Replit.com Link: https://replit.com/@RacheySnachey/c11hwstartjigsaw#README.md

Overview

Implement a Puzzle class constructor and its methods so that you can read a data file that describes a randomized collection of puzzle pieces – one piece per line of the file - and then print the "ASCII ART" image it portrays.
Each piece of the puzzle will be described by a single integer (which represents an ascii character value), along with its row and column integer location within the puzzle, counting from zero.
Your goal is to read in all puzzle pieces first, storing them in a dynamic array of pointers. Then you will assign each of their pointers to their correct location in a dynamic 2D array (a grid) of pointers. You will print out the puzzle grid to reveal the picture.
You must accomplish these goals by using dynamic memory allocation techniques to refer to each piece through a pointer to instance of a Token object (where each Token represents a piece of the puzzle), and by using the classes and methods provided in the puzzle start repl at the linked below.
This assignment will exercise our new-found understanding of dynamic memory management. All array members of the Puzzle class that hold data for the Puzzle class will need to be dynamically allocated AND deleted via a Puzzle destructor. No containers will be used.

Files

The repl provided will include the following files which you will need:

- puzzle_input_basic.txt - input data file if you are NOT attempting the bonus
- Puzzle.cpp - your implementation for the Puzzle class
- Puzzle.h - declarations for the Puzzle class (provided for you)
- Token.cpp - your implementation of the game Token class (provided for you)
- Token.h - declarations for the Token class (provided for you)
- puzzle_input_bonus.txt - input data file if you ARE attempting the bonus
- Token_bonus.h - a hint file if you are pursuing the bonus
- art_credits.txt - credit for artists who created these pictures

Sample Output

Your final puzzle grid could look something like this:
(Art by Shanaka Dias)

```
               ,,,,
           ,;) .';;;;',
 ;;,,_,-.-.,;;'_,|I\;;;/),,_
  `';;/:|:);{ ;;;|| \;/ /;;;\__
      L;/-';/ \;;\',/;\/;;;.') \
      .:`''` - \;;'.__/;;;/  . _'-._
    .'/   \     \;;;;;;/.'_7:.  '). \_
  ."/     | '._ );}{;//.'    '-:  '.,L
.'. /       \  ( |;;;/_/        \._./;\   _,
 . /        |\ ( /;;/_/            ';;;\,;;_,
. /         )__(/;;/_/              (;;''''
 /        _;:':;;;;:';-._            );
/        /   \  `'`   --.'-._        \/
       .'     '.  ,'        '-,
      /    /   r--,..__       '.\
    .'    '  .'       '--._     ]
    (     :.(;>        _ .' '- ;/
    |      /:;(    ,_.';(   __.'
     '- -'"|;:/    (;;;;-'--'
           |;/      ;;(
 snd       "      /;;|
                 \;;|
                  \/
```

The Input Data File

- An input text file called “puzzle_input_basic.txt” that starts with a single integer that represents the **pieceCount** (total number of pieces that will follow), followed by **pieceCount** lines, each which represent a single puzzle piece. For example, the first three lines could look like this below, and describe a puzzle with 1218 pieces, followed by the first 2 pieces of the puzzle. The first piece is represented by the ascii character 32 (a space), which will belong at row 21 and column 1 in the puzzle grid. The second piece is represented by the ascii character 46 (a period), which will go in row, column = (5,30) of the puzzle grid. Remember that row & column numbers start at zero.

```
1218

32 21  1

46 5  30
```

The input file will start with a single number that identifies how many pieces are in the puzzle (and how many lines of input will follow). We’ll call this value the **pieceCount** for the puzzle. You will not be told the row or column dimensions of the puzzle You will need to determine the puzzle’s row and column dimensions when you read the file, by monitoring the largest row and column values you encounter in the input file for any particular piece. Warning: the number of rows

is 1 larger than the largest row number found in the file, since row numbers start with zero. Same for columns.
Each piece of the puzzle will be described by a line in the input data file (following the initial **pieceCount** on the 1st line of the file). Each line puzzle piece line in the file will have 3 integers:

- An integer which represents the ascii code of a single character. We will refer to this character as a "letter" even though it really could be any ascii character. (you will need to convert this integer type value to a char-type character after you read it).
- An integer which represents the row number of the puzzle grid at which this letter must be placed. Rows start at zero.
- An integer which represents the column number of the puzzle grid at which this letter must be placed. Columns start at zero.

For example:
```
32 21  1
```
would be a puzzle piece that uses a character with ASCII code 32, a should be placed at row 21 and column 1 of the grid member of the puzzle (counting from zero)

Token Class

- A Token class has been provided for you to use to store each piece of the puzzle. (see Token.cpp and Token.h)
- The token will provide a char-type data member called **letter** which will hold the ascii value of a single puzzle piece, along with an integer **row** and **column** location that the puzzle will eventually be located in the puzzle grid. (Note that the Token constructor will automatically convert from the integer representation of a character into the native char-type of the **letter** data member when assigned.
- The Token class will also provide a **printMe()** function which will print out the description of a single puzzle piece. This method has also been provided for you and you will need to use it for the work described below (hint).

