COMPENG 2SH4 Project – Peer Evaluation [30 Marks]

Your Team Members Prestion McMann (mcmannp) Katarina Rakulj (rakuljk)

Team Members Evaluated Garish Manivannan (manivang) Aashish Paul (paula41)

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 **30 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 program follows the overall flow of collecting input, running logic, drawing on the screen, and cleaning up. The main() function is a little challenging to interpret as the program should exit the while loop without needing Runlogic() to return a value (changing the flag in the gamemechs object). The Draw screen function is hard coded with a board instead of being easily modifiable by collecting the dimensions from the gamemechs class. The rest of the code where you add the snake/player element into the board which was implemented correctly by interacting with the game board size in the gamemechs class. The print statements were also done well as they grab the values from the objects such as the score, state, and keys pressed.

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:

- Encapsulations: Having classes like player, gamemechs and food promote modularity which makes the code neater and easily interpretable
- Reusability: Classes like player or food can be used more than once to allow for more complex game modes using repeated lines
- Improved debugging: having private/protected data and functions inside an object increase the rate of find a bug as it will be easy to identify based on the behaviour of an object instead of the entire project.

Cons:

- Memory Leaks: Higher chance of memory leaks because of dynamic memory management such as heap allocation
- Complex: Follows a different flow than our brains are used to as objects can interact with each other
- Repetitive: Writing getting's and setters and following rule of 6 can get tedious for every class

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.

The code has sufficient comments which make it super easy to follow along. The player class especially is well documented. The only criticism is that there are too many comments which hinders my ability to effectively understand the code. For example, an improvement would be to make a quick summary above a section to explain its purpose instead of commenting on each line individually.

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.

The code does a great job with indentations and how they format the display text. They have a good use of the '\n' which makes the program display format much more user friendly. The code is very easy to follow which either a single line or 2 between each function that clearly distinct the end of a section of code. They could have added spaces ahead of the text to make the writing centered with the display board as a minor improvement.

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)

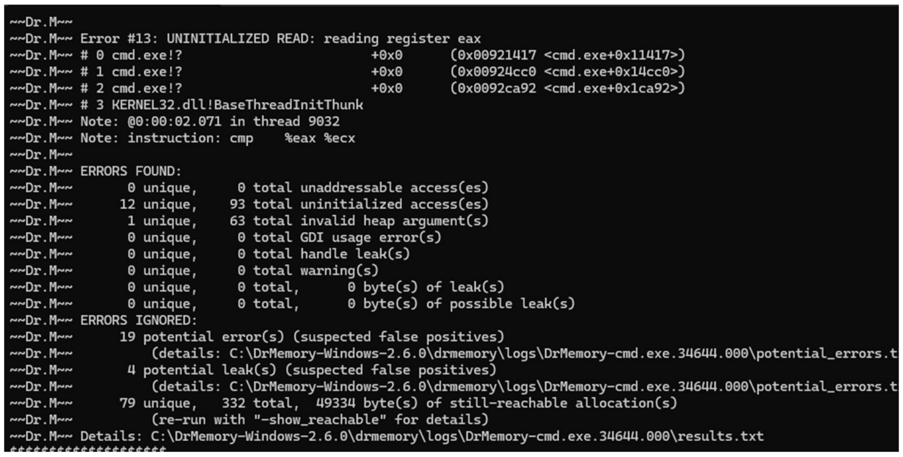
The Snake Game provides a smooth, bug-free playing experience. During testing, all core functionalities performed seamlessly: the snake moved fluidly around the screen, direction changes were responsive (respecting the 180 deg parameter), and score updates occurred instantly when food was consumed. The game handled both natural game-over scenarios and forced exits without delays or glitches, ensuring a consistently enjoyable and reliable user experience.

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 Snake Game does not cause any memory leaks. The developers have correctly implemented destructors, such as:

```
objPos::~objPos() {
   delete pos;  // Deletes the dynamically allocated struct to free memory
}
```

