COMPENG 2SH4 Project – Peer Evaluation [25 Marks]

Your Team Members	hello world -	lebely1	micham19
Team Members Evaluated	ScottChen1A -	dcostd1	bojcevg

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).

Peer Code Review: OOD Quality

1. Primary Program Loop Interaction

- The workflow is simple because it's what we've been doing: taking inputs, executing their logic, drawing code onto the screen, and delaying execution. It is representation of modularized design to say that here interaction is through passing the references of GameMechs and Food into Player class.

2. OOD C++ vs Procedural C Design

Pros:

- Better encapsulation by such classes as GameMechs, Player, etc., making the design much more modular and repetitive in code.
- Management of states and objects in the game is simplified by clarity of boundaries on responsibilities.

Cons:

- Sometimes it's hard to fully understand the logic.
- Some design decisions, like passing references between a multitude of objects, may be very difficult to untangle without good enough documentation.

Peer Code Review: Code Quality

1. The comments are sufficient in explaining why and how a particular function is written. For example, position generation, player movement, and major parts of game mechanics. Moreover, descriptive methods and variable names amply contribute to the self-documenting style of the code, making it easily understandable.

2. The code has almost perfect spacing and indentation at every spot for great, read-on appearances of presentation of the code. It provides sensible use of newlines to enhance visual structure. Regular indentation allows easier tracing by a human through the source code. In this perspective, no shortcoming could be observed, and formatting followed the common principles of programming closely. Because of that, the code is quite readable.

Peer Code Review: Quick Functional Evaluation

1. The Snake Game runs smoothly and is mostly bug-free. However, occasional issues with snake body collision detection suggest that edge cases, such as rapid directional changes, could cause self-collision logic to fail. Using debug print statements or step-by-step logging during gameplay could help identify and resolve such issues.

2. No memory leaks were found within normal gameplay, since destructors for dynamic objects such as Player and food were well implemented.

```
~~Dr.M~~ Note: @0:00:02.936 in thread 30952
~~Dr.M~~ Note: instruction: cmp    %eax %ecx
~~Dr.M~~
~~Dr.M~~ ERRORS FOUND:
~~Dr.M~~       0 unique,     0 total unaddressable access(es)
~~Dr.M~~       8 unique,    11 total uninitialized access(es)
~~Dr.M~~       1 unique,    62 total invalid heap argument(s)
~~Dr.M~~       0 unique,     0 total GDI usage error(s)
~~Dr.M~~       0 unique,     0 total handle leak(s)
~~Dr.M~~       0 unique,     0 total warning(s)
~~Dr.M~~       0 unique,     0 total,      0 byte(s) of leak(s)
~~Dr.M~~       0 unique,     0 total,      0 byte(s) of possible leak(s)
~~Dr.M~~ ERRORS IGNORED:
```

Project Reflection

1. The objPos class compound object design makes sense and is effective; adding a Pos struct adds sensible cohesion between position (x and y) and symbol data into one manageable unit, which in turn helps to manage entities on the game board. The design uses the additional Pos struct to simplify a number of operations, such as collision checks, boundary handling, and comparison of positions, while the main objPos class remains flexible for future extensions. All this greatly enhances modularity and readability of the code.

2. Another possible design is to not use the structure Pos and instead include x and y in the objPos class as a private member. This may be reducing the complexity, but this may make your code less modular and will be complicated to scale in case other attributes about positions need to be added in the future (for example, velocity, direction). However, if such change is implemented, then any of the functions that handle the position of the object need to be split into two separate functions for x and y. For example, the getObjPos() function inside the class would have to be split into two getter functions for each of the positions instead of the getObjPos() function. The other could be a dynamic array, vector that keeps the position with greater flexibility in the case of multi-dimensional game boards. Overall, the current design provides a good balance between simplicity and extensibility; further improvements can be explored depending on specific project requirements.

ObjPos
-x: **int** -y: **int** -symbol: **char**
+ objPos() + objPos(xPos: **int**, yPos: **int**, sym: **char**) + setObjPos(o:objPos): **void** + setObjPos(xPos: **int**, yPos: **int**, sym: **char**): **void** + getPosX(): **int const** + getPosY(): **int const** + getSymbol(): **char const** + isPosXEqual(refPosX: const int): **bool const** + isPosYEqual(refPosY: **const int**): **bool const** + getSymbolIfPosEqual(refPosX: **const int**, refPosY: **const int**): **char const**