Recitation 03

Focus

- Basics of Object Oriented Programming: Encapuslation and Data Hiding

Task

The game Minesweeper has certainly occupied an amazing number of hours of people's time. The author of a free game like that should either be thanked or severely punished, depending on your point of view.

In the hopes of regaining some element of productivity from that game's invention, here we will attempt to implement it and learn something about OOP in the process.

The Game

Well, no, I'm not going to go into the details of the game. If you have never played it or need a reminder, doing a websearch will pull up many hits. The first hit I came across was minesweeperonline.com, where you can play the game. And there is, of course, a wikipedia page.

The main thing you have to understand for programming is what happens *when a player selects a tile*. Whatever is under it, now becomes visible and...

- If it is bomb, the player dies, end of game.
- If the tile is not a bomb but is adjacent to at least one, then the tile will show the number of bombs that the tile is adjacent to.
- What happens if the tile is not a bomb and is not adjacent to one? This is the once complicated part. Now there is a rippling effect. Not only does the tile become visible, but all 8 adjacent tiles are also made visible, following the same rules we just discussed. As each becomes exposed, we again consider whether it is adjacent to a bomb or not. If it is then just show the count of bombs it is adjacent to. If it is not adjacent to one, then again we make it visible and continue in the same way. (Note that it won't *be* a bomb.) This can result in a large number of squares suddenly becoming visible with a single tile selection.Note that this is very much like a "search" or "traversal" that you learned in data structures. What were the easiest

versions of a search that you learned? Breadth first search and depth first search. I would recommend doing a depth first search, as you only need a vector to keep track of your *todo* list.

Here is some pseudo-code for **depth-first search**:

```
put first item on todo list
while(todo list not empty)
    take next item off, call it current
    if current has already been processed continue
    process current
    if appropriate, add current's successors to the the todo list
```

- What is a todo list? A vector
- How do you add something to the list? `push_back`
- How do you remove something? `pop_back`. But not that pop_back does not actually return anything (void return type), so use the method `back` first to access the item.
- What is an item here? A tile or the coordinates of a tile.
- What does it mean to process an item? Make it visible
- What are its successors? the 8 tiles / coordinates around it.
- When is it appropriate to add successors to the todo list? Only if current has no bomb neighbors and the neighbors are with-in the bounds of the border.

GUI?

Our point is not to worry about using a graphical library. Instead we will use a simple text interface. I don't wish to tell you how to design your interface. Hopefully it will be clear and intuitive enough that the player will quickly accomodate.

Actually, I am going to provide a user interface to save you a few lines of code. You can certainly write your own! But don't think about that till you have implemented the Minesweeper class.

Program Design

First, you will be implementing a class, `Minesweeper` to represent the game. Below we list the required public methods for the class. These methods allow us to provide you with the user interface mentioned above.

Second, there is a question about how large the game board should be. Being lazy, I chose a fixed size of 10 x 10. You are welcome to be more flexible, if you like. But whatever size you pick, those *numbers* should only appear in one place in your code. Everywhere else they should appear as field names, e.g. `rows` and `cols`.

Probably the biggest decision you will make initially is how to represent the information about each tile. Don't obsess over this. Just pick something that looks like it will work for you and implement it. First, what do you need to know about each tile?

- Is it a bomb?
- If it is not a bomb, how many bombs are around it?
- Has the count of neighboring bombs been "made visible" to the player yet?

Make a tile class/struct? Fine. Use some other encoding? Fine, too. I thought about defining such a type, chose not to, and will likely go back at some point and reimplement, just to see how much cleaner the revised code will look. (I'm pretty sure it will look a little better, but not dramatically.) If you are curious, I used:

- a 2d vector (i.e. a vector of vectors) of ints to hold the counts, using a value of -1 for a bomb.
- a second 2d vector of bools to track whether the squares are visible or not.

Make a border? Just as with Conway's Game of Life, it is possible that a border will simplify your code. In my implementation, I did not use a border and I only see a few lines of code that could have been avoided if I had had one. But I avoided having to create one.

So, to help you along, I am going to specify a set of methods that must be public. You may have additional public or private methods, but these are the methods our driver program will require, and they seem like a reasonable set for you to provide.

- `Minesweeper`. The constructor to initialize the game.
- `display`. To display the board. Take a boolean argument specifying whether to display the bombs or not. We won't want to for normal turns, but will want to at the end.
- `done`. Returns true if the game is over, false otherwise.
- `validRow`. Takes a row number. Returns true if the row number is valid, and false otherwise.

- `validCol`. Takes a column number. Returns true if the column number is valid, and false otherwise.
- `isVisible`. Takes a row number and a column number. Returns true if the corresponding cell is visible and false otherwise.
- `play`. Takes a row number and a column number, changing the state of the board as appropriate for this move. Returns false if the move results in an explosion.

Randomness

It would be boring to play the game with the exact same bombs every time so you will want to generater bombs randomly. Perhaps the easiest way to generate a random number is with rand() from the <cstdlib> header file.

By default, every time you run your program you will see the *same sequence* of random numbers. You can change the sequence by "seeding" the random generator. This is done with the function srand(int). to provide a "seed" for rand(). If you are using srand, you are going to only call it *once* in your program. A common mistake is to call it in a loop.

One common hack is to call srand(time(NULL)) to provide the seed. That uses the current time. Note that the function time requires that you include <ctime>.

Other Things

The option for a player to "flag" a square that he is sure is a bomb would make it more pleasant to play the game. I have not assumed this ability in the interface that I provided. Feel free to add a method and extend the interface to use it, but best to first get the rest working.

Maintained by John Sterling (john.sterling@nyu.edu). Last modified: 02/07/2018

Driver Program / User Interface

```
int main() {
    Minesweeper sweeper;
    // Continue until only invisible cells are bombs
    while (!sweeper.done()) {
        sweeper.display(false); // display board without
bombs
```

```
        int row_sel = -1, col_sel = -1;
        while (row_sel == -1) {
            // Get a valid move
            int r, c;
            cout << "row? ";
            cin >> r;
            if (!sweeper.validRow(r)) {
                cout << "Row out of bounds\n";
                continue;
            }
            cout << "col? ";
            cin >> c;
            if (!sweeper.validCol(c)) {
                cout << "Column out of bounds\n";
                continue;
            }
            if (sweeper.isVisible(r,c)) {
                cout << "Square already visible\n";
                continue;
            }
            row_sel = r;
            col_sel = c;
        }
        // Set selected square to be visible. May effect
other cells.
        if (!sweeper.play(row_sel,col_sel)) {
            cout << "Sorry, you died..\n";
            sweeper.display(true); // Final board with
bombs shown
            exit(0);
        }
    }
    // Ah! All invisible cells are bombs, so you won!
    cout << "You won!!!!\n";
    sweeper.display(true); // Final board with bombs shown
}
```