# COMPENG 2SH4 Project – Peer Evaluation [25 Marks]

Your Team Members Muhammad Haseeb Aslam, Sammy Abbas_

Team Members Evaluated laeeqm, Khana421

Provide your genuine and engineeringly verifiable feedback. Ungrounded claims will lead to deductions. Completing the peer code evaluation on time will earn your team a total of **25 marks**. Do not exceed 2 paragraphs per question.

## Peer Code Review: OOD Quality

1. **[3 marks]** Examine the main logic in the main program loop. Can you easily interpret how the objects interact with each other in the program logic through the code? Comment on what you have observed, both positive and negative features.

   The main program loop effectively demonstrates clear object interactions, with each object (like the player and food) being managed through well-defined classes. The objPos class is particularly well-structured, using dynamic memory allocation to handle object positions and symbols. This ensures each object maintains its own state and behaves independently within the game. The use of constructors—default, parameterized, and copy—allows for efficient object initialization, while methods like setObjPos and getObjPos help keep the code clean and readable by encapsulating the logic for object management.
   However, there are a couple of areas for improvement. One would be to pass objects by reference (e.g., const objPos&) in methods like setObjPos, which would prevent unnecessary object copying and improve performance. Additionally, the assignment operator could be optimized by checking for existing memory allocations before copying, ensuring that memory is handled more robustly and reducing the chance of memory issues.

2. **[3 marks]** Quickly summarize in point form the pros and cons of the C++ OOD approach in the project versus the C procedural design approach in PPA3.

   Pros:
   Modularity: Using classes and objects made the code more modular, facilitating easier management and future feature extensions without disrupting existing code.

   Reusability: Inheritance enables code reuse, reducing redundancy.

   Clearer Structure: The use of classes for real-world entities makes the code structure more intuitive compared to PPA3's procedural approach.

   Cons:
   Complexity: OOD introduced more complexity, making the architecture more complicated.

   Performance Overhead: Object instantiation and method calls created some memory and processing overhead compared to the procedural approach

## Peer Code Review: Code Quality

1. **[3 marks]** Does the code offer sufficient comments, or deploys sufficient self-documenting coding style, to help you understand the code functionality more efficiently? If any shortcoming is observed, discuss how you would improve it.

   Overall, the code is well-commented and strikes a good balance between being over or under commented. The comments are especially helpful in explaining more complex sections of the code, like the initialization of the player's position and the use of object references. Additionally, the use of good variable names allows for easy understanding of what is being updated or referenced, without the need to constantly check what the variable represents. This is crucial for writing clean and understandable code. I found it easy to follow the logic due to these comments and variable choices.
   However, there is one area I would suggest improving: the use of the phrase "self-explanatory" in comments. For example, in the updatePlayerDir() method, the comment "PPA3 input processing logic, self-explanatory" may not be very helpful for someone unfamiliar with the specific implementation. It's better to describe what the logic does or why it's important. A more helpful comment could be: "Processes player input to update movement direction, ensuring no reverse movement." This makes the intent clearer for anyone reading the code, even if they're not familiar with the specifics of the PPA3 input processing.

2. **[3 marks]** Does the code follow good indentation, add sensible white spaces, and deploys newline formatting for better readability? If any shortcoming is observed, discuss how you would improve it.

   Overall, the code follows good indentation and utilizes sensible whitespace, making it easy to read and follow. The logical structure is clear, and the code is well-organized. It's easy to follow the flow of the program, which is a testament to the well-thought-out formatting.
   However, there are a couple of minor improvements that could be made. First, adding spaces around operators (such as =, ==, and !=) would enhance readability. This small adjustment would make the code more visually consistent and easier to scan. Secondly, there is some inconsistency in the use of newline spacing between functions. While most functions are separated by a single blank line, a few have multiple blank lines. To improve consistency, I would recommend using a single blank line between each function throughout the entire code. These small changes would make the code even more polished and easier to navigate.

## Peer Code Review: Quick Functional Evaluation

1. **[3 marks]** Does the Snake Game offer smooth, bug-free playing experience? Document any buggy features and use your COMPENG 2SH4 programming knowledge to propose the possible root cause and the potential debugging approaches you'd recommend the other team to deploy. (NOT a debugging report, just technical user feedback)

   After playing the game a few times and reaching a higher score, there weren't any major bugs found in the gameplay. However, as the snake length increased, it felt like the game started lagging a bit. One of the reasons could be frequent DMA while running the program. If new memory for the snake's body is dynamically allocated and deallocated during gameplay, this can cause performance issues as the snake's size increases. Using stack-based variables might be able to slightly improve the gameplay.

Another reason might be due to using an inefficient data structure like an Array List. Larger arrays mean more iterations for collision detection, leading to slower updates. We can try using relatively efficient data structures like a **linked list** for dynamic resizing without reallocation.

2. **[3 marks]** Does the Snake Game cause memory leak? If yes, provide a digest of the memory profiling report and identify the possible root cause(s) of the memory leakage.

