IC210 Fall 2013
Programming Project 3: Console Collapse

Executive Summary
You will create a console version of a game you might have played on your journeys through the internet. Your skills learned thus far will serve you well.

Online Examples
http://www.oopixel.com/games/samegame/
http://www.mah-jongg.ch/samegame/samegame.html
Try the game at the links to get a feel for the general style of the game. Search for “collapse game” to get a slightly different flavor of the game. As you “research” the game, think about what tasks are being performed and how they can be broken up.

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

- Friday November 15 (COB): a mandatory milestone is due. You must complete 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 18, or 4 pts if submitted by COB on Tues Nov 19. 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 encouraged but not graded at this point.
- Monday, November 25 (COB): the final project is due. See the course policy for details about “close of business” and late penalties.
- OPINFO participants: Required – submit electronically by deadline (paper submission can be made when you return). Recommended – complete & submit before you leave!

Honor
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/resources/ProgrammingPolicy.pdf

Good News?
You have more time for this project…but it is also more challenging. Start now! Don’t wait!

Confused on what the program should do? See the calendar for a “sample executable” to try it out.
Project Description

The object of this game is to remove all the tiles from the board. There are several versions of this game out there, but we’ll use the following rules:

- The board will be a collection of tiles with N rows by M columns. It is initially filled with a random distribution of k “colors”. We will use different ASCII characters to represent each “color.”
- 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 space 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 2D array and that functions will make it easier by breaking the game up into more manageable tasks. If you aren’t thinking that, start!

Code Implementation Details

Your code will involve multiple steps (to be submitted incrementally to Blackboard), and each step includes new functions. Your main() will be quite small in the end. No starter code is provided, but you might find useful ideas in the TicTacToe lab (Lab 11).

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. This includes having good comments!
- Your grade will also depend on the reasonable use of C++ code. Don’t use 50 lines of code where 5 would do.
- Do not use any “global variables” – all variables should be defined in main() or another function.

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 backup copies of your program in case your sponsor’s dog eats your hard drive.
- Start early! Allow time for you to get new ideas and to see the instructor for help.
- DEBUGGING: A “segmentation fault” probably means you accessed an invalid array or array location. Use “cout” statements to help identify how far your code makes it before crashing. Note: in these cases, include a “endl” (newline) with each cout statement -- otherwise the output may not appear on the screen before your program crashes.
  - And/or see online notes about how the use the “gdb” debugger to locate crashes.
- 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.
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”. There are four possible difficulty levels (1, 2, 3, 4) 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>3</td>
</tr>
<tr>
<td>Level 2</td>
<td>10</td>
<td>14</td>
<td>4</td>
</tr>
<tr>
<td>Level 3</td>
<td>14</td>
<td>16</td>
<td>6</td>
</tr>
<tr>
<td>Level 4</td>
<td>28</td>
<td>26</td>
<td>8</td>
</tr>
</tbody>
</table>

Write a function called setDifficulty() that displays a starting menu and asks the user to set his difficulty (1-4). If the user doesn’t choose one of the four set difficulties (for instance, by entering “0” or “7”), the function will select level 1 by default. The function should print out the level that was selected, then return a struct that holds the appropriate number of rows, columns, and colors. **You must use a struct called “GameBoard” to hold this information!**

In main(), call the setDifficulty() function, then (back in main()) print out the number of rows, columns, and colors. Make your output look just like the example below (except insert your name AND alpha on the first line).

Step 2: Build a Board
Now that you have a struct GameBoard, you need to represent the board itself inside your struct. Add a member to your struct for this (use a 2D array of characters). You will now write a function to initialize this array.

Write a function called createBoard() that creates a 2D array of characters, fills in the 2D array with random “colors”, and returns a pointer to the array. You should pass information about the number of rows, columns, and colors to this function so that it can do its job. The size of the array is 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 with particular values. 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:

```c
char tileArray[] = {'o', 'x', '*', '#', '=', 'l', '-', '+'};
```
The statement above will create a “static character array” of size 8 filled with the listed characters. 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 (inside the createBoard() function), 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:

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

Include the <stdlib> library in order to use the rand() function. When the function is finished creating the board and filling it with random characters, it should return a pointer to the board. Your main() function should then store the pointer inside your GameBoard struct.

Now it doesn’t do much good to create the board if you can’t display it. Write a function called displayBoard() that takes your GameBoard struct as a parameter and draws the board on the screen (see picture below). For an N by M game board your function will display the row number from N at the top to 1 at the bottom. Row numbers will be separated from the game board with the ‘|’ character. The columns should be labeled with letters, beginning with ‘A’. Column labels should be separated from the game board with a line of ‘-’ characters. You must match the screenshot exactly. Exception: depending on how you create the array, the specific “colors” you see in each location may be different than the provided screenshots -- that is fine.

