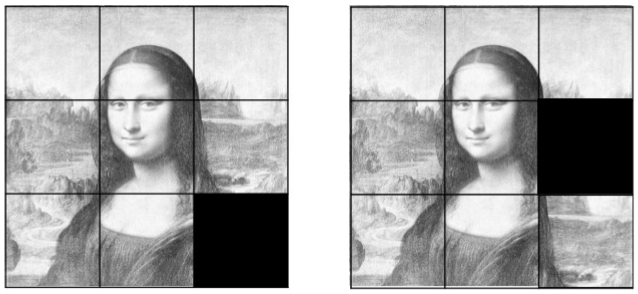
University of Cape Town
Department of Computer Science

CSC3022F
Assignment 2B - Image-based Slide Puzzle Game

March 2025

You are given an image and asked to turn this into the beginnings of a slide puzzle game — these are 2D tile-based board games where you slide tiles left, right, up and down leaving behind a space. After several moves, the game board appears as a collection of randomized image tiles. To play the game one would then have to figure out how to shift them around again to rebuild the input image. The image below shows an initial tile image state, and then one move later in which a tile at the right edge was moved into the empty space below. Note that the black lines on the image are for illustration only.



In this assignment we're interested in generating candidate play states (represented as a sequence of images) for others to solve.

Note: solving the puzzle (returning to the initial state) given such a permuted image state is outside the scope of the assignment — but something you can definitely tackle in your spare time.

You will need to load in an image — stored as a dynamically allocated array — and then carve it up into appropriate image tiles and use these image sub-arrays to generate new play states (represented as 'tiled' images) after a given number of single tile moves. Each move should be randomly chosen, but must be valid — we can only move a tile into the currently empty slot if it is adjacent to the empty slot, and in moving it, the empty slot moves to the tile's previous location (a **swap** basically). After each move, you must generate a new *tiled image* which shows the current state of the 'board'. This will differ from the previous image by only one "tile" move.

Your program must accept an input image (in PGM image format) as well as an odd number (3,5,7 etc) which will be used to subdivide the image into tiles. For example, if the number 3 was specified, you would divide the input image into a 3 x 3 grid of sub-images.

In addition to these command line parameters, you will also add an option to specify how many moves to compute, as well an option to specify the output image name (the 'base name'). For example, if the base name is "myimage" you would produce a sequence of PGM images (one per move), each of which shows the currents state of the tile board, named as "myimage-1.pgm", "myimage-2.pgm" and so on. You can write out the initial board state – before any moves – as "myimage-0.pgm".

1 Core requirements — 85%

The requirements for this assignment are as follows:

1. Provide a command line interface which supports the following parameters (<x> shows the type of argument expected):

   -s <int>, integer (odd number) size of tile grid (same in both x and y)

   -o <string>, base name be used for output pgm board images

   -n <int>, number of moves to produce

   Your program would be invoked as: `programName <options> inputImageName`

2. The initial 'empty' tile should always be placed in the bottom right corner for all starting tile grids.

3. When computing the next move, ensure that this is a valid transformation of the previous grid state into the next.

4. After a new move has been computed, you must write out a new image which shows the new state of the tile board. The image sequence number should be increased by 1 and the sequence number must appear in the output image name.

