Interactive Fiction Project, Part 5

Due 7/19/2019

June 20, 2018

1 Description

In this part of the project, you and your partner will finish implementing the code for the interpreter. In addition, you will reflect on your design for part 3 and evaluate which parts did or didn't work, what elements were missing, etc. Finally, each of you will submit a brief peer evaluation. The specifications for your interpreter are described in Section 5, while the reflection and peer evaluation are described in sections 2 and 3, respectively.

2 Design reflection

Before submitting your final code, you will write a reflection on the differences between your class design for Project part 2 and your final implementation of the IF interpreter. In the report, you should address each change that you made to your class design, including *what* was changed about your design and *why* this change was made. Each of these changes should be related back to some feature of the interpreter that your original design lacked, would have had trouble implementing, or would have been unnecessarily complicated. You should organize changes that were made for the same (or closely related) reasons into paragraphs. Simple changes (requiring no more than a sentence to explain and justify) can be included as a bulleted list at the end of the report.

The report should be typed, in 12pt Times New Roman font. There is no length requirement for the report, but it should fully address the differences between your class design from Project Part 3 and your implementation. Your reflection should be submitted in PDF format. Most word processors have the ability to export files to PDF documents, which may be an option under Save as... or Export.

3 Peer evaluation

In addition to the group submission, every group member should submit a peer evaluation. The peer evaluation will be worth 30 points of the lab grade. This evaluation should include your name, your partner's name, what each of you

contributed to the project, contribution scores for both of you, and any concerns you've had working with your partner. **Note**: in addition to the information in the peer eval for part 2, your evaluation should also describe what parts of the code each of you implemented.

The contribution scores should be integers in the range 0–10, and the two scores should add up to 10. For example, if you feel that the work was split equally between you, you should assign 5 to both. Or, if you feel you did the majority of the work, you might assign yourself 8 and your partner 2. If you are assigned to a group of three, you should include the same information for everybody, but your scores should be in the range 0–15 and add up to 15 (equal contribution is 5-5-5).

4 Submission instructions and grading

You should submit your source code and design reflection to the "Project part 5 (Group)" assignment on Canvas, and you should submit your peer evaluation to the "Project part 5 (peer eval)" assignment. Your code and reflection should be submitted as a zip archive, with the reflection in PDF format. Your code should be able to be compiled with `g++` using the command

```
g++ *.cpp -o if
```

Code	**70 points**
Compiles	
Able to display IF stories of increasing sophistication	
Good coding style	
Design reflection	**30 points**
Peer eval	**30 points**

5 Specifications

This section will detail the various components of an interactive fiction story. Much of this information will be syntactic and repeat information from the previous assignment, though these sections will also detail how these parts should act and connect together to form a story. Your implementation for this project should support the features described below.

5.1 The interpreter

Your IF interpreter should start by opening the file `if.html` and reading in the story data, as described in the sections above. You may assume that the input does not contain any syntax or logical errors (e.g., testing a variable in an `(if:)` command before it has been `(set:)`). Once it has constructed objects to represent the structure of the story, the interpreter should start by displaying the first passage defined in `if.html`. When displaying a passage, your program should execute all commands appropriately and properly display all links.

Note that your interpreter will need to keep track of the variables that have been defined, as well as their values. You should use an `unordered_map` object for this purpose (`#include <unordered_map>`); you do not need to define a specialized class to match variables to their values. We will cover `unordered_map`s more when we discuss the Standard Template Library in class.

After displaying a passage, the interpreter should print out a numbered list of all links in the passage, and prompt the reader to select one.

When displaying a passage, your program should execute any commands that appear in that passage appropriately and properly display all links (see Section 5.3 below). Afterwards, the interpreter should print out a numbered list of all links in the passage and prompt the reader to select one. This list should start numbering at 1; e.g.,

```
1. Take the red pill
2. Take the blue pill
```

Once the reader has selected a link, your interpreter should display the corresponding passage (see Links section below). This should continue until the reader reaches a passage with no links, at which point the interpreter should terminate.

