Assignment #4: Polymorphism and Design Patterns

Due Date 1: Friday, November 5th, 2021, 5:00 pm EST
Due Date 2: Friday, November 12th, 2021, 5:00 pm EST
Online Quiz: Monday, November 15th, 2021, 5:00 pm EST

Topics that must have been completed before starting Due Date 1:

1. Relationships and Inheritance (all the module)
2. STL (all the module)
3. Object-Oriented Programming: Advanced object uses

Topics that must have been completed before starting Due Date 2:

1. Error handling with exceptions (all the module)
2. Design patterns: Observer
3. Design patterns: Decorator

Learning objectives:

- Relationships, Inheritance, and Polymorphism in C++
- STL vectors
- Decorator and Observer design patterns

- **Questions 1, 2a, and 3a are due on Due Date 1; questions 2b and 3b are due on Due Date 2. You must submit the online quiz on Learn by the Quiz date.**
- On this and subsequent assignments, you will take responsibility for your own testing. This assignment is designed to get you into the habit of thinking about testing *before* you start writing your program. If you look at the deliverables and their due dates, you will notice that there is *no* C++ code for Q2 and Q3 due on Due Date 1. Instead, you will be asked to submit test suites for C++ programs that you will later submit by Due Date 2.

  Test suites will be in a format compatible with that of the latter questions of Assignment 1, so if you did a good job writing your `runSuite` script, that experience will serve you well here.
- Design your test suites with care; they are your primary tool for verifying the correctness of your code. Note that test suite submission zip files are restricted to contain a maximum of 40 tests. The size of each input (`.in`) file is also restricted to 300 bytes. This is to encourage you not to combine all of your testing eggs in one basket. There is also a limit for each output file (`.out`), but none of your tests should be able to create such a large output file that you would normally encounter it.
- You must use the standard C++ I/O streaming and memory management (MM) facilities on this assignment; you may **not** use C-style I/O or MM. More concretely, you may `#include` the following C++ libraries (and no others!) for the current assignment: `iostream`, `fstream`, `sstream`, `iomanip`, `string`, `utility`, `stdexcept`, and `vector`. Marmoset will be set to **reject** submissions that use C-style I/O or MM, or libraries other than the ones specified above.
- 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`. This means that you must use vectors instead of variable-length arrays.

- We will manually check that you follow a reasonable standard of documentation and style, and to verify any assignment requirements that are not automatically enforced by Marmoset. Code to a standard that you would expect from someone else if you had to maintain their code. Further comments on coding guidelines can be found here: `https://www.student.cs.uwaterloo.ca/~cs246/F21/codingguidelines.shtml`
- We have provided some code and sample executables in the subdirectory `a4`. **These executables have been compiled in the CS student environment and will not run anywhere else.**

Note: We suggest creating the directory `~/cs246/f21/a4` and creating all of your assignment solutions in this directory.

Practice Question

Question 1 is a coding practice question and may be publicly discussed on Piazza. Students can publicly discuss solution strategies so long as solutions are not revealed. Publicly sharing any part of your code is not allowed.

Question 1

(40% of DD1) In this exercise you will write a C++ program that can mutate and update a number sequence composed of addition, multiplication, subtraction, and division operations. Your program should define the sequence to have an initial value of 0 and be defined by the following function (known as the *identity function*):

$$f(x) = x$$

Your program should read characters from standard input, where each of the following characters defines a command telling your program to do something:

- `'s'` - if the character `'s'` is read your program should next read an `int` from the stream and reset the **s**equence being maintained to the identify function with an initial value of the `int` read from stream.
- `'n'` - if the character `'n'` is read your program should update the sequence to the **n**ext number in the sequence and print that value out.
- `'+'` - if the character `'+'` is read your program should next read an `int` $readIn$ from the stream and add a new operation to function that defines the sequence of adding $readIn$ to the value. That is if your old sequence defining function was $f(x)$ then your new sequence defining function becomes $g(x) = f(x) + readIn$.
- `'*'` - if the character `'*'` is read your program should next read an `int` $readIn$ from the stream and add a new operation to function that defines the sequence of multiplying the value by that $readIn$. That is if your old sequence defining function was $f(x)$ then your new sequence defining function becomes $g(x) = f(x) * readIn$.
- `'-'` - if the character `'-'` is read your program should next read an `int` $readIn$ from the stream and add a new operation to function that defines the sequence of subtracting $readIn$ from the value. That is if your old sequence defining function was $f(x)$ then your new sequence defining function becomes $g(x) = f(x) - readIn$.
- `'/'` - if the character `'/'` is read your program should next read an `int` $readIn$ from the stream and add a new operation to function that defines the sequence of dividing the value by $readIn$. That is if your old sequence defining function was $f(x)$ then your new sequence defining function becomes $g(x) = f(x)/readIn$.