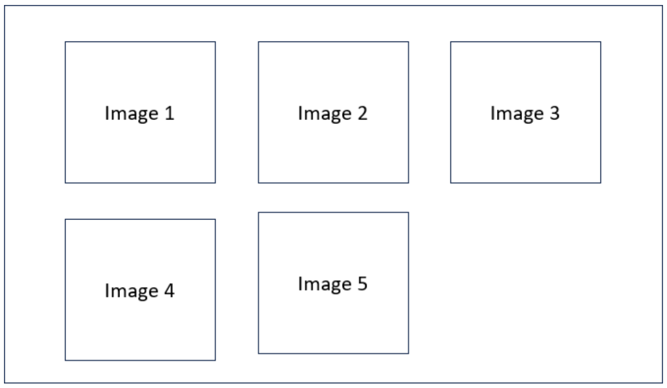
5. You must create a single class, *TileManager*, which encapsulates all the information for the tile board. You must also have a *Tile* class to encapsulate individual tile state and image data. Each tile must contain at least a *dynamically allocated array of unsigned char values*. This will store the grey-scale pixel intensities (0..255) for each tile image. Both these classes should have correctly formed constructors, and their destructors should ensure that all dynamically allocated memory is recovered when they go out of scope. They can be implemented in one common implementation file, and share the same header file, since they are logically tightly coupled. You may use any suitable data structure to represent the complete tile board, including a vectors of vectors. Important: the Tile objects **store** the actual image pixel data for each rectangular sub-array of the initial board layout; you must determine where these tile arrays need to be copied to in the output board image you build after each new move.

6. You may not use smart pointers of any kind — manage all dynamic memory yourself though explicit calls to new and delete. This will ensure that you understand how these work.

Completing the above core work will enable you to score up to 85%. To achieve a higher mark, you should tackle the mastery work outlined below.

2 Mastery work — 15%

Create a single large "summary image" which shows the board state after each move. Figure out a good heuristic to lay out the game state images so this summary image is as square as possible. The game state images should be ordered as Image 1 ... Image N, from top to bottom, left to right. The image below might be one way to lay out 5 game state sub-images within the larger summary image —– each 'box' would be the appropriate image for that tile arrangement:



You can embed the state images on a white background and will need to space them out appropriately.

1. Add a new flag, -x <string>, which will specify the output name for an additional PGM image which will contain all the sub-images created as part of your N moves.
2. The sequence of board sub-images must be arranged into a single larger image, with a white background. You should make this images as square as possible. You can use the white space around the sub-images to space them out to achieve this.
3. You may add additional flags to control properties such as how these images are spaced; you may also want to add some control over the creation of the board images themselves - perhaps to allow a slight 'margin' /separator between tiles to show them up better (As in the original figure at the top of this assignment). Make sure to document all this in your README.

Implementation Notes

1. When parsing command line parameters, remember to include

   ```
   int main(int argc, char *argv[])
   ```

   when declaring main(). You will then have access to argc (argument count) and argv (an array of argument char* strings) which you can then process with string processing and conversion operations. Remember to do error checking when parsing your command line (there is a BOOST library that can help, but it is not part of the C++ core and your program may not use non-standard libraries. This example is also simple enough to avoid that, and its good practice). Note that argc also counts the program name itself, which is accessible as argv[0].

2. Since we have not fully covered move semantics and more complex class features, please ignore the usual requirements for move/copy constructors, special assignment operators and so on. You must provide a (parameterised) constructor as well as a destructor for all your defined classes. If your destructor doesn't need to do anything – please do still indicate this by having empty braces {} for its implementation. You should also define appropriate private/public methods to set/read and manipulate the class data/state.
3. When subdividing the image into tiles **each** tile must be the same size. To avoid issues you may slightly shrink image dimensions in your output state image e.g. for a 400 x 400 pixel image, with a 3 x 3 tile grid, each sub-image would be int(400/3) = 133. But 3 x 133 = 399 pixels. This is fine we'd consider only the 399 x 399 portion of this 400 x 400 input image. Just remember to reduce the range of pixels rows/cols that you iterate over if you do 'shrink' your input image in his way. Note that in this case, the output image you must generate would now be of size 399 x 399.
4. You cannot assume square images.
5. You will be reading in PGM (portable grey map images). **I have provided code for this, but you are welcome to write your own implementation if you wish**. The spec is explained in more detail in this document. You can export arbitrary images (greyscale images) in this format using "the Gimp" image editor, or apply some of the image conversion tools that exist under Unix (pngtopnm, for example, which will turn a greyscale PNG image into a PGM or pnmtopng to go the opposite way).

