CS246—Assignment 4 (Fall 2014)

Due Date 1: Friday, November 7, 11:55pm
Due Date 2: Friday, November 14, 11:55pm

Questions 0, 1, 2 (test suite), 3a (test suite), 4 (test suite) are due on Due Date 1; the remainder of the assignment is due on Due Date 2.

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: 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: Beginning with this assignment, you are now required to submit a `Makefile` along with every code submission. Marmoset will use this Makefile to build your submission.

0. This question will help you prepare for Problem 3 and Assignment 5.

   - Problem 3 asks you to work with XWindows graphics. To start that question you must first 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++ window.cc graphicsdemo.cc -o graphicsdemo -lX11
     ./graphicsdemo
     ```

     Note: (thats lower case L followed by X and one one)

     You know that the above test is successful if a new window opens and displays the string "Hello" and a blue rectangle.

     Note for Mac OS users: On machines running newer 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++ window.cc graphicsdemo.cc -o graphicsdemo -lX11 -L/usr/X11/lib -I/usr/X11/include
```

- This is also the time to find a partner 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 for the assignment and let us know.

Due on Due Date 1:

- Submit a file named `graphics.txt` containing the word `Installed`. By submitting this file you are declaring that you have configured your environment to run XWindows graphics and have successfully compiled and executed the provided graphicsdemo. (Every student must do this individually.)
- Submit the name of your project partner to Marmoset. (`partner.txt`) **Only one member of the partnership should submit the file. If you are working alone, submit nothing.** The format of the file `partner.txt` should be

  ```
  userid1
  userid2
  ```

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

1. Two files, `arrays.c` and `arrays2.cc` have been made available. Take a look at these files and note the difference between them. Then compile both of these as follows:

   ```
   g++ -DSIZE=5000 arrays.cc -o arrays
   g++ -DSIZE=5000 arrays2.cc -o arrays2
   ```

   Using the `time` command, measure the execution time of each of these programs:

   ```
   time ./arrays
   time ./arrays2
   ```

   If the numbers are not significantly different, use a larger number when you compile the files. If the programs crash when you run them, use a smaller number. Then answer the following in `q1.txt`:

   (a) What is the difference between these two programs?

   (b) What is the difference between the timings of these two programs (report user times).

   (c) Read the following article on locality of reference, specifically spatial locality:
   `http://en.wikipedia.org/wiki/Locality_of_reference`
   Pay particular attention to what it says about hierarchical memory. Based on what you have read, how do you explain the difference in running time?

   Due on Due Date 1: `q1.txt`

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

- lone integers
- 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. For example, the input

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

denotes the expression $-(12 * (34 + 7))$. Your program must read in an expression, print its value in conventional infix notation, and then print its value. For example (output in italics):

```
1 2 + 3 4 - * ABS NEG
-|((1 + 2) * (3 - 4))|
= -3
```

To solve this question, you will define a base class `Expression`, and a derived class for each of the three kinds of expressions, as outlined above. Your base class should provide virtual methods `prettyprint` 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, 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 the input is exhausted, the stack will contain a pointer to a single object that encapsulates the entire expression.

For the stack, use an array of pointers to expression objects. The array should have size 10. If at any point, you require more than this amount of stack space, you should print "Stack overflow" to `cerr` and terminate the program.

Once you have read in the expression, print it out in infix notation with full parenthesization, as illustrated above. Then evaluate the expression and print the result.

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.

Note: We have provided one test case in the a4/q2 directory. We have also provided a compiled executable of the solution for you to experiment with / generate .out files.

Due on Due Date 1: Submit a test suite (`suiteq2.txt`) and UML diagram (in PDF format `q2UML.pdf`) for this program. There are links to UML tools on the course website. **Neatly** handwritten and scanned to pdf is also acceptable. Your UML diagram will be graded on the basis of being well-formed, and on the degree to which it matches the `.h` files that you submit on Due Date 2.

Due on Due Date 2: Submit your solution. You must include a Makefile, such that issuing the command `make` will build your program and create an executable named a4q2.

3. (a) In this problem, you will use C++ classes to implement the game of Flood It, a one player game. You can play a graphical version of the game at `http://floodit.appspot.com/`. An instance of Flood It consists of an $n \times n$-grid of cells, each of which can be in one of 5 states, 0, 1, 2, 3, or 4 (with the default state being 0). These states can be thought of as colors with the mapping 0(White), 1(Black), 2(Red), 3(Green) and 4(Blue). Before the game begins, we specify an initial configuration of the state of cells. Once the cells are configured, the player repeatedly changes the state of the top-left (0,0) cell. In response all neighbouring cells (to the north, south, east, and west) switch state if they were in the same state as a neighbouring cell which changed states. The object of the game is to have all cells in the grid be in the same state before running out of moves.

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

   - `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`).

   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). The grid can call `Cell::notify` on a given cell and ask it to change state. Note that because of the way the game is played, it only makes sense for the grid to call `Cell::notify` on the (0,0) cell with a single parameter, the new state of the cell. The notified cell must then call a `notify` method on each of its neighbours (each cell is told who its neighbours are when the grid is initialized). In this notification, you might find it useful to send the cell's current and previous states as parameters. Moreover, the `TextDisplay` class is an observer of every cell. When a cell's status changes, it must invoke `TextDisplay::notify` to publish its new state to the observer.

   You are to overload `operator<<` for the text display, such that the entire grid is printed out when this operator is invoked. 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, reads triples of integers `r c s` and sets the cell at row `r`, column `c` to state `s`. 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. A set of invalid co-ordinates and state will be ignored. When initialization mode ends, the board should be displayed.
   - `include f` The file `f` is a list of cell initializations of the same form of initialization from init. This file does not have to end with the coordinates `-1 -1`.
   - `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. `game` cannot be called once a

game has been started.

- `switch s` Within a game, switches the top-left (0,0) cell to s, changes all appropriate neighbours, 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 in one state 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 4
init
0 0 4
0 2 3
0 3 2
1 1 1
1 3 3
2 0 4
2 2 2
2 3 1
3 1 2
3 3 3
-1 -1
4032
0103
4021
0203
game 4
4 moves left
switch 0
0032
0103
4021
0203
3 moves left
switch 4
4432
4103
4021
0203
2 moves left
switch 0
0032
0103
0021
0203
1 move left
```

```
switch 2
2232
2103
2221
2203
0 moves left
Lost
```