Puzzle Class

- Holds the entire puzzle (see Puzzle.h, you will need to implement Puzzle.cpp completely)
- See Puzzle.h for a description of each data member and method

Implementation Steps and Scoring

In Puzzle.cpp: Implement the Puzzle class constructor that:

- **5 points**: Opens a file using the **fileName** string passed to the constructor, and reads the **puzzleCount** value from the first line of the file, and then dynamically allocates space for the **pieces** array, to hold enough POINTERS to for every piece of the puzzle. The POINTERS will be of type Token. Remember: The **pieces** data member is an array of POINTERS, not an array of instances!
- **5 Points**: From within a loop in the constructor, reads 3 integers for from the file for each puzzle piece, storing the integers locally as a **letter**, **row**, and **column**;

- **5 Points**: From within the above loop, for each 3 integers read, dynamically creates a Token instance, by using “new”. Each Token instance will hold the ascii character that describes a single piece (**letter**), and the **row** & **column** integer that describes the final location of that puzzle piece in a grid. Store a POINTER to each new Token instance (as returned by new) in the dynamically allocated **pieces** array described above. When your loop completes, the **Puzzle::pieces** data member will hold “puzzle Pieces” pointers, which are each pointing to a dynamically allocated Token instance. This represents your “box of random puzzle pieces”.

Note: You will not be initializing the the 2D array called grid in the Puzzle constructor. This will be done in the loadGrid() method described below.

- **15 Points**: In Puzzle.cpp, in the above Puzzle constructor, while you are loading pieces, keep track of the highest row and column number that you have encountered. Set the Puzzle::**rowCount** and Puzzle::**colCount** data members based on these max values. WARNING The largest value you find for a row and column in the input file will be 1 less than the total number of rows or columns in the puzzle. If you set rowCount and colCount values to the largest row or column encountered, your program will eventually crash because you will not have enough rows and columns in your dynamically allocated arrays to hold the entire puzzle. Be sure you set **rowCount** and **colCount** accordingly, to avoid data corruption later in your project. Also be sure to initialize the Puzzle::**pieceCount** integer number of total pieces in the puzzle, based on the first line of text in the file.
- **10 Points**: From main(), Create a POINTER to an instance of your puzzle called “myPuzzle_ptr” by using the **new** keyword, and by passing your Puzzle constructor the string name puzzle_input_basic.txt. Your Puzzle constructor should open the file and load and initialize the pieces array and data member variables **rowCount**, **colCount**, and **pieceCount**, as described above. Please Note: your constructor must open a file based on the name string passed to the constructor, not a hard-wired string that you use from within your constructor. Also, your Puzzle constructor will NOT allocate or fill the grid member... that will be done by the loadGrid() method below
- **10 Point**s, In Puzzle.ccp, implement the **Puzzle::printPieces()** method of the Puzzle class, which should print out a description of each Token instance pointed to by the pieces array. Your output should look something like this below (note that you will not be able to see all the pieces printed in your console due to console buffer limitations in repl):

```
Printing letter # 0:  Puzzle piece '_' is located at at row, col = (1, 17)

Printing letter # 1:  Puzzle piece 'L' is located at at row, col = (10, 22)

Printing letter # 2:  Puzzle piece ' ' is located at at row, col = (25, 41)

…
```

- **5 Points**: From main.cpp, print out all of the puzzle pieces from your myPuzzle_ptr instance by using the **printPieces()** method implemented above. Remember that myPuzzle_ptr is a POINTER and so you will need to call the printPieces method by using the arrow "->" syntax after the myPuzzle_ptr.

In Puzzle.cpp, Implement the **Puzzle::loadGrid()** method.

