CS246—Assignment 4 (Spring 2018)

C. Kierstead A. Moss V. Sakhnini

Due Date 1: Tuesday, July 03, 11:55pm

Due Date 2: Monday, July 09, 11:55pm

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: Each question on this assignment asks you to write a C++ program, and the programs you write on this assignment 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.

Note: A proper object-oriented design is achieved through the use of virtual methods that allow you to perform the required operations, without ever knowing exactly what kind of object you have. Therefore any attempt to "query" the run-time types of objects, and then make decisions based on the discovered types, will not earn marks.
Note: Problem 5 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:

```
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)

Run the program
./graphicsdemo
```

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. **(Note that the method that carries out this operation should return a string representation of the expression, and your main function should do the actual printing.)**
   - `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. **All of this is to be done within `operator>>` for expression pointers:**

```
istream &operator>>(istream &in, Expr *&e);
```

There is to be no use of stacks outside of `operator>>`.

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. This problem continues Problem 1. Suppose you wish to be able to copy an expression object. The problem is that if all you have is an `Expression` pointer, you don't know what kind of object you actually have, much less what types of objects its fields may be pointing at. Devise a way, given an `Expression` pointer, to produce an exact (deep) copy of the object it points at. Do not use any C++ language features that have not been taught in class.

   When you have figured out how to do it, implement it as part of your solution for Problem 1, and add a `copy` command to your interpreter. If the command is `copy`, you will execute the following code:

```
Expression *theCopy = ( ... create a copy of your main Expression object ...)
cout << theCopy->prettyprint() << endl;
theCopy->set("x", 1);
cout << theCopy->prettyprint() << endl;
try { cout << theCopy->evaluate() << endl; }
catch(...){ /* Do the appropriate thing */ }
delete theCopy;
```

   The copy will contain the same variable assignments as the original expression. However, setting the variable `x` to 1 in the copy should not affect the value of `x` in the original expression. `x` may or may not be the only unset variable in the expression. If there are other unset variables, then naturally, an exception will be raised, and you should handle it in the same way as previously.

   This problem will be at least partially hand-marked for correctness. A test suite is not required. **Note that Marmoset will provide only basic correctness tests of output. If it is found during handmarking that you did not complete this problem according to our instructions, you correctness marks from Marmoset will be revoked.**

   Due on Due Date 2: Your solution. Submit it as part of your Q1 solution. Your Q1 UML should not include your solution for Q2.

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`). Assuming the program is named `a4p3`, the program can be executed as follows:

```
./a4p3 source.ppm destination.ppm flip rotate flip sepia
```

   `source.ppm` is the name of the file that contains the `Image` to be transformed. `destination.ppm` is the name of the file to which the transformed image is written. The format for `ppm` files is discussed below. After the source and destination files come 0 or more of `flip`, `rotate` and `sepia` in any order and any number of times. The program constructs a custom `Image` processor from this list of decorators and applies the transformations to the Image in `source.ppm`

   Internal Representation of Image: The internal representation for an `Image` used by the program is already provided to you in a class `PPM` defined in `ppm.h`. Note that you have also been provided with `ppm.o` (an object file that contains the implementation of methods and functions declared in `ppm.h`). In particular, we have already implemented overloaded input and output operators for `PPM`. The overloaded input operator for `PPM` will update a PPM object to create the internal representation of the Image available on the input stream. The overloaded output operator for `PPM` converts the internal representation of an `Image` back to PPM format. You need not know how these have been implemented.

Format of source and destination files (Plain PPM Format): The source image and the destination image are represented in “Plain PPM format” which is a simple image encoding format. Since the input and output operators of `PPM` have already been implemented, you do not need to know the details. However, if you are curious, refer to `http://netpbm.sourceforge.net/doc/ppm.html` for details (Plain PPM section). Some documentation is also present in `ppm.h`.

We have provided you with two sample `.ppm` files to help you test your program. `small.ppm` is a small image with multiple colors. `castle.ppm` is a very large image of a castle. To see the effects of your transformations, you can open ppm files with certain image viewing software (e.g. gimp).

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 `PPM` 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 (`PPM`) 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 (`PPM`) so that it now represents an image that has been rotated 90° clockwise. If `i` is the column index and `j` is the row index of a pixel in the rotated image, then the following will give the required values for the new pixels, 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 (`PPM`) 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 in the command line argumnets.. The area of code you would need to modify is clearly marked.

