# CS246—Assignment 4 (Fall 2013)

R. Ahmed B. Lushman M. Prosser

Due Date 1: Friday, November 8, 5pm
Due Date 2: Friday, November 15, 5pm

**Questions 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.
**Note:** Problem 3 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++ window.cc graphicsdemo.cc -o graphicsdemo -lX11
```

**Due on Due Date 1:** 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 a 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 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.

**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.

3. (a) In this problem, you will use C++ classes to implement Conway's Game of Life. (`http://en.wikipedia.org/wiki/Conway's_Game_of_Life`). An instance of Conway's Game of Life consists of an $n \times n$-grid of cells, each of which can be either alive or dead. When the game begins, we specify an initial configuration of living and dead cells. The game then moves through successive generations, in which cells can either die, come to life, or stay the same, according to the following rules:

   - a living cell with fewer than two live neighbours or more than three live neighbours dies;
   - a living cell with two or three live neighbours continues to live
   - a dead cell with exactly three live neighbours comes to life; otherwise, it remains dead.

   The neighbours of a cell are the eight cells that immediately surround it (cells on the edges of the grid naturally have fewer neighbours).

   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). Thus, when the grid calls `Cell::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. When a cell's status changes, it must invoke `TextDisplay::notify` to publish its new state to the observer.

An iteration of the game (which corresponds to one invocation of `Grid::tick`) happens in two steps, as follows:

- (Step one) The grid calls each cell's `notifyNeighbours` method.
- When the cell's `notifytNeighbours` method is called, the cell, if alive, tells all of its neighbours that it is alive, by calling each neighbour's `notify` method).
- When a cell's `notify` method is called by one of its neighbours, it updates its record of how many of its neighbours are alive.
- (Step two) After the grid has called each cell's `notifyNeighbours` method, the grid calls each cell's `recalculate` method.
- When a cell's `recalculate` method is called, it calculates its alive or dead status for the next round, based on the number of messages it received from living neighbours, and notifies the text display.

You are to overload `operator<<` for the text display, such that the entire grid is printed out when this operator is invoked. Each living cell prints as `X` and a dead 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 alive. 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 simulation is running.
- `step` Runs one tick of the simulation (i.e., transforms the grid into the immediately succeeding generation).
- `steps n` Runs $n$ steps of the simulation.
- `print` Prints the grid.

The program quits when the input stream is exhausted. You may assume that inputs are valid.

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

```
new 5
init
1 2
2 2
3 2
-1 -1
print
_____
__X__
__X__
__X__
_____
step
```

```
print
-----
-----
_XXX_
-----
-----
```

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

**Bonus Opportunity:** Modify your solution to support the following two additional commands:

- `scrollup` – scrolls the grid one position upward. The top row becomes the bottom row, and all other rows move up.
- `scrollleft` – scrolls the grid one position to the left. The leftmost row becomes the rightmost row, and all other rows move left.

Note that these two operations will cause some cells' sets of neighbours to change. To earn credit for this problem, you must implement these commands such that they run in constant time. Moreover, no operations in your original solution that were constant time may become non-constant as a result of this change. If you are attempting the bonus, you may make modifications to the public interfaces of the given classes, but you must still follow the Observer pattern. Submit your solution to the bonus as a separate program. You do not need to submit a test suite for the bonus.

**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.

(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 should only need black and white rectangles. 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` and `undraw` methods to the `Cell` class to draw either a black or white rectangle to the correct spot on the board (as determined by each cell's coordinates).
- When a cell is turned on via a call to `Cell::setLiving`, it should call its `draw` method.
- When a cell goes from dead to alive, it should call its `draw` method.

- When a cell goes from alive to dead, it should call its `undraw` method.

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 life-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.

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

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:

- `SwapCase` Change all upper case letters to lowercase and all lowercase letters to uppercase. Leave all other characters unchanged.
- `DoubleVowels` Double up all vowels in the string. Vowels are a, e, i, o, u, and their capital equivalents.
- `SkipWord` Return only every second word from the input. The first, third, fifth, etc., words should be suppressed.
- `Translate` Change all occurrences of a given source character to a given destination character (you may assume that neither of these characters denotes whitespace).

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 translate e 3 swapcase doublevowels
h3LLOO
wOORLD
sample.txt skipword translate o 0
WOrld
```

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.