Assignment #4

Due Date 1: Monday, 5 November, 2018, 5:00 pm
Due Date 2: Monday, 12 November, 2018, 5:00 pm

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 4 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: this is a dash followed by lower case L followed by X and then one one)

Run the program
./graphicsdemo
```

Due on Due Date 1: Make a project group for Assignment 5. Each group may have up to 3 registered students from any section. 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`.

Part 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 $-(12*(34+7))$. Expression tokens are separated by one or more whitespace. 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 `Expr`, 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`. Be sure to zip all your .cc , .h and Makefile into `a4q1.zip`.

Part 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 `Expr` 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 `Expr` 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, extend your solution to Problem 1 to provide this ability. Also, add a `copy` command to your interpreter. If the command is `copy`, you will execute the following code:

```
Expr *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. The executable that is produced from your submitted Makefile should be called `a4q2`. Submit your updated solution to `a4q2b`. Your Problem 1 UML should not include your solution for Problem 2.

Part 3

In this problem, you will use C++ classes to implement Conway's Game of Life. (https://en.wikipedia.org/wiki/Conways_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 and corners 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`);

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.

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 `alertNeighbours` method.
- When the cell's `alertNeighbours` method is called, the cell, if alive, tells all of its neighbours that it is alive, by calling each neighbour's `neighbourIsAlive` method).
- When a cell's `neighbourIsAlive` 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 `alertNeighbours` 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.

You are to overload `operator<<` for cells, such that a 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 results in printing each of its cells, arranged as a grid.

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 `x y` and set the cell at (x, y) as alive. x represents the horizontal direction, and y represents the vertical direction. The top-left corner is $(0, 0)$. 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. Make sure your program does not leak memory.

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

```
new 5
init
2 1
2 2
2 3
-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.
Due on Due Date 1: Design the test suite suiteq3.txt for this program. Submit the zipped file `a3q1a.zip` containing your test suite.
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 `a4q3`.

Part 4

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 options `-L/usr/X11R6/lib -lX11` to the compiler. For example:

```
g++14 -L/usr/X11R6/lib -lX11 *.cc -o life-graphical
```

Note: Your program should be well documented and employ proper programming style. The X11 library is known to leak memory. However, your program should not cause any additional memory leaks. Markers will be checking for these things.
Due on Due Date 2: Submit your solution. The executable produced by your submitted Makefile should be called `a4q4`. Submit your updated solution to a4q4b.

Part 5

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 `a4q5`.