

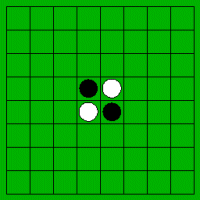
**Joel Youngberg Project 02**

**27 October 2013**

**Instr. Dr. Mark Lehr**

This program is an implementation of the game Othello or Reversi! It was quite an adventure putting this together. The program’s core is a dynamically allocated two-dimensional array of objects. The algorithm which I designed checks in all eight cardinal directions after a location on the two dimensional array is input. Each slot object in the 2D array has a numeric value member variable along with three additional string display variables. I ended up adding a fourth string variable later, which I will explain as we get to that part. The program initializes the object array values using a constructor, which was quite convenient! The member variables begin set to a value of zero while empty, and they display a string set of brackets for empty. Later as the game progresses the board fills with a capital letter “O” for black, and an asterisk for white (in Qt with the black background asterisks are white). This all works based on a display function which displays the respective string in the object based on the value stored in the numeric value member variable. (i. e. the brackets show when value is 0, white when value is 1, and black when value is 2.

The program begins by asking the user to input a single integer between 8-36 for the size of the board. I put the limits on the board size in order to prevent the board from being too small to get a decent game out of it and also to prevent the board from being so big that it would not display properly in my little black Qt window. Once input, the number is duplicated for the column value and a board is dynamically allocated as a square of the input number X the input number. This board is our awesome dynamically allocated 2D array of objects!

 After that the four center pieces of the board must be set to two black and two white as shown here: This was one of the first spots in the program design that I had to stop and think. Since the board is being dynamically allocated, I needed to determine where those four places would be so I could initialize them no matter what size was chosen for the board. After taking a break for a minute, I realized after I had drawn out an 8X8 sample array (which is my smallest size and the official size for the actual game), that the diagonal going through the center of the array was a pattern of duplicate ascending values.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 |
| 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
| 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 |
| 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 |
| 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 |
| 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 |
| 60 | 61 | 62 | 63 | 64 | 65 | 66 | 67 |
| 70 | 71 | 72 | 73 | 74 | 75 | 76 | 77 |

Because I saw this pattern, I was able to come up with the formula as follows:

Top left number = maximum board size (or input number) / 2 -1

This is true as long as the board size is an even number, which it must be for the game to work anyway. In order to get the other three spots, I simply added and subtracted the top left grid coordinate values to move to the corresponding board spot (i.e. to get to the left bottom simply add one to the row value : [3 +1] [3] = [4][3]. So I implemented this formula to assign the four game starting values.

Next, I wrote a function that checked in one direction, returning a true (or 1) if it was a valid move. Then if it was validated as a legal move, (in Othello/Reversi to place a piece you must cause a flip to happen or you have to pass the turn if you cannot make one) the board piece is changed to the color being placed, and the corresponding line (or single) piece are flipped to the opposite color.

After creating the two functions for checking and flipping, the second dependent upon the first, I called the functions for each direction and reversed their flipping properties for their respective colors.

The move is then validated for the respective color by checking to see if any of the functions successfully flipped (which means that they return a 1). If none did then the user is prompted that the move is not valid and that they must choose another or pass the turn. Upon testing, I discovered that the second condition was still returning 1 when the while loop just before it never occurred, so I added a local variable called check and iterated it as the loop went. I made it so that if the while loop does not execute at all (check > 0), then it will not return 1.

Upon testing play again, I realized that when I chose a number on the border of the 2D array, that the program crashed. Because it was only crashing on the outer edge, I immediately suspected an out of bounds error was occurring. Upon pondering over it a little more and taking a few more breaks… I realized that it was indeed an out of bounds array error caused by my checking algorithm. For example, my checker function checked for values that didn’t exist near any border as illustrated below – if a user selected value 3 0 for example, it went out of bounds to the left in its check….

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 |
|  | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
|  | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 |
|  | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 |
|  | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 |
|  | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 |
|  | 60 | 61 | 62 | 63 | 64 | 65 | 66 | 67 |
|  | 70 | 71 | 72 | 73 | 74 | 75 | 76 | 77 |

To solve this issue, I decided to create a fourth string for the object array. It was simply two spaces: “ “. Then I called it invisible. I then changed the board creation and initialization to *secretly* create a board with one more outer edge (by adding 2 to the input values), and initialized my new edge to invisible spaces. These are displayed as blank spaces at value 3.

I also raised the value of the slots able to be chosen on the displayed board so that I ended up with the following new grid:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |
|  | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |  |
|  | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 |  |
|  | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 |  |
|  | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 |  |
|  | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 |  |
|  | 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 |  |
|  | 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 |  |
|  | 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 |  |
|  |  |  |  |  |  |  |  |  |  |

Then selecting an edge value was not a problem anymore!

I finished off the fix by validating the grid coordinate choices to make sure they could not select the invisible spots (i. e. in this instance each choice must be between 1-8).

I then closed the program by summing the totals of black and white pieces and comparing them, then printing the results to the screen and to file, keeping running totals.

I added a template class per the requirements of the assignment to demonstrate use of templates, although the game does not really need a template. I also added a polymorphic calling of a member function called praiseWinner. The function gives a phrase of praise and bonus flip-points to any winner that is either 10 or 20 points over the other at the end of the game. When praise winner is called on class/object slot it gives one phrase and 50 bonus points. When the same method is called on class/object poly it gives a different phrase and 100 bonus points.

The winner is the color/player with the most pieces of their own color on the board when no more moves can be made. This was a great challenge and I ran into some interesting issues which took a bit of innovation to fix. I had a great time and the program runs great now!