IC210 Fall 2009
Programming Project 3
Console Collapse

Executive Summary
Who doesn’t like a good game? Using the skills you’ve learned to this point you are going to write a program to play a console version of a game you might have played on your journeys through the internet...

Same

http://www.springfrog.com/games/same/

Take a look at the included link or do a search for “same game” to get a feel for the general style of the game. A search for “collapse game” will get a slightly different flavor of the game you are going to create. As you “research” the game think about what tasks are being performed and how they can be broken up.

Due Dates and Honor
A friendly reminder: Submit all steps as they are completed to Blackboard!!

- **OPINFO Participants**: The milestone described below is due before you leave.
- **Friday November 20, 2009** (“by close of Business”): a mandatory interim milestone is due. This milestone consists of completing steps 1 and 2. Meeting this milestone is **worth 10 points of your project grade**. If you miss the deadline, you can still get 6 pts if submitted by COB on Mon Nov 23, or 4 pts if submitted by COB on Tues Nov 24. Here is what to submit:
  - Blackboard: submit your code under “Step 2”
  - Hardcopy: Turn in a copy of your code and a screenshot showing the board printed. Style/comments are encourage but not graded at this point.
- **Thursday December 3, 2009** (“by close of Business”): the final project is due. See the course policy for details about “close of business” and late penalties.

Again, this is a Programming Project, and it is very important that you understand and abide by the Department’s policy concerning programming projects. In particular, this means that you may **give no assistance** and may **receive no assistance** whatsoever from anyone except for your instructor. Please view:
http://www.usna.edu/cs/academics/ProgrammingPolicy.pdf

**Good News and …. more Good News?**
You have more time for this project. It also is more challenging – start now! Note especially that the project is due shortly after Thanksgiving – **don’t wait till the end!**
**Project Description**
The object of the game Same is to remove all the tiles from the board. There are several versions of the game. We'll use the following rules:

- The board will be an \(N\) rows by \(M\) columns array. It is initially filled with a random distribution of \(k\) “colors”.
- Only tiles that are orthogonally adjacent (directly above, below, left or right) to at least one other tile of the same “color” may be removed.
- When removing a tile all the orthogonally contiguous tiles of the same “color” will also be removed. This means if you remove a tile, any other tile that can be reached by moving only left, right, up, or down, all the while staying on the same color, will also be removed.
- The space left behind by any removed tiles will be filled by any tiles above the hole dropping down to fill the gap. Empty spaces at the top of the game board will remain empty.
- If an entire column of the game board is empty any columns to the right will move to the left to fill the space.
- The game is over if there are no more legal moves or all the tiles are removed.

Hopefully you are thinking that the game board is best represented by a two dimensional array and that functions will make it easier by breaking the game up into more manageable tasks. If you aren’t thinking that, start!

**Details**
The project is divided up into several steps which add additional functions to the game until you arrive at a complete game. This will allow us to break the seemingly complex game into more manageable tasks. In fact, most of your program will be implemented through a series of function calls, and \texttt{main()} will be quite small.

**No starter code** is needed or provided for this project. The key concepts you need to understand are 2D arrays and recursion, so you should review those first. Also, you will probably find ideas from the \texttt{TicTacToe lab} (Lab 10) very useful.

**Step 1: Choose Your Challenge**
The difficulty is determined by the number of rows and columns in the game board, and by the number of “colors”. The are five possible difficulty levels (1, 2, 3, 4, default) as follows:

<table>
<thead>
<tr>
<th>Level</th>
<th># of Rows</th>
<th># of Columns</th>
<th># of “Colors”</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 1</td>
<td>8</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>Level 2</td>
<td>10</td>
<td>14</td>
<td>6</td>
</tr>
<tr>
<td>Level 3</td>
<td>14</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>Level 4</td>
<td>28</td>
<td>26</td>
<td>10</td>
</tr>
<tr>
<td>Default</td>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

Write a function called \texttt{setDifficulty()} that displays a starting menu and asks the user to set his difficulty (1, 2, 3, or 4). If the user doesn’t choose one of the four set difficulties (for instance, by entering “0” or “7”), the function will select the “default” difficulty. The function should return a \texttt{struct} that holds the number of rows, columns, and colors. You must use a \texttt{struct} called “GameInfo” to hold this information!
Step 2: Build a Board
Write a function called `createBoard()` that creates and fills in the board. You should pass information about the number of rows, columns, and colors to this function so that it can do its job. The function should create a two dimensional character array for the board, based on the number of rows and columns provided. The number of “colors” will be used by the function to determine which symbols will be used to randomly fill in the board.

After creating the 2D array, you need to fill it in. To make things easier, it is possible to declare a “statically allocated array” to hold a selection of characters which will serve as “colors”. We haven’t covered this yet, but you just need to define it like this:

```
char tileArray[] = {2, 3, 5, 11, 14, 15, 21, 30, 35, 42, 43, 64};
```

The statement above will create a “static character array” of size 12 filled with the characters with those specific ASCII values. Don’t worry too much right now about how this array is being created. All you need to know is that once you include that statement in your code, below it you can access `tileArray` just like any other array. For instance, suppose the number of colors is 6. In that case, we want to randomly pick one of the first 6 entries in the array (we ignore the rest). You could randomly pick a number for the color, then convert it to a character for the board as follows:

```
int color = rand() % 6; // pick random number between 0 and 5
char ch = tileArray[color]; // pick character based on that number
```

When the function is done creating the board and filling it with random characters it should return a pointer to the board.

