CS 162 Group 7:
- Matt Dienhart
- Jacob Carter
- Kevin Allen
- Brian Hoglund

- Group Project Reflection -

Part 1: Program Design

Critter Class:

Protected members:
- X location
- Y location
- Age of critter in turns
- Critter's symbol ("O" for Ant, "X" for doodlebug)

Public members:
- Get X and Y location
- Get critter's age
- Increment critter's age
- Move (virtual function, not defined in Critter)
- Breed (not virtual, since the project spec says this works the same way for Ants as for Doodlebugs)
  - Similar to Move in that the critter must choose a direction N, S, E, or W to spawn a new object of the same type as it, with age 0
  - We can either make this function check if the object's age is high enough before breeding, or we can just tell the Game class not to call Critter.breed() unless the critter is old enough. I think the latter way allows breed() to be non-polymorphic.

Ant Class:

Private members:
- None?

Public members:
- Move (polymorphed from base class)
  - There are four possible directions to move (N, S, E, W)
  - The ant knows its own location, so it needs to ask the board whether the spaces north (current X, current Y-1), east (current X+1, current Y), south (current X, current Y+1), or west (current X-1, current Y) are blocked as it tries to move randomly in one of those directions.

Doodlebug Class:

Private members:
- Number of turns left until starvation

Public members:
- Move (polymorphed from base class)
  - First it must check if there are any ants in the adjacent cells, and randomly choose to eat one of them and take their place

- If there are no ants nearby, then the code should be basically the same as the Ant's move function.

Board Class:

Private members:
- Size of board (X)
- Size of board (Y)
- Pointer to dynamic array containing pointers to Ant objects
  - (size of dynamic array is board x size * board y size, probably will never need that many objects but it saves us from having to resize the array during run-time)
- Pointer to dynamic array containing pointers to Doodlebug objects

Public members: (these are sort of dependent on how the person who writes the dynamic arrays wants to implement them)
- Add Ant
- Remove Ant
- Add Doodlebug
- Remove Doodlebug
- Get Ant location (takes an integer index, returns X and Y location as integers)
- Get Doodlebug location (takes an integer index, returns X and Y locations)

Game Class:

Private members:
- Current turn
- Maximum number of turns

Public members:
- Void playGame()
- Get valid input from the user
- Print board state

Detailed psuedocode for playGame() function:

Call the createBoard() function to make the board (possibly best done in the Board constructor)
Call the initializeBoard() function to set up the game.
Get a pointer to the board via a Board class member function (To access the 2d array).

For the number of steps specified by the user, do the following
- Loop through all the pointers in the board array
  - if pointer is null, leave it alone.
  - If (pointer points to a doogleBug)
    - Call the makeMove() function on the doodleBug.
    - Increment age of doodlebug by one (unless done in the makeMove() function).
    - If the age of the doodlebug is 8, create a new doodlebug.

Loop again through all the characters in the board array
- if pointer is a nullptr leave it alone.
- If (pointer points to an Ant)

Call the makeMove() function on the ant.
Increment the age of the ant by one.
If the age of the ant is 3, create a new ant

Part 2: Test Plan

Test Case	Input Value	Driver Functions	Expected Outcome	Actual Outcome
string input	"foo"	main() play or quit selection	Error, repeat prompt	Error, repeat prompt
invalid integer input	-3		Error, repeat prompt	Error, repeat prompt
double input	1.5		decimal is truncated, program advances	decimal is truncated, program advances
too high input	3		Error, repeat prompt	Error, repeat prompt
too low input	0		Error, repeat prompt	Error, repeat prompt
highest valid input	2		program advances	program advances
lowest valid input	1		program terminates	program terminates
string input	"foo"	menu() number of steps entry	Error, repeat prompt	Error, repeat prompt
invalid integer input	-3		Error, repeat prompt	Error, repeat prompt
double input	5.5		decimal is truncated, program calls Game.playGame(5)	decimal is truncated, program calls Game.playGame(5)
too low input	0		Error, repeat prompt	Error, repeat prompt
lowest valid input	1		program calls Game.playGame(1)	program calls Game.playGame(1)
high valid input	200		program calls Game.playGame(200)	program calls Game.playGame(200)
string input	"foo"	after game completes, main() play again or quit selection	Error, repeat prompt	Error, repeat prompt
invalid integer input	-3		Error, repeat prompt	Error, repeat prompt
double input	1.5		decimal is truncated, program calls Game.playGame(), no data remains from previous game	decimal is truncated, program calls Game.playGame(), no data remains from previous game
too high input	3		Error, repeat prompt	Error, repeat prompt
too low input	0		Error, repeat prompt	Error, repeat prompt
highest valid input	2		program calls Game.playGame(), no data remains from previous game	program calls Game.playGame(), no data remains from previous game
lowest valid input	1		program terminates	program terminates
No room to move	all adjacent board spaces are occupied, or are an edge	Ant.move()	Ant remains where it is	Ant remains where it is
Multiple spaces to move	all adjacent spaces are empty		Ant moves to a randomly selected adjacent empty space	Ant moves to a randomly selected adjacent empty space

