C++

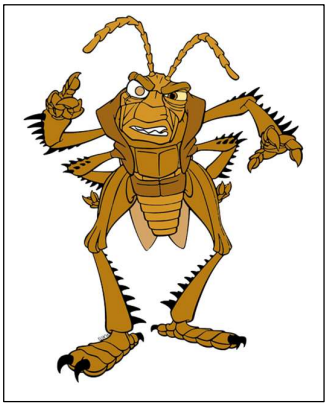
Continuous Assessment
CA3 - Stage 2

Weighting: 30%

Stage 1 (2%), Stage 2 (28%)

Semester 2 – 2024-2025

B.Sc. (Hons) in Computing in Games Development, Year 2, Semester 2
B.Sc. (Hons) in Computing in Software Development, Year 2, Semester 2



Assignment: A Bug's Life **Deadline: See Moodle**

Assignment is to be completed **in pairs or individually**.

Aim

Develop a system that will simulate the movement and interaction of various bugs placed on a Bug Board. The Bug Board is marked with grid lines so that it has 10 x 10 cells (squares). When we "Tap" (shake) the board it will cause all bugs to **move** in accordance with their specified behaviour. Bugs arriving on the same cell after a "Tap" will fight, and the biggest bug will eat all others on that cell. An end point is reached when there is only one bug remaining. You are also required to implement a console-based user interface using the menu items specified below.

Menu Items

1. Initialize Bug Board (load data from file)
2. Display all Bugs
3. Find a Bug (given an id)
4. Tap the Bug Board (cause all to move, then fight/eat)
5. Display Life History of all Bugs (path taken)
6. Display all Cells listing their Bugs
7. Run simulation (generates a Tap every second)
8. Exit (write Life History of all Bugs to file)

Introduction of more Bug types and the use of Inheritance

Due to the success of your Bug's Life App your senior product managers have decided to expand the App to facilitate a range of different Bug types, with different attributes and behaviours. Your System Architect has pointed out that the system will be easier to maintain and expand if you create an *Abstract Class* called ***Bug*** and use it to share the behaviours that are common across all bugs.

1. Bug (Abstract class)
The following attributes and behaviours are common to all Bugs.

`int id;`	Identification number (id) for a bug (101,102…)
`Position position;`	Co-ordinate pair (x,y) representing a board position. (0,0) is top left cell on the board.
`int direction;` *(or better, use enum class)*	direction in which the bug is facing: 1=North, 2=East, 3=South, 4=West (or use *enum*)
`int size;`	Size of the bug (initially 1-20); biggest bug wins in a fight and others on same cell are eaten. Winner grows during a fight by the sum of the sizes of other bugs eaten.
`bool alive;`	Flag indicating life status of a bug. All bugs are set to 'true' initially. When eaten, this flag is set to 'false'. true => alive, false => dead *Consider the pros and cons of marking a bug as deleted over actually removing the bug when it dies.*
`List<Position> path;`	Path taken by a bug. (i.e. the List of Positions (on grid) that a bug has visited during its lifetime.)
`virtual void move() {}`	All **derived** classes (sub-classes) that must implement the logic required to move a bug from its current position to a new position based on the movement rules for the particular bug type. The *move*() function must be declared as a **pure virtual function** in the **Bug** class (with **no** implementation permitted in the Bug base class).
`bool isWayBlocked()` `{}`	Checks if a bug is against an edge of the board AND if it is facing in the direction of that edge. If so, its way is blocked. [This method is used by the *move*() function]

2. Crawler

As described in the Stage-1 brief. The Crawler class may need to be *refactored* to reflect attributes and behaviour that has been relocated to the Bug base class.