Now it doesn’t do much good to create the board if you can’t display it. Write a function called `displayBoard()` that draws the game board on the screen (see picture below). For an N by M game board your function should display the row number from N at the top to 1 at the bottom. Row numbers should be separated from the game board (try '|'). The column should be labeled with letters, beginning with ‘A’ and proceeding through to the Mth letter of the alphabet. Column labels should also be separated from the game board with a line of ‘-’ characters (see picture).

Finally, modify `main()` to call `setDifficulty()`, then use your new functions to create the board (based on values from `setDifficulty()` ) and then display it.
Step 3: It’s Your Move!
Now it’s time to start playing our game! After displaying the board we need to get a move from the player. Write a function, getMove() that prompts the user for a move in the form <char><int> (i.e. A2, D12, H3, ...). The function should check to ensure that the user’s input is somewhere on the board (not out of bounds) and continue to prompt until the user provides a move that is on the board. For example, if the board is a 4 X 4 game and the user inputs C6 the function should prompt for another input (since there is no 6th row). The function should then convert the input to the correct array indices and return them to main().

Once main() has a move from the user you should check to see if it was a legal move (a move that will actually remove pieces). Remember, in order for a move to be legal the chosen tile must be orthogonally adjacent to another tile of the same “color”. Write a function legalMove() that checks above, below, left and right one square (watch out for the edges!) and returns true if there is at least one matching adjacent tile and false otherwise.

Note: You may assume your user will always enter a valid input (i.e. will not enter lower case letters, will not enter a number followed by a character or two numbers, etc).

Hint: Remember C2 means the third column and the second row – but think carefully about what that means for your array indices.

Modify your main() function to repeatedly draw the board, get an input from the user and then display whether or not it is a legal move. For now there will be no way to exit this loop. We’ll fix this later.
Step 4: Get Those Tiles Out of Here!

Now that we know the user’s move is valid you need a function that will remove all the contiguous tiles of the same “color”. You need a removePiece() function that will take as arguments the board and the current position being removed. HINT: it is probably easier if you also pass (as an argument) the actual character value being removed!

This function is not as complicated at it may first appear. The task of removing 100 tiles is really just removing one tile and calling other functions to remove 99 tiles. Hmmm... sounds kind of like recursion! Another thing to think about is that you need to possibly move in four directions so maybe you shouldn’t limit yourself to just one function call inside removePiece()...

Your main() function needs to be changed to include the removePiece() function in the loop from Step 3. Now your program should get a move from the user, check that it’s a legal move, remove the appropriate pieces, display the new board and loop back up to repeat the process. Note that in this step the loop is still an infinite loop. There is no way to end the game. Yet...

HINT: See in-class recursion project hint for ideas on how to implement this step!

One more thing for this step: you need a way to clear the screen after each move so that you can start with a blank slate for displaying your board. The statement system("cls") will clear the screen and place the write cursor back at the upper left corner of the screen. You will have to include the cstdlib library in order to use the system() function.
Step 5: Watch for Falling Pieces
So you've removed all the pieces but the ones above the new spaces are just hanging there. They need to drop down. After all, the game is called Console Collapse. Write a function called `collapse()` that looks at each column of the game board and moves any “floating” tiles down into any spaces left after the last call to `removePiece()`. When this function is done running, any spaces on the game board should all at the very top of the board.

Add the `collapse()` function to your game loop so that after each call to `removePiece()` the board collapses before asking the user for his next move.

ALSO: modify the `legalMove()` function created in step 3 such that it will return false if the selected (row,col) of the user’s move is a space (i.e. the piece has already been removed from the board).
Step 6: Everyone Squeeze Together
After several moves you may find that you’ve managed to clear an entire column. But
the problem is that you can’t clear the board if all the tiles on the board aren’t all
connected. This means that any empty columns need to be compressed so that
separate regions of tiles are merged into one. You need to write a function called
compress() which will move any columns of tiles that are to the right of an empty
column to the left. In other words after a call to compress() all the columns that still
contain tiles will be on the left side of the game board and any empty columns are on the
right side of the game board.

Add the compress() function to the loop in main() so that the board is compressed
after the call to collapse().
Step 7: It’s Over Already?
Up to this point we’ve been ignoring the fact that there is no way to end the game and we were stuck in an infinite loop. We should fix that. Write a function called anyMoveRemaining() that takes the game board as input and looks through the board to see if there are any legal moves remaining. The function should return true if there is at least one legal move left and false if there are no moves left.

Now that we have a function that tells us if there is a move left or not we should consider using it to control the game loop. Modify main() to incorporate anyMoveRemaining() to cause the loop to exit when there are no more moves left.

