COMPENG 2SH4 Project – Peer Evaluation [25 Marks]

Your Team Members Team_Aaryon	Aaryon Arora (aroraa68) and Jacob Chisholm (chishj3) –
Team Members Evaluated Lena Ma & Davina Cao	Lena Mao (mal55-lenama) and Davina Cao (caod12) -

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.
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.

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.
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.

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)
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.

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?
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. You are expected to facilitate the discussion with UML diagram(s).

OOD Quality:

1. The main logic in the program loop is quite easy to follow. This group made good use of programming structure by calling their functions in a linear order. Every function that does an action in the game is all called in the main Project.cpp file. This minimizes the number of functions that call other functions. If you drew out a visualization of function calls for this project, it would be linear without much branching out. I can easily see how objects interact with each other.

   The most complex files in this project are the player class and the project file. This is simply because it has the bulk of the logic and where multiple objects interact with each other. Even in these files, it is fairly easy to follow along. In the player class, you have functions handling movement change based on key presses, move mechanics with the snake growing and shrinking as well as food consumption detection. The project.cpp file is the core of the game with functions handling the run logic, a function dedicated to printing everything on the screen and some functions to initialize the game and delete any storage on the heap.

   Some positive features I like is the addition of a win condition which is not something we even thought about and employs adding additional member functions and member fields in the GameMechs class. There weren't any negative features that came to mind while observing the code.

2. Right away the greatest pro of the OOD approach in C++ is greater code organization and formatting. Having code in multiple files and modularized into functions makes it much easier to keep track of. A con of this could be that it is harder to get started with a project because you need to dedicate more time into learning what each function does and the purpose of each class. Another pro to the OOD approach is code reusability. The most prevalent example of this is objPos class. The snake and food items depend on this to create an item on the board and allows an easy way to keep track of the objects symbol and position. By making it into a class it allows an easy way to create a new object in just one line of code and print it out in another. By extension the objPosArrayList class is also very useful because it allows you to easily manipulate the snake while keeping the inner mechanics of it hidden in another file that you don't even need to touch once properly implemented.

   Overall, OOD is far superior to the procedural design approach in C but it is still a bit of a learning curve. Although this project was a huge undertaking, coding in C is sometimes faster for smaller game. For bigger games like snake, taking the time to lay the foundation for all the classes reaps huge payoffs when coding the logic of the game.

Code Quality

3. The code offers a great number of comments that easily explains the logistics of what each part of the code does in a function and what each function does. I would say the only shortcoming is not explaining what each class is supposed to do (The player class holding the logic for the snake for example) but it still offers concise and efficient explanations for every piece of code in every file. For example, in the player class for the updatePlayerDir(), the group provides insight on the if statements in each case of the switch statement. They said that a snake can't turn 180 degrees so if the snake is already moving down, it can't switch up. Although this is basic knowledge for someone who's already coded this game before or even played snake, to a complete novice it offers a good explanation of the coding decisions made.

4. The code makes great use of all formatting practices, and as such, reading the code is both easy and intuitive. Though only very minor, the only perceptible shortcoming in this regard is the lack of whitespace and newline formatting used on the getters amongst some of the classes, such as in GameMechs. I do not personally see the issue with formatting the getters this way, as getter methods are quite simple and skeletal in nature, however I can see how a novice coder might get confused trying to interpret the large block of one line getter methods. If not to implement newline formatting on the getters, maybe just including spaces between the getters themselves may prove helpful to people reading the code.

Quick Functional Evaluation:

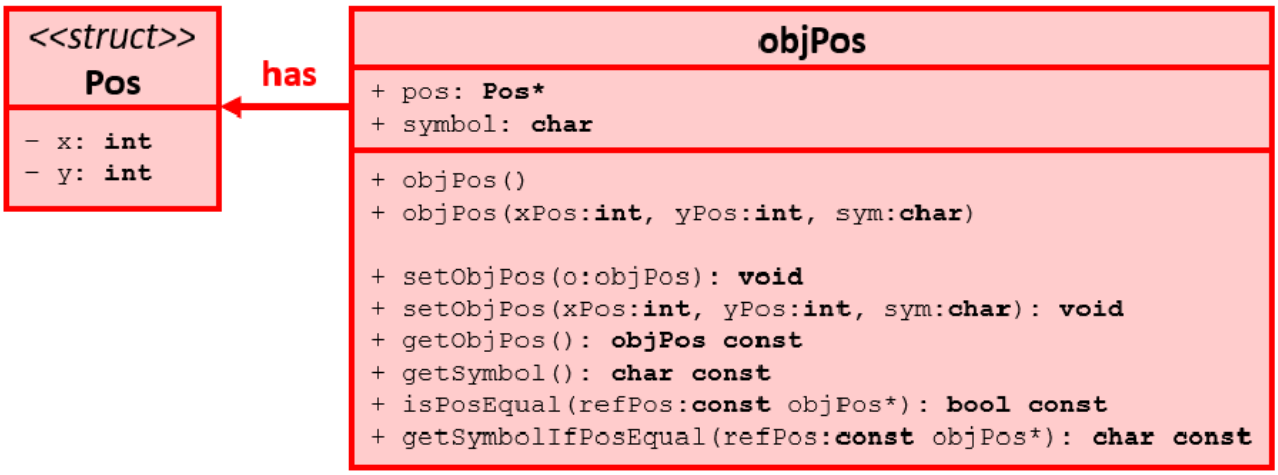
5. The game from my evaluation does not present any obvious bugs or technical issues that interfere with a smooth gameplay. Once again, while minor, there are only two points of feedback I would possibly provide to make the gameplay more robust, though to reiterate they are not bugs. One, I think the game UI would benefit from more instructions for the player. For example, I intuitively assumed the game controls would be WASD, but they could've just as easily have been arrow keys. The game window does not specify this. Second, it might be less visually confusing if the symbol for the snake and the food were differentiated. They both use the * symbol.

6. The game does cause memory leak. Most prominently, the Dr memory report of the code shows that compiling the code incurs 8 bytes worth of memory leaks. The most likely cause of this issue is rooted in the destructor in the objPos class. In the destructor, the group deallocates memory related to the position in the form: delete[] pos;. However, pos is not dynamically allocated on the heap as an array. Pos, rather, is a single struct object. Thus, using the [] suffix on delete may lead to undefined behaviours.

Project Reflection:

7. I do not think the compound object design of the objPos is very sensible. Having a Pos struct does make sense in theory but in application it makes the code very messy. In both the other groups code and ours, we must use the individual x and y position of the object very frequently. To do this, you need to call object.pos->x and object.pos->y. Calling this many times just increases the complexity of reading your code and impacts the organization of your overall program. Having a struct wasn't very sensible to keep track of the x and y coordinates of the player.

8. Given that we believe that the compound design and the inclusion of a struct inhibits code readability, we do believe that there is a more sensible approach. Simply, it would be more beneficial to remove the struct and to handle position declarations internally from within the class.



As we can see from the given UML diagram, there is an association between the objPos class and the Pos struct, but the objPos class merely aggregates the Pos struct. The objPos class contains a Pos type pointer that allows the x-y coordinates to be instantiated on the heap, but there is no composition relationship present here. Thus, the Pos* pointer can be removed, and alternatively, we could declare two int* pointers for x and y respectively. Functionality would still carry over the same to the setters and getters present in the class provided the code is refactored (from pos->x = thisX to x = thisX, for example).