Programming Assignment #1: DupeyDupe

COP 3502, Spring 2019

Due: Sunday, January 20, *before* 11:59 PM

Abstract

In this assignment, you will write a *main()* function that can process command line arguments – parameters that are typed at the command line and passed into your program right when it starts running (as opposed to using *scanf()* to get input from a user *after* your program has already started running). Those parameters will be passed to your *main()* function as an array of strings, and so this program will also jog your memory with respect to using strings in C.

On the software engineering side of things, you will learn to construct programs that use multiple source files and custom header (*.h*) files. This assignment will also help you hone your ability to acquire new knowledge by reading technical specifications. If you end up pursuing a career as a software developer, the ability to rapidly digest technical documentation and work with new software libraries will be absolutely critical to your work, and this assignment will provide you with an exercise in doing just that.

Finally, this assignment is specifically designed to require relatively few lines of code so that you can make your first foray into the world of Linux without a huge, unwieldy, and intimidating program to debug. As the semester progresses, the programming assignments will become more lengthy and complex, but for now, this assignment should provide you with a gentle introduction to a development environment and software engineering concepts that will most likely be quite foreign to you at first.

Deliverables

DupeyDupe.c

Note! The capitalization and spelling of your filename matter!

Note! Code must be tested on Eustis, but submitted via Webcourses.

1. Important Note: Test Case Files Look Wonky in Notepad

Included with this assignment are several test cases, along with output files showing exactly what your output should look like when you run those test cases. You will have to refer to those as the gold standard for how your output should be formatted.

Please note that if you open those files using an older version of Notepad on Windows, they will appear to contain one long line of text. That's because Notepad used to handle end-of-line characters differently from Linux and Unix-based systems. One solution is to view those files in a text editor designed for coding, such as Atom, Sublime, or Notepad++. For those using Mac or Linux systems, the input files should look just fine.

2. Overview

In this assignment, you will have to loop through an array of strings and print the *last* string in the array that is a duplicate of any other string in the array. For example, consider the following array of strings:

{ "apple", "banana", "coconut", "domination", "apple" "coconut", "banana", "cake" }

Three of those strings are duplicates (i.e., they appear more than once in the array): "apple", "banana", and "coconut". The very last string in the array that is a duplicate is "banana". If this array were passed to your program, you would have to print "banana" to the screen (followed by a newline character). Also, if there are two ***consecutive*** occurrences of the string "dupe" ***anywhere*** in the array, you will print "dupe dupe!" (followed by a newline character) on a second, separate line of output. If there are no duplicates in the array your program is processing, you'll simply print "no dupey dupe :(" to the screen (followed by a newline character).

The strings you process will be passed to your program as command line arguments. The process for setting up your program's *main()* function to accept command line arguments is described below in Appendix A ("Processing Command Line Arguments") (pg. 9).

3. DupeyDupe.h (*Super Important!*)

Your code for this assignment will go in a file named *DupeyDupe.c*. At the very top of that file, write a comment with your name, the course number, the current semester, and your NID. Directly below that, you **must** include the following line of code:

```
#include "DupeyDupe.h"
```

The "quotes" (as opposed to <brackets>) indicate to the compiler that this header file is found in the same directory as your source file, not a system directory. Note that filenames are case sensitive in Linux, so if you `#include "dupeydupe.h"` (all lowercase), your program might compile on Windows, but it won't compile when we test it. You must capitalize "DupeyDupe.h" correctly.

The *DupeyDupe.h* file we have included with this assignment is a special header file that will enable us to grade your program. If you do not *#include* that file properly, your program will not compile on our end, and it will not receive credit. Note that you should also *#include* any other standard libraries your code relies upon (such as *stdio.h*).

From this point forward, you will always have to have *DupeyDupe.h* in the same folder as your *DupeyDupe.c* file any time you want to compile your code. You should not send *DupeyDupe.h* when you submit your assignment, as we will use our own copy of *DupeyDupe.h* when compiling your code. You should also be very careful not to modify *DupeyDupe.h* while working on this assignment, except as described below on pg. 4.

4. Running All Test Cases on Eustis (*test-all.sh*)

The test cases included with this assignment are designed to show you some ways in which we might test your code and to shed light on the expected functionality of your code. We've also included a script, *test-all.sh*, that will compile and run all test cases for you.

Super Important: Using the *test-all.sh* script to test your code on Eustis is the safest, most sure-fire way to make sure your code is working properly before submitting.

