CS246—Assignment 4 (Spring 2017)

Due Date 1: Sunday, July 2, 5pm
Due Date 2: Monday, July 10, 5pm

Note: There is very little due on DD1. You should not take this to mean that you can delay starting on DD2 deliverables until DD1 has passed. There is significant coding required for DD2 and you are highly encouraged to start early.

Note: You must use the C++ I/O streaming and memory management facilities on this assignment. Marmoset will be programmed to **reject** submissions that use C-style I/O or memory management.

Note: You may include the headers `<iostream>`, `<fstream>`, `<sstream>`, `<iomanip>`, `<string>`, `<utility>`, `<stdexcept>`, and `<vector>`.

Note: For this assignment, you are **not allowed** to use the array (i.e., `[]`) forms of `new` and `delete`. Further, the `CXXFLAGS` variable in your `Makefile` **must** include the flag `-Werror=vla`.

Note: Questions on this assignment ask you to write C++ programs that can each span multiple files. For this reason, we **strongly** recommend that you develop your solution for each question in a separate directory. Just remember that, for each question, you should be *in* that directory when you create your zip file, so that your zip file does not contain any extra directory structure.

0. This question will help you prepare for Problem 2 and Assignment 5.

   (a) **Note:** Problem 2 asks you to work with XWindows graphics. Well before starting that question, make sure you are able to use graphical applications from your Unix session. If you are using Linux you should be fine (if making an ssh connection to a campus machine, be sure to pass the `-Y` option). If you are using Windows and putty, you should download and run an X server such as XMing, and be sure that putty is configured to forward X connections. Alert course staff immediately if you are unable to set up your X connection (e.g. if you can't run `xeyes`).
   Also (if working on your own machine) make sure you have the necessary libraries to compile graphics. Try executing the following (files are present in the q2 provided-files directory):

   ```
   g++14 window.cc graphicsdemo.cc -o graphicsdemo -lX11
   (Note: that is a dash followed by lower case L followed by X and then one one)
   ./graphicsdemo
   ```

   Note for Mac OS users: On machines running Mac OS you will need to install XQuartz. `http://xquartz.macosforge.org/`. Once installed, you might have to explicitly tell g++ where `X11` is located. If the above does not work, browse through your

Mac's file system looking for a directory `X11` that contains directories `lib` and `include`. You must then specify the lib directory using the `-L` option and the include directory using the `-I` (uppercase i) option. For example, on my MacBook I used:

```
g++14 window.cc graphicsdemo.cc -o graphicsdemo -lX11 -L/usr/X11/lib -I/usr/X11/include
./graphicsdemo
```

You know that the above test is successful if the following happens:

- Two windows open
- The big window prints the strings Hello!, ABCD, Hello! followed by rectangles containing a rainbow of colours
- The small window prints ABCD

(b) This is also the time to find partner(s) for **assignment 5** in which you will be developing a medium size game. At this point all you need to do is choose your partner(s) for the assignment and let us know. A project group can be formed by up to three students who are registered in this term's CS246 (students can be in different sections).

Due on Due Date 1: Submit the name of your project group members to Marmoset(`group.txt`). **Only one member of the group should submit the file. If you are working alone, submit nothing.** The format of the file `group.txt` should be

```
userid1
userid2
userid3
```

where `userid1`, `userid2`, and `userid3` are UW userids, e.g. `j25smith`.

1. In this problem, you will write a program to read and evaluate arithmetic expressions. There are four kinds of expressions:

   - lone integers
   - variables, which have a name (letters only, case-sensitive, but cannot be the words `done`, `ABS`, or `NEG`)
   - a unary operation (`NEG` or `ABS`, denoting negation and absolute value) applied to an expression
   - a binary operation (`+`, `-`, `*`, or `/`) applied to two expressions

   Expressions will be entered in reverse Polish notation (RPN), also known as postfix notation, in which the operator is written after its operands. The word `done` will indicate the end of the expression. For example, the input

   ```
   12 34 7 + * NEG done
   ```

   denotes the expression $-(12 * (34 + 7))$. Your program must read in an expression, print its value in conventional infix notation, and then initiate a command loop, recognizing the following commands:

- `set var num` sets the variable `var` to value `num`. The information about which variables have which values should be stored as part of the expression object, and not in a separate data structure (otherwise it would be difficult to write a program that manipulates more than one expression object, where variables have different values in different expressions).
- `unset var` reverts the variable `var` to the unassigned state.
- `print` prettyprints the expression. Details in the example below.
- `eval` evaluates the expression. This is only possible if all variables in the expression have values (even if the expression is `x x -`, which is known to be 0, the expression cannot be evaluated). If the expression cannot be evaluated, you must raise an exception and handle it in your main program, such that an error message is printed and the command loop resumes. Your error message must print the name of the variable that does not have a value (if more than one variable lacks a value, print one of them).