When the game is over main() should print the game board one last time and tell the user the game is over.
Step 8 Extra Credit: Replay (maximum 2 pts.)
A nice touch is to ask if the user wants to play again. If the user says yes the game should start all over again by calling the `setDifficulty()` function and creating a new board.

Step 9 Extra Credit: Scores (maximum 5 pts.)
Add a scoring feature to your `removePiece()` function. The value for each tile will increase by a power of 2 for each level away from the original tile. The first tile will be worth $2^0$ or 1 point. The tiles immediately adjacent to the original tile are worth $2^1$ or 2 points, from there points increase to 4, 8, 16, ... for each level. The value of each tile removed should be totaled up and returned to `main()` which will then display the score below the board after each move. See below for an example of the scoring method. If the original tile is in the third row down and fourth column over the scoring for matching tiles might look like this:

```
+---+---+---+---+
|   |   |   |   |
+---+---+---+---+
| 16| 4 | 2 | 4 |
+---+---+---+---+
| 8 | 4 | 2 | 1 |
+---+---+---+---+
| 8 | 2 | 4 |
+---+---+---+---+
|32|16|4|8 |
+---+---+---+---+
|32|
+---+---+---+---+
```

Total score: 151

Step 10 Extra Credit: Animation (maximum 6 pts)
Change your game such that the tiles selected by the user are removed, then the remaining tiles slowly drop into their new positions, and the empty columns are slowly compressed. **WARNING: this is very difficult** – partial credit for something that achieves some of the desired effect.

**Important grading points:**
- The default method of passing variables is pass by value.
- Appropriate use of functions to implement this program is critical to receiving the maximum possible score.
- **If your program does not compile as submitted**, you will receive a zero.
- Your program must read input and write output in the exact format specified in this project description.
- Your program’s source code must comply with the Required Style Guide in order to maximize the grade you receive on this project.
- Your grade will also depend on the reasonable use of C++ code. Don’t use 50 lines of code where 5 would do.
There will be both a paper and an electronic part to your submissions. The paper submission can be handed to your instructor or slid under their office door (but do not put your project in their mailbox). For the purposes of any late penalties, your project is not considered submitted until your instructor receives BOTH the electronic and paper portions of your submission. Any disagreement between your paper and electronic submissions will result in a grade reduction.

Electronic submission: Unless otherwise specified by your instructor, your electronic submission should be made via Blackboard. As you complete each step you should submit your project3.cpp file. When completely finished, submit your project3.cpp file under the “Final submission” link on Blackboard.

If you desire to submit a revised solution to any Blackboard-submitted step notify your instructor via email so they can clear your previous submission. However, unless changes are dramatic this is not required for intermediate steps – the final submission is what is graded.

Milestone submission: On-time milestone submission is worth 10 pts, included with the Step 2 grading below. See the first page for the earlier deadline and details on this.

Step 1 submission: (max 10 points total)

Paper: Submit a printout of your source code and a screen capture showing your program being run. You will also submit a signed copy of this cover sheet.
Electronic: The .cpp file containing the source code for your solution.

Step 2 submission: (max 40 points total)

Paper: Submit a printout of your source code and a screen capture showing your program being run and displaying your board. You will also submit a signed copy of this cover sheet.
Electronic: The .cpp file containing the source code for your solution.

Step 3 submission: (max 55 points total)

Paper: Submit a printout of your source code and a screen capture showing your program being run and properly handling all the various categories of user input. You will also submit a signed copy of this cover sheet.
Electronic: The .cpp file containing the source code for your solution.

Step 4 submission: (max 75 points total)

Paper: Submit a printout of your source code and a screen capture showing your program being run. Provide a couple of screen shots of your program before and after the call to removePiece(). You will also submit a signed copy of this cover sheet.
Electronic: The .cpp file containing the source code for your solution.
Step 5 submission: (max 85 points total)

**Paper:** Submit a printout of your source code and a screen capture showing your program being run. Provide a couple of screen shots of your program before and after each move. You will also submit a signed copy of this cover sheet.

**Electronic:** The .cpp file containing the source code for your solution.

Step 6 submission: (max 95 points total)

**Paper:** Submit a printout of your source code and a screen capture showing your program being run. Provide a couple of screen shots of your program before and after each move. You will also submit a signed copy of this cover sheet.

**Electronic:** The .cpp file containing the source code for your solution.

Step 7 submission: (max 100 points total)

**Paper:** Submit a printout of your source code and a screen capture showing your program being run. Provide a screen shot of your program at the conclusion of a game when there are no more moves left. You will also submit a signed copy of this cover sheet.

**Electronic:** The .cpp file containing the source code for your solution.

**TIPS:**

- Follow the Style Guide as you go! It’s easier to indent and comment as you write than at the end, and you’ll make it easier on yourself to write your program. And you’ll maximize your grade by following the guidelines.
- Compile often so you don’t end up with a snake’s nest of errors all at once
- Save a backup copy of your program’s source code in case your sponsor’s dog eats your X drive.
- **Start early!** Allow time for you to get new ideas and to see the instructor for help.
- Remember that this project is not to be discussed with anyone besides your instructor! Don’t defeat your learning AND jeopardize the honor of yourself or your friends by talking to them about it.