonging to different classes within a hierarchy to operate according to an appropriate type-specific behavior.

Abstract class:
A class that can’t be instantiated and serves as a blueprint for other classes, enforcing a common interface for its derived classes. The derived classes that inherit from it, must provide implementations for all the pure virtual methods to become concrete instantiable classes.

Run-Time Type Information:
Allows to determine the type of an object at runtime, typically using typeid operator or dynamic casting. With this last one we try to downcast: if the condition fails, it returns the null pointer; otherwise it returns the pointer to the derived class; it also works with references but in this case if the condition is not satisfied, it throws an exception.

Function:
Identified by name, number and type of its parameters, const qualifier, type of the enclosing class, the return type is not part of the function identifier.

Callable object:
An object that can be called as if it were a function. It can be functions, function pointers, member function pointers, functors, lambda functions, function wrappers.

Functor / function object:
A class object which overloads the call operator (operator ()). If it returns a bool, it’s a predicate. A characteristic of a functor is that it may have a state so it can interact with other objects and store additional information to be used to calculate the result. Some functors are predefined in the Standard Template Library (STL), like std::negate<type> and std::back_inserter<type>.

Lambda functions:
Lambda functions are similar to the MatLab anonymous functions and they are used to create short inlined functions quickly, thus only when the function takes up one line (for Python) or a few (for C++) of code.

In both languages we don't need to specify the return type, with the difference that in Python we don't write anything, while in C++ we write `auto` before the definition and we can also specify with `->` (example later).
In Python they are created with the keyword lambda, for example `add_one = lambda x: x + 1`.
In C++ the square brackets are used to capture specification, which allows to use variables in the enclosing scope inside the lambda either by value or by reference. For example, for doing the same thing done with Python, we can write:
```bash
auto f = [] (double x) { return x + 1; };
```
Which can be also specified with
```bash
auto f = [] (double x) -> double { return x + 1; };
```
If we want to capture all the variables by making a copy we can write:
```bash
auto g = [=] (float x, float y) { return x * y; };
```
And so on with other symbols inside [].

Function wrappers:
Useful when you want to have a common interface to callable objects. The class std::function<> provides polymorphic wrappers that generalize the notion of a function pointer.

Generic programming:
Programming paradigm that aims to write code in a way that's independent of data types, creating reusable and versatile code by using templates or type abstractions.
Examples of it are STL (offers a collection of generic data structures and algorithms), function templates, class templates.

Function templates:
Allow to write a function once and use it with different data types. The actual compilation occurs only when the template is instantiated. constexpr can be applied to ensure that variables, functions and constructors, are evaluated at compile time.
When templates are long, separate their declarations from definitions with a module.tpl.hpp and then #include "module.tpl.hpp" at the end of module.hpp.
To save compile time when debugging, you can use explicit instantiation of the template.

Template template:
Allow to define templates as template arguments; they are used for defining higher-order templates that accept template classes.

```cpp
template <typename T, template <typename> class Container>
class ContainerWrapper {
private:
    Container<T> data;
};

ContainerWrapper<type1, container> wrapper; // Instantiation of ContainerWrapper
object
```

Variadic templates:
Allow functions and classes to accept a variable number of arguments, writing … after the name of the argument (ex. Args…).

Containers:
Sequence containers are ordered collections of elements with their position independent of the element value.
Associative containers have elements whose position depends on their content. They are divided into maps (elements are key-value pairs) and sets (elements are just values (or key, for sets it's interchangeable)) and they can be ordered or unordered. If we write the same key, in a map it will overwrite the previous value, while in a set it won't be added.

Iterators:
Generalization of pointers that allow to work with different structures (for example containers and ranges) in a uniform way. An iterator is any object that allows iterating over a succession of elements, typically stored in a standard container.

Algorithms:
STL provides an extensive set of algorithms to operate on containers, more precisely on ranges. They allow a more uniform optimized way. They can be modifying (like sorting algorithms) or non-modifying (like finding minimum and maximum).