To run *test-all.sh* on Eustis, first transfer it to Eustis in a folder with *DupeyDupe.c*, *DupeyDupe.h*, all the test case files, and the *sample_output* directory. Transferring all your files to Eustis with MobaXTerm isn't too hard, but if you want to transfer them from a Linux or Mac command line, here's how you do it:

1. At your command line on your own system, use *cd* to go to the folder that contains all your files for this project (*DupeyDupe.c*, *DupeyDupe.h*, *test-all.sh*, the test case files, and the *sample_output* folder).

2. From that directory, type the following command (replacing `YOUR_NID` with your actual NID) to transfer that whole folder to Eustis:

   ```
   scp -r . YOUR_NID@eustis.eecs.ucf.edu:~
   ```

   Warning: Note that the dot (".") in the command above refers to your current directory when you're at the command line in Linux or Mac OS. The command above transfers the ***entire contents*** of your current directory to Eustis. That will include all subdirectories, so for the love of all that is good, please don't run that command from your desktop folder if you have a ton of files on your desktop!

Once you have all your files on Eustis, you can run *test-all.sh* by connecting to Eustis and typing the following:

```
bash test-all.sh
```

If you put those files in their own folder on Eustis, you will first have to *cd* into that directory. For example:

```
cd DupeyProject
```

That command (*bash test-all.sh*) will also work on Linux systems and with the bash shell for Windows. It will not work at the Windows Command Prompt, and it might have limited functionality in Mac OS.

5. Checking the Output of Individual Test Cases

If the *test-all.sh* script is telling you that some of your test cases are failing, you'll want to compile and run those test cases individually to inspect their output. This section tells you how to do that.

There are two types of test cases included with this assignment: (1) the test cases where you compile your program and run it with the command line arguments listed in one of our text files (*arguments01.txt* through *arguments08.txt*), and (2) the test cases where you have to compile one of our source files (*UnitTest01.c* or *UnitTest02.c*) along with your source file (*DupeyDupe.c*) in order to run.

If you want to run one of these test cases individually in order to examine its output outside of the *test-all.sh* script, here's how you do it:

1. **Instructions for running your program with the command line arguments given in one of the *.txt* files:**

   a. Place all the test case files released with this assignment in one folder, along with your *DupeyDupe.c* file.

   b. In *DupeyDupe.h*, make sure line 14 is commented out <u>*exactly*</u> as follows, with no space directly following the "//". This is the <u>*only*</u> line of *DupeyDupe.h* that you should ever modify:

   ```
   //#define main __hidden_main__
   ```

   c. At the command line, *cd* to the directory with all your files for this assignment, and compile your program:

   ```
   gcc DupeyDupe.c
   ```

   d. To run your program and redirect the output to *output.txt*, copy and paste the contents of one of the *argumentsXX.txt* files to the command line after `./a.out`, like so:

   ```
   ./a.out chocolate lava cake lava cake > output.txt
   ```

   e. As an alternative to step (d), type the following to have the command line automatically insert the arguments from one of the text files for you:

   ```
   ./a.out $(cat arguments01.txt) > output.txt
   ```

   f. Use *diff* to compare your output to the expected (correct) output for the program:

   ```
   diff output.txt sample_output/arguments01-output.txt
   ```

2. **Instructions for compiling your program with one of the unit test cases:**

   a. Place all the test case files released with this assignment in one folder, along with your *DupeyDupe.c* file.

   b. In *DupeyDupe.h*, make sure line 14 uncommented and appears <u>*exactly*</u> as follows. This is the <u>*only*</u> line of *DupeyDupe.h* that you should ever modify:

   ```
   #define main __hidden_main__
   ```

   c. At the command line, *cd* to the directory with all your files for this assignment, and compile your program with *UnitTestLauncher.c* and any <u>*one*</u> of the *UnitTestXX.c* files you would like to test:

   ```
   gcc DupeyDupe.c UnitTestLauncher.c UnitTest01.c
   ```

d. To run your program and redirect the output to *output.txt*, execute the following command:

```
./a.out > output.txt
```

e. Use *diff* to compare your output to the expected (correct) output for the program:

```
diff output.txt sample_output/UnitTest01-output.txt
```

Super Important: *Remember, using the test-all.sh script to test your code on Eustis is the safest, most sure-fire way to make sure your code is working properly before submitting.*

6. Function Requirements

In the source file you submit, *DupeyDupe.c*, you must implement the following three functions.

