C++

Continuous Assessment 2 (CA2)

Weighting: 40%

Semester 2 – AY2324

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

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

Assignment to be completed **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 particular type of 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 implemented a graphical interface (using SFML) that will display graphics representing the underlying simulation/model.

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 (causes move all, 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)
```

Text File: "bugs.txt"
Format for a line (a record), delimited by semi colons (";"), ending in newline.

Type of bug	'C' for crawler, 'H' for Hopper
ID for a bug	Unique integer ID value (e.g. 101,102,… etc.)
X coordinate	(X,Y) coordinate system where (0,0) is top left hand cell -
Y coordinate	and X increased to right (East), and Y increases as we go down(South)
Direction bug is facing	Direction values : 1=North, 2=East, 3=South, 4=West (or use *enum*)
Size of bug	Measure of bug size (1-20), bigger bugs eat smaller bugs and grow accordingly
Hop Length	Distance (length) of a hop that a Hopper bug can make. (This field is **not** present for a Crawler.)

Sample file data:

```
C;101;0;0;4;10
H;102;9;0;1;8;2
C;103;9;9;2;5
```

Class Details (Specification)

You are required to create the following classes using separate header (.h) and source (.cpp) files.

You **MUST** use the fields shown below (as named), and you may need to introduce more fields and/or functions to implement the logic described in the later functional specifications.

1. Bug
(Abstract Base Class) (Data members to be declared as "protected")

`int id;`	Identification number (id) for a bug (1,2,3,4,…)
`pair<int, int> position;`	Co-ordinate pair (x,y) represented in a 'pair' struct from <utility> standard library. (0,0) is top left cell on 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 set to 'true' initially. When eaten, this flag is set to 'false'. true => alive, false => dead
`list<pair<int,int>> path;`	Path taken by a bug. (i.e. the List of positions (on grid) that a bug visited.)
`virtual move() {}`	All derived classes must implement logic to move a bug from its current position to a new position based on movement rules for the particular bug type. The *move*() function must be made a **pure virtual function** in the Bug class (with no implementation 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 (a **derived** class that inherits from Bug base class)

`void move(){}`	A Crawler bug moves according to these rules: - moves by 1 unit in the direction it is currently facing - if at edge of board and can't move in current direction (because its way is blocked), then, set a new direction at random. (Repeat until bug can move forward). - record new position in the crawler's path history

3. Hopper (inherits from Bug)

`int hopLength;`	The distance/length that a particular hopper bug can hop (in range (2-4 units). Field to be **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 direction at random. (repeat until bug can move forward) and then move. - if bug can't move/hop the full '*hopLength'*, then the bug does move but 'hits' the edge and falls on the square where it hit the edge/wall - record new position in 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 we want to be able to store all bugs in a vector that will store both Crawlers and Hoppers, so that we can iterate over all bugs and treat then in a similar way. However, we can't use a vector of Bug objects (vector<Bug>), because the derived class objects (e.g. Hopper) would not fit into a Bug object vector element. (The are different sizes). So, one option is to create a **vector of pointers to Bug objects** (**`vector<Bug*>` vect**). 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).

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 *move*() function (using overrides), we **MUST declare a virtual *move*() function in base class, and implement that method in each derived class. In order to avail of the runtime polymorphism behaviour provided by virtual functions, we must use a vector of pointers to bug objects.** This allows the correct *move*() function for the particular derived class type object to be called at runtime.

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 Bug objects dynamically on the Heap, and adding their addresses to the vector. The owner of these object must also remember to free the associated memory.

```
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 GiLab, and make your lecturer a collaborator, and **commit & push** each features 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.

1. **Initialise the bug Board.**
Read the "bugs.txt" file and populate the Bug vector. Assume the file contains valid data. Bugs must be allocated from the heap.

