ECE 150 - Project 3 Geesespotter

Due Dates

This project has two submissions, both are required.

Project_3_geesespotter_first: submit geesespotter.cpp by Friday November 3

Project_3_geesespotter_FINAL: submit geesespotter.cpp by Friday November 10

Course Material Focus

- Functions
- Bitwise Operations
- Arrays
- Dynamic Memory Allocation

Estimated time to complete:

These estimates assume an even division of time across two-to-three weeks.
DO NOT attempt in a single sitting! This tends to increase the time needed for debugging.

- Previous C++ experience: ~4-8 hours
- Previous coding experience: ~6-10 hours
- No coding experience: ~8-14 hours

(The early submission covers about 1/3 of the work of the assignment)

Table of Contents

0. Recomended Task List

- [ ] 1. Read this entire document and make notes
- [ ] 2. Create a Project3 directory/folder
- [ ] 3. Download the starter files into the created directory
- [ ] 4. Create an empty geesespotter.cpp file and add compiler directives:

```
#include "geesespotter_lib.h"
#include <iostream>
```

- [ ] 5. Create a skeleton of all eight required functions in geesespotter.cpp
- [ ] 6. Compile the skeleton using:

```
g++ -std=c++11 geesespotter.cpp geesespotter_lib.cpp -o main
```

- [ ] 7. Complete functions in the following order:
  - [ ] 7.1 - create_board() - uses Lecture 2.3 Dynamic allocation of arrays
  - [ ] 7.2 - clean_board() - uses Lecture 2.3 Dynamic allocation of arrays
  - [ ] 7.3 - print_board()
  - [ ] 7.4 - hide_board()
  - [ ] 7.5 - mark()
- [ ] 8. **Submit**: Project_3_geesespotter_first
- [ ] 9. Complete the remaining functions in the following order:
  - [ ] 9.1 - computer_neighbours()
  - [ ] 9.2 - is_game_won()
  - [ ] 9.3 - reveal()
- [ ] 10. **Submit**: Project_3_geesespotter_FINAL

1. Introduction

Geese are part of Waterloo life! Everyone knows not to get too close to a goose. In this project you will develop “GeeseSpotter”, a simple, text-based, single-player game.

The game is played on a 2-dimensional playing board. The playing board is made up of individual fields (x-horizonal, y-vertical). Each field may or may not contain a goose. Geese are messy, and the neighboring fields of a field that contains a goose contain goose droppings. An observant Waterloovian can tell how many geese are in neighboring fields by observing the amount of goose droppings in the current field. The University is asking us to determine all fields that contain a goose, without disturbing/revealing a goose.

1.1. Similar to Minesweeper

The GeeseSpotter game is similar to the classic game Minesweeper. It is recommended for anyone who has not played Minesweeper before to play the game in order to get a better intuition about how the game works. A free version of Minesweeper can be played by searching for "Minesweeper" on Google.

2. Code Project Setup and Compilation

This project combines four files to create the game. In this project you will be creating and submitting the fourth and final file.
The main function is written in the provided code. You are responsible for writing functions that store and manipulate game data, as well as functions responsible for printing the board. The main game loop and user interface are part of the provided code.

2.1 Starting files

Files for this project are seperated into two parts:

- Function declarations: in *filename* **.h** files (both are provided)
- Function definitions: in seperate *filename* **.cpp** files (only one is provided).

The full program consists of two halfs:

- *geesespotter_lib*: This code is responsible for the main game loop and processing input
    - geesespotter_lib.h - the declarations (provided)
    - geesespotter_lib.cpp - the definitions (drovided)
- *geesespotter*: This code has the game data, game data manipulation, and board output
    - geesespotter.h - the declarations (provided)
    - **geesespotter.cpp** - (To be created by you)

You are responsible for writing *geesespotter.cpp* which completes the functions declared in geesespotter.h

1. Put **"geesespotter_lib.h"**, **"geesespotter_lib.cpp"**, and **"geesespotter.h"** in a folder you create.
2. Create a file called **"geesespotter.cpp"** inside the same folder. There should now be 4 files in your folder.
3. At the top of your **geesespotter.cpp** file, make sure to write the following statement:

```
#include “geesespotter_lib.h”
#include <iostream>
```

4. Start by creating the skeleton definitions of every functions in your **geesespotter.cpp** file as stated in Your Tasks.
5. The **"geesespotter.cpp"** file will be the only file you need to submit on Marmoset. Submit this file with the skeleton function definitions for all eight required functions to make sure it compiles on Marmoset.