Smart pointers:
Smart pointers control the lifespan of the resource they point to. `std::unique_ptr` means that the owned resource gets destroyed when the pointer goes out of scope; while for `std::shared_ptr` it happens when the last pointer owning that resource is destroyed. In fact in this last one, you have several objects that refer to the same resource.
`std::weak_ptr` is a non-owning (thus weak) pointer to a shared resource; to access the referenced object it must be converted to a shared pointer; it's used to safely observe the shared object without affecting the ownership.

Lvalues and rvalues:
An lvalue is an expression that may appear on the left and on the right-hand side of an assignment. It refers to a memory location and allows us to take its address via the & operator. Examples of it are the value held in a variable, the returned value reference of a function.
An rvalue is an expression that can only appear on the right-hand side of an assignment. Examples of it are the returned value of a function.

Assertions:
When a function has a condition, its requirement can be asserted at runtime with assert (which can be disabled by defining the NDEBUG preprocessor) or at compile-time with static_assert.

Exceptions:
An exception is an anomalous condition that disrupts the normal flow of a program's execution when left unhandled. You can handle them using throw to indicate that an exception has occurred or using try-catch to catch and handle exceptions.
You can raise an error when something bad is being done, and when executing that operation, you can use a try-except block to handle the exception.

Library:
A collection of pre-written code or routines that can be reused by computer programs. They usually contain functions, variables, classes and procedures that perform common tasks, saving time and effort by leveraging existing code rather than writing everything from scratch. It's formed by a set of header files (that provide the public interface of the library) and one or more library files (that contain, in the form of machine code, the implementation of the library; it can be static or shared). The first ones are only used in the development phase, the needed ones are the library files.
Header-only libraries are libraries whose implementation is only contained in header files (thanks to inline functions and templates), thus are the easiest to use.

Generate random numbers:
Including the header <random>, you can generate random numbers using engines or distributions.
Engines serve as a stateful source of randomness, proving random unsigned integer values uniformly distributed in a range. They use seed data as an entropy source for generating pseudo-random numbers. Usually a seed can be generated using std::random_device (will be called seed) and then using it as argument of std::default_random_engine (will be called gen(seed)). It's not directly used the first one because it is slower than other engines.
Distributions are template classes that specify how the values generated by the engine have to be transformed to generate a sequence with prescribed statistical properties (for example std::uniform_int_distribution for a uniform distribution of integers,

std::binomial_distribution for a binomial distribution, std::geometric_distribution for a geometric distribution, etc).
An example for the uniform distribution of a 12-face dice is:
```cpp
std::random_device seed;
std::default_random_engine gen[seed()};
std::uniform_int_distribution<> dice{1,12};
for (unsigned int n = 0; n < 10; ++n)
        std::cout << dice(gen) << " ";
std::cout << std::endl;
```

Static VS shared libraries:
A static library (often denoted by .lib or .a extension), contains compiled code that is linked directly into an executable at compile time. Building it, a copy of its code is included in the final executable, so the resulting executable is independent of the original library file.
A shared library (often denoted by .dll or .so or .dylib extension) contains code that is loaded at run-time. Instead of being included in the executable, the program references the shared library, loading it into memory when needed.

Pros and cons of static libraries:
Pros: the executable is self-contained.
Cons: if an external library receives an update, you have to relink its code against the new version; can't load symbols dynamically on the base of decisions taken at run-time; executable might become large.

Pros and cons of shared libraries:
Pros: updating a library has immediate effect, no recompilation or relinking is needed; executable is smaller since the code in the library is not duplicated; we can load libraries and symbols runtime.
Cons: executables depend on the library; both linking and loading phase need careful management.

Target of Makefile:
Is the desired output or action; it can be an executable or object file or a specific action like "clean".

Cache variables:
Used to interact with the command line setting options on or off.

Code optimization:

Process of enhancing a program's performance, efficiency and resource utilization without changing its functionality; improving execution speed and reducing memory usage.

Debuggers:
Software tools that enable developers to inspect, analyze and troubleshoot code during the development process; they provide a set of features for identifying and fixing errors in programs.

Debugging types:
Static debugging analyzes core dump if code aborts.
Dynamic debugging executes through a debugger, breaking at points to examine variables.