5.2 Passages

Interactive fiction works are divided into passages, which appear in the HTML tags `<tw-passagedata>`. Each passage will start with `<tw-passagedata ...>` and will end with `</tw-passagedata>`. In addition to starting with `<tw-passagedata`, the opening tag will specify some attributes, like `pid`, `name`, `tags`, and `location`, and the body of the passage will be between the opening and closing tags.
Example passage:

```
<tw-passagedata pid="1" name="start" tags="" location="100,100">
The body of the passage will be here.
</tw-passagedata>
```

Your interpreter only needs to pay attention to the `name` attribute of each passage, the other attributes can be safely ignored. In the body of a passage, there are 3 different types of things to deal with: text, links, and commands. When a passage is being displayed, text should appear as typed (including spacing), however, links will display differently than they appear in the input file, and commands can cause a variety of different things to happen.

5.3 Links

Links in the body will be denoted by double brackets: `[[` and `]]`. Links are treated differently depending on whether or not they contain the characters `-&gt;`. A link that doesn't contain these characters should appear in the text without the double brackets, and this link should be presented to the reader as

an option to further the story after the passage has displayed. When selected, the link should display the passage with the name that matches the link text.

A link that contains the characters `->` should display as the characters to the left of `->`; however, it should link to the passage whose name matches the characters to the right of the `->`.
Example links:

```
[[Simple]]
```

Displays as "Simple"; links to passage named "Simple"

```
[[Take the blue pill-&gt;Bad dream?]]
```

Displays as "Take the blue pill"; links to passage named "Bad dream?"

5.4 Commands

Commands are denoted by a single word and a colon immediately after an open parenthesis. Your IF interpreter should support 5 different commands, `(go-to:`, `(set:`, `(if:`, `(else-if:`, and `(else:`.

5.5 Go-to command

The go-to command should cause the program to immediately change to the given passage. Any other text, commands, or links that appear after the go-to command should be ignored, and any links before the go-to can be safely ignored, as though the reader had selected the given passage as their choice. The name of the passage to go to will appear between two copies of `&quot;`.

Example go-to command:

```
(go-to: &quot;start&quot;)
```

Continues the story with the passage named "start", as though the reader had selected a link leading to this passage.

5.6 Set command

The `(set:` command allows the IF author to define and set the value of a variable. Note that the `(set:` command will never display any text; however, it will execute any time a passage containing it is displayed. Variables that do not exist are created, while variables that do exist are updated. While the full specification for Harlowe allows for three different types of variables (numeric, string, and Boolean), you should treat all variables as Boolean.

The first word after the colon in the `(set:` command will be a variable name, which always starts with $. The second will be the keyword `to`, and the third will be the assigned value (`true` or `false`).[1]
`Example set command:`

[1] While Harlowe allows much more sophisticated versions of this syntax, including arithmetic operations (+, −, etc.), relational operators (<, ≤, etc.), and logical connectives (and, or, not), we will only support simple assignments.

```
(set: $ateCake to true)
```

Stores `true` as the value of the `$ateCake` variable

5.7 If/Else if/Else

The `(if:`, `(else-if:`, and `(else:` commands act much like they do in C++. The `(if:` and `(else-if:` commands will be followed by a variable, the keyword `is`, and a value to test the variable against, followed by the closing parenthesis. `(else:)` has no condition.

The blocks that if, else-if, and else apply to are denoted by brackets `[]`. **Note:** links and other commands (including other `(if:` commands) may be embedded in these blocks. While PassageTokenizer extracts these blocks as a single token, you can create another PassageTokenizer to parse the contents of the block.

`Example if command`

```
(if: $ateCake is true)[You are quite full.]
(else-if: $ateCookie is true)[You sigh contentedly.]
(else:)[You still feel a mite peckish.]
```

6 Project recommendations

I strongly recommend that you divide up the classes with your partner so that you can finish the project faster. In order to do this, though, you will need to meet up and decide who will implement what, and you will also need to make sure that you understand exactly what all of the member functions you are implementing should do.

As with Project Part 4, you are strongly encouraged to test code as you go, rather than waiting until the end to start testing. Writing a test harness will make it easier to unit test every member function you implement, though a reasonable set of test cases can also work. You might first test if your program can display a story that has only text, then one that just has text and links, then another that has a goto, then several that have various types of if and set commands. At each step, you'll want to think about the fewest number of functions you can implement to test the next most sophisticated test case, then implement and test those.

You have been provided with couple of example stories to get you started, though you are welcome to type up your own, or develop new stories using Twine 2: `https://twinery.org/2/`.