Finally, modify main() to call setDifficulty(), then createBoard(), and call displayBoard() to display it. See the screenshot below.
Step 3: It’s Your Move!
Now it’s time to start playing! 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, ...). You may assume that the user will enter sensible input (one uppercase letter and then one number).
- Checks that the input is somewhere on the board (not out of bounds). If not, continue to prompt until the user provides a move that is on the board. For example, if the board is 4x4 and the user inputs C6, the function should prompt for another input.
- Once you have an input that is somewhere on the board, the function should convert the input to the appropriate array indices and return them to `main()`. Note that ‘C2’ is the third column and second row – think carefully about what that means for your array indices!!

Once `main()` has a move from the user (using `getMove()` ) you should check to see if it was a legal move (a move that will remove pieces). 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 follows these rules:

1. Returns false if the selected tile is a blank character (a space, from removing tiles)
2. Returns true if there is at least one matching adjacent tile either above, below, left or right one square (watch out for the edges!)
3. Returns false otherwise.

You decide what argument s the `legalMove()` function needs in order to do its job.

Modify your `main()` function to draw the board, the repeatedly (a) get an input from the user and (b) display whether or not it is a legal move (using your new functions). For now there will be no way to exit this loop.

Below is some sample input. Note that “A0” and “J4” are rejected (silently, by `getMove()` ) because they are “off the board.”
Step 4: Get Those Tiles Out of Here!
We now have a valid move from the user. Now write a function to remove the chosen tile, as well as contiguous tiles of the same “color”. A tile is “removed” by setting the array location to a space.

Write a function called removePiece() that takes as arguments the board and the current position being removed, then updates the board to remove the appropriate tiles. **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. Hmm…**sounds 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, and check that it’s a legal move. If it is a legal move, remove the appropriate tiles, 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 “lave flow” recursion project hint for ideas on how to implement this step! If you haven’t seen this yet, look on the calendar for this example.

**Clear screen:** One more thing: you need to do is clear the screen after each move so that you can start with a blank slate for displaying your board. The statement `system("clear");` will clear the screen and place the write cursor back at the upper left corner of the screen. Include the `cstdlib` library in order to use the `system()` function.
STOP: Make sure you have thoroughly, exhaustively, really tested step 4 before continuing!

---

**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. 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 finished running, any spaces on the game board should be 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/her next move.

![Game Board Before](image1.png)

![Game Board After](image2.png)

**HINT**: There are many possible ways to do this step (and the next step). Here’s one idea: at first, don’t try to write a function that completely solves the problem. Instead, write a function that looks for tiles that should fall down (e.g., any tile that has a space underneath it). When you find such tiles, go ahead and move them down without worrying about the impact on other tiles, or whether that tile needs to fall further than one space. Now, you have a function that moves some tiles closer to where they belong -- and you can just call this function repeatedly until all the tiles are in the right place. You’ll have to think about this: how do you know when you are finished?

You do NOT have to use this hint, but it is an idea that may be helpful.

It is NOT required that your solution be super-efficient, but it should have run fast enough so that you can play the game without much delay.
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 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 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 removePiece(). 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, ... The sum of all removed tiles should be returned to main(). Then display the score below the board after each move. See below for an example. If the original tile is in the third row down and fourth column over the scoring for matching tiles might look like this:

<table>
<thead>
<tr>
<th></th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>32</td>
<td>16</td>
</tr>
<tr>
<td>32</td>
<td></td>
</tr>
</tbody>
</table>

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 could be very difficult** – partial credit for something that achieves some of the desired effect.

**Grading**

See “Important grading points” on page 2.

**Submission**

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.

**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.

**Final electronic submission:**

1. Submit your `project3.cpp` code for the furthest step you completed electronically under “Final submission” on Blackboard.

**Final paper submission:**

1. Follow the instructions for the particular step you completed:

   **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 the project coversheet.

   **Step 2 submission: (max 35 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 the project coversheet.

   **Step 3 submission: (max 50 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 the project coversheet.

   **Step 4 submission: (max 70 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 the project coversheet.

   **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 the project coversheet.

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 the project coversheet.

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 the project coversheet.

TIPS AGAIN:

- 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 backup copies of your program in case your sponsor’s dog eats your hard drive.
- Start early! Allow time for you to get new ideas and to see the instructor for help.
- DEBUGGING: A “segmentation fault” probably means you accessed an invalid array or array location. Use “cout” statements to help identify how far your code makes it before crashing. Note: in these cases, include a “endl” (newline) with each cout statement -- otherwise the output may not appear on the screen before your program crashes.
  - And/or see online notes about how the use the “gdb” debugger to locate crashes.
- 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.