3. Hopper (inherits from Bug)

`int hopLength;`	The distance that a particular hopper bug can hop (in the range 2-4 squares). This field must be declared **private**.
`void move(){}`	A Hopper bug moves according to these rules: - moves by "***hopLength***" units in current direction - if at edge of board and can't move/hop in current direction (as it is against an edge), then set a new available direction at random. (repeat until bug can move forward) and then move. - if a bug can move, but cannot move/hop the full '*hopLength'*, then the bug does hop but 'hits' the edge and falls on the square where it hit the edge/wall - record new position in the hoppers path history

Our Bug class holds fields and functions that are common to all Bugs. We don't want to allow anyone to instantiate a 'generic' Bug, so, we make our Bug class **Abstract** by adding a **pure virtual function**. ("=0"). (A class containing a virtual function becomes an abstract class)

Notice that the main difference between the derived classes is in behaviour. The behaviour of the various bugs differ as determined by their implementation of the *move*() function. A Crawler bug behaves differently from a Hopper bug.

For convenience and maintainability of code, we want to be able to store all bugs in a *vector* that will store **both** Crawlers and Hoppers (and any additional bugs), so that we can iterate over all bugs and treat them in a similar way. However, we can't use a vector of Bug objects `vector<Bug>` because the derived class objects (Hoppers and Crawlers) would not be of the same size, and elements in a vector/array **must be** of the same size. (e.g. A Hopper would not fit into a Bug object vector element (Hopper has additional field and is bigger)).

So, one option is to create a **vector of pointers to Bug objects** (`vector<Bug*> bug_vector`). The elements of this vector are of type 'pointer to Bug', so ***they can point at any derived class*** objects of the base class "Bug" (e.g. Crawler or Hopper). A Bug pointer (pointer to Bug) can be used to point at a Hopper object, or a Crawler object.

We also wish to be able to 'move *()*' all the bugs after a "Tap" on the board. As there is more than one derived bug type, each implementing its own version of the *move*() function (using overrides), we **MUST declare a virtual *move*() function in the base class, and implement that method in each derived class. To avail of the runtime polymorphism behaviour provided by virtual functions, we must use a vector of pointers to Bug type objects.** This allows the correct *move*() function for the particular derived class type object to be called at runtime. This selection of the right move method is called *Dynamic Binding*.

Therefore, we must declare a vector of pointers to Bug objects [in main()], and populate it by reading data from a text file ("bugs.txt"), instantiating derived class objects dynamically on the Heap, and adding their addresses to the Bug vector. The owner of these object **must also remember to free** the associated memory when it is no longer required. `vector<Bug*> bug_vector;`

Board Class

Board class *encapsulates* the vector and cells. No access to internal workings of board is to be 'leaked' outside the Board class, so no references or pointers to any internal objects are to be returned. Return only **copies** of data if required. (So, internally pointers can be passed around, but for public interface functions, provide only copies of objects. Consider the interface.)

Features/Functionality

(Complete in the order shown below. (Increasingly challenging))

You **MUST** use GitHub or GitLab, and make your lecturer a collaborator, and **commit & push** each feature as you complete it. Not having a verifiable record of regular incremental commits to your GitLab code repository ***will incur substantial penalties***. You must share your repository with your lecturer (by making them a collaborator), and placing the link in the spreadsheet provided.

1. Initialise the bug Board.

Read the "bugs.txt" file and populate the Bug vector. Assume the file contains valid data (so no validation is required). Bugs must be allocated from the Heap.

2. Display All Bugs

Display all bugs from the vector, showing all relevant fields: *id, type, location, size, direction, hopLength, and status* - all in human-readable form. E.g.

```
101 Crawler (3,4) 18 East Dead
102 Hopper  (5,8) 13 North 4 Alive
```

3. Find a Bug

User is to be asked to input a **bug id**, and the system will search for that bug.
Display the bug details if found, otherwise display "bug xxx not found".