2.2 Compilation Instructions

6. To test your **geesespotter.cpp** file, type the following command into terminal while in the folder directory.

```
g++  -std=c++11 geesespotter.cpp geesespotter_lib.cpp -o main
```

7. Run `.\main` to run the test file.

Your geesespotter.cpp file does NOT require a main function or function declarations. Only function definitions are required. Note that as you are not submitting a `main()` function, you will not require `#ifndef MARMOSET_TESTING` pre-processor directives.

3. Game Setup

The player is first asked for the desired dimensions of the board and then for the number of geese that will start hidden on the playing board. After an empty board is created, geese are placed randomly in sperate fields in the board.

Let’s look at a simple four by four board. Two geese are hidden on the board, represented with the number **9** on the field. The x coordinate increases from left to right (horizontally) and starts at zero. The y coordinate increases from top to bottom (vertically) and also starts at zero.

The two geese are at coordinates (`x:2, y:0`) and (`x:1, y:2`) highlighted with a yellow background. These boards represent the underlying game data, during play all fields start hidden (see: Playing the game).

Empty board (4 x 4)

y\x	0	1	2	3
0				
1				
2				
3				

Two geese placed randomly

y\x	0	1	2	3
0			9	
1				
2		9		
3				

The other fields contain no geese, but instead contain the number of neighboring fields that contain a goose. That number can range from zero to eight. The value of a field is determined by looking at the adjacent fields and counting the number of geese.

x:0, y:0 has no adjacent geese

y\x	0	1	2	3
0	0		9	
1				
2		9		
3				

x:2, y:2 has 1 adjacent goose

y\x	0	1	2	3
0			9	
1				
2		9	1	
3				

x:2, y:1 has two adjacent geese

y\x	0	1	2	3
0			9	
1			2	
2		9		
3				

Once the number of neighboring fields that contain a goose is calculated - for all empty fields - the final board setup would look like this. With the board values in place the board is ready to be *hidden* and for play to start.

A fully setup board

y\x	0	1	2	3
0	0	1	9	1
1	1	2	2	1
2	1	9	1	0
3	1	1	1	0

4. Playing the Game

The game gebins with the player being presented a playing board with each field hidden. Hidden fields are represented with an asterisk (*****). Each turn, the player is given the option to either **S**how a field that they believe is empty or to **M**ark/unmark a field that they believe is likely occupied by a goose. After entering **S** or **M** the player is then asked to provide the two coordinates (**x**/column, **y**/row) of the field they are showing or marking.

If the player selects to show a field, there are two possible outcomes:

1. The field contains a goose, and the player upsets the mighty goose. The game is lost immediately and a new game is started.
2. The field does not contain a goose; the selected field is shown. When shown a hidden field (*****) will now display the fields value (0-8).
   If the field and none of its neighbours contain a goose (as in the x:0,y:0 example above), the neighboring *unmarked* fields are also shown, and the player learns how many geese are in the neighbours of each of the neighboring fields. Values can range from 0 to 8.
3. Only hidden fields can be shown. Marked fields must be unmarked before they can be revealed.

If the player selects to **M**ark or unmark a field, the status of the field is changed from hidden to marked or from marked to hidden, depending on the current status of the field. A player marks a field if they believe that it contains a goose, this is represented with an **M**. Only hidden fields can be marked (**M**) and only marked fields can be unmarked.

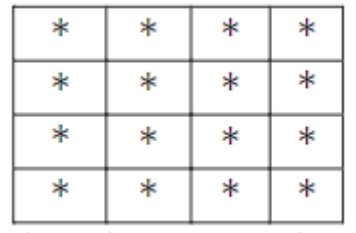
The current status of the board is printed after each action of the player (i.e. show, sark/unmark).

The game is won when all fields that do not have a goose have been revealed/shown. Once a game is won or lost, the game starts again by asking for the dimensions of the next playing board.

The player is free at any turn to choose to **R**estart with a new game or to **Q**uit the game and exit the program.