Profiler:
Tool or set of tools designed to analyze the runtime behavior and performance of a computer program. It provides detailed information about resource utilization, execution times and function calls during the program's execution.

Verification VS validation:
Verification is conducted during development and tests individual components separately. Specific tests demonstrate correct functionality, covering the code and checking for memory leaks.
Validation is performed on the final code. Assesses if the code produces the intended outcome-convergence, reasonable results and expected computational complexity.

Types of testing:
Unit testing tests individual components (functions, methods, classes) to ensure each behaves as expected; focuses on a specific piece of code in isolation.
Integration testing verifies that different components/modules of the software work together as intended; deals with interactions between different parts of the system.
Regression testing ensures recent code changes do not adversely affect existing functionalities; involves re-running previous tests on the modified codebase to catch unintended side effects.

Test-Driven Development (TDD)
Software development approach where tests are written before the actual code. It encourages modular and testable code, ensuring all parts of the codebase are covered by tests. The process is writing a test, implementing the code to pass the test, refactoring.

Continuous Integration (CI):
Frequently integrating code changes into a shared repository. Automated builds and tests ensure new changes don't break existing functionalities.

Code coverage:
Metric used in software testing to measure the extent to which source code is executed during the testing process; providing insights into which parts of the codebase have been exercised and which remain untested. It's often expressed as a percentage of lines of code that have been executed by tests, so the goal is to have it as close to 100% as possible.

Python:

args and kwargs
*args is a tuple of arbitrary positional arguments, **kwargs is a dictionary of variable keyword arguments. They are collected respectively with * and ** because * is used to unpack iterable objects into a tuple; ** is used to unpack the key-value pairs from a dictionary.

Constructor in Python:
Is def __init__ and takes self as first input argument. Self is a variable that refers to the object itself. It's equivalent to the this pointer in C++.

List comprehensions:
List comprehensions are a convenient and compact way to build lists, tuples, sets and dictionaries in just one line. For example instead of writing:
```bash
message = ["The", "cat", "is", "purring"]
first_letters = []
for word in message:
    first_letters.appen(word[0])
print(first_letters)
```
We can just write
```bash
letters = [word[0] for word in message]
print(letters)
```

Class VS static methods:
Class methods (@classmethod) act on the class itself, often serving as alternative constructors.
Static methods (@staticmethod) operate independently of instances and classes but are relevant to the class.

Magic methods:

Aka dunder (double underscore) methods, are special methods that start and end with double underscores and are automatically invoked by the Python interpreter in response to certain events or operations.

Decorators:
Powerful way to enhance the functionality of functions or methods. They act as wrappers, allowing you to extend or modify the behavior of the original function. They can be imagined as a shortcut to calling a wrapper function, which is a function that wraps around another function so that it can do something before or after the inner function.
A popular example is the decorator that measures the execution time of a function:

```python
Import time

def execution_time(function):
    def wrapper(*args, **kwargs):
        start = time.time()
        result = function()
        end = time.time()
        print(f"{function.__name__} ran in {end – start:.3f} sec")
        return result
    return wrapper

@execution_time
def some_function():
    # …
    return
```

In this way once some_function is called, the decorator will be applied, measuring its execution time with a precision of 3 decimal places and a fixed-point format.

Generators:
In Python, generators are functions that produce a sequence of values in a lazy way: one at a time, rather than generating the entire sequence all at once. This provides an efficient use of memory, especially when dealing with large datasets because it avoids to store the entire sequence in memory simultaneously.
We can get values from a generator in four main ways: with next() (just writing next(gen) for having the next value), with list() (with list(gen)), looping (for i in gen: print(i)) and with the yield statement. This last one is used in the definition of the generator, for example:

```python
def gen():
    for n in range(10):
        yield (n, n**2)
```

```
g = gen()
next(g) # gives (0,0)
next(g) # gives (1,1)
next(g) # gives (2,4)
```
Using the yield statement, the computation is not actually done until it is asked to.

