Benjamin Richard Olson
March 6, 2014
Introduction to Computer Science 1
Final Project: DocuWiki to Json Document Converter

PROBLEM:

I need to convert a group of DocuWiki text files (that have the exact format of the example below) representing stories as picture-text pairs, to Json for the purpose of displaying the stories in HTML pages.

The expected format of each DocuWiki input file, <story>.txt, is as follows:

====== Title ======

{{dir:subdir:...pic1.jpg?nolink&640x360}}

Test corresponding to above picture link.

{{dir:subdir:...pic2.jpg}}

Test corresponding to above picture link.
...and so on, to the end of the story, followed by...

//footer text//

SPECIFICATION/EXPECTED PROGRAM BEHAVIOR:

The DocuWiki example above, <story>.txt, should be converted to the following Json file, saved as <story>.json in the working directory:

```
{
        "title":"Title",
        [
                {"picture":"dir/subdir/pic1.jpg?nolink&640x360",
                 "text":"blah, blah, blah"},
                        ...
        ],
        "footer":"footer text"
}
```

In order to do this, the program will be run as a command-line utility in the following manner:

program_name dir/story.txt (...optional additional text files)

On input error, exit with error message indicating what caused the error and ask user to rerun the program.

program_name (this outputs a messsage to the user about how to run the program)

GENERAL DESIGN (See final_project_design.pdf for a more thorough design):

If argv[1] is "-help" or argc is 1 (no arguments given), display help message to user.
For each file name given on command line, do the following (and :

1. Check file name for extension .txt (exit with error if extenion is not .txt).
2. On valid filename, try opening the file (exit with error if unsuccessful).
3. Parse file, using a parsing object (container of parsing functions), into struct representing the data (exit with error if docuwiki syntax contains errors).
4. Iterate/traverse through this object data to create a json string.
5. Output thes create string into a new, formatted json text file.
6. Output message to user listing the json text files successfully created.
7. Ask user to randomly choose one of the json file names to view its contents or exit the program.

TESTING SUMMARY (See final_tests.cpp and test files - DocuWiki input and Json output - for details):

Testing was done throughout development phase.
First some functions were tested separately in final_tests.cpp.
Then, the complete program was tested using a few different combinations of text file and CLI input, to test for successful file conversions as well as error handling of CLI input and input file syntax errors.

Testing of Separate Functions in final_tests.cpp revealing errors:

string processSyntax(string textLine, int x, char c):

Test 3: returned " Title " but should exit with error
Test 4: returns "Title" but should exit with error
Resolution: shifted an else-if statement up in the code

Final test cases performed on complete program (refer to *.txt test files, which are read into the program):

Scenario 1: Test input handling of input file (DocuWiki text file)that contains a syntax error.
Scenario 2: Test input handling of incorrect filename given on CLI.
Scenario 3: With no ClI errors or errors in the DocuWiki file itself, test successful conversion to Json file.
Scenario 4: Test a run of the program that produces some Json files but then exits with an input file syntax error.
Scenario 5: Test a run of the program that prints a help message on command line usage and then exits.

REFLECTION:

DESIGN:

I have made notes here from beginning to end, from discovering and defining the problem to a fully implemented solution. The most significant difference between this project and the previous projects was that I discovered the problem myself and defined requirements myself for how the program should behave. The second most signifcant thing to mention was that I designed test scenarios differing by the nature of CLI and file input, to verify that the program behaves correctly in each scenario. Thirdly, I spent more time designing in hopes to spend less time coding.

To start the project, I defined the problem and detailed how I wanted the finished program to behave, with expected output file given an example input file.

I made some changes to the design during implementation because I realized some elements of my design were not practical. In the DocuWikiParser class I chose not have a bool to keep track of whether the text being parserd was DocuWiki syntax or an actual text value to be stored, becuase it would be redundant with int ID used in a switch statement to figure out what kind of file line was being parsed.

Also, I had planned to have two overloaded getData functions, but then realized it would be more concise to have only one and eliminate its formal argument, string syntaxToken, unecessary because int ID essentially does the same thing - that is - to keep track of what kind of line is being parsed (title, picture directory, text, or footer...given IDs 1, 2, 3, and 4, respectively) in the getData function. Then I added a recursive function which was also overloaded with another function in the class, both named "processSyntax." I realized it would alse be more concise to remove the function syntaxID() because its definition would be very short and its usage was restricted to the switch statement in "DocuWikiParser::getData(string text)." As I was implementing processSyntax, though, I realized I didn't need to overload that function either. During the implementation, I also realized that there were a few other functions I needed to write that were not included in the design.

To help me keep my code organized, I had the mindset of limiting the scope of as many variables as I could (as many variables as possible local to a function) to minimize the possibility of conflicts and reuse variable names from one function to the next.

DEBUGGING:

Most useful was simple cout statements to catch all three kinds of errors - syntax, run-time, and logical errors. The cout statements helped me locate where in the code the error was being produced and also confirm that values being produced or changed was happening as I expected. Secondly, running compiled code helped me catch run-time and logical errors, so once again, thorough testing throughout the lifecycle of this project proved to be most valuable in guiding me through the process to the final product. I believe I spent as much or more time testing and debugging as I did writing code.

REFERENCE TO LOCATE FULFILLED PROJECT REQUIREMENTS:

1. Binary: line 477...
2. Two's Compliment: line 477...
3. Pre-Defined Macro: lines 151, 198, 207, 256, 279, 290, 312, 366, 394
4. Simple Output: throughout
5. Simple Input: line 163
6. Type Casting: lines 438, 452, 526
7. Conditionals: throughout
8. Logical Operator: throughout
9. Loops: throughout
10. Random Number: lines 464, 465
11. Error Categories: (see "DEBUGGING" under "REFLECTION")
12. Debugging: lines 116-121, 140, 200, 202, 226, 231, 237, 242, 247, 270, 283, 284, 316, 317, 373, 397, 401, 411, 416, 528, 529
    and final_function_tests.cpp and final_tests.txt
    (also see "DEBUGGING" under "REFLECTION")
13. Functions: throughout
14. Functional Decomposition: seen in main(), line 111
15. Variables Scope: use of local variables
16. Pass-by-value: throughout
    Pass-by-reference: throughout
17. Function Overloading: lines 265... and 299...
18. String Variables: throughout
19. Recursion: lines 265... and 299...
20. Multidimensional Array: lines 71 and 402, 407, 412, 445, 446, 448, 449
21. Dynamically Declared Arrays: lines 131, 191, 426, 428, (170, 171)
22. Command Line Arguments: lines 75, 187...
23. Struct: lines 68..., 399-412
24. Define Class: lines 214-352
    Create Object: line 66
25: Pointer to an Array: line 127
26. Pointer to a Struct: line 125
27. Pointer to an Object: line 112
28. Does something awesome: File reading/parsing/writing