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 all your Due Date 2 correctness marks for this question being revoked.** Your program must not leak memory.

Due on Due Date 1: A UML diagram (in PDF format `q3UML.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. **Note that we have not asked you to create any test cases. We highly recommend creating them for your own use.**

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 `a4q3`.

4. In this problem, you will use C++ classes to implement the game of Reversi. (`https://en.wikipedia.org/wiki/Reversi`). An instance of Reversi consists of an $n \times n$-grid of cells, each of which can be either empty, black, or white. When the game begins the middle 4 cells are populated following the pattern Black-White-Black-White. Reversi is a two-player game where players take turns placing a piece of their colour in a cell. Black plays first. The goal of Reversi is to have the most cells holding pieces of your colour at the end of the game. If a new piece A would form a line segment with an existing piece B of the same colour, such that all of the cells in between are occupied and of the opposite colour, those in-between pieces are flipped to the same colour as A and B. There is one slight difference from standard Reversi: whereas a legal move in standard Reversi must cause at least one flip, your program is not required to enforce this rule. This means that on any turn players can play a piece on any cell, so long as that cell is within the grid and not currently occupied by another piece.

   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`);
   - `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`);
   - `struct State` — provides a representation of a cells state (see provided `state.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.

   As a hint for implementation, consider how the rules of the game above can be replicated using an observer pattern where a notification can either be a notification of a **new piece**, a **relay** of that new piece message, or a **reply** to that new piece being found. A cell that receives a new piece can notify its observers (adjacent cells) that it has received a new piece of a certain colour. The cells notified of a new piece by their neighbour can relay that message along the line. When a message is received by a cell that contains the same colour piece as the new piece, it can reply back in a similar fashion. Hence, when a piece matching the colour of the new piece is reached, the message stops relaying and instead replies back. Similarly if there are no pieces in a cell to relay a message then it should stop. Since the observer pattern doesn't distinguish what observers are in the collection the cell receiving a new piece can't send out specific information about the direction of each line to its appropriate neighbours. However, when a neighbour receives a notification of a new piece they can check the information passed along and determine what direction they are from the original piece, and relay that information along. For more information on these messages see the `state.h` file provided to you.

You are to overload `operator<<` for the text display, such that the entire grid is printed out when this operator is invoked. Each empty cell prints as `-` and a cell with a white or black piece prints as `W` or `B` respectively. 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 $n \geq 4 \wedge n \equiv 0 \pmod 2$. If there was already an active grid, that grid is destroyed and replaced with the new one. When setting up the new grid your program should intialize the middle 4 squares following the Black-White-Black-White pattern.
- `play r c` Within a game, plays a piece at row `r`, column `c` of the colour corresponding to the player who's move it is. If the row and column entered correspond to a cell that already has a piece, or a position outside the grid, then the input is ignored and nothing is done.

The program ends when the input stream is exhausted, or when the game is over. The game is over when there are no more empty cells on the grid.,

When the game is over, if the black player wins the program should display `Black Wins!` to stdout before terminating; if the white player wins it should display `White Wins!`; otherwise it should display `Tie!`. 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 8
--------
--------
--------
---BW---
---WB---
--------
--------
--------
play 3 5
--------
--------
--------
---BBB--
---WB---
--------
--------
--------
play 3 6
```

```
--------
--------
--------
---BBBW-
---WB---
--------
--------
--------
play 3 2
--------
--------
--------
--BBBBW-
---WB---
--------
--------
--------
play 2 3
--------
--------
---W----
--BWBBW-
---WB---
--------
--------
--------
play 0 0
B-------
--------
---W----
--BWBBW-
---WB---
--------
--------
--------
play 3 1
B-------
--------
---W----
-WWWBBW-
---WB---
--------
--------
--------
play 7 7
```

```
B-------
--------
---W----
-WWWBBW-
---WB---
--------
--------
-------B
^d
```

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 `a4q4`.

5. **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 4 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, white, and blue rectangles. In the graphical representation let blue represent an empty cell, black represent a cell containing a piece from the black player, and white represent a cell containing a piece from the white player. To do so you must complete 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.
   - 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 `GraphicsDisplay` class should have a composition relationship with Xwindow. So your constructor for `GraphicsDisplay` should also create a `Xwindow` object which is a member of the `GraphicsDisplay` class. Each time a new game is created you should create a new `GraphicsDisplay` to set up an appropriate window for the game size.
   - 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 *.o -o a4q5 -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 `a4q5`.