COMPENG 2SH4 Project – Peer Evaluation [25 Marks]

Your Team Members Kamyar Lakdashi [lakdashk] Kai Hughes [hughek26]

Team Members Evaluated Elijah James [jamese13] Grace Jonker [jonkerg]

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.

Observing how objects interact with each other is easy by use of appropriate variable names and general formatting. The use of special character models is a creative feature and adds a stronger "game" aspect to the project – this was well done. I do find, however, that the debug mechanism in DrawScreen(), called from GameMechs, should be less invasive in the final completion of the main project file. A possible solution to this would be create a separate modular function that can be used for debugging – this would improve the overall readability and help future developers who would work on the code. Also, it is generally frowned upon to call the same method more than once without any change to the value. To improve this, observe the Drawscreen() method where playerPositions->getElement(i) is called each time throughout the for loop, it would be more efficient to set that to a temp objPos so that the method is only called once, rather than 3 times. For example:

```
1.  for (int i = 0; i < playerPositions->getSize(); ++i)
2.     {
3.       objPos thisSeg = playerPositions->getElement(i);
4.       if (thisSeg.pos->x == col && thisSeg.pos->y == row)
5.          {
6.            contained = true;
7.            symbol = (thisSeg.getSymbol());
8.          }
9.     }
```

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:

- Encapsulation: The use of object classes encapsulates data and behavior, improving code modularity and maintainability.
- Separation of Concerns: Different parts of the game logic (player movement, game mechanics, food generation) are separated into distinct classes, making the code more organized and easier to modify.
- Code Reusability: Object-oriented design facilitates reuse of code through inheritance and polymorphism (though not heavily used here).

- Ease of Extension: New features (like additional game mechanics or player abilities) can be added easily by extending existing classes.

Cons:

- Complexity: Creating a game with very similar features was more difficult because it required a level of discipline to manage how objects interact with and reference each other.
- Performance Overhead: The overhead of object management (e.g., memory allocation and deallocation, virtual function calls) could potentially impact performance in resource-constrained environments, especially for a terminal game like snake game

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.

For the general readability, from someone who developed a similar project, I had no issues going through and understanding the code functionality. However, there is a clear inconsistency between code documentation in each class development. More specifically, some examples of over commenting can be seen in the movePlayer() method within the Player class - each movement does not need to be described as exact movement and type of wraparound. Given that the enums were labeled according, commenting for the exact same movement for each case seems redundant. Likewise, throughout the ArrayList class, each method will not benefit from such heavy annotations especially simple getter methods – infact, it disturbs the general readability for other developers. Opposingly, more complex algorithms like randomly generating food contain no comments or in movePlayer() the increasePlayerLength() is called and commented “check for food consumption”, this comment should be moved into the actual increasePlayerLength() method when the checkFoodConsumption() is called.

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.

Throughout the source code, there is a clear attention towards code formatting. A great use of indentations, and white spaces where applicable in for loops, conditionals etc. The only issue with the readability, again, stems from some areas being over commented.

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 core functionality, as a user, had no bugs and worked smoothly. The wraparound, FSM, food consumption, and self collision appears to work flawlessly. This would be a good indication that all the objects are interacting with each other well. I did find when playing the game that it was easy to accidently interact with the 'f' key on the keyboard due to the nature of the WASD layout of the game – this changed the location of the food which, from deeper observation, was a debugging feature. To fix this issue, either removing the debugging features before final publication, or changing the food debugging to a more discreet key would be helpful to the user. Also, before closing the program, the user will often run into a bug when attempting to exit the program where it will buffer for a couple seconds before properly exiting out. To fix this, double check that all destructors are properly written and there are no segmentation errors throughout the code.

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.

