CS246—Assignment 4 (Fall 2016)

J. Avery   C. Kierstead   B. Lushman

Due Date 1: Monday, November 7, 5pm
Due Date 2: Wednesday, November 16, 5pm

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: 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++14 window.cc graphicsdemo.cc -o graphicsdemo -lX11
```

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.
```

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. 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;;
cout << theCopy->evaluate() << endl;
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. (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 `a4q3a`.

(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 3a 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 a4q3b -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 `a4q3b`.

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:

   - `dropfirst n` Drop the first `n` characters of each word. If `n` is greater than or equal to the length of some word, that word is eliminated.
   - `doublewords` Double up all words in the string.
   - `allcaps` All letters in the string are presented in uppercase. Other characters remain unchanged.
   - `count c` The first occurrence of the character `c` in the string is replaced with 1. The second is replaced with 2, ... the tenth is replaced with 10, and so on.

   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 doublewords dropfirst 2 count l
1 12o
2 34o
3 r5d
4 r6d
sample.txt allcaps
1 HELLO
2 WORLD
```

The numbers at the beginning are word numbers (word 1, word 2, etc.), and are supplied by the test harness. You do not need to generate them. Your program must be clearly written, must follow the Decorator pattern, and must not leak memory.

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`.