COMPENG 2SH4 Course Project [100 marks]

Watch Briefing Video! **Not An Autograded Evaluation**

You must submit all the source codes to both the Git Repository and Avenue to Learn dropbox before the posted deadline.

IMPORTANT: Your submission must compile without syntactic errors to receive grades. Non-compilable solutions will not be graded.

IMPORTANT: Submissions not respecting mandatory OOD criteria will not be graded. Procedural Programming Styled Solution will not be graded.

About Course Project

- Use the following link to accept the project invitation: https://classroom.github.com/a/mLqiHWLE
- Project code submission is due on **Monday, Dec 2, 2024, at 11:59 pm** on both Github and Avenue.
- Project peer evaluation and reflection is due on **Thursday, Dec 5, 2024, at 5:00 pm** on Avenue.
- The course project is a **2-person group project** (Maximum 2 members per group).
- Project evaluation is submission-based – no in-person demonstration will be required.
  The teaching team will pull your submissions from the GitHub repo, compile your project locally, and functionally evaluate your submission based on the provided marking scheme.
- Invitations to in-person demonstration will be issued on case-by-case basis in case of grading issues.

Tips for Completing the Course Project

- **Attend live coding sessions in the lectures and tutorials.** There will be professor-led code development activities to help guide you to complete key milestones in the project.
- **Watch briefing videos.** The videos will give you critical tips and OOD design strategies to expedite your project development.
- **Use the Manual.** The manual, as usual, contains a recommended workflow to provide guidance in good OOD programming practices and sustainable OOD development strategies. This is particularly helpful for students with less OOD programming experience.
- **Use the provided Unit Test plan.** Certain common objects in the project should be tested and proven to be working before being deployed. Do not skip steps – it will develop into a bigger debugging hole very quickly.
- **Use all the debugging techniques.** Do not shoot in the dark.
- **Respect Incremental Engineering.** Do not ever write a large chunk of code without confirming its correctness in all the intermediate steps – it's a very bad practice that results in heavy and unnecessary debugging effort, and most certainly considered unprofessional in the industry.

Project Introduction

Snake game is one of the most common introductory programming projects that allows many programming principles to be applied in practical scenarios. The snake game can be further customized with different data structures and algorithms for improved user experience and increased variety of gaming modes. As a result, the CE2SH4 project will take on the C++ OOD version of **snake game**, constructed from the ground-up using all the C procedural and C++ OOD knowledge we've covered in the course. Moving into the subsequent COMPENG courses, more customizations and improvements will be added to the CE2SH4 project, thereby illustrating the advantages and applicability of different data structures and algorithms in CE2SI3 and CE3SM4. It is important to note that the design experience covered in this project is mostly transferrable to other programming languages – even including the assembly language in CE2DX3.

A "Classic Mode" snake game starts with randomly generated snake food anywhere on the game board, and a snake head with zero body segments in another non-repeating random location somewhere on the game board. The snake can move around the game board freely using the four-direction keys and can wrap around the boundaries. Upon collecting each food item, the snake body segment should increase by 1 unit, and the player score will increase by 1 accordingly. The player fails if the snake head runs into its body and the game is over.

Your project team's goal is to create the same game using the OOD principles described in the Project Procedure section. Submissions not written in C++ OOD structure will not be accepted.

A sample executable is provided for your reference. All above-and-beyond features are not included in it.

Project Starter Code

The project starts with the same skeleton code as in the PPAs, with only the game loop and the MacUI Library implemented in the program. In addition, the skeleton code includes one composite class, "objPos", with a "Pos" struct pointer member that tracks the x-y coordinate information on the heap, as well as a placeholder data member "symbol". More discussions on the purpose of this class will be given in later sections.

A separate test suite project is provided to allow you to develop an Array List class independently of the main project, using the test-driven development approach you've used in all the labs. The validated Array List class will then be included as a data member of the Snake object in the project implementation, similar to how you've done in lab 3 for the custom string library/class.

Recommended Workload Breakdown

One of the most common parallel software development processes is to assign two developers to two parallel developmental tasks. While your team may want to engage with the project in different task breakdowns, here is the recommended workload distribution in case your team is looking for a starting point.

Preparation (Both Developers Together)

- Review and understand the model code and the mandatory objects
- Complete the implementation of objPos class by filling in all the missing special member functions.

Phase 1

- Developer 1: Setting up game mechanisms: input system, game condition system, score system.
- Developer 2: Implementing player object phase 1 – FSM transplant from PPA3, single segment movement validation after OOD refactoring.

Phase 2

- Developer 1: Implementing food object – random position generation, validating on game board.
- Developer 2: Developing and validating the object position array list.

Integration (Both Developers Together)

- Player object phase 2 – array integration to deploy the snake body movement.
- Player object phase 3 – collision detection, snake body growth, score system update.
- Player object phase 4 – Game over condition.

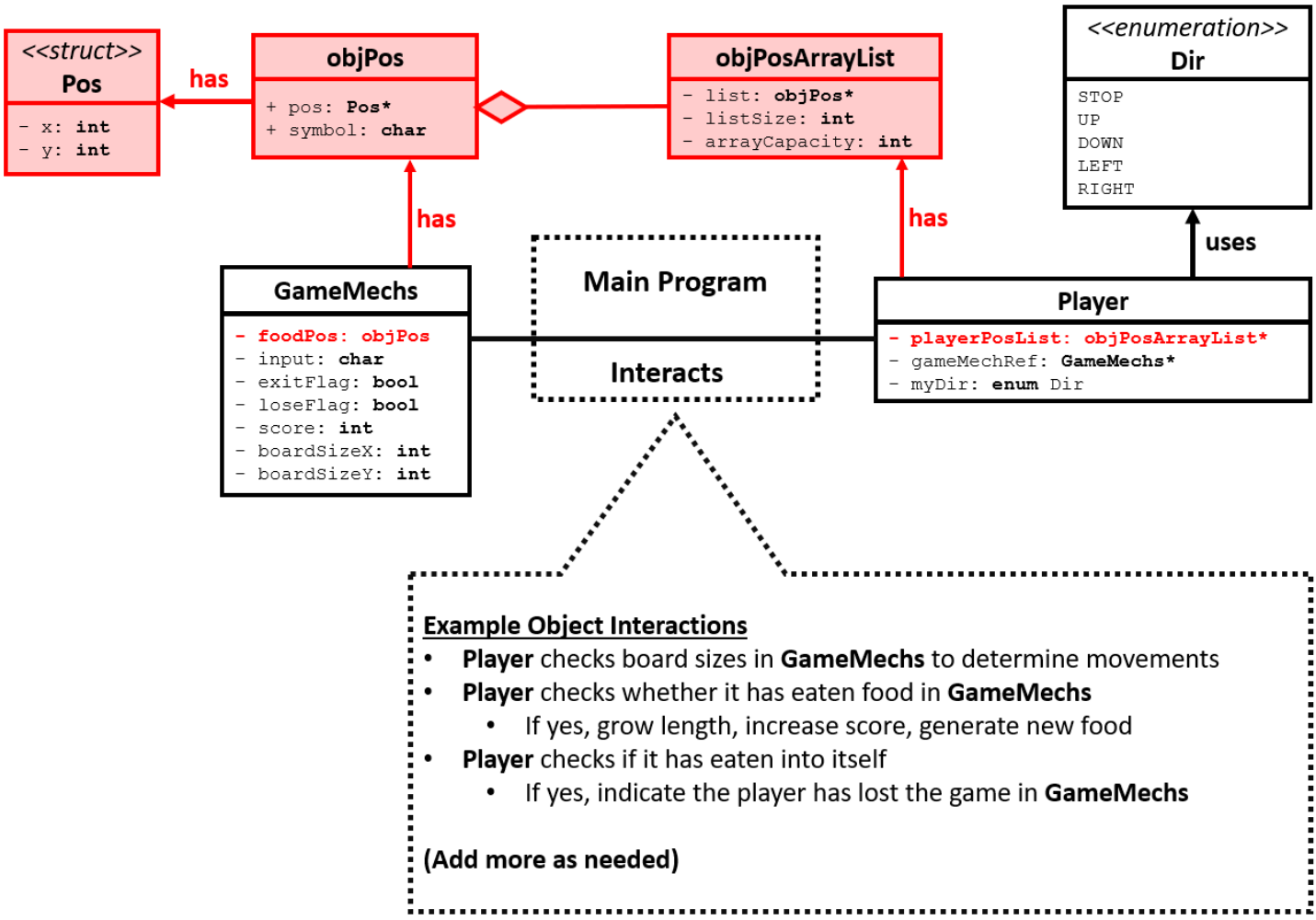
Above-and-Beyond (Your turn to plan your workload)

