CS246—Assignment 4 (Fall 2015)

Due Date 1: Friday, November 13, 4:55pm
Due Date 2: Friday, November 20, 4:55pm

Questions 0, 2a (UML), 3 (test suite) are due on Due Date 1; the remainder of the assignment [1, 2a(Code), 3(Code)] and the bonus (2b) is due on Due Date 2.

Please note that there is not much work for Due Date 1. The only reason that Due Date 1 is so far in the future is because some of the UML concepts needed for Q2a will be covered in Tuesday's lecture (Nomair's Sections). You are highly encouraged to start your Due Date 2 implementations before Due Date 1.

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: If you have segmented your code into multiple subdirectories, then you should use the `zip` command with the `-r` option while inside the directory that contains the Makefile. This option recursively zips files in any subdirectories.

0. **This is 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 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`.

   Please follow these instructions carefully. Failure to do so results in long delays.

1. There are many different ways to represent lists of data when programming; you have used at least linked lists and arrays in the past. While we may choose to represent a list one way or another in our own code, we'd still like to write code that other people can reuse irrespective of our choices. This involves creating a common interface that the different representations of a data structure support.

   In this question, you will implement classes which store lists of `int`s using different internal representations, and still provide a common interface so that elements can be inserted into, found in, and removed from each list without needing to know the type of the list in advance and how the functionality is implemented.

   You are to provide the class `List`, which serves as a common interface to all of these data types. All lists should support the following operations:

   - `int at(int index)`, which takes in a position in the list and returns the `int` stored at that position. This is equivalent to the `arr`[index] operation on an array `arr`. You may assume that `index` is a valid index in the list.
   - `int first()`, which returns the first element in the list. You may assume that the list is nonempty.
   - `int last()`, which returns the last element in the list. You may assume that the list is nonempty.
   - `int size()`, which returns the number of elements currently in the list.
   - `void add(int num)`, which adds a number to the list.
   - `int find(int num)`, which searches through the list for `num`. If found, it returns the index of the first occurrence of `num`. Otherwise, it returns `-1`.
   - `bool remove(int num)`, which searches through the list for `num`. If found, it removes the first occurrence and returns `true`. Otherwise, it returns `false`.

   You will support the `List` interface by implementing the classes: `Vector`, `UnsortedList` and `SortedList`.

   `Vector` should be implemented as an array whose size is adjusted as necessary to contain every element, and where `add` adds new elements to the end of the array and `remove` removes the selected element and immediately fills the gap that is created.

   `SortedList` and `UnsortedList` should be implemented as linked lists, with the **only** difference that `add` inserts elements into a `SortedList` in sorted order, whereas it inserts elements into an `UnsortedList` at the end of the linked list. The linked list should be implemented using the standard definition of a `Node` class.

   Note that while marmoset will test that your classes behave correctly, it cannot enforce that the classes are implemented according to the specifications (linked list or resizing array). Because of this, there is a significant handmarking component which will determine whether your classes actually followed the implementation requirements. Submissions that produce the desired outcome but do so without following the requirements mentioned above will be heavily penalized (including reducing marks already obtained by passing Marmoset test cases).

Your code will also be marked on good design, including maximal code reuse, good choice of privacy modifiers and the const keyword.

Provided Files: We have provided a `main` function in the file `main.cc` (located under a4/q1) which exercises some of the functionality of the interface. While no test cases are required for this question, it is your job to test your implementations beyond the expected behaviour of `main.cc`. We have also provided the files `vector.out`, `unsorted.out` and `sorted.out` which contains the redirected output when the provided `main` function executes, given a correct implementation of all the classes. Note, by definition, an `UnsortedList` and a `Vector` produce identical output.

Due on Due Date 2: Submit your solution. Your submission must NOT contain the file `main.cc` (or any other file containing a main function). Marmoset is programmed to reject submissions containing this file. We will automatically place our own `main.cc` in your submission directory. Your submission must contain at a minimum the files `vector.{h,cc}`, `sorted.{h,cc}` and `unsorted.{h,cc}` along with ANY other files you might have created. It is essential that you create the `vector.h`, `sorted.h` and `unsorted.h` headers such that if our code intends to use one of these implementations all it needs to do is include the appropriate header, i.e., no other includes should be necessary. You must also include a Makefile, such that issuing the command `make` will build your program and create an executable named `a4q1` (assume the presence of `main.cc` in the compilation directory).

