Documentation of Lab 5

1. An introduction where the problem is described. For example, what should the program do? Which classes do you have to define? Which methods do you have to implement for these classes?

This project aims to develop, in object-oriented C++, a simplified interpreter of the orders to move a graphical turtle. The program should process a list of commands, execute them in the context of a turtle, and test the outcome by showing it.

For that purpose, the following classes will be defined:

- Instruction: An abstract class that provides a common interface to all kinds of specific instruction types; this allows the use of polymorphism.
- TurtleInstruction: A derived class of Instruction that will carry out actions such as movement and rotation on a Turtle object.
- Function: Another derived class of Instruction that will execute a list of statements stored in a Program.
- Logo: Manages a list of instructions and offers functionality to execute them in a controlled environment.

Key methods include:

- For Instruction: A virtual destructor and the toString method for representation.
- For TurtleInstruction: Methods to execute operations on the Turtle object.
- For Function: Methods to manipulate and execute a stored program.
- For Logo: Methods to add instructions to a dictionary and execute them by name.

2. A description of possible alternative solutions that were discussed, and a description of the chosen solution and the reason for choosing this solution rather than others. It is also a good idea to mention the related theoretical concepts of object-oriented programming that were applied as part of the solution.

For implementation, a few different solutions had to be considered:

- Using Direct References vs. Pointers:
    - Direct References: this hugely simplifies the memory management; however, one couldn't take advantage of the polymorphism in instruction processing-for example, storing different types in the same container.
    - Pointers: The use of pointers here is because, through polymorphism, the Logo class is able to store, in one single list, different types of instructions comprising TurtleInstruction and Function amongst others. This is a better approach toward object-oriented principles at least with regards to polymorphism and inheritance.
- Monolithic vs. Modular Design:
    - Monolithic Design: The other would have thrown everything in one class, from moving turtles to processing instructions. It was discarded for the reason that it is not going to be flexible and hard to maintain.
    - The module design will look like this: Each class has one job: a Turtle is responsible for moves, an Instruction for generic logic, and a Logo for maintaining the flow of a program. In return, this will provide good maintainability, scalability, with adherence to object-oriented design principles such as encapsulation and separation of concerns.

The theoretical concepts put into action here were thus:

- Inheritance: The classes TurtleInstruction and Function are derived from the Instruction class to build specific types of instructions.

- Polymorphism: Virtual methods such as execute() enable the program to be manipulated uniformly for different types of instructions. Encapsulation: Each class is responsible for its state, and through methods, it exposes to the outside interface so that the data is taken care of appropriately. Composition: The class Logo consists of elements of type Instruction, while it collaborates with Turtle to realize most of its activities.

3. A conclusion that describes how well the solution worked in practice, i.e. did the tests show that the classes were correctly implemented? You can also mention any difficulties during the implementation as well as any doubts you might have had.

It also worked just fine in real life. The tests gave proof that everything is implemented correctly-the turtle reacted to the commands as needed. The output after running the test showed that, after every instruction was performed, the state of the turtle did change; the flow was correct.

Debugging also utilized std::cout for validation of the implementation. After executing the instructions which were MoveForward and Rotate, this is where the position and angle came in handy; it was easily verifiable in the functionality of the program. For instance, after executing "MoveForward," the position of the turtle moved just right along the Y-axis and, after "Rotate," the angle was updated.

Several complications ensued during the implementation mainly of how to manage pointers, specifically when classes did communications with each other. But that is resolved by being mindful of what goes on in constructors, destructors, and pointers in usage. Also, having proper use of polymorphism, notably in the Logo class storing instruction when required from C++ pointer management should be well understood.

Overall, the design was able to successfully implement a working turtle graphics system; the codebase was modular and, hence, extendable and maintainable. Further work could be done on adding more complex instructions or error handling for undefined instructions and testing with larger programs for robustness.