4.1. Example Gameplay

Let's take the earlier playing field. Initially, all fields are hidden:



As a first move, the player selects to reveal field (`x:0, y:0`). The value of that field is 0 (zero) - which indicates that none of its neighbours contain a goose - so the neighboring fields are revealed. The new board is now:

0	1	*	*
1	2	*	*
*	*	*	*
*	*	*	*

Next, the player decides to reveal field (`x:2, y:1`). The value of that field is 2 - indicating that there are two adjacent fields that conatin a goose, so no further fields are revealed. The new board is now:

0	1	*	*
1	2	2	*
*	*	*	*
*	*	*	*

The player now guesses that field (`x:1, y:2`) contains a goose and decides to mark the field. The board is now:

0	1	*	*
1	2	2	*
*	M	*	*
*	*	*	*

The player reveals field (`x:2,y:0`). A goose inhabits that field and the game is lost.

0	1	9	*
1	2	2	*
*	M	*	*
*	*	*	*

4.2 Example Console Output

The following is a sample interaction with the game as it appears in the console/terminal:

First, the game is setup by determining the dimensions of the board and the number of geese.
The board is created, printed, and begins fully hidden.

```
Welcome to GeeseSpotter!
Please enter the x dimension (max 60): 5
Please enter the y dimension (max 20): 5
Please enter the number of geese: 2
*****
*****
*****
*****
*****
```

When a player chooses to show a location it is revealed. The first location (`x:0,y:0`) has zero adjacent geese and immediatly reveals all non-marked fireds adjacent to it. The second location (`x:4,y:0`) has one goose adjacent and only reveals itself. The third location shown (`x:1,y:4`) also has zero adjacent geese and reveal the five adjacent fields.

```
Please enter the action ([S]how; [M]ark; [R]estart; [Q]uit): S
Please enter the x location: 0
Please enter the y location: 0
00***
11***
*****
*****
*****
Please enter the action ([S]how; [M]ark; [R]estart; [Q]uit): S
Please enter the x location: 4
Please enter the y location: 0
00**1
11***
*****
*****
*****
Please enter the action ([S]how; [M]ark; [R]estart; [Q]uit): S
Please enter the x location: 1
Please enter the y location: 4
00**1
11***
*****
111**
000**
```

The player then chooses to show a location that has already been revealed (x:2, y:4). Because the location is already revealed no changes are made to that location however, because there are zero adjacent geese at that location the adjacent fields are revealed.

```
Please enter the action ([S]how; [M]ark; [R]estart; [Q]uit): S
Please enter the x location: 2
Please enter the y location: 4
00**1
11***
*****
1110*
0000*
```

5. Data Structures

5.1 The Board

The board is represented as a **one-dimensional array of chars**, in row-major order. For our example board:

0	1	9	1
1	2	2	1
1	9	1	0
1	1	1	0

The char array will contain the following values:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Value	0	1	9	1	1	2	2	1	1	9	1	0	1	1	1	0

In this example, the first row (0,1,9,1) of the board becomes the first values of the array (indicies 0-3). The second row of the board (1,2,2,1) follows (indicies 4-7), followed by the third row, etc. In this example each row contains four fields, but in practice the width of a row changes. Remember that the parameter `x_dim` should be used to defines how wide each row is, and the parameter `y_dim` defines how many rows the board has.

5.2 Fields

Fields are represented as chars. We use the lower four bits of a char to represent the *value* of goose neighbours and the value 9 to represent a goose. We set markedBit() to mark a field and we set hiddenBit() to hide a field. Combining a field value with the valueMask() always determines whether a field contains a goose or how many geese neighbours there are.

Fields are written in hex: 0x##. The each number in a hex code represents four bits.
Example, the hex number 0x21 is the binary number 0b 0010 0001.

Consider a field that has two geese neighbours. Its value is `0x02` initially. When the field is hidden, its value is changed to `0x22`, by setting the hidden bit `0x20`. If the player then decides to mark that field, its value is changed to `0x32` (both the marked bit `0x10` and the hidden bit `0x20` are set to true). The value can then be unmarked, going back to `0x22`. And finally, the field can be revealed, changing it to `0x02`.

5.3 Binary Encoding

The following section describes how information about the game board is stored in the board array. Each element of the board array is an 8-bit (char) value that represents one "field" of the game board. There are three pieces of information stored within this 8-bit value:

- whether the field has been revealed by the player or not
- whether the field has been marked by the player or not
- the number of geese adjacent to the field, or if there is a goose in the field

The following diagram shows which bits are used to encode this information.

```
        [7][6][5][4] [3][2][1][0]
              |  |   |  |  |  |
hiddenBit -----+  |   +--+--+--+-- number of adjacent geese/goose in field
markedBit --------+
```

Note that bits 6 and 7 are unused and so should always have a value of 0.