- **5 points**: First, Initialize the **grid** member of the Puzzle class to have enough empty rows and columns to store POINTERS to all the puzzle pieces. You will already know the **rowCount** and **colCount** values based on Puzzle constructor. So you will need to create **rowCount** rows of pointers to pointers to Tokens, and in each row, you will want to create **colCount** null Token pointers. Note: the grid is a grid of pointers to Tokens, not a grid of Token instances or characters. You will not yet know the pointers to the specific Token instances, so this step is all about allocating the grid, and then filling out the grid with null pointers that we will overwrite in the next step below.
- **5 points**: For each previously loaded Token instance POINTER in the **pieces** array, assign that POINTER to its correct location in the grid of pointers. That is, each puzzle piece POINTER will need to be placed at grid[row] where row is the row member of the token and col is the col member of the Token. The value at grid[row] will be a pointer to the Token that comes from the **pieces** array.
- **5 Points**: From main(): call your **loadGrid()** method from main to build and initialize the Puzzle **grid**, based on the already constructed **pieces** array you created above. Remember that myPuzzle_ptr is a POINTER and so you will need to call the loadGrid method by using the arrow "->" syntax after the myPuzzle_ptr.
- **10 Points**: In Puzzle.cpp, Implement the **Puzzle::printGrid()** method. Iterate through the already initialized grid 2D array, to print the final puzzle one row at a time (with each row of the puzzle starting on a new line of output). Remember that the grid holds POINTERS to Token instances, so you will access grid members using a syntax like grid[r]->letter, where the arrow "->" is needed because the value at location [r] is a POINTER, and the letter is a data member belonging to the Token instance. Note also that because Token designates Puzzle as a friend, we are able to access the private letter data member directly in this way (otherwise we would have an error)
- **5 Points**: From main(); Call your **printGrid()** method from main to verify that your puzzle looks good! Remember that myPuzzle_ptr is a POINTER and so you will need to call the printGrid method by using the arrow "->" syntax after the myPuzzle_ptr.
- **10 Points**: In Puzzle.cpp, implement your **Puzzle::~Puzzle()** destructor to delete every Token instance that is pointed to by the **pieces** array. Then, delete the dynamically created **pieces** array itself, which you created in your constructor. You will use the **delete** keyword for both of these tasks, but recall that you will need to also use the [] brackets syntax with delete when you delete the pieces array (since that is presumably how you created it!) Lastly, you must delete the 2D array of pointers to Token pieces that you created in your Puzzle::loadGrid() method. For every "new", there must be a delete. This means that for each row of pointers you allocated using new,you will need to delete that row. Be careful not to delete any individual Token instance twice, via different pointers to the same memory location, as that will crash you program.
- **5 Points**: From main(), delete the Puzzle instance by calling **delete** on the POINTER to the puzzle instance you created at the start of your main(). Remember that for every new, we use a corresponding delete, and that we are "feeding" the same pointer back to delete that we were given by new. This still will automatically call the Puzzle destructor you

implemented above. Do not attempt to call the Puzzle destructor directly in main by using the tilde symbol.

Bonus (15 points)

Develop a Puzzle.cpp and Main.cpp which will read the file puzzle_input_bonus.txt which is provided with the starter repl link, instead of the puzzle_input_basic.txt file.

- This separate bonus file describes a puzzle, with one piece per line. However, there is no piecesCount integer at the start of the file, and the individual piece descriptions on each line of the file do not include a row or column designation of their ultimate location in the grid. Instead, piece descriptions include the following information comprised by 6 integers:
- An integer of the ascii code that represents the puzzle piece letter. This integer can be cast to a char to achieve the final appearance of the puzzle piece, exactly as was done in the basic part of this assignment.
- An integer which represents a unique id for this specific puzzle piece. No two pieces in the puzzle will have the same integer id.
- An integer which represents the id of the puzzle piece directly above this piece, or zero if this puzzle piece is along the top edge of the puzzle.
- An integer which represents the id of the puzzle piece directly to the right of this piece, or zero if this puzzle piece is along the right edge of the puzzle.
- An integer which represents the id of the puzzle piece directly below this piece, or zero if this puzzle piece is along the bottom edge of the puzzle.
- An integer which represents the id of the puzzle piece directly to the left of this piece, or zero if this puzzle piece is along the bottom edge of the puzzle.

Your task is to load all puzzle pieces into the pieces array as done in the basic part of this assignment in your Puzzle constructor (ignoring row & column data members in the Token class), and thereafter, in your loadGrid() routine, to assign each piece to its correct location on the grid in any manner you choose, using methods or modifications to the Token.h and Puzzle.h files that you required. You may choose to start from the Token_bonus.h and Token_bonus.cpp files provided. (please submit it as Token.h and Token.cpp if you do). You may not using containers for the bonus.
For the bonus,
Note that your grade for the bonus will BE your grade for the entire assignment. (So you do NOT need to submit solutions for the basic part of the assignment if you decide to pursue the bonus.) Hopefully this saves you time, but it does pose a risk that you will be graded entirely based on your ability to complete the bonus.
Bonus Hint: You are allowed to use containers to help you achieve the bonus, but you must still use dynamic allocation for the final **pieces** array and **grid** member of the Puzzle class as described above.

Constraints

- You may not read the input file more than one time!

- You must initialize the grid based on the Dynamic Array of Puzzle Piece Pointers (not directly from reading the input data file)
- You may not use any containers for this assignment
- There should only be one instance of each puzzle piece (of each Token), where each Token instance will be pointed to exactly and only twice: once by the pieces array, and once by the grid.

Submission

- If you are not doing the bonus, submit your repl.it link via canvas, or at least your Puzzle.cpp file and your main.cpp file and (if not using rep).
- If you ARE doing the bonus, please submit to canvas your modified Puzzle.cpp, Main.cpp, and your .h files. Also please submit your repl link, (but only if you are actively using repl.)

May the force be with you! ;)

Concepts Covered in this Assignment

- Pointers
- Dynamic Allocation and Destruction of memory from the heap (class instances)
- ASCII file input
- Arrays of pointers to arrays of (anything)
- Classes (advanced): Multiple Constructors, Methods, and Destructors
- Class prototypes
- .h Files
- Include Guards
- Containers (Bonus only)