- Upgrading item to an Item Bin class using array list, enabling at least 3 randomly generated food items around the game board.
- Including at least 1 “special food” feature that can do something special to the snake and the score.
    - Ex. Shorten the snake body while increasing the score by 10, etc.

Project Overview – UML

The fundamental principle of OOD is encapsulation of code implementation, so to achieve a coding style that promotes object interaction rather than procedural execution.

In the simplified UML diagram illustrated here, objects and data members that are marked **RED** are mandatory – they are part of the skeleton code and will have to be implemented. You may expand on them, but not modifying or dwindling them. Other objects built on top of these mandatory features are customizable to your design likings.



Mandatory Objects Implementation

These items in the UML diagram are made mandatory because 1) they enhance your understanding and practice of OOD for organizing basic building blocks of the game, and 2) they create a more versatile OOD platform allowing more complex data structures and algorithms to be incorporated into your project in more advanced software courses (2SI, 3SM, etc.).

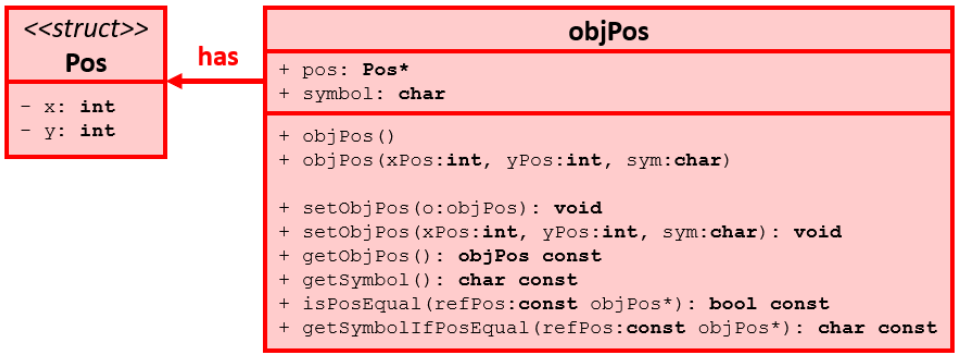
- Object Position Class with Pos Struct holding X-Y Coordinate Information
- Object Position Array List Class
- Data Members involving the above two classes

IMPORTANT: Submissions that do not respect this design criteria will not be graded.

The design of these mandatory objects are intended to be somewhat overcomplicated, and you will have a chance to apply your OOD intuitions on how you'd improve this portion of the design at the end of the project.

objPos – Object Position Class

For this OOD project, the bottom-up design approach is deployed. Recall the game board we've been using in PPA2 and PPA3 – every single ASCII character on the game board has a unique x-y coordinate as well as its ASCII character representation, or ***symbol*** so to speak. Every single feature drawn on the game board, whether stationary or moving, occupies exactly one ASCII character space. As a result, it makes sense to *objectify* the ASCII character space on the game board into a class "object position", or **objPos** in short.



Important Advantage: This encapsulation results in a critical simplification – throughout the remaining design process, every new class we deploy will remain coherent in handling game board coordinate information, as long as they all use the objPos class to track its positional and symbol information. Analogously, objPos is our 1x1 Lego block, the smallest unit from which all bigger Lego pieces can be derived and positioned upon.

Unusual Design Choice: However, this encapsulation design deploys an additional struct, **Pos**, as positional struct containing the x-y coordinate information. The objPos class holds a pointer that can refer to an x-y coordinate instance on the heap, or simply leave it uninstantiated. While one can claim that this is a memory-conserving strategy, it's up to you and your teammate to discuss at the end of the project whether this is a good design choice, and if not, how would you improve it. However, throughout the project code development phases, you will have to accept the objPos class design as is – as if you are tasked by your supervisor to develop the software with the existing and unchangeable design blocks.

Rule of Six / Minimum Four: As we plan to instantiate, copy, assign, and pass objPos instances very frequently in Array List as well as in the game implementation, it makes sense to apply the Rule of Minimum Four. You and your teammate must implement the required special member functions to ensure no memory leakages or errors occur throughout the game runtime.

objPosArrayList - Object Position Array List Class

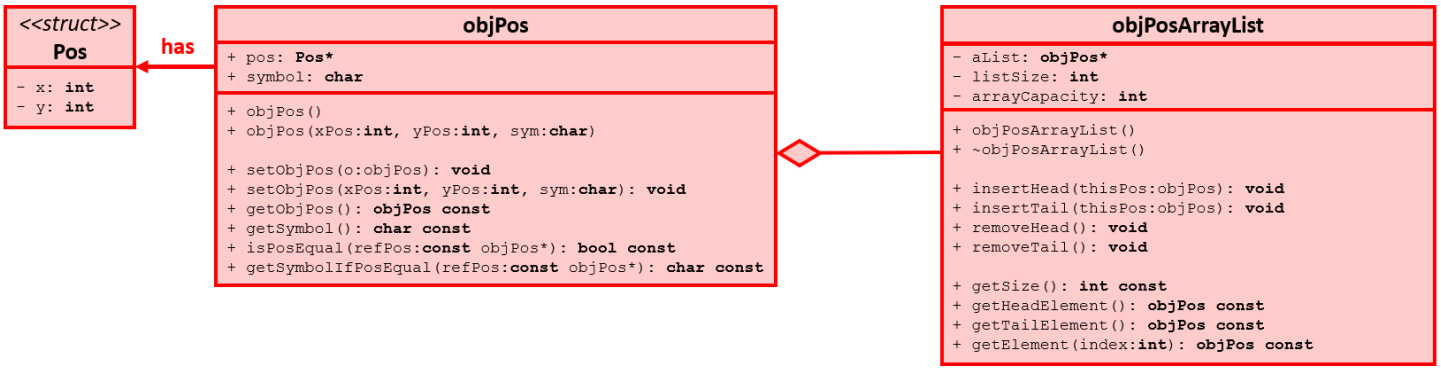
As discussed in class, Array List, or list in general, is a continuous sequence of items organized in an orderly fashion. An array list of objPos class therefore is a continuous and orderly sequence of ASCII symbols on the game board. We can use it for at least the following purposes in our project:

1. Tracking the entire body of the Snake. (Required Feature)
2. Tracking the collection of randomly generated food items. (Above-and-Beyond Feature)

For any additional features, you may find list being an optimal building block as well. We will examine more of these features in COMPENG 2SI3.