For example (output in italics):

```
1 2 + 3 x - * ABS NEG done
-|((1 + 2) * (3 - x))|
eval
x has no value.
set x 4
print
-|((1 + 2) * (3 - 4))|
eval
-3
set x 3
print
-|((1 + 2) * (3 - 3))|
eval
0
unset x
print
-|((1 + 2) * (3 - x))|
eval
x has no value.
```

(Note: the absolute symbol is vertical bars but appears slanted above since all output is shown in italics)

Numeric input shall be integers only. If any invalid input is supplied to the program, its behaviour is undefined. **Note that a single run of this program manipulates one expression only. If you want to use a different expression, you need to restart the program.**

To solve this question, you will define a base class `Expression`, and a derived class for each of the the four kinds of expressions, as outlined above. Your base class should provide virtual methods `prettyprint`, `set`, `unset`, and `evaluate` that carry out the required tasks.

To read an expression in RPN, you will need a stack. Use cin with operator `>>` to read the input one word at a time. If the word is a number or a variable, create a corresponding expression object, and push a pointer to the object onto the stack. If the word is an operator,

pop one or two items from the stack (according to whether the operator is unary or binary), convert to the corresponding object and push back onto the stack. When `done` is read, the stack will contain a pointer to a single object that encapsulates the entire expression.

Once you have read in the expression, print it out in infix notation with full parenthesization, as illustrated above. Then accept commands until EOF.

Note: Your program should be well documented and employ proper programming style. It should not leak memory. Markers will be checking for these things.

Note: The design that we are imposing on you for this question is an example of the Interpreter pattern (this is just FYI; you don't need to look it up, and doing so will not necessarily help you).

Due on Due Date 1: A UML diagram (in PDF format `q1UML.pdf`) for this program. There are links to UML tools on the course website. Do **not** handwrite your diagram. Your UML diagram will be graded on the basis of being well-formed, and on the degree to which it reflects a correct design.

Due on Due Date 2: The C++ code for your solution. You must include a Makefile, such that issuing the command `make` will build your program. The executable should be called `a4q1`.

2. (a) In this problem, you will use C++ classes to implement the game of Lights Out. (`http://en.wikipedia.org/wiki/Lights_Out_%28game%29`). An instance of Lights Out consists of an $n \times n$-grid of cells, each of which can be either on or off. When the game begins, we specify an initial configuration of on and off cells. Lights Out is a one-player game. Once the cells are configured, the player chooses a cell and turns it on if it is off, and off if it is on. In response the four neighbouring cells (to the north, south, east, and west) all switch configurations between off and on as well. The object of the game is to get all of the cells turned off.

   To implement the game, you will use the following classes:

   - `class Subject` — abstract base class for subjects (see provided `subject.h`);
   - `class Observer` — abstract base class for observers (see provided `observer.h`);
   - file `subscriptions.h` — contains an enumeration of possible subscription types;
   - `class Cell` — implements a single cell in the grid (see provided `cell.h`);
   - `class Grid` — implements a two-dimensional grid of cells (see provided `grid.h`);
   - `class TextDisplay` — keeps track of the character grid to be displayed on the screen (see provided `textdisplay.h`);
   - file `info.h` structure definition for queries issued by the observer on the subject.

   Note: you are not allowed to change the public interface of these classes (i.e., you may not add public fields or methods), but you may add private fields or methods if you want.

   Your solution to this problem must employ the Observer pattern. Each cell of the grid is an observer of all of its neighbours (that means that class `Cell` is its own observer). Thus, when the grid calls `notifyNeighbours` on a given cell, that cell must then call the `notify` method on each of its neighbours (each cell is told who its neighbours are when the grid is initialized). Moreover, the `TextDisplay` class is an observer of every cell (this is also set up when the grid is initialized). A subject's collection of observers does not distinguish what kinds of observers are actually in the collection (so a cell could have

arbitrary cells or displays subscribed to it). When the time comes to notify observers, you just go through the collection and notify them. However, not all observers want to be notified at all times. Displays want to be notified whenever the cell's state changes, so that their display will be accurate. Neighbouring cells, however, only want to be notified when a cell has been `switch`ed by the user, because they will then change their state as well. However, since this latter change of state was not the result of a `switch` call, *their* neighbours should not change state, and therefore should not be notified (otherwise, the switching would cascade throughout the grid, which is not how the game works). So there are two kinds of subscriptions: subscriptions to all events, and subscriptions to switch events only. The file `subscriptions.h` has an enumeration for these subscription types, and the class `Observer` has a pure virtual method whereby an observer object can indicate what kind of a subscriber it is. The cells will use this information to determine which observers need to be notified on a given event.

