C++

Continuous Assessment 4

Weighting: 40%

B.Sc. (Hons) in Computing in Software Development, Stage 2, Semester 2
B.Sc. (Hons) in Computing in Games 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 the others. An end point is reached when there is only one bug remaining.

Menu Items

```
1. Initialize Bug Board (load data from file)
2. Display all Bugs
3. Find a Bug (given 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 and their Bugs
7. Exit (write Life History of all Bugs to file)
```

Text File: "bugs.txt"
Format for a line (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
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. (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 can be declared with "protected:" access rather than "private:" so that derived class member functions can access those fields directly)

`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.
`int direction;` *(or use enum class)*	direction in which the bug is facing : 1=North, 2=East, 3=South, 4=West
`int size;`	Size of the bug (initially 1-20); biggest bug wins in a fight and others are eaten. Winner grows by sum of sizes of other bugs eaten.
`bool alive;`	Flag indicating life status of a bug. All initialized to true. When eaten, this is set to false. true => alive, false => dead
`list<pair<int,int>> path;`	Path taken by a bug. 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. No implementation is require 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. [Used by the move() function]

2. Crawler (inherits from Bug)

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

3. Hopper (inherits from Bug)

`int hopLength;`	The distance/length that a particular hopper bug can hop (in range (2-4 units)
`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 over edge in current direction, set a new direction at random (repeat until bug can move forward) an then move. - if can't 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 own path

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 define our Bug class Abstract by adding a **pure virtual function**. ("=0").

Notice that the main difference between the derived classes is in behaviour. The behaviour of 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 of some type - 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. So, our best option is to create a vector of pointers to Bug objects (vector<Bug *>). The elements of this vector are of type 'pointer to Bug', so they can point at any derived class objects of 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 implementing different move() functions (overrides), we **MUST use a virtual move() function and pointers to our bug objects** in order to achieve **runtime polymorphism**. This allows the right move() function for the particular derived class object to be called at runtime.

Therefore, we 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 in the Heap, and adding their addresses to the vector.

```
vector<Bug*> bug_vector;
```

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

1. Initialise Bug Board.
Read the "bugs.txt" file and populate the Bug vector. Assume the file contains valid data.

2. Display All Bugs
Display all bugs from the vector, showing: id, type, location, size, direction, hopLength, status, 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 for a bug id, and system will search for that bug. Display bug details if found, otherwise display "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 of 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.
I recommend that you implement only move() initially. The fight and eat behaviour can be developed at the end, 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).
101 Crawler Path: (0,0),(0,1),(1,1),(2,1),(3,1) Eaten by 3
102 Hopper Path: (2,2),(2,3), Alive!

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

7. 6.Display all Cells
Display all cells in sequence, and the name and id of all bugs occupying each cell.

```
(0,0): empty                    { meaning: cell (0,1) is empty }
(0,1): empty
(0,2): Crawler 101, Crawler 103 { there are 2 Crawled bugs in this cell }
(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 Man Standing. Two or more bugs equal in size won't be able to overcome each other and will all continue to live.
You should discuss your approach to this with your lecturer before implementation

Marking Scheme

Base and Derived classes implemented and tested.	10
Bug vector initialized with bugs from text file	10
Display all Bugs	10
Find a Bug	10
Tap the Bug Board (move all bugs)	10
Display Life History of all Bugs	10
Display All Cells and their Bugs	10
Exit (write Life History of all Bugs to file)	10
Implement **and fighting/eating** extension to option 4	20

Criteria

Functional correctness, quality of code (clarity, accuracy, maintainability) and user interface (interaction and clarity) will form part of the criteria for grading in each case.

Upload Requirements

Upload to contain:

1. **Screencast** showing your app working. No code. State your name and class at beginning. 3 minutes max.
2. Zipped file containing all **source code** and **data files**
3. **CA sheet** with your name typed in place of a signature.

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

Standard late submission penalties apply.