In Lego analogy, objPosArrayList is our 1xN Lego strip. Who doesn't love 1xN Lego strips? It may not be as atomic as the 1x1 block but is much more convenient and structurally sturdy. With different varieties of 1xN strip, we can achieve different designs faster and more reliably. This is the power of List in software design.

Since List is such a powerful building block, encapsulating it into an object is the next sensible move in the bottom-up OOD process.

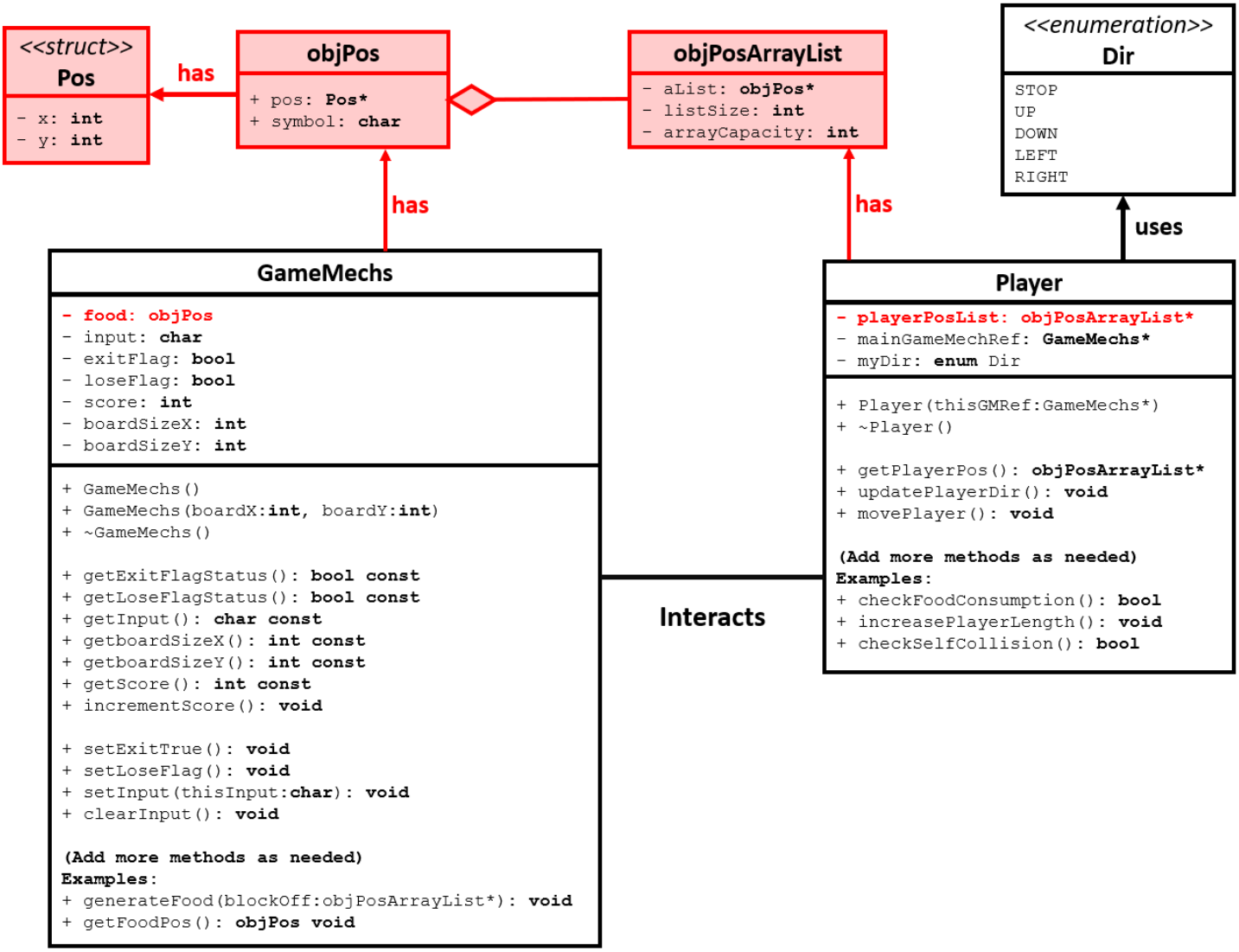


As far as Rule of Six is concerned, objPosArrayList can be designed with just the default constructor and destructor, as the typical usages of a list, at least in this project, do not involve “copying list contents to make another list”, or “passing the entire list by value into a function”. As a result, copy constructor and copy assignment operator overload are optional.

The objPos and the objPosArrayList classes together complete the groundwork of our project. We are now ready to take on the Snake Game using OOD principles in C++.

Recommended Basic UML Design

For your reference, here is the more detailed UML design containing our recommended features. Your implementation can deviate from this recommended design, as long as the **RED** mandatory features are identical.



Marking Scheme

The project has a total of 100 marks and the following two evaluation components:

1. Peer Code Review and Project Reflection [25 marks]
2. Functional Evaluation by Teaching Team [75 Marks]

The evaluation schemes of these two components are outlined as follows.

Peer Code Review and Project Reflection

All project groups will be randomly paired up with another team. Completing the peer code evaluation on time will earn your team a total of **25 marks**.

For Peer Code Review, do not exceed 2 paragraphs per question.

For Project Reflection, limit the discussion in two pages with supporting UML diagrams.

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 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?
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.

Please provide your genuine and engineeringly verifiable feedback. Ungrounded feedback will lead to deductions.

Functional Evaluation by Teaching Team

Your project will be evaluated functionally by the teaching team with the provided marking scheme, and the peer code review results will serve as a validation reference. Functional evaluation by the teaching team will earn you **75 marks**.

[Submission Not Accepted] The project must be compilable from GitHub pull. The teaching team will not debug for any syntax errors. **Triple check** your submission before Git push.

[50% Penalty] The project must be developed on top of the objPos class and the objPosArrayList class. If not, the maximum score will be capped at 50%.

Full Functional Evaluation **[Total 55 Marks]**

- **[6 marks]** Correct input collection and screen drawing logic
- **[5 marks]** Correct random food generation within the white spaces of the game board.
- **[16 marks, with breakdown]** Correct snake movement
  - [12 marks] Regular snake movement using list operations, 4 marks per direction.
  - [4 marks] Wraparound logic implemented correctly at the game board boundary
- **[15 marks, with breakdown]** Correct snake food consumption and length growth mechanism
  - [6 marks] Correct food collision detection mechanism
  - [4 marks] Correct invocation of random food generation logic to keep the game going
  - [5 marks] Correct snake length growing logic
- **[3 marks]** Correct end-game mechanism – snake death and forced exit – with custom message before shutdown.
- **[10 marks]** No memory leakage, segmentation failure, or heap errors.