Here are some examples of how this encoding is used.

hex value	binary value	meaning
0x03	0b000 00011	field is revealed; not marked; three adjacent geese
0x29	0b001 01001	field is hidden; not marked; goose in the field
0x39	0b001 11001	field is hidden; field is marked; goose in the field

A working knowledge of bitwise operations is required for this project. To review this required knowledge, refer to the lecture content on this topic (1.27 Bitwise and bit-shift operators).

5.4 Bitwise Operators Order of Operations Warning

It should be noted that the bitwise AND (**&**), XOR (**^**), and OR (**|**) operators have lower precedence for order of operations than logical comparison operators (e.g. ==, !=, <, >, etc.). It is recommended that bitwise operations be wrapped in parentheses to ensure they are evaluated first. For reference: https://en.cppreference.com/w/cpp/language/operator_precedence

6. Provided Code

We provide you with the basic game skeleton and you are asked to implement the details.
Note: functions 5 (marked_bit),6 (hidden_bit), and 7 (value_mask) are useful for bitwise calculations.

We provide the following eight functions - found in *geesespotter_lib.cpp*:

1. `int main();`
   The main entry point for the program - immediatly calls game().
2. `bool game();`
   One complete game. This is the main function that controls gameplay.
3. `void spread_geese(char *board, std::size_t x_dim, std::size_t y_dim, unsigned int num_geese);`
   Randomly spreads num_geese geese over the board. This function is called after a board has been created and before neighbours are calculated.
4. `std::size_t x_dim_max();` and `std::size_t y_dim_max();`
   Helper to determine maximum x (horizontal) and y (vertical) dimensions.
5. `char marked_bit();`
   Returns a bitwise mask: A field with bit `0x10` set is a marked field (0b00010000).
6. `char hidden_bit();`
   Returns a bitwise mask: A field with bit `0x20` set is a hidden field (0b00100000).
7. `char value_mask();`
   Returns a bitwise mask: The mask `0x0F` can be used to easily remove the marked and hidden bits from a field value (0b00001111).

7. Your Task

Your task is to implement a number of key functions and data elements of the GeeseSpotter game. Some code that contains the basic game user interface is provided (see: Provided Code section above).

7.1 Implement the following functions:

All of the following functions are called by the main game code of *geesespotter_lib.cpp* thoughout the course of play. You are responsible for their functionality, *geesespotter_lib.cpp* will call your functions while running the game.

1. `char *create_board(std::size_t x_dim, std::size_t y_dim);`
   Allocate a *1-dimensional* char array with x_dim * y_dim elements and initialize each element with zero. Note that the initialization value is the value 0 and not the character '0'. The data type of the array is char not because ASCII characters are going to be stored in each element, but because char is an 8-bit data type. See Data Structure section above for details.

2. `void clean_board(char *board);`
   Deallocate the given board - clears that board from memory.

3. `void print_board(char *board, std::size_t x_dim, std::size_t y_dim);`
   Prints the content of the board to `std::cout`. Each field is represented with a single character. If a field is hidden this function prints an asterisk '*****', if a field is marked an '**M**' is printed, and otherwise the field *value* is displayed ('**0**' to '**9**'). Note that the only time a field with a goose is displayed is at the end of the game (either won or lost). There are no spaces between printed board elements.
   Refer to the Data Structure section below for further information on how information is encoded in each array element.

4. `void hide_board(char *board, std::size_t x_dim, std::size_t y_dim);`
   Hide all the field values. The hide_board function is called after a board has been created, geese spread (by geesespotter_lib.cpp), and neighbours calculated.

5. `int mark(char *board, std::size_t x_dim, std::size_t y_dim, std::size_t x_loc, std::size_t y_loc);`
   This function is called when the player attempts to mark a field. If the field is already revealed, its state should not be changed and the function returns the value 2. Otherwise, the Marked bit for the field is toggled and the function returns 0. If the Marked bit is on, then this function turns it off. If the Marked bit is off, then this function turns it on.

6. `void compute_neighbours(char *board, std::size_t x_dim, std::size_t y_dim);`
   This function receives a board array with all elements set to the values 0 or 9. The function updates the board array such that all fields that do not have a goose (value 9) have their value set to the number of geese adjacent to that field.