2. (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://unixpapa.com/floodit/`. 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:

   - `Cell` - implements a single cell in the grid (see provided cell.h)
   - `Game` - contains the grid of cells and implements the game logic (see game.h)
   - `View` - abstract class to provide the interface for any display (see view.h)
   - `TextDisplay` - responsible for displaying the text version of the grid. `TextDisplay` "is a" `View` (see textdisplay.h)
   - `Controller` - responsible for getting user input and communicates the input to the Game. The controller also receives updates from `Game` and is responsible for communicating updates to the `View` accordingly (see controller.h and controller.cc)

   A `main.cc` is also provided.

   Your solution to this problem must employ the Model-View-Controller (MVC) pattern for its implementation. MVC is a combination design pattern employing the Observer pattern as part of its implementation. The basic idea behind MVC is to break the application into three parts:

Model : The Model is the data, which in this case is the game state and consists of the `Cell` and `Game` classes.

View : The View is the display, which in this case consists of the `TextDisplay` and `GraphicDisplay` (see part b) classes.

Controller : The Controller receives user input and is responsible for mediating between the Model and View. The Model and View should not directly communicate, which facilitates designing reusable Model and View classes. **For this question, we have provided you with an almost complete implementation of the controller (see controller.h and controller.cc)**.

The `Game` contains a grid of `Cell`s. Each `Cell` is an observer of its neighbours (that means that class `Cell` is its own observer)[1]. `Game` 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 `Game` 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 (to prevent infinite notification loops). Each time a `Cell` changes state, it must notify `Game` of its new state which in turn will notify any views.

The Model does not know about the View or Controller. However, other objects can ask the Model to send them notifications when the Model is updated. In our implementation, the `Game` class (part of the Model) sends notifications by using a `GameNotification` object. Any object of type `GameNotification`, i.e. any object that "is a" `GameNotification`, can register with a `Game` in order to be informed whenever a `Game`'s state is updated. Specifically, `GameNotification` is a simple abstract class that provides a known interface for the `Game` object to send notifications. Every time the state of a `Cell` is updated, it sends an update to the `Game` which sends an update to the registered object. Note, we have provided you the `GameNotification` class.

For this question, the `Controller` should register with the `Game` to be the `GameNotification` object. Hence, the `Controller` "is a" `GameNotification`. When the `Game` notifies the `Controller` of an update, the `Controller` will then communicate these updates to the appropriate `View` (i.e., `TextDisplay` and/or `GraphicDisplay`).

The `View` class declares a pure virtual `print` method. Calling `TextDisplay::print` prints the grid to the screen (see example below).

When you run your program, it will listen on stdin for commands. The program accepts the following commands:

- `new n` Creates a new $n \times n$ grid, where $n \geq 2$. 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. If the triple `r c s` refers

[1] Each `Cell` has at most four neighbours.

to invalid co-ordinates, the triple is ignored. When initialization mode ends, the board should be displayed.

- `include f` The file `f` is a list of cell initializations of the same format of initialization from init. Reading from `f` will end either when end-of-file is reached or a -1 -1 is read. Include is called independently of init.
- `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 ($g > 0$). `game` and `new` 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.

Note: notice that most of the above has already been implemented in controller.cc. There are a few features left for you to implement. These are clearly marked in controller.cc with TODO comments.