OOD Code Quality Evaluation **[Total 20 Marks]**

- **[6 marks]** Sensible object modularization and behaviour design – No procedural decomposition
- **[4 marks]** Sensible variable and method naming for promoting self-documenting coding style
- **[5 marks]** Sufficient comments to promote code readability and collaboration
- **[5 marks]** Good use of indentation, newlines, and white spaces for code readability

Bonus Feature – Multiple Food items with Special Food Feature **[Total Bonus: 5 Marks]**

WARNING! These features are only evaluated if your project delivers all the basic Snake game features. If your submission doesn't result in a functional classic Snake Game, you won't score any additional marks in this section.

- **[3 marks]** Food bin class implemented with objPosArrayList composition
- **[1 marks]** Sensible random food generation logic for generating multiple random food items around the board
- **[1 marks]** At least 1 special food feature leading to different snake growth and/or scoring behaviour

Recommended Workflow

As usual, you may choose to use the recommended workflow to complete the project using our recommended OOD approach for your learning purpose. Besides the mandatory features described in the previous section, your team is free to deploy your own design to complete the project, as long as the OOD principles are respected.

Reminder AGAIN!! Unless you are an experienced programmer, **DO NOT** attempt to develop this program in one shot. Always, **ALWAYS** make sure the current feature is working as intended before adding more features to prevent creating a debugging nightmare.

If this is your first time working with an OOD language, please watch the briefing video and attend all the in-class/in-tutorial coding activities. Most of the implementation procedures will be demonstrated in the live coding sessions.

Use Iterative Workflow! Iterative design strategy is an absolute must as you progress on to higher level software courses. With less hand-holding instructions provided in the project manual; iterative workflow will help you keep the software development under control.

Pay Attention to Dynamic Memory Allocation! Remember to return every single piece of heap memory back to the computer when done with it! In OOD, this translates into:

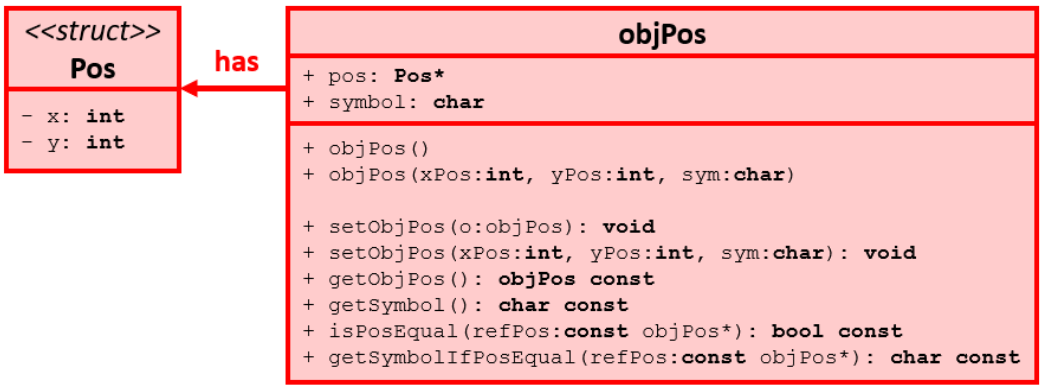
1. Deploy the destructor correctly to delete all the heap data members correctly
2. Correctly delete all object instances on the heap at the end of the program

Iteration 0 – Getting Familiar with the Skeleton Code and Complete the Implementation of objPos

- Read the skeleton source code and the comments to get a good understanding of the code structure.
- Apply Rule of Six / Minimum Four to complete the implementation of the provided **objPos** class.
- **STOP!** You must complete the objPos class implementation with all the required special member functions to avoid memory errors and other disasters. **Do not proceed until done.**

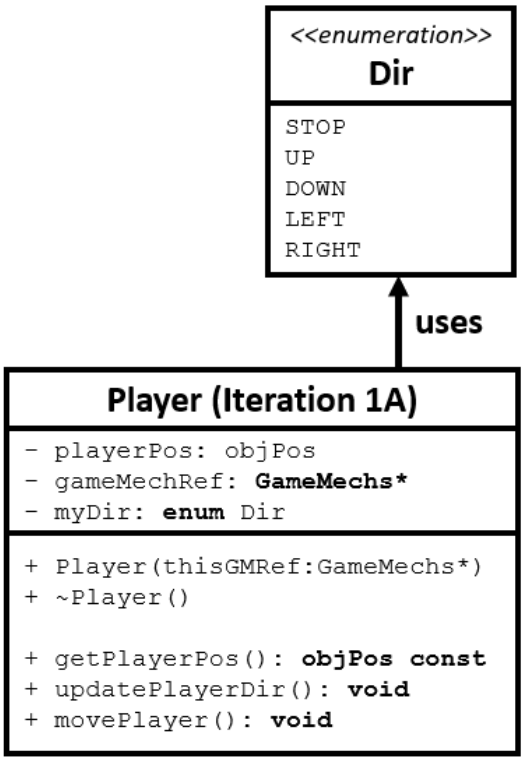
- Create a new Draw routine implementation to draw the game board border using '#' as its symbol, and a few ASCII characters on the game board with arbitrarily chosen coordinates and symbols.
- Many features will be very similar to what you've seen in PPA2 and PPA3.

- **STOP!** Make sure the above implementation is validated using objPos class, NOT your own custom implementation.
- **DO NOT PROCEED UNTIL BOTH TEAM MEMBERS ARE FAMILIAR WITH OBJPOS CLASS USAGE**



From this point onward, the recommended workflow assumes that the two members are working in parallel. Each pair of parallel paths in the same iteration is denoted as A-path or B-path. Nonetheless, your team may still choose to follow these iterations sequentially, allowing both team members to learn the OOD workflow together. Neither approach is incorrect.

Iteration 1A – Player Class Implementation



It may not be obvious at first glance, but the movement control of the Snake in the project is identical to that of the Scavenger in PPA3. The only difference is that a Snake moves its head while growing its body, which in turn limits the player movement. As a result, it makes sense to port the Scavenger implementation, particularly its FSM, into the Snake game to save us some development time. The goal in this iteration is to refactor the C implementation of the Scavenger in PPA3 into the C++ OOD implementation as a Player class, such that you can move an instance of the player object on the game board identically as in PPA3.

This refactoring task may appear rather unnecessary with the current project scope; however, Snake class encapsulation will prove to be an extremely beneficial task as we move into COMPENG 2SI3.

The class diagram of the Player class is not completed yet. It only contains the features to be refactored from PPA3. The rest of the features will be implemented in Iteration 3.

- **Private Data Members (Fields)**
  - `objPos playerPos`
    - Holding the player position and its ASCII symbol
  - `enum Dir myDir`
    - The direction enumeration instance for tracking the direction state of the player.
  - `GameMechs* mainGameMechsRef`
    - This is a typical OOD-based object feature – object holding the reference of another object it will interact with in the program runtime.
    - Upon Player class instantiation, the reference of the GameMechs in the main program must be passed into the Player object in the constructor, so that the player can call the methods in the GameMechs class for whatever information / interaction is required.
    - **Hint**: Moving forward, any new classes added to the program that will interact with the Player object must pass its reference into the Player object. The Player class in turn must hold a reference to every object it will interact with.

- **Public Member Functions / Methods (Interface)**
  - `enum Dir {…}`
    - This is the same enumeration from PPA3. Just copy it over and place it in either public or private scope.
  - `Player(GameMechs* thisGMRef);`
    - Constructor
      - Must require the GameMechs reference so to access the boardSizeX and boardSizeY when determining the movement wraparound.
  - `~Player()`
    - Destructor. Delete heap members here to prevent memory leak.
  - **`objPos`** `getPlayerPos()` **`const`**
    - Getter method for obtaining the current position of the player (Snake head).
    - The objPos is returned by value.