You are to overload `operator<<` for the text display, such that the entire grid is printed out when this operator is invoked. Each on cell prints as `X` and an off cell prints as `_` (i.e., underscore). Further, you are to overload `operator<<` for grids, such that printing a grid invokes `operator<<` for the text display, thus making the grid appear on the screen.

When you run your program, it will listen on stdin for commands. Your program must accept the following commands:

- `new n` Creates a new $n \times n$ grid, where $n \geq 1$. If there was already an active grid, that grid is destroyed and replaced with the new one.
- `init` Enters initialization mode. Subsequently, read pairs of integers `r c` and set the cell at row `r`, column `c` as on. The top-left corner is row 0, column 0. The coordinates `-1 -1` end initialization mode. It is possible to enter initialization mode more than once, and even while the game is running. When initialization mode ends, the board should be displayed.
- `game g` Once the board has been initialized, this command starts a new game, with a commitment to solve the game in `g` moves or fewer.
- `switch r c` Within a game, switches the cell at row `r`, column `c` on or off, and then redisplays the grid.

The program ends when the input stream is exhausted or when the game is won or lost. The game is lost if the board is not cleared within `g` moves. You may assume that inputs are valid.

If the game is won, the program should display `Won` to stdout before terminating; if the game is lost, it should display `Lost`. If input was exhausted before the game was won or lost, it should display nothing.

A sample interaction follows (responses from the program are in italics):

```
new 5
init
1 2
2 2
3 2
-1 -1
```

_____
__X__
__X__

```
__X__
-----
game 3
3 moves left
switch 2 2
-----
-----
_X_X_
-----
-----
2 moves left
switch 3 1
-----
-----
___X_
XXX__
_X___
1 move left
switch 3 3
-----
-----
-----
XX_XX
_X_X_
0 moves left
Lost
```

Note: Your program should be well documented and employ proper programming style. It should not leak memory. Markers will be checking for these things.

Due on Due Date 2: Submit your solution. You must include a Makefile, such that issuing the command `make` will build your program. The executable should be called `a4q2a`.

(b) **Note: there is no sample executable for this problem, and no Marmoset tests. This problem will be entirely hand-marked.** In this problem, you will adapt your solution from problem 2a to produce a graphical display. You are provided with a class `Xwindow` (files `window.h` and `window.cc`), to handle the mechanics of getting graphics to display. Declaring an `Xwindow` object (e.g., `Xwindow xw;`) causes a window to appear. When the object goes out of scope, the window will disappear (thanks to the destructor). The class supports methods for drawing rectangles and printing text in five different colours. For this assignment, you should only need black and white rectangles. To make your solution graphical, you should carry out the following tasks:

- Create a `GraphicsDisplay` class as a subclass of the abstract base class `observer`, and register it as an observer of each cell object. It should subscribe to all events, not just switch events.
- The class `GraphicsDisplay` will be responsible for mapping the row and column numbers of a given cell object to the corresponding coordinates of the squares in the window.

- Your main function will create a window, and pass a reference to this window to your `GraphicsDisplay` object, which will use this reference to update the contents of the window.
- Your cell objects should not have to change at all.

The window you create should be of size 500×500, which is the default for the `Xwindow` class. The larger the grid you create, the smaller the individual squares will be.

Note: to compile this program, you need to pass the option `-lX11` to the compiler *at link time*. For example:

```
g++-5 -std=c++14 *.cc -o a4q2b -lX11
```

This option is not relevant during compilation, so it should not be put in your `CXXFLAGS` variable. You should only use it during the linking stage, i.e., the command that builds your final executable.

Note: Your program should be well documented and employ proper programming style. It should not leak memory (note, however, that the given `XWindow` class leaks a small amount of memory; this is a known issue). Markers will be checking for these things.

Due on Due Date 2: Submit your solution. You must include a Makefile, such that issuing the command `make` will build your program. The executable should be called `a4q2b`.

3. In this problem you will implement an extensible image processing package that can apply `flip`, `rotate` and `sepia` transformations to an image using the Decorator pattern.

   You are provided with a partially-implemented mainline program for testing your image processor (`main.cc`). The mainline interpreter loop works as follows:

   - You issue commands of the form

     ```
     source-file destination-file list-of-decorators
     ```

     all in one line. The `source-file` is the name of a file that contains the `Image` to be transformed. The format for this file is discussed below. The `destination-file` is the name of the file to which the transformed image is written. The `list-of-decorators` can be any of `flip`, `rotate` and `sepia` in any order and any number of times.
   - The program constructs a custom Image processor from `list-of-decorators` and applies the transformations to the Image in `source-file`
   - You may then issue another command. An end-of-file signal ends the interpreter loop.