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

```
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: main.cc cell.h controller.h controller.cc game.h view.h textdisplay.h

Due on Due Date 1: No test cases are needed for this question. However, you might still want to create test cases as it is considered good practice. Create and submit a UML diagram for the program you are asked to create for Due Date 2. All information needed to create this diagram is available in the files provided above. You need not include main.cc in your UML. Name the file `q2UML.pdf`.

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) **This is an OPTIONAL bonus question worth a 5% bump to your A4 marks**. If you feel like you might run short of time, complete the required questions first.

In this problem, you will adapt your solution from problem 2 to produce a graphical display. The solution to Problem 2 part (a) could be executed as:

```
./flood
```

In part (b), the program can take an optional parameter, a string `-graphics`

```
./flood -graphics
```

This should produce a graphical display for the game (the game is still controlled through the keyboard). The textual display continues to work as before. Also note that if your solution to part (b) is executed without the command line argument, it should have the exact same behaviour as your solution to part (a).

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. Instructions on installing an XServer were provided to you in the Getting Started PDF document made available through Piazza at the beginning of the term. It can also be found in the repository. To test your X connection, see if you can run the program `xeyes`.

Also (if working on your own machine) make sure you have the necessary libraries to compile graphics. From within the a4/graphicsdemo directory, try executing the following:

```
g++ window.cc graphicsdemo.cc -o graphicsdemo -lX11
./graphicsdemo
```

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

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
```

You know that the above test is successful if the following happens:

- Two windows open
- The big window prints the strings Hello!, ABCD, Hello! followed by rectangles containing a rainbow of colours
- The small window prints ABCD

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 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.

The one additional class you will need to create is:

- `GraphicDisplay` - responsible for displaying the graphical version of the grid using Xwindow. `GraphicDisplay` "is a" `View` and will therefore inherit the `notify` and `print` pure virtual methods. Note that since this class will implement a graphical display for the game, the `print` method need not do anything (but still needs to be implemented for the class to be concrete).

To make your solution graphical, you should carry out the following tasks:

- Alter you `main` function to accept the command line argument (see comment in main.cc)
- add a field to the `Controller` class representing the pointer to a `GraphicDisplay` object, so that it can be updated when the game state changes. Some hints on where to make changes are given in the controller.cc file.

- create the `GraphicDisplay` class. `GraphicDisplay` should accept update notifications about each cell in its display and update the corresponding rectangle of the display (based on the coordinates and colour) accordingly.
- the `GraphicDisplay` class should have a pointer to the `Xwindow` class so it can draw to the display.
- When `Controller` is notified of an update, it should send a corresponding notification to `GraphicDisplay`.
- The program should accept the additional 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 (and any other options that you might have had to use to get the graphicsdemo to work for your machine). For example:

```
g++ *.cc -o flood -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: Additional files provided: 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`. Note that Marmoset runs on the student environment. Test your Makefile to make sure that it works in this environment.

3. 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 are also provided a fully implemented mainline program for testing your text processor (`main.cc`). Note that the mainline program will not compile unless the header files mentioned below are created.

   You are not permitted to modify the two given classes or `main.cc` 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 the length of some word, the entire word is dropped (though the following word is `not` affected). Files to create: `dropfirst.h` and `dropfirst.cc`
- `DoubleWords` Double up all words in the string. Files to create: `doublewords.h` and `droublewords.cc`
- `AllCaps` All letters in the string are presented in uppercase. Other characters remain unchanged. Files to create: `allcaps.h` and `allcaps.cc`
- `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. Files to create: `count.h` and `count.cc`

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

The mainline interpreter loop has already been implemented and works as follows:

- You issue a command of the form `source-file list-of-decorators`. If `source-file` is `stdin`, then input is 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
12o
34o
r5d
r6d
sample.txt allcaps
HELLO
WORLD
```

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

Provided Files: Sample test cases can be found under a4/q3/sample-test. Starter code can be found under a4/q3/starter-code

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. Your submission must NOT contain the files `main.cc`, `echo.{h,cc}` and `textprocess.{h,cc}`. Marmoset is programmed to reject submissions containing any of these files. These files will be automatically placed in your submission directory. Your submission must contain at a minimum the files `doublewords.{h,cc}`, `allcaps.{h,cc}`, `dropfirst.{h,cc}` and `count.{h,cc}` along with ANY other files you might have created. You must also include a Makefile, such that issuing the command `make` will build your program and create an executable named `a4q3` (assume the presence of `main.cc`, `echo.{h,cc}` and `textprocess.{h,cc}` in the compilation directory).