2. **Display All Bugs**
Display all bugs from the vector, showing: *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 to be asked to input a bug id, and the system will search for that bug.
Display 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 and Hopper. (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 in sequence, and the **name** and **id** of all bugs 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 is 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 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) to your system, that has a specific move() behaviour different to those already created. Refactor all code 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

Criteria

Understanding of code presented and ability to explain it. Functional correctness, adherence to specification, quality of code (data structures, readability, maintainability, efficiency) and, quality and functionality of user interface (interaction and clarity) will form part of the criteria for grading.

Upload Requirements

Upload to contain:

1. **Screencast** showing your app working AND a walkthrough of your code identifying the core functionality and the data structures used. Please state your name and class at the beginning. Screencast to be no longer than 5 minutes.
2. Zipped file containing all **source code** and **data files.** (Repo URL alone is not acceptable) ZIP the project folder for upload. Please name your project folder "Lastname_Firstname_CA3" 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.

Standard late submission penalties apply.

Grading Rubrik Student:

C++ (AY2223) Continuous Assessment 3 - PROJECT

Grade Descriptor	**Criteria / Characteristics**	**Grade/ Mark**
Exceptional (80%-100%)	- **All** *Marking Scheme Items* and supporting functionality fully implemented, **and** - work demonstrates thorough understanding of all concepts, algorithms principles and C++ structures/idioms used in the presented solution **and** - **highest quality** code delivered [code readability, variable names, DRY principles applied, use of functions, efficiency, correct and most appropriate data types, quality comments], and - **verifiable evidence** of incremental development of project based on commit/push from repo and in-class discussions with lecturer, and - all project specification requirements delivered, and - exceptional game GUI and game playability	
Excellent (70% - 79%)	- **All** *Marking Scheme Items* and supporting functionality fully implemented, **and** - work demonstrates **very good understanding** of all concepts, algorithms and C++ structures/idioms used in the presented solution, **and** - **good quality code** delivered [code readability, variable names, DRY principles applied, use of functions, efficiency, correct and most appropriate data types, quality comments] (with some deficiencies) , **and** - **verifiable evidence** of incremental development of project based on commit/push from repo with quality commit messages and in-class discussions with lecturer, **and** - all project specification requirements delivered, **and** - good game GUI and playability	
Very Good (60%-70%)	- **almost** all *Marking Scheme Items* * and supporting functionality fully implemented (**missing up to 2 main features) and** - demonstrates **good understanding** of all concepts, algorithms and C++ structures/idioms used in the presented solution, **and** - **good quality code in most cases** delivered [code readability, variable names, DRY principles applied, use of functions, efficiency, correct and most appropriate data types, quality comments] (with some deficiencies), **and** - **verifiable evidence** of incremental development of project based on commit/push from repo with quality commit messages and in-class discussions with lecturer, **and** - most project specification requirements delivered, **and** - basic GUI delivered	
Good (50% - 59%)	- **no less than 2/3rd of** *Marking Scheme Items** and supporting functionality fully implemented, **and** - demonstrates **fair understanding** of **most** concepts, algorithms and C++ structures/idioms used in the presented solution - **quality of code lacking** [code readability, variable names, DRY principles applied, use of functions, efficiency, correct and most appropriate data types, comments] - **verifiable evidence** of incremental development of project based on commit/push from repo and in-class discussions with lecturer (but number/quality of commits lacking) - most project specification requirements delivered - rudimentary or NO GUI delivered	
Pass (40%-49%)	- **majority of** *Marking Scheme Items** (excluding GUI) and supporting functionality implemented, and/or - demonstrated **lack of understanding** of **core** concepts algorithms and C++ structures/idioms used in the presented solution under interview, - **poor quality code** apparent throughout (variable names, DRY principles applied, use of functions and correct data types) - **GitLab commit/push evidence lacking**	
Fail (<40%)	- **less than 5 of** the *Marking Scheme Items** completed, and/or - **poor/no understanding** of concepts, algorithms and C++ structures/idioms used in the presented solution under interview - unable to explain own code (consider possibility of plagiarism) - **no GitLab repo made available with staged commits**	

*****(*relative degree of difficulty of features will be taken into consideration)*
DRY (Don't Repeat Yourself)
Students will normally be ***interviewed*** *to assess their understanding of the work submitted.*