This snake game does have memory leaks, the memory profiler indicates:

```
1. ERRORS FOUND:
2.       2 unique,     2 total unaddressable access(es)
3.       8 unique,    64 total uninitialized access(es)
4.       1 unique,     2 total invalid heap argument(s)
5.       0 unique,     0 total GDI usage error(s)
6.       0 unique,     0 total handle leak(s)
7.       0 unique,     0 total warning(s)
8.       3 unique,     4 total,   3240 byte(s) of leak(s)
9.       0 unique,     0 total,      0 byte(s) of possible leak(s)
```

Interpreting this, line 2 indicates that there is likely a segmentation error throughout the code as the program is attempting to access a location that is "unaddressable". Line 3 indicates that an object on the heap was not properly terminated before the program ended. These observations also explain the bug mentioned in the previous question.

To fix these bugs, firstly, rewrite the Player destructor. The objPosArrayList, is simply an object initialized on the heap from "new objPosArrayList", but, in the destructor the delete[] keyword is called. Next issue is with the copy assignment that does not properly perform deep copy.

Rewrite copy assignment:

```
1. objPos &objPos::operator=(const objPos &sObjPos)
2. {
3.     if (this != &sObjPos)
4.     {
5.         *pos = * sObjPos.pos;
6.         symbol = sObjPos.symbol;
7.     }
8.     return *this;
9. }
10.
```

With both these changes, there will be no more memory leaks.

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).

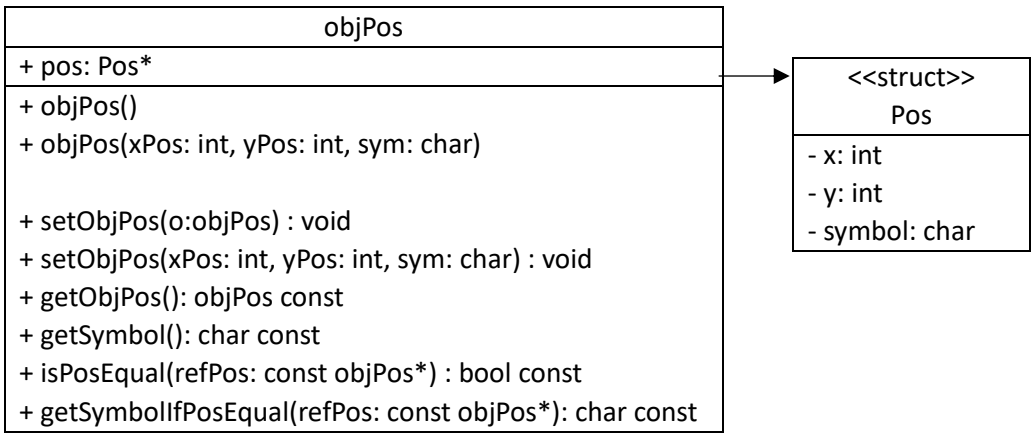
1.

The design of the objPos class is functional but can be greatly simplified when the sole objective is to create this snake game. The class encapsulates pos (x, y) and a symbol, which is useful for representing game objects. However, the dynamic allocation of the Pos struct (Pos* pos) seems unnecessary. Given the lack of complexity and only containing two integers it would make more sense to use a plain Pos member directly, avoiding the complexity of dynamic memory management (e.g., new/delete), reducing the risk of memory leaks or errors. Additionally, the copy constructor and assignment operator work but are more complicated than needed from this use of pointers. If Pos were stored directly with x and y, these methods could be simplified with the same functionality and much easier to debug for other developers. Either x,y, and symbol are within the compounded struct, or x and y are brought into public scope within the objPos class.

Also, methods provided like getSymbolIfPosEqual() return 0 when positions don't match, which does not seem particularly sensible because the return boolean is not useful. Our team found this so useless that we overwrote it and had it return ' ' if it was not equal so that it could serve some use when called in the drawscreen().

Ultimately, the objPos class, although functionally "sensible", it can be improved by removing the pointer-based memory management for Pos which would simplify and avoid issues. The class design is functional but could be made more efficient, maintainable, and easier for other developers to read.

2.



This design example still would utilize the dynamic memory but would include symbol in the struct.

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

This design example would be able to get rid of the need for dynamic memory allocation as each objPos would have an accessible x,y, and symbol element. This seems more intuitive and would eliminate the need for pointers that are potentially dangerous when not handled properly.