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

Note: Provided files: cell.h, grid.h, textdisplay.h

Due on Due Date 1: Submit a test suite for this program. Call your suite file `suiteq3.txt`.

Due on Due Date 2: Submit your solution. You must include a Makefile, such that issuing the command `make` will build your program calling the executable `flood`.

(b) In this problem, you will adapt your solution from problem 3 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 need white, black, red, green, and blue rectangles which correspond to the states 0, 1, 2, 3, and 4 respectively. To make your solution graphical, you should carry out the following tasks:

- add fields for the x- and y-coordinates, as well as width and height, and a pointer to a window in the `Cell` class.
- add a method `setCoords` to the `Cell` class, whose purpose is to set the above fields. The `Grid` object will call this method when it initializes the cells.
- add a field to the `Grid` class representing the pointer to the window, so that it can be passed on to the cells. Change `Grid`'s constructor so that it can initialize the window field.
- add `draw` methods to the `Cell` class to draw a rectangle to the correct spot on the board (as determined by each cell's coordinates and state).
- When a cell changes state, it should call its `draw` method.
- The program accepts the command `?` which prints out

  ```
  White: 0
  Black: 1
  Red:   2
  Green: 3
  Blue:  4
  ```

  which is the encoding between the text version and graphics version.

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. For example:

```
g++ *.cc -o flood-graphical -lX11
```

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.

Note: Provided files: window.h, window.cc, graphicsdemo.cc

Due on Due Date 2: Submit your solution. You must include a Makefile, such that issuing the command `make` will build your program calling the executable `flood-graphical`.

4. In this problem you will have a chance to implement the Decorator pattern. The goal is to write an extensible text processing package. You will be provided with two fully-implemented classes:

   - `TextProcessor` (`textprocessor.{h,cc}`): abstract base class that defines the interface to the text processor.
   - `Echo` (`echo.{h,cc}`): concrete implementation of `TextProcessor`, which provides default behaviour: it echoes the words in its input stream, one token at a time.

   You will also be provided with a partially-implemented mainline program for testing your text processor (`main.cc`).

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

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

   - `L337:` Translate the string to its L337 equivalent using the following translation key.

Char	a/A	e/E	t/T	o/O
L337	4	3	7	0
Char	b/B	g/G	i/I	s/S
L337	8	6	1	5
Char	z/Z	!	q/Q	
L337	2	1	9	

     For example, the string bait is translated to 8417.
   - `Lower:` All letters in the string are presented in the lowercase. Other characters remain unchanged. For example, HelloWorld! is changed to helloworld!
   - `DoubleCon:` All consonants occurring in the string are doubled. For example, apple is changed to apppplle
   - `Caesar X:` Each letter in the string is replaced by the letter occurring X places down alphabetically. For example, with `X` equals 7, d is replaced by k, v is replaced by c. With X equals 25, N is replaced by M. `X` is always non-negative not exceeding 26.

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

   The mainline interpreter loop works as follows:

   - You issue a command of the form `source-file list-of-decorators`. If `source-file` is `stdin`, then input should be taken from `cin`.
   - The program constructs a custom text processor from `list-of-decorators` and applies the text processor to the words in `source-file`, printing the resulting words, one per line.

- You may then issue another command. An end-of-file signal ends the interpreter loop.

An example interaction follows (assume `sample.txt` contains `Hello World!`):

```
sample.txt lower caesar 2 l337
jgnn9
y97nf1
sample.txt l337 caesar 4 doublecon
LL3pppp0
A0vvpphh1
```

Your program must be clearly written, must follow the Decorator pattern, and must not leak memory.

Due on Due Date 1: Submit a test suite for this program. Call your suite file `suiteq4.txt`.

Due on Due Date 2: Submit your solution. You must include a Makefile, such that issuing the command `make` will build your program and create an executable a4q4.