This ensures that dynamically allocated memory (such as the **pos** struct in **objPos**) is properly deleted when the object is destroyed, preventing memory leaks. The destructor's role in releasing allocated memory is confirmed in the code, and as shown in the screenshot below, there are no instances of memory leakage.



```
~~Dr.M~~
~~Dr.M~~ Error #13: UNINITIALIZED READ: reading register eax
~~Dr.M~~ # 0 cmd.exe!?                          +0x0      (0x00921417 <cmd.exe+0x11417>)
~~Dr.M~~ # 1 cmd.exe!?                          +0x0      (0x00924cc0 <cmd.exe+0x14cc0>)
~~Dr.M~~ # 2 cmd.exe!?                          +0x0      (0x0092ca92 <cmd.exe+0x1ca92>)
~~Dr.M~~ # 3 KERNEL32.dll!BaseThreadInitThunk
~~Dr.M~~ Note: @0:00:02.071 in thread 9032
~~Dr.M~~ Note: instruction: cmp    %eax %ecx
~~Dr.M~~
~~Dr.M~~ ERRORS FOUND:
~~Dr.M~~       0 unique,     0 total unaddressable access(es)
~~Dr.M~~      12 unique,    93 total uninitialized access(es)
~~Dr.M~~       1 unique,    63 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:
~~Dr.M~~      19 potential error(s) (suspected false positives)
~~Dr.M~~          (details: C:\DrMemory-Windows-2.6.0\drmemory\logs\DrMemory-cmd.exe.34644.000\potential_errors.t
~~Dr.M~~       4 potential leak(s) (suspected false positives)
~~Dr.M~~          (details: C:\DrMemory-Windows-2.6.0\drmemory\logs\DrMemory-cmd.exe.34644.000\potential_errors.t
~~Dr.M~~      79 unique,   332 total,  49334 byte(s) of still-reachable allocation(s)
~~Dr.M~~          (re-run with "-show_reachable" for details)
~~Dr.M~~ Details: C:\DrMemory-Windows-2.6.0\drmemory\logs\DrMemory-cmd.exe.34644.000\results.txt
```

Project Reflection

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

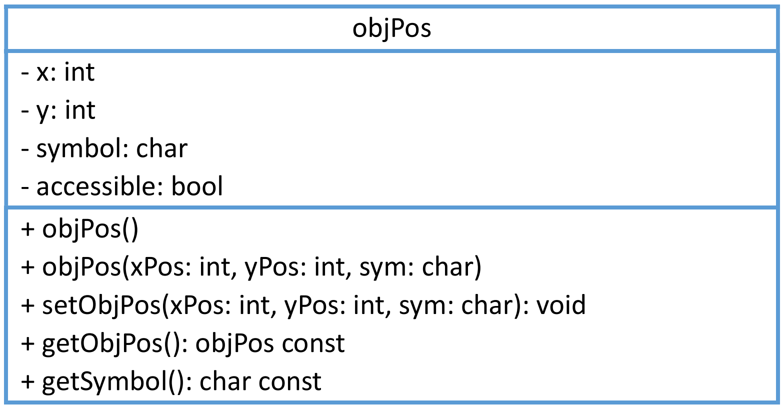
1. **[3 marks]** Do you think the compound object design of objPos class is sensible? Why or why not?

Yes, I think the compound object design of the **objPos** class makes sense. By using a separate Pos structure for position data, the design keeps things organized and clear, separating the position details from the overall object. This setup can be helpful if the Pos structure needs to be expanded later—for example, if you wanted to add more attributes like a z coordinate for 3D positions.

Using dynamic memory allocation ensures that each **objPos** instance has its own independent position data, which helps avoid issues with shared data. The implementation of copy constructors and assignment operators shows good handling of deep copies, which is important for preventing memory leaks and keeping things stable.

That said, since Pos only has two integers, the dynamic allocation might be a bit more complicated than necessary. You could simplify the design by having **int x** and **int y** directly in the **objPos** class without losing much functionality. So overall, it's a solid design, especially if it needs to be extended in the future, but there's room to simplify depending on the project's needs.

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.</u>



objPos

- x: int
- y: int
- symbol: char
- accessible: bool

+ objPos()
+ objPos(xPos: int, yPos: int, sym: char)
+ setObjPos(xPos: int, yPos: int, sym: char): void
+ getObjPos(): objPos const
+ getSymbol(): char const

+ isPosEqual(refPos: const objPos*): bool const
+ getSymbolIfPosEqual(refPos: const objPos*): char const

The diagram above presents an alternative design for the **objPos** class compared to the one used in the project. In this approach, the x and y coordinates are directly integrated into the class as separate integer attributes, rather than being encapsulated within a **Pos** structure and accessed through a pointer.

Advantages of this design:

- **Cleaner code structure:** Accessing the coordinates becomes more straightforward, as there is no need to dereference pointers, making the code easier to read and maintain.
- **Reduced risk of pointer-related bugs:** Eliminating pointers minimizes the chances of encountering bugs related to null pointers or dangling references, enhancing the overall stability of the program.

Disadvantages of this design:

- **Scalability concerns:** For larger programs, the use of pointers can be more efficient, especially when passing objects, as it avoids the overhead associated with copying entire structures. Removing the pointer-based approach may limit flexibility if additional position-related attributes need to be added in the future.

Another change could also include a new boolean attribute named **accessible**, which indicates whether a specific position can be used or blocked. This feature could enhance gameplay by preventing certain coordinates from being accessed. For instance, during food generation, the program could check if the **accessible** flag is false, indicating that the spot is occupied (e.g., by the snake), thereby preventing food from spawning in that location. This approach offers simplicity and reduces potential errors but may require reconsideration if the project scales significantly.