Ant is not old enough to breed	Ant age is less than 3	Ant.breed()	No new Ant object is created	No new Ant object is created
No room to create new ant	all adjacent board spaces are occupied		No new Ant object is created	No new Ant object is created
Only 1 space to create new ant	all but one adjacent spaces are occupied		New Ant object is created in empty space with age 0	New Ant object is created in empty space with age 0
Multiple spaces to create new ant	more than one adjacent space is empty		New Ant object is created in a randomly selected empty adjacent space with age 0	New Ant object is created in a randomly selected empty adjacent space with age 0
No room to move	all adjacent board spaces are occupied by Doodlebugs, or are an edge	Doodlebug.move()	Doodlebug remains where it is	Doodlebug remains where it is
Multiple spaces to move	all adjacent spaces are empty		Doodlebug moves to a randomly selected adjacent empty space	Doodlebug moves to a randomly selected adjacent empty space
One Ant in adjacent space	An Ant is in an adjacent space, other adjacent spaces are empty		Doodlebug eats the Ant and takes its place	Doodlebug eats the Ant and takes its place
Multiple Ants in adjacent spaces	More than one adjacent spaces are occupied by Ants		Doodlebug randomly selects one of the adjacent ants, eats it, and takes its place	Doodlebug randomly selects one of the adjacent ants, eats it, and takes its place
Doodlebug not old enough to breed	Doodlebug's age is less than 8	Doodlebug.breed()	No new Doodlebug is created	No new Doodlebug is created
No room to create new doodlebug	all adjacent board spaces are occupied		No new Doodlebug object is created	No new Doodlebug object is created
Only 1 space to create new doodlebug	all but one adjacent spaces are occupied		New Doodlebug object is created in empty space with age 0	New Doodlebug object is created in empty space with age 0
Multiple spaces to create new doodlebug	more than one adjacent		New Doodlebug object is created in a randomly selected	New Doodlebug object is created in a randomly selected

	space is empty		empty adjacent space with age 0	empty adjacent space with age 0
Doodlebug eats before starving	Doodlebug eats an Ant within 3 turns	Doodlebug.starve()	Doodlebug remains on the board and starvation counter resets	Doodlebug remains on the board and starvation counter resets
Doodlebug starves	Doodlebug goes more than 3 turns without eating		Doodlebug is removed from the board	Doodlebug is removed from the board
End of round	print the board state	board.printBoard()	board state is printed correctly after every turn	board state is printed correctly after every turn
after several back-to-back simulations		main() program execution	no errors or memory leaks	no errors or memory leaks

Part 3: Reflection

The original concept for the Board class for this project consisted of a pair of arrays containing dynamically allocated critter objects: one would be a list of all the Ant objects, the other would be a list of all the Doodlebug objects. The objects would store their own position in their data members, and functions like critMove() and printBoard() would iterate through these arrays, asking each object for its location information.

However, we decided to switch to a different implementation, one where a single 2D array (theBoard) contained pointers to both types of critter objects, and the row/column index of the pointer to each object would also serve to track that object's position on the board. Empty spaces would be represented by the pointer for that row/column index pointing to NULL. This allowed us to more easily access the board state from the Game class, since "theBoard" array always contains an up-to-date state of the board in one place.

When we started implementing the critMove() and critBreed() functions for the Ants and Doodlebugs, we realized that it would be necessary to keep track of which critters had already moved during a round, and which ones had not. This is because the Game::playGame() function loops through every row and column index in "theBoard", starting from the upper-left corner and finishing in the lower-right, and it processes the move and breed commands for every critter it finds in that order. So if a critter happens to move to the right one column or down one row, it will accidentally be allowed to move and/or breed twice because the playGame() loop will find it again in a later iteration. So we needed to have each critter keep track of it's movement history, and playGame() checks that for each critter in a conditional statement before proceeding.

Work Distribution

Matt Dienhart wrote the main.cpp function, performed testing on the program, and wrote the reflections document.

Jacob Carter wrote the Board class.

Brian Holgrund wrote the Game class.

Kevin Allen wrote the Critter class and the derived Ant and Doodlebug classes.