4. Tap the Bug Board

This option simulates tapping the bug board, which prompts all the bugs to move. This will require calling the *move*() function on all bugs. The *move()* method must be implemented differently for Crawler, Hopper and other bugs. (See class details above). Later you will be asked to implement fight/eat. *We recommend that you implement only move() initially. The fight and eat behaviour can be developed later, when all other functionality has been implemented.*

5. Display Life History of all bugs

Display each bug's **details** and the **path** that it travelled from beginning to death. The history will be recorded in the **path** field (which is a chronological **list** of positions). (A **list** container must be used)
101 Crawler Path: (0,0),(0,1),(1,1),(2,1),(3,1) Eaten by 203
102 Hopper Path: (2,2),(2,3), Alive!

6. Exit - Write the life history of all bugs to a text file called "bugs_life_history_date_time.out"

7. Display all Cells

Display all cells coordinates in sequence (in order left-to-right, top-to-bottom), along with the **name** and **id** of all bugs that are alive and currently occupying each cell.

```
(0,0): empty                      // meaning: cell (0,0) is empty
(0,1): empty
(0,2): Crawler 101, Crawler 103 // i.e. the 2 Crawler bugs in this cell
(etc…)
(1,0): Hopper 102
(1,1): Crawler 105, Hopper 107, Crawler 109
```

In order to achieve this, you must design and implement a mechanism to record which bugs are occupying which cells (squares). This is a challenging task.
You should discuss your approach to this with your lecturer before implementation.

8. (Expand option 4) Eat functionality

Implement functionality that will cause bugs that land on the same cell to fight. This will happen after a round of moves has taken place – invoked by menu option 4. (Tap ….). The biggest bug in the cell will eat all other bugs and will grow by the sum of the sizes of the bugs it eats. The eaten bugs will be marked as dead ('alive=false'). We can keep 'tapping' the bug board until all the bugs are dead except one – the Last Bug Standing. Two or more bugs equal in size won't be able to overcome each other so the winner should be resolved at random.
You should discuss your approach to this with your lecturer before implementation.

9. Run Simulation

Implement functionality to simulate the tapping of a board every 1 second (or suitable interval) until the game is over. Display progress on screen as simulation proceeds and write results to file.

10. New Bug Type

Define a new bug type (derived from Bug) and add it to your system. Your new bug type must have a specific move() behaviour different from those already created. Refactor all code where necessary to integrate this addition.

11. Implement a GUI for the project using the SFML library.

Use the SFML graphics to display a board and bugs (sprites) as they move using your model (from above) as the underlying data structure. Introduce a **Super-Bug** that you can move independently of the simulation by using the arrow keys, and that can interact (fight) with other bugs.

Marking Scheme Items

1. Base and Derived classes implemented and tested
2. Bug vector initialized with bugs from text file
3. Display all Bugs
4. Find a Bug
5. Tap the Bug Board (move all bugs)
6. Display Life History of all Bugs
7. Display All Cells and their Bugs
8. Exit (write Life History of all Bugs to file)
9. Implement a **fighting/eating** extension to option 4
10. Introduce new bug type
11. Add a new Bug type to your system with new behaviour
12. GUI (SFML) Implementation

Grading Criteria

Demonstrated understanding of code presented and ability to explain the concepts used in the code during interview. Functional correctness and adherence to specification, quality of code (Maintainability, Readability, exploitation of standard library data structures, efficiency), and quality and functionality of User Interface will form part of the criteria for grading. Evidence of continuous development evidenced by Git commit history available on GitHub is essential, and failure to provide this evidence will result in severe penalties.

Upload Requirements

Upload to contain:

1. **Screencast** showing your app working AND a brief walkthrough of your code identifying the structure, core features, and the data structures used. Please **state your name** and class group at the beginning. Screencast to be no longer than 5 minutes.
2. Zipped file containing all **source code** and **data files-** but NOT executable and other compiler files) (GitHub Repo URL alone **is not acceptable,** but **link** must be included in a text file**.**) ZIP the project folder for upload. Please name your project folder "Bugs_Lastname_Firstname" and your zip file "Lastname_Firstname.zip"
3. **Completed CA cover sheet**.

Upload completed project as a single zipped file to Moodle by the deadline.

Late project submissions will be grade capped at 40%.