- `void updatePlayerDir()`
  - This method should contain the player FSM.
  - It obtains the most current input from the GameMechs reference, updates the FSM as needed, and clears the input buffer via GameMechs reference.
  - This method should be called in every game loop in the main logic.

- `void movePlayer()`
  - This method should contain the player position update algorithm.
  - This method should be called in every game loop in the main logic.

The high-level steps for porting the Scavenger code from PPA3 into the Player Class are as follows:

- For ease of development, define and implement the Player class at the top of project.cpp. Only package the contents into Player.cpp and Player.h when done with the class development.
- Implement all the data members, the direction enumeration, the constructor, and the destructor.
  - In the constructor, make sure to initialize the Player initial position and the symbol the same way as in PPA3.
- For simplicity, you should use the ASCII character ' * ' (asterisk) for representing the player object.
- Implement the getPlayerPos() method.

- **STOP!** We need to test the above features first before proceeding to implementing FSM and position update.
  - Declare a pointer to Player object in the global scope
  - During initialization, instantiate a player object on the heap and keep its reference in the global pointer.
  - Update the draw routine such that it draws the Player object at the correct position with the correct symbol.
  - Do not proceed until you can confirm that the player object is correctly drawn on the game board.

- Next, port the FSM implementation from PPA3 to updatePlayerDir() method.
  - Make sure to update the statements involving user input – you now have to access it via GameMechs reference.
  - Deploy debug messages to confirm your FSM is correctly responding to the keyboard input.
  - Then, in the main program loop, call this method on the Player object in the suitable routine.

- **STOP!** Do not proceed until you can confirm your FSM is responding to keyboard inputs correctly!

- Finally, port the player position update logic from PPA3 to movePlayer() method.
  - Again, remember that boardSizeX and boardSizeY are now part of the GameMechs class. Use the reference to access them when dealing with boarder wraparound.
  - In the main program loop, call this method on the Player object in the suitable routine after updatePlayerDir().

- **STOP!** Do not proceed until you can control the player object movement the same way as in PPA3.

Iteration 1B – Refactor PPA2/PPA3 Code and Deploy the Game Mechanism (GameMechs) Container Class

The goal of this iteration is to port all the usable features from the C implementation in PPA2 and PPA3, and refactor them into C++ equivalence in the project code. As we've learned in class, C and C++ are almost identical in procedural programming style, thus you may find that porting your code from PPA2/PPA3 appears rather straightforward. These features include:

1. Game board size parameters, X and Y
2. Input collection routine
3. Exit command
4. Clean-up routine – including end-game messages

However, you may realize in PPA3 that these features result in rather scattered variables in the global scope (bad practice). When accessing them, you'd need to be extra careful to prevent tempering the values in them. In addition, you need to be highly aware of their visibilities around the program, so to access them correctly in the correct routine. This is one of the shortcomings of procedural programming – when designing a program without too much organizational effort, your program works well but doesn't sit well with feature expansion. The OOD features in C++ are intended to enforce code organization through syntax features, so that your code can be more expansion friendly. Of course, this is assuming that you are not using C++ only for procedural programming.

As a result, we recommend deploying a GameMechs class. This class can be referred to as a container class, because its main purpose is like a **toolkit** used to collect and manage scattered parameters related to the underlying game mechanics under a single class for ease of parameter management. Such a class may not be necessary in smaller software projects, because the organization effort of creating such a class is not expected to outweigh the effort of maintaining individual parameters without functional confusion. However, as a sustainable practice, for any work under development, container classes like this are often deployed anyways so to "leave the hooks" for future feature expansions.

There is no right or wrong here, just engineering trade-offs.

The class diagram provided is the most basic design. You may need to add more fields and methods as needed in future iterations.

- **Private Data Members (Fields)**
  - `char input`
    - Holding the most recent input collected via MacUILib_getChar()
  - `bool exitFlag`
    - Self explanatory.
  - `bool loseFlag`
    - The Boolean flag recording whether the player has lost the game (Snake eating into itself).
    - This flag should not be set to true if the player presses the exit key to terminate the game.
    - Use this flag to determine what messages to display at the end of the game.
  - `int score`
    - Holding the current score of the player
  - `int boardSizeX` / `int boardSizeY`
    - Self explanatory.

- **Public Member Functions / Methods (Interface)**
  - `GameMechs() / GameMachs(int boardX, int boardY)/ ~GameMechs()`
    - Constructors
      - Initialize the game mechanics-related parameters.
      - Recommended default board size: X = 30, Y = 15.
    - Destructor
      - Deallocate all heap data members
      - Defense against memory leakage
      - May not be needed if you do not have heap data members.

  - `bool getExitFlagStatus() const / void setExitTrue()`
    - Getter and setter of the exit flag. Purpose is self explanatory.

  - `bool getLoseFlagStatus() const / void setLoseFlag()`
    - Getter and setter of the lose flag. Purpose is self explanatory.

  - `char getInput() const / void setInput(char thisInput)`
    - Getter and setter of the most recently collected ASCII character input from MacUILib_getChar()

  - `void clearInput()`
    - Clear the most recently collected ASCII character input from the field.
    - Use this to make sure no input is double-processed.

  - `int getBoardSizeX() const / int getBoardSizeY() const`
    - Self explanatory getter methods for board dimensions.

  - `int getScore() const`
    - Self explanatory getter method for the game score.

  - `void incrementScore()`
    - Specialized setter for the score field. Assumption is that the score can only be incremented 1 at a time for every food item collected.
    - You may consider changing this method to allow the score to be increased by numbers other than 1.

To refactor the PPA2/PPA3 C code into the project with the GameMechs class, you would need to incrementally carry out the following actions:

- Create a pointer to GameMechs class in the global scope. You should delete all other global variables related to game mechanics (in fact, you should delete all global variables other than the GameMechs pointer)
- During initialization, create a GameMechs object on the heap, and initialize its fields accordingly.
- Update the program loop so that it is accessing the exit flag in the GameMechs object instead of the global flag (which should now be removed)
- In input collection stage, collect the input ASCII character into the corresponding field in GameMechs object using the correct setter method.
- In input processing stage (main logic), access the correct field in the GameMechs object through the getter method to process the input character, choose the correct action, then clear the input field in GameMechs to avoid double-processing the input.
- In the clean up stage, make sure all the required end-game actions are taken before deleting the GameMechs object from the heap.

- **STOP!** Make sure the above features are implemented such that you can start the program and shut it down using your selected exit key before proceeding onward.

- After the basic GameMechs class is validated, you should further implement and confirm the following features:
    - Set up a debug-use key to test whether you can increment the score in GameMechs object. Print appropriate debug messages to confirm its functionality.
    - Set up another debug-use key to test the lose flag. Make sure you can print different messages at the end of the game according to the lose flag status.

- **STOP!** Make sure the above features are implemented and validated before proceeding onward.