   Format for source-file: We have provided you with a few `source-file`s (extensions `img`). These contain the representation of some Images that you can use to test your program. You should not need to create new `img` files for testing. However, if you would like to create your own, the file contains a sequence of RGB values (range 0-255), the width of the image and the height of the image. All input is separated by whitespace.

   Internal Representation of Image: The internal representation for an `Image` used by the program is already provided to you in `ppmArray.h`. Note that you have also been provided with `ppmArray.o` (an object file that contains the implementation of methods and functions declared in `ppmArray.h`). In particular, we have already implemented overloaded input and output operators for `PpmArray`. The overloaded input operator for `PpmArray` will update a PpmArray to create the internal representation of the Image available on the input stream.

The overloaded output operator for `PpmArray` converts the internal representation of an `Image` into PPM format (see below). You need not know how these have been implemented.

Format for destination-file (Plain PPM Format): The output operator for `PpmArray` will convert the internal representation of the (possibly) transformed input image into "Plain PPM format". Since this function has already been implemented for you, you do not need to know the details. However, if you are curious: PPM is a simple image encoding format. Refer to `http://netpbm.sourceforge.net/doc/ppm.html` for additional details (Plain PPM section). The file starts with P3 which is the magic number for a Plain PPM file. This is followed by two whitespace separated integer values representing the width and height of the image, respectively. This is followed by the number 255 which represents the maximum colour value for the pixel data. Each line following this header, contains the pixel data for a row of the image in order from top to bottom. The pixel data for an entire row is on its own line. A single pixel consists of three numbers, between $0 - 255$, representing the RGB value for that pixel. Consecutive RGB values are separated by white space. For a single pixel, the individual RGB values are also separated by white space.

Transformations through the Decorator Pattern: You are also provided with the following fully-implemented classes:

- `Image` (`image.{h,cc}`): abstract base class that defines the interface to the image processor.
- `BasicImage` (`basic.{h,cc}`): concrete implementation of `Image`, which provides default behaviour. In particular, the already implemented method `BasicImage::render` uses the input operator for `PpmArray` to create the internal representation from the input stream.

You are not permitted to modify these two classes in any way.

You must provide the following transformations that can be added to the default behaviour of `BasicImage` via decorators:

- `flip`: Modifies the internal representation (`ppmArray`) so that it now represents an image that has been flipped horizontally. In other words, Pixel 1 of a row with n Pixels becomes Pixel n, Pixel 2 becomes Pixel n-1 and so on.
- `rotate`: Modifies the internal representation (`ppmArray`) so that it now represents an image that has been rotated 90° clockwise. If `i` is the row index and `j` is the column index of a pixel in the rotated image, then the following will give the required values for the new pixel array, where `ppm` contains the current image.

  ```
  newPixel[i * ppm.height + j] = ppm.pixels[ppm.width * (ppm.height - j - 1) + i];
  ```

- `sepia`: Modifies the internal representation (`ppmArray`) so that it now represents an image that has a sepia filter applied to the image. Apply the following formulae (exactly as given) to each pixel in the image. For `Pixel p` with result stored in `Pixel np`:

  ```
  np.r = p.r *.393 + p.g * .769 + p.b * .189
  np.g = p.r *.349 + p.g * .686 + p.b * .168
  np.b = p.r *.272 + p.g * .534 + p.b * .131
  ```

  If any of the final values above exceed 255, then set them to the maximum value of 255.

These functionalities can be composed in any combination of ways to create a variety of custom Image processors.

You must change `main.cc` so that it **updates** the `img` pointer variable to an appropriate decorated Image transformer based on the transformations given for each line of input. The area of code you would need to modify is clearly marked.

An example input to the program is as follows (`public.in`):

```
small.img small_none.ppm
small.img small_rotate_flip.ppm rotate flip
castle.img castle_multi.ppm rotate flip sepia rotate rotate
```

In the first line, the file small.img is the file that will be used by `BasicImage` to read in the input image, the output will be written to small_none.ppm and no transformations will be applied. The second line uses small.img for the image, stores the transformed image to small_rotate_flip.ppm and applies rotate and flip, in this order.

Your program must be clearly written and must follow the Decorator pattern (with each class in its own .h and .cc file). Since the purpose of this question is to practice writing the Decorator pattern, failure to follow these instructions will lead to severe penalties during hand marking. Your program must not leak memory.

Due on Due Date 1: A UML diagram (in PDF format `q3aUML.pdf`) for your proposed implementation of this program. There are links to UML tools on the course website. Do **not** handwrite your diagram. Your UML diagram will be graded on the basis of being well-formed, and on the degree to which it reflects a correct design.

Due on Due Date 2: Submit the UML diagram (in PDF format `q3bUML.pdf`) that best represents the solution you implemented (this can be different from the UML you submitted in in Due Date 1). Also, submit your solution. You must include a Makefile, such that issuing the command `make` will build your program. The executable should be called `a4q3`.