Packages in Python:
Are folders of modules with a special __init__.py file indicating that the folder contains Python modules; this file is executed when the package or module is imported, so to set package-level variables or define what should be accessible. Packages provide a hierarchical organization for modules.

NumPy
Core library for numerical computing in Python, offers an efficient interface for working with arrays and matrices. Its arrays provide an efficient way to store and manipulate numerical data, offering advantages over traditional Python lists in terms of less memory, more speed and convenience.

Boolean indexing in NumPy:
With NumPy an array of boolean values can be used as an index of another array, where each element of the boolean array indicates whether or not to select the elements from the array (respectively True and False).

```python
a = np.arange(1,100)
b = a % 2 == 0 # When printed returns [True, False, True, False,...]
c = a[b] # When printed returns odd numbers
```

Reference VS deep copy
For higher performance, assignments in Python don't usually copy the underlying object. If you want a deep copy you have to explicitly write it with numpy.copy(value).

Type casting:
Cast an array of some type to another (with astype), creating a new array of new type.

SciPy
Offers higher-level scientific algorithms and modules for specialized tasks like optimization, integration, signal processing, linear algebra. It uses NumPy arrays for efficient data representations.

Matplotlib:

Library for data visualization, offers a wide range of plotting tools with various types of plots like scatter plots, bar plots, histograms; provides a high-level interface for drawing attractive statistical graphics.

Seaborn:
Library for data visualization, provides a more intuitive interface for creating statistical plots, excelling in creating complex ones like heatmaps, pair plots and facet grids. One of its strengths is the ability to work seamlessly with Matplotlib, customizing the plots further.

Pandas:
Library for data manipulation and analysis, provides fast, flexible data structures like Series and DataFrame, designed to work with structured data intuitively and efficiently.
Series are a 1D labeled array that can hold various data types.
DataFrame are 2D size-mutable potentially heterogeneous tabular data structure with labeled axes.
Pandas excels in time series data analysis, offering functionalities for resampling, shifting and window operations.

Pybind11:
Header-only library suitable for a range of projects, enhancing problem-solving skills.

Creating bindings:
PYBIND11_MODULE macro creates a function that will be called when an import statement is issued from within Python. The first macro argument is the module name, the second defines a variable of type py::module_<module> which is the main interface for creating bindings. The method module::def generates binding code that exposes the defined function to Python. Then to import it, you will write import <module> and module.function()
For example:
```bash
#include <pybind11/pybind11.h>
#include <pybind11/stl.h>

// ----------------
// Regular C++ code
// ----------------

void do_something(const std::vector<int> &list) {
  // ...
}

// Wrap as Python module.
PYBIND11_MODULE(example, m) {
```

```
  m.doc() = "pybind11 example plugin";

  m.def("do_something", &do_something, "Do something");
}

// In test.py:
import example
list = [1, 2, 3, 4]
example.do_something(list)
```

Dynamic attributes:
In Python it's possible to pick up new attributes dynamically, but by default classes exported from C++ don't support this. For doing it, we have to use class_::def_readwrite or class_::def_property with py::dynamic_attr when defining the class.

Similarities and differences in object-oriented features between C++ and Python and how they implement inheritance and polymorphism differently:
- A main difference is the lack of pointers in Python, because it automatically manages the memory, contrary to C++.
- Python offers also magic methods, which are special methods automatically invoked in response to certain events or operations, like __add__, __eq__, __getitem__ etc.
- In Python all class members are public and all methods are virtual, which means they can be overloaded anytime. In C++ members can also be private or protected, which is also why you can declare a class as friends of another one (to access its protected members).
- In the class methods, Python requires a first name that refers to the object itself and it's called self; it's the equivalent of the this pointer in C++.
- What said above is also for the constructor in Python, which must take self as first argument and initializes every member with self.nameofmember.
- They both support multiple inheritance.
- While in C++ it is specified with `class DerivateClass : public BaseClass`, in Python it's `class DerivateClass(BaseClass)`.
- If there isn't a constructor for the derived class, the one of the abstract is automatically called in Python.
- When calling a function of the BaseClass, Python can either use its name or the function super().functionToCall.