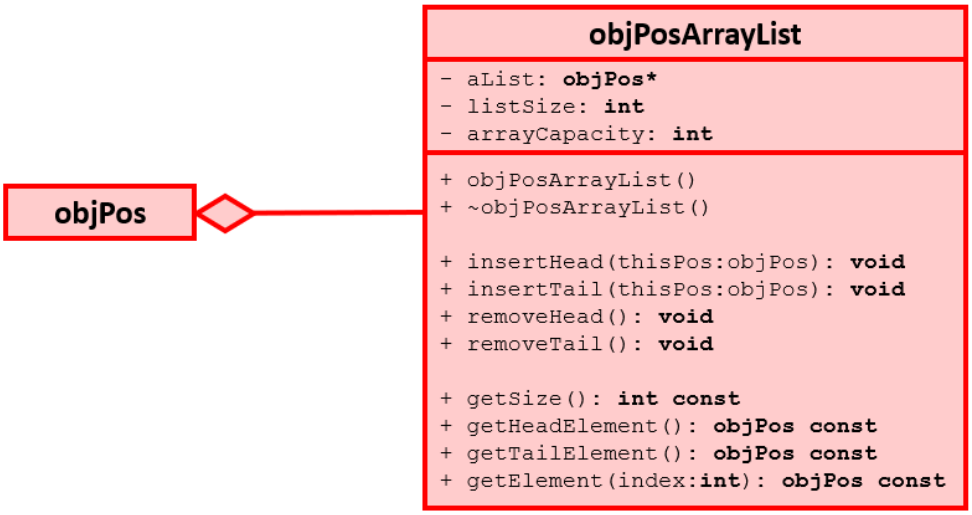
It is important to note that the above GameMechs features are NOT the complete set of features. Your team has 100% freedom in adding / removing features to and from this container class as needed to make your game more feature rich.

By completing iteration 1, you will have refactored your PPA2 from procedural C program into the Object-Oriented C++ program. Please confirm that your project code works identically as your PPA2 code (yes, NOT PPA3).

Iteration 2A – Deploying and Validating objPosArrayList Functional Class

The objPosArrayList class can be referred to as a functional class, because its instances serve as the functional backbone of many other larger objects in the project. A separate folder named "objPosArrayList_TestSuite" is provided in the project repo to allow you to develop the objPos array list using the provided test bench project. You can also set the workspace temporarily to the Array List project folder to run the debugging session on it. The UML class diagram of the objPosArrayList class is provided for your reference.

We will discuss different usages of the array list in the project in later iterations.



The description of each data member and member function is provided below.

- **Private Data Members (Fields)**
  - `objPos* aList`
    - Pointer to the start of an objPos array list, with the array size specified by sizeArray.
    - **Heap Data Member**
  - `int sizeList`
    - The number of valid list elements in the list
  - `int sizeArray`
    - The number of array elements allocated in the heap array

- **Public Member Functions / Methods (Interface)**
  - `objPosArrayList() / ~objPosArrayList()`
    - Array List Constructor
      - Initialize the sizeArray and sizeList data members.
      - Allocate an array on the heap with the size specified by sizeArray.
      - The default sizeArray is 200. You may change it as needed.
    - Array List Destructor
      - Deallocate all heap data members
      - Defense against memory leakage
  - `int getSize() const`
    - Returns the size of the list (i.e. sizeList)
    - **NOT the sizeArray (Array Capacity!!)**
  - `void insertHead(objPos thisPos)`
    - Inserting thisPos as a new objPos element to the head of the list

        - Do you need to "shift" the elements? Think about it.

    - **void** `insertTail(objPos thisPos)`
        - Inserting thisPos as a new objPos element to the tail of the list
        - Do you need to "shift" the elements? Think about it.

    - **void** `removeHead()`
        - Remove the head element from the list
        - Do you need to "shift" the elements? Think about it.

    - **void** `removeTail()`
        - Remove the tail element from the list
        - Do you need to "shift" the elements? Think about it.

    - **objPos** `getHeadElement()` **const**
        - Return a copy of the objPos element at the head of the list

    - **objPos** `getTailElement()` **const**
        - Return a copy of the objPos element at the tail of the list

    - **objPos** `getElement(int index)` **const**
        - Return a copy of the n-th objPos element from the list, where *n* is specified by index.

STOP! Make sure you have passed all the default test cases, as well as some additional test cases of your own, before proceeding. The robustness of your array list implementation will critically affect the debuggability of your code in future iterations.

STOP! TRUST ME, VALIDATED LIST ROBUSTNESS IS AN ABSOLUTE MUST!

Once fully validated, you may copy the objPosArrayList.h and .cpp from the test project folder into the main project folder.

Your custom Array List for handling objPos is now ready.

Iteration 2B – Random Food Generation

In this iteration, the random food generation mechanism will be deployed in the project. You have two design choices to implement the Snake Food:

1. Incorporate the food generating mechanism in the GameMechs class, and keep the Snake Food as a simple objPos member of the GameMechs class, as described in the UML diagram below on the left, or
2. Creating a Snake Food class as described in the UML diagram below on the right.

- **Private Data Members (Fields)**
  - `objPos foodPos`
    - Holding the most recently generated food position and its ASCII symbol
- **Public Member Functions / Methods (Interface)**
  - **`void`** `generateFood(objPos blockOff)`
    - The random food generation algorithm should be placed here. (copy from PPA3)
    - blockOff should contain the player position, on which the new food should NOT be generated.
  - **`objPos`** `getFoodPos()` **`const`**
    - Getter method for obtaining a copy of the current position of the food.

There is no right or wrong here. The only difference between the two design choices is that the former one (on the left) saves you time right now but may present more challenges later when developing advanced features (bonus part), while the latter one (on the right) will incur more work right now but will give you an easier time if your team chooses to implement above-and-beyond bonus features.

Either way, here are the high-level steps you should take to make sure the food generation mechanism can be implemented. The steps below are only for generating SINGLE FOOD randomly on the board.

- Either in the GameMechs class or in your own Snake Food class, create an objPos member to track the food position.

- Make sure the class containing the snake food objPos member supports the following methods:
    - Random food generation method (you can port the code from PPA3 and revise it for the project purpose)
    - Food objPos getter method
- In the initialization routine, generate a randomly positioned food item on the game board.
    - If you have created a Snake Food class, you need to create a global pointer to this class, allocate it on the heap and initialize it in the initialization routine, and delete it from the heap in the clean up routine.
- In the draw routine, make sure you can get the objPos information of the food, and print its symbol at its corresponding x-y position.

- **STOP!** Before advancing to the next step, make sure you can see a randomly generated food item showing up on the game board at different locations every time launching the program.

- Then, you should test whether you can repeatedly generate new food at random locations on the board on-the-fly. In the input routine, create a debug-use key to clear away the current food and generate another random food. You do not need to literally use the **delete** and **new** operations here – simply overwriting the old objPos information of the food with the new one is good enough.

- **STOP!** Make sure you can repeatedly generate new food at random positions (and wipe away old food) using the debug key before proceeding to the next iteration.

We now have completed the C-to-C++ OOD refactoring for the all the in-game objects in PPA3, and are ready for further feature expansion using our objPos array list.