3 PGM Images (for information)

PGM images are greyscale — meaning they have no colour, and use only one value to encode an *intensity* that ranges from (black = 0) through to (white=255). Each value is thus stored as an 'unsigned char'. The images we will provide are 'raw' PGM i.e. binary files. However, they have a text header, so you usually open the file as binary (ios::binary), but use the >> and << operators to read/write the header information. Getline() is very useful for reading and discarding comment lines. The header is followed by a block of bytes, which represent the image intensities. You can use the read() and write() methods to manipulate this data. Look at the function prototypes to see what arguments they expect (www.cplusplus.com can help here if you have no C++ reference). The PGM images you will receive will have the format:

```
P5
# comment line (there can be more than 1 of these comment lines)
# the characters P5 will always be the first item in the file.
width height
255
binary_data_block
```

where width and height are integers representing the number of columns and rows making up the image matrix. There is a newline after each line — use “ws” to process this correctly.

After reading 255 (and using the ws manipulator) you will be at the start of the binary data (width*height unsigned chars) which you can read in using with one (!) call to read(). If you write out a PGM image you needn’t include any comments, although it is good practice to indicate what application generated the image. You can open the output image as a binary file, write out the text header using the usual text operator << and then use write() to output the binary data (in one statement!) before closing the file.

Have a look at https://en.wikipedia.org/wiki/Netpbm for more information on the image format (and it’s history).

Remember that image data (being a 2D array of unsigned char) is indexed from (0,0) and that this is the top left hand corner of the image. Generally we process image data line by line, starting from the top. This is in fact how the binary image data is stored in the PGM format — so you can read/write the entire image with one read()/write() statement! This is very convenient.

We will provide some PGM images. You can view the output of your application using an image viewer like Gimp on Ubuntu.

Please Note:

1. A working Makefile must be submitted. If the tutor cannot compile your program on nightmare.cs by typing make, you will only receive **50%** of your final mark.

2. You must use version control from the get-go. This means that there must be a .git folder alongside the code in your project folder. A **10%** penalty will apply should you fail to include a local repository in your submission.

   With regards to git usage, please note the following:

   -10% - usage of git is absent. This refers to both the absence of a git repo and undeniable evidence that the student used git as a last minute attempt to avoid being penalized.

   -5% - Commit messages are meaningless or lack descriptive clarity. eg: “First”, “Second”, “Histogram” and “fixed bug” are examples of bad commit messages. A student who is found to have violated this requirement for numerous commits will receive this penalty.

   -5% - frequency of commits. Git practices advocate for frequent commits that are small in scope. Students should ideally be committing their work after a single feature has been added, removed or modified. Tutors will look at the contents of each commit to determine whether this penalty is applicable. A student who commits seemingly unrelated work in large batches on two or more occasions will receive this penalty.

   Please note that all of the git related penalties are cumulative and are capped at -10%

(ie: You may not receive more than -10% for git related penalties). The assignment brief has been updated to reflect this new information.

We cannot provide a definitive number of commits that determine whether or not your git usage is appropriate. It is entirely solution dependent and needs to be accessed on an individual level. All we are looking for is that a student has actually taken the time to think about what actually constitutes a feature in the context of their solution and applied git best practices accordingly.

3. You must provide a README file explaining what each file submitted does and how it fits into the program as a whole. The README file should **not** explain any theory that you have used. The README be used by the tutors if they encounter any problems.

4. Do **not** hand in any binary files. Do **not** add binaries (.o files and your executable) to your local repository.

5. Please ensure that your tarball works and is not corrupt (you can check this by trying to downloading your submission and extracting the contents of your tarball - make this a habit!). Corrupt or non-working tarballs will not be marked - **no exceptions.**

6. A 10% penalty per day will be incurred for all late submissions. No hand-ins will be accepted if later than 1 day (24 hours).

7. **DO NOT COPY. All code submitted must be your own.** ***Copying is punishable by 0 and may result in academic misconduct being noted on your academic record.*** **Scripts will be used to check that code submitted is unique.**