All arithmetic is done on integers. For example given the following input:

```
s 1
n
n
+ 5
n
n
```

```
* 10
n
n
```

you would generate the following output:

```
1
1
6
11
160
1650
```

Because after the command `s 1` the function is set to $f(x) = x$ and the current value is set to 1. The two `n` commands say to apply f to the current value, update the current value to the result, and print it. However these each print 1 because $f(1) = 1$, and thus $f(f(1)) = f(1) = 1$. Then after the command `+ 5` the function now becomes $f(x) = x + 5$ so the next two `n` commands produce 6 and 11. Then after the `* 10` command the function becomes $f(x) = (x + 5) * 10$ so the next two `n` commands produces 160 and 1650.

Submission:

Due on Due Date 1: Submit `a4q1.zip` to Marmoset with all the source code files needed to build your program and a `Makefile` that builds your program and names it `a4q1`.

Assessment Questions

Questions 2 and 3 are part of the coding assessment and may be publicly discussed on Piazza only for general clarifications. Solutions cannot be publicly discussed nor revealed.

Question 2

(60% of DD1; 60% of DD2)

In this question you will be building a fun ASCII art drawing and animation program! You'll start with a blank canvas, and will draw rectangles consisting of characters of your choosing on that canvas (remember that any point is a 1x1 rectangle, so you can draw anything!), and choose how your drawing will evolve with time.

The key to the functioning of this program is the Decorator pattern. A blank canvas will respond with a blank space to any query about the contents of a cell at a particular row or column. However, it can be decorated by rectangles, which will respond in their own way to the same query, or will pass it along to the ASCII art element behind them. Moreover, the key to the animation aspect of this program is that the response to a query for a cell can be time-varying. So a query to a cell actually involves three parameters: the row, the column, and the tick count (where the latter is the number of rendering events that have occurred since the last time the animation was reset).

You are to support the following kinds of ASCII art elements:

- *filled box:* A simple solid rectangle, specified by top and bottom row (inclusive) and left and right column (inclusive). Does not move.
- *blinking box:* A box that is displayed when the tick count is even, and is transparent otherwise.
- *moving box:* A box that moves one position (either up, down, left, or right) with each tick.
- *mask box:* A box that is invisible unless there is a visible art element beneath it. In that case, the part of the mask that lies above the art below becomes visible and covers that art. Does not move.

The provided starter code comes with a test harness that supports the following commands:

- `render` — causes the current artwork to be displayed, with a frame around it, so that the boundaries are clear (see the provided executable for details). Counts as a tick.
- `animate n` — renders the artwork `n` times.
- `reset` — resets the tick count to 0.
- `filledbox t b l r c` — adds a filled box with given top, bottom, left, right boundaries, filled with the character `c` (assumed to be non-whitespace and printable ASCII in the range 0–127). Invalid if top exceeds bottom or left exceeds right.
- `blinkingbox t b l r c` — adds a blinking box, with parameters understood as above.
- `movingbox t b l r c dir` — adds a moving box, with first five parameters understood as above and a direction `dir`, which can be one of `u d l r`. **Clarification:** These parameters are understood to be the position of the box when the tick count is 0. If the tick count is not 0 when the moving box is placed, it will show up shifted by a number of positions equal to the current tick count.
- `maskbox t b l r c` — adds a mask, with parameters understood as above.

Note: Some implementation notes for this problem can be found in the provided `README.txt`. Be sure to read and follow it!