Note: If adding a separate **Food class**, you need to also modify the **makefile** as follows:

In Line 3 add:

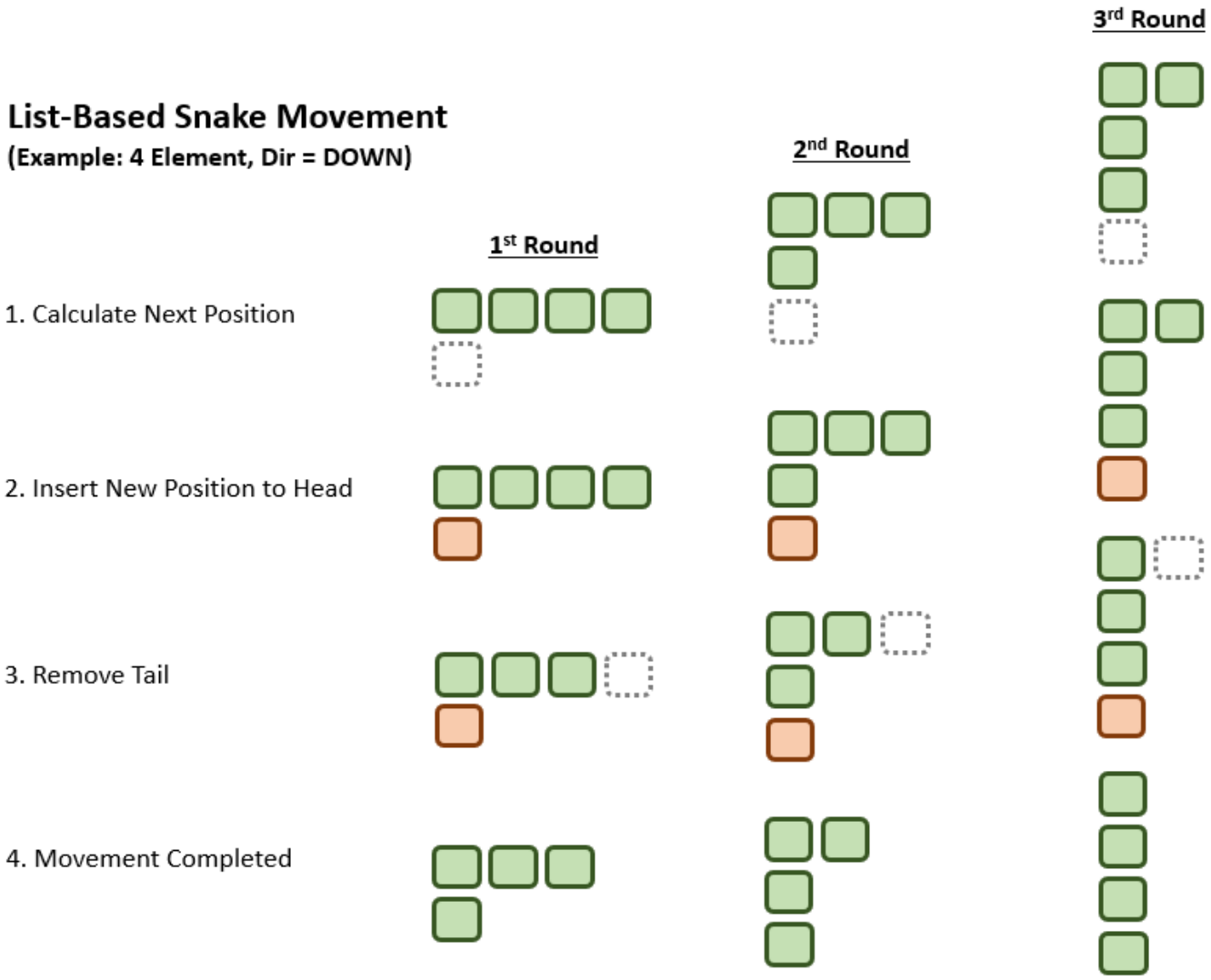
```
OBJ = GameMechs.o objPos.o objPosArrayList.o Food.o MacUILib.o Player.o Project.o
```

Iteration 3 – Player Class Functional Expansion and Feature Integration

We now have all the required features ported from PPA3 and refactored into C++ OOD style, and there are only three more features to implement in the Player class to complete the Classic Snake game implementation.

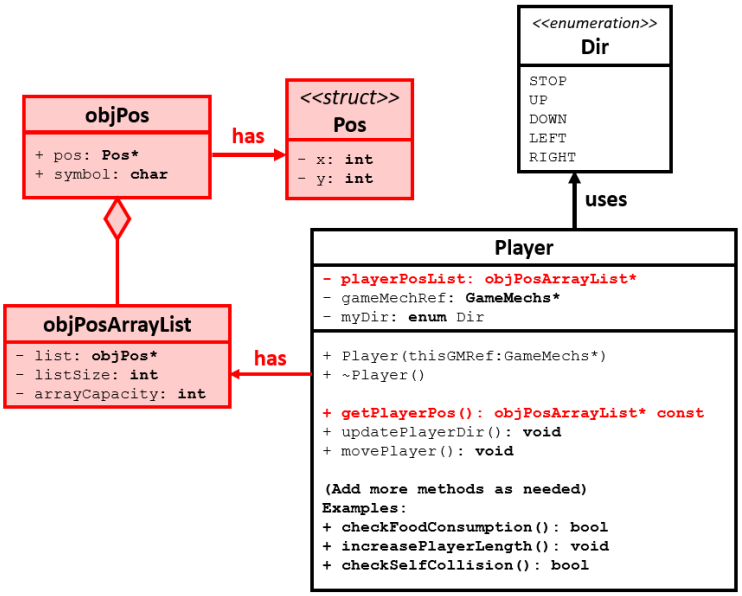
Feature 1 – Snake Body Implementation using Array List

The key to keeping track of all the snake body segments on the game board is the use of Array List. The diagram below illustrates the basic idea.



The conceptual trick here is that the snake movement takes place by inserting a new snake segment to the head of the list at the target x-y location after the movement, and then remove the last snake segment from the tail of the list. This way, the visual snake movement can be completed as easily as one list insert and one list removal.

To implement this clever snake movement algorithm, we need to deploy one of the most commonly used OOD principles – **Composition**. More specifically, we need to make use of the Array List object we've previously created, turn it into a member of the Player object, such that the Player object can be said to be "an object composed of another object".



Here is the high-level procedure for realizing this composition relationship:

- Just for your peace of mind, TRIPLE CHECK the correctness of your objPosArrayList implementation.
    - Composition works best under the assumption of functional underlying object implementations.
    - If your underlying object malfunctions, the debug scope will explode from just within the Player Class to both the Player and objPosArrayList classes, as well as every interaction between the two classes. **It will be a nightmare.**

- Player needs to own an instance of objPosArrayList. As a result, we first need to replace the data member objPos playerPos with objPosArrayList* playerPosList.
    - playerPosList should be instantiated on the heap in the constructor, then with the first element inserted at the starting x-y position with '*' as the symbol. This will make the list with a size of 1.
    - All methods that are simply returning the playerPos should be updated to return the objPos of the head element of the array list.

- Then, in movePlayer() method, we need to make the following changes:
    - For every newly calculated x-y coordinate we are to move the player towards, we need to insert the equivalent objPos into the head of the list.
    - Then, remove the last element from the list.

- For getPlayerPos() method, we need to return the reference of the entire playerPosList, so that the draw method can access all the elements in the array list for drawing purpose.

