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 one 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.
        - In case the phrase "depth first" seems unfamiliar, on binary trees the pre-order traversal would be a good example of "depth first" traversal. As you came to a "node", you examined that node and then dove down the left side before handling the right side.
        - I would recommend depth first rather than breadth first, 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.
- What should the items in the vector be? Anything that holds the necessary information: the coordinates of the tile. Could be a struct just for this purpose, or vector of two items, etc.
- How do you add something to the list? `push_back`
- How do you remove something?
    - With the vector, if you actually *want* the thing that you are removing, first access it with the method `back`.
    - Then use `pop_back`. to remove it. `pop_back` does not actually return anything, i.e., `pop_back`'s return type is `void` .
    - 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, but to saveyou time, I have provided a function for the UI below. You can certainly write your own, but please don't think about that till you have fully implemented the Minesweeper class.

Program Design

First, you will be implementing a class, `Minesweeper` to represent the game.

When designing a class, there are two major decisions you will need to make.

- What is the *public interface* of my class? That boils down to, what methods do you want to make public. Below we list the required *public* methods for the class. These methods allow us to provide you with the user interface mentioned above, and strike me as a reasonable set.
- What is the *internal representation* for my class? Clearly for Minesweeper, the question is how to represent the board. A two dimensional board is likely a vector%lt;vector%lt;something>>. So what will the something be? We'll discuss that below.

There Is also a question about how large the game board should be. Being lazy, I chose a fixed size of 10 x 10. Feel free to do the same. (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. Where would you put them? Fields in your class would be a good option. Marking them as `const` would also be good. Remember that constants are named with all uppercase, other than underscores to separate words.

So 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 tile, i.e. count of neighboring bombs, been "made visible" to the player yet?

You could, for example, make a tile struct. Or use some other encoding, also fine.

If you use a struct Tile, then your board might be defined as a `vector<vector<Tile>>`. And this vector would be a field in your class.

One last, I think, comment on the representation. If you found worrying about corners and edges troublesome in the Game of Life, you might consider putting a "border" of tiles around the actual board, that will have no bombs. These tiles would never be displayed or modified; they would only be there to simplify some of your code.

So, to help you along, I am specifying 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. It's main task to to fill the board with a reasonable number of bombs, and mark the other non-bomb tiles with a count of how many neighboring tiles hold bombs. See below for comments on how to do this "randomly".
- `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.

In my program I found it worked pretty well to assign a bomb to a square with a 10% probability. How would you do that?

- Call rand(), which will return an non-negative int.
- Mod that with 100 to get a value from 0 to 99, i.e. 100 possibilities. BTW, the modulo operator in C/C++ is the percent sign, so the expression: `n % 100` computes the desired value.
- All that's left is to compare that result with your probability value, which I am expressing as an int, not a double: `if ((rand() % 100) < BOMB_PROBABILITY) { ... }`, where BOMB_PROBABILITY is a constant integer, for example 10 representing 10%.

By default, every time you run your program you will see the *same sequence* of random numbers. That can be handy for testing a program.

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 *only once* in your program. A common mistake is to call it in a loop.

One common hack for setting a seed value is to use the call `srand(time(NULL))` to the seed the random number generator. 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: 09/20/2018

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
}
```