Important: As the point of this problem is to use the Decorator pattern, if your solution is found in handmarking to not employ the Decorator pattern, your correctness marks from Marmoset will be revoked.

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: Submit your test suite (`suiteq2.txt` and all necessary files) in the file `a4q2a.zip`.

Due on Due Date 2: Submit your solution in the file `a4q2b.zip`. You must include a `Makefile`, such that issuing the command `make` will build your program. The executable should be called `a4q2`.

Question 3

(0% of DD1; 40% of DD2) Note: This problem asks you to work with XWindows graphics. Well before starting this question, make sure you are able to use graphical applications from your student.cs 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. If you are using a Mac, download and run XQuartz, and pass the -Y option when making ssh 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
```

Be sure to add the `-lX11` option to the linking line of your Makefile (that's lowercase-L, X, eleven, and it goes at the **end** of the linking line in your Makefile).

This question extends the previous question, in which you wrote an ASCII art animator program. In that problem, you had a studio object that managed the tick count, and was responsible for rendering your artwork to an output stream (`std::cout`). In this problem, we'll take that second responsibility away from the `Studio` class by employing the Observer pattern. Your `Studio` class will become a concrete "subject" class in this new model.

If you wish to actually see your artwork, you will need to create an observer object and attach it to the subject. But why stop at one? You could, in fact, create several observers! (And why should they all be necessarily text-based? More on that in a bit.) A bunch of observers all observing and rendering the same text is boring. But who says they all have to be watching the same part of your artwork? And who says your artwork has to be limited to just a 10x10 grid?

No, in this problem, your art has rows and columns spanning **the entire range of 32-bit integers!** But each of your observers, of course, will only see a much smaller portion of it.

In this problem, you will restructure your solution so that, in addition to following the Decorator pattern, it also conforms to the Observer pattern. As well, incorporate the other changes described above.

The test harness for this problem includes all of the commands from the previous problem. Those whose meanings have changed are described below, along with the new commands specific to this problem:

- `render` — causes all observers to render their images. Images should be displayed in the order the observers were registered with the subject.
- `addtext t b l r` — adds a text-based observer watching a portion of the image with the given top, bottom, left, right boundaries. Invalid if top exceeds bottom or left exceeds right. This observer, when notified, will display its view to `std::cout`.
- `addgraphics t b l r` — adds a graphical observer, with parameters understood as above.

For the graphical observers, you will use the `Xwindow` class that has been provided to you. **Make sure you have a working XWindows server running on your computer, such that you can run graphical applications over an SSH connection. Test this EARLY.** Your windows will be of size $10r$ by $10c$, where r is the number of rows being observed, and c is the number of columns being observed. Represent the rendered objects by colour-coded 10x10 squares:

- A red square will represent any lower-case letter.
- A green square will represent any upper-case letter.
- A blue square will represent any digit.
- A black square will represent any other printable character.
- A white square will represent no character.

If you have difficulty distinguishing colours, such that this portion of the assignment would be difficult for you to complete, then please contact an ISA to request an alternative means of completing the graphical display.

Note: Some implementation notes for this problem can be found in the provided `README.txt`. Be sure to read and follow it!

Important: As the point of this problem is to use the Decorator and Observer patterns, if your solution is found in handmarking to not employ these patterns, your correctness marks from Marmoset will be revoked.

Something to think about: In the Observer pattern, the subject does not own its observers, and is not responsible for deleting them. Therefore, you must think about who is going to be responsible for the observers that you create and how they should eventually be deleted.

Something else to think about: You will probably find that your graphical display, when running anywhere other than on campus, will be quite slow. Think about how you might optimize the performance of your graphics observer, so that it renders more quickly. This is not an assignment requirement, but something to think about for your own interest.

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: Get X11 forwarding working on your system. When you have done so, submit a text file `a4q3a.txt` with the contents (on one line, with a newline at the end) `I successfully ran xeyes.` for one mark towards Due Date 2. You do not have to submit testing for this problem.

Due on Due Date 2: Submit your solution in the file `a4q3b.zip`. You must include a `Makefile`, such that issuing the command `make` will build your program. The executable should be called `a4q3`.