   No, the game doesn't cause any memory leakage as after each **new** that we see in the constructor of various classes or in the Initialize() function, the other team made sure to implement a **delete** in the destructor or cleanup method of each class. There are both new and delete in ObjPosArrayList, objPos, Player classes and the Project.cpp.
   (We weren't able to run the memory report using Dr. Memory, as both of can't run it, and we already let Dr. Athar know).

## Project Reflection

Recall the unusual objPos class design with the additional Pos struct. After reviewing the other team's implementation in addition to your own, reflect on the following questions:

1. **[3 marks]** Do you think the compound object design of objPos class is sensible? Why or why not?

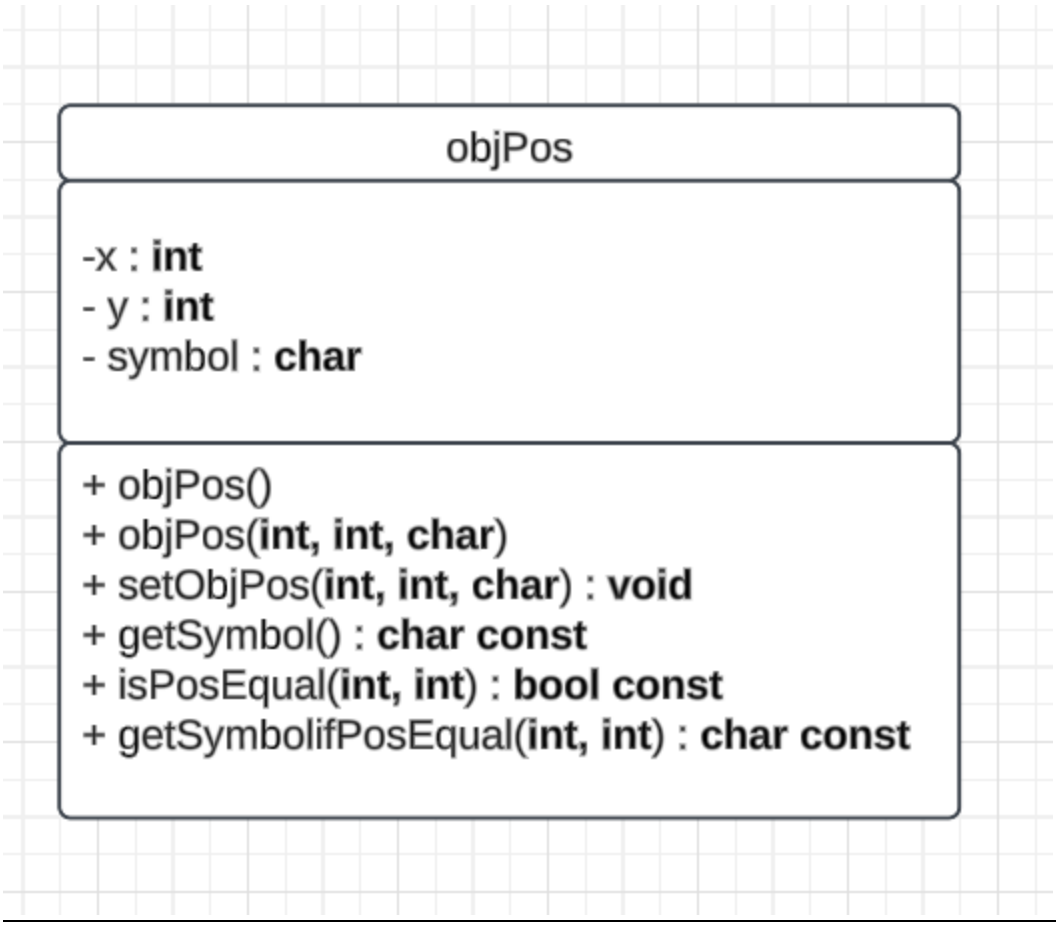
I don't thing that the design of objPos is entirely sensible due to the following reasons:

a- **Unnecessary Complexity**:
Using `Pos*` for a simple struct like `Pos` adds in a few complications. Since `Pos` only contains two integers (`x` and `y`), it could easily be stored directly as a member variable of the `objPos` class, reducing memory management complexity and potential errors (like forgetting to delete pos).

b- **Performance**: Dynamic memory allocation using new is slower compared to using stack-based variables. This could impact performance in cases where many `objPos` objects are created and destroyed frequently.

2. **[4 marks]** If yes, discuss about an alternative objPos class design that you believe is relatively counterintuitive than the one in this project. If not, explain how you'd improve the object design. <u>You are expected to facilitate the discussion with UML diagram(s).</u>

   Instead of using a dynamically allocated pointer for `Pos`, the `Pos` struct should be directly included as a value-type member of the `objPos` class. This simplifies memory management, enhances performance, and reduces the risk of bugs. By avoiding heap allocation, this leads to faster object creation and destruction. There is Less risk of memory leaks, which simplifies debugging and maintenance.




*objPos Class UML Diagram*