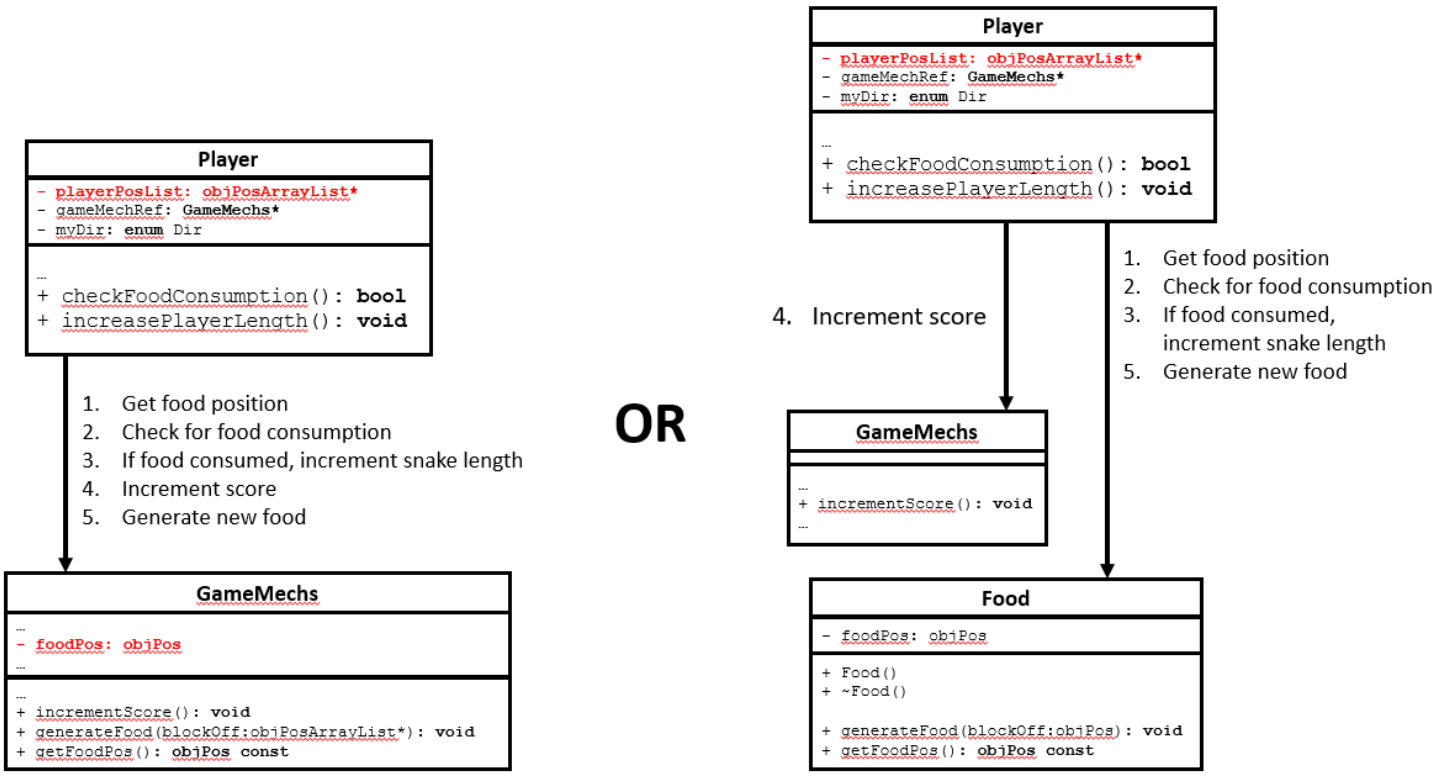
- Finally, in the draw routine, use getPlayerPos() method to gain access to the entire playerPosList, and draw every single element on the game board from head to tail.

STOP! Make sure you have validated the composition implementation carefully with at least three different initial snake lengths (1, 3, 5, etc.). Check all the wraparound logics, movement correctness, etc.

DO NOT PROCEED UNLESS FULLY VALIDATED

Feature 2 – Snake Food Consumption and Snake Body Growth (also Score)

Once you get the first feature implemented, you've completed the most challenging OOD feature of the project. The remaining two features simply build on top of feature 1.



For snake food consumption, here is what you need to do:

- Modify the player class constructor so it can obtain the reference of the food object (if not part of the GameMechs class)

- In movePlayer() implementation, check whether the newly positioned head overlaps with the objPos of the food.
    - If no, carry out regular insert + remove to complete the snake movement.
    - If yes, perform the following actions
        - **Insert the head, but DO NOT remove the tail.**
        - This will cause the list size, hence the snake length, to grow by 1.
        - Then, call the generateFood() method to generate the new food item on the game board

- **STOP!** Make sure your snake can consume food and increase its length before proceeding to the next section. Don't worry about new food overlapping the snake for now.

- Then, in order to make sure we don't generate new food on top of the snake body, you need to make the following changes in generateFood() method
    - Accept the objPosArrayList reference as the input, so that it can handle the playerPosList reference.
    - When generating food, iterate through every single objPos element in the playerPosList, and make sure the position of the new food item does not overlap with any of the elements.

- **STOP!** Test play your game until you are 100% sure no food will be generated on top of the snake body before proceeding.

As for the score system, its as simple as getting the list length of the playerPosList from the Player Class. This is left for you to implement on your own.

- Just remember that list size of 1 maps to the player score of 0.

Feature 3 – Snake Death Check (Game Over Condition)

At this stage, you should be fairly comfortable with the gist of OOD – incremental design within the class and/or classes involved. This design flow guarantees that changes are only made in the confined scope of classes and won't accidentally spill over to unrelated components of the project by accident.

Now, looking at the game over condition, you probably can see that this is a behaviour change within the Player Class, and you probably have a clear game plan of how to implement this.

Here is the high-level procedure for your reassurance:

- In movePlayer() method, check whether the new head position collides with any element in the playerPosList.
  - If not, carry forward with the movement and food consumption detection
  - If yes, set both lose flag and exit flag in the GameMechs object
    - This will force the game to shut down and trigger the different exit message with the lose flag set to true.
    - Well, assuming that the GameMechs class is implemented correctly.

And, at the end of the COMPENG 2SH4 project, you should know very well the drill – **STOP!!!** Do not proceed until you've tested the feature well.

Additional Iterations – Above and Beyond Feature (Bonus 5% for Project)

As usual, this section contains only tips and high-level guidelines for deploying the above-and-beyond features. However, note that in the project, the above-and-beyond features count as an additional bonus (unlike the PPAs which had no bonus). Do not attempt these features unless your team has completed and validated the base implementation of the project.

WARNING! It is **HIGHLY** recommended to keep a working copy of the base project implementation as a fallback submission. (Or learn about how you can revert your project back to an earlier version using an appropriate Git command)

Remember, we will only accept compilable solutions.

Bonus Feature – Generating more than 1 food objects at random positions with special food functions

- For example, generate 5 random food items like in PPA3, and randomly choose one or two of them to have different symbols – we refer to them as *special food*.
- Snake eating a special food will gain special effect.
    - Ex. 1 Scores 10 points without increasing snake length
    - Ex. 2 Scores 50 points but increases snake length by 10

- To achieve this feature, you will certainly need to implement the Food class with a composing objPosArrayList data member on the heap – the "Food Bucket".
    - Modify all the relevant methods to cater to the Food Bucket list member on the heap
    - Food consumption algorithm requires Player class to visit every single food element in the bucket to check for consumption

- Your team has all the freedom to implement this feature using different OOD approaches.

  The only unacceptable solution is an OOD-decomposed solution, a.k.a. a solution with only procedural programming approach with absolutely no regard for OOD principles.