7. `bool is_game_won(char *board, std::size_t x_dim, std::size_t y_dim);`
   Determine whether the board is in a won state. The game is won when all fields that do not have a goose have been revealed. Returns `true` if the game is won and `false` otherwise. Note the following special case. By creating a 5 by 5 board with 25 geese on it, you would expect that you win immediately as every field contains a goose, however, this check is only made after a player has made an action.

8. `int reveal(char *board, std::size_t x_dim, std::size_t y_dim, std::size_t x_loc, std::size_t y_loc);` This function reveals a field if it is hidden *and* unmarked. Revealing a field means to set the hiddenBit to zero for the specific field. If the field is marked, the function returns the value 1 without modifying the field. Marked fields can only be unmarked. If the field is already revealed, the function returns the value 2 without modifying the field. If there is a goose in the field the function reveals the field and then returns the value 9. For all other cases, the function returns the value 0.
   If an empty field is revealed (it contains the value zero), also reveal the values of all *unmarked* neighboring fields. Do not recursively reveal neighboring fields if they also are fields with zero geese. Reveal only the neighboring 8 fields. The following shows the difference between the expected result and a recursive reveal, when the location (2, 3) is revealed.

7.2 Differences from Minesweeper

When an empty space in Minesweeper is selected - the 8 surrounding squares are revealed. If any of those revealed squares contain an empty square the 8 squares surrounding that square are revealed recursively in a chain reaction. In Geesespotter - there is no recursion and no chain reactions.

```
The expected result when revealing x:2, y:3
(i.e. only the surrounding 8 non-marked fields are revealed)
******
******
*210**    < - notice the zero in the top left corner
*100**
*101**
******

Result using incorrect recursive reveal of  x:2, y:3
(i.e. typical minesweeper game behavior)
**101*
**101*
*2101*    < - notice the zero incorrectly recursively revealed neighbours
*1001*
*10111
*10000
```

7.3 Hints and Recommendations

- **Helper Functions**
  You may, in addition to the functions above, implement helper functions. Theses functions should be declared and defined in the *geesespotter.cpp* file (do not put declarations in the *geesespotter.h* file). Remember that functions can call other functions.

- **Work in coordinates!**

  Where possible you should design your code to work in x,y (column, row) coordinates. Creat/Use a helper function to translate an x,y-coordinate into a 1D-array index. This will dramatically simplify your calculations in the compute_neighbours function.

- **Temporary Modifications for Debugging**
  Temporarily modifying print_board to print extra information about the data contained in each field can be a useful debugging tool. Similairly, disabling hideboard can be a useful tool when debugging compute_neighbours().

8. Marmoset Submission Details

Project name: Project 3 Geesespotter
Submission filename: geesespotter.cpp

Project 3 has **two** required submissions. The first submission is a subset of the final submission. Each student is required to submit to **both** parts.

The two submissions also helps students to know which functions to focus on first. The purpose of requiring students to make two submissions is to assist students learn to tackle a larger scale codebase, which requires work over a longer period of time. With a single submission, students tend to leave too much coding to be done close to the final deadline, resulting in higher stress and ineffective learning.

8.1 First Submission

Marmoset project: Project_3_geesespotter_first
Due: 11:59pm on Friday, **November 3**, 2022

- The functions tested in *First* submission are the following:
    1. `create_board()`
    2. `clean_board()`
    3. `print_board()`
    4. `hide_board()`
    5. `mark()`

Note: In order to compile the first submission requires function skeletons for these last three functions.

8.2 First Grading + Bonus

- The tests are out of 10, but we will mark it out of 5
    - ex: 7/10 will be marked as 7/5 (i.e. 5/5 + 2 bonus)
- Which means:
    - Bonus: any grade above 5/10 will count as bonus toward the project total grade

8.3 Final Submission

Marmoset project: Project_3_geesespotter_FINAL
Due: 11:59pm on Friday, **November 10**, 2023

- The *FINAL* submission tests *all* functions (including those from the First Submission):

8.4 Final Grading

Your final grade will be the sum of *First Submission* out of 5 AND (*Final Submission* + Bonus)

- If you got 7/10 on First (5/5 + 2 Bonus) and 20/25 on final:
- Then your overall grade would be (5 + 2 + 20) out of (5 + 25) = 32/35 (plus secret tests)

The overall project grade will be capped at 100%. It is not possible to earn a grade higher than 100% on the overall project grade.

9. Plagiarism detection software

We analyze all submissions with automated plagiarism detection software. Please consult the syllabus for more information.