Super Important! These are the only three functions you're permitted to write for this assignment. You cannot write any auxiliary functions (helper functions). Please be sure the spelling, capitalization, and return types of your functions match these prototypes exactly.

```
int main(int argc, char **argv)
```

Description: Process each command line argument (except for *argv[0]*, which contains the name of the program being executed), and print the expected output to the screen as described above in Section 2 of this document ("Overview"). You should also consult the output files included with this assignment to see examples of the exact output formatting expected in this assignment.

Note that the strings we pass to your program could be arbitrarily long. Note also that there are some special restrictions on how you can solve this problem (given on the following page).

Return Value: Your *main()* function should always return zero.

Related Test Cases: *arguments01.txt* through *arguments08.txt*

```
double difficultyRating(void)
```

Description: Return a double indicating how difficult you found this assignment on a scale of 1.0 (ridiculously easy) through 5.0 (insanely difficult). This function should not print anything to the screen.

Related Test Case: *UnitTest01.c*

```
double hoursSpent(void)
```

Description: Return an estimate (greater than zero) of the number of hours you spent on this assignment. Your return value must be a realistic and reasonable estimate. Unreasonably large values will result in loss of credit. This function should not print anything to the screen.

Related Test Case: *UnitTest02.c*

7. Special Restrictions (*Super Important!*)

You must abide by the following restrictions in the *DupeyDupe.c* file you submit. Failure to abide by any one of these restrictions could result in a catastrophic loss of points.

- ★ Please do not create any string variables or arrays in this assignment (other than the array of command line arguments itself). You should not create any new copies of the command line argument strings.
- ★ Do not read or write to files (using, e.g., C's *fopen()*, *fprintf()*, or *fscanf()* functions). Also, please do not use *scanf()* to read input from the keyboard.
- ★ Do not declare new variables part way through a function. All variable declarations should occur at the *top* of a function, and all variables must be declared inside your functions or declared as function parameters.
- ★ Do not use *goto* statements in your code.
- ★ Do not make calls to C's *system()* function.
- ★ Do not write malicious code, including code that attempts to open files it shouldn't be opening, whether for reading or writing. (I would hope this would go without saying.)
- ★ No crazy shenanigans.

8. Style Restrictions (*Super Important!*)

Please conform as closely as possible to the style I use while coding in class. To encourage everyone to develop a commitment to writing consistent and readable code, the following restrictions will be strictly enforced:

- ★ Any time you open a curly brace, that curly brace should start on a new line.
- ★ Any time you open a new code block, indent all the code within that code block one level deeper than you were already indenting.
- ★ Be consistent with the amount of indentation you're using, and be consistent in using either spaces or tabs for indentation throughout your source file. If you're using spaces for indentation, please use at least two spaces for each new level of indentation, because trying to read code that uses just a single space for each level of indentation is downright painful.
- ★ Please avoid block-style comments: /* *comment* */
- ★ Instead, please use inline-style comments: // *comment*
- ★ Always include a space after the "//" in your comments: "// *comment*" instead of "//*comment*"
- ★ The header comments introducing your source file (including the comment(s) with your name, course number, semester, NID, and so on), should always be placed *above* your *#include* statements.
- ★ Use end-of-line comments sparingly. Comments longer than three words should always be placed *above* the lines of code to which they refer. Furthermore, such comments should be indented to properly align

with the code to which they refer. For example, if line 16 of your code is indented with two tabs, and line 15 contains a comment referring to line 16, then line 15 should also be intended with two tabs.

★ Please do not write excessively long lines of code. Lines must be no longer than 100 characters wide.

★ Avoid excessive consecutive blank lines. In general, you should never have more than one or two consecutive blank lines.

★ When defining a function that doesn't take any arguments, always put *void* in its parentheses. For example, define a function using *int do_something(void)* instead of *int do_something()*.

★ When defining or calling a function, do not leave a space before its opening parenthesis. For example: use *int main(int argc, char **argv)* instead of *int main (int argc, char **argv)*. Similarly, use *printf("...")* instead of *printf ("...")*.

★ Do leave a space before the opening parenthesis in an *if* statement or a loop. For example, use use *for (i = 0; i < n; i++)* instead of *for(i = 0; i < n; i++)*, and use *if (condition)* instead of *if(condition)* or *if(condition)*.

★ Please leave a space on both sides of any binary operators you use in your code (i.e., operators that take two operands). For example, use *(a + b) - c* instead of *(a+b)-c*. (The only place you do <u>not</u> have to follow this restriction is within the square brackets used to access an array index, as in: *array[i+j]*.)

★ Use meaningful variable names that convey the purpose of your variables. (The exceptions here are when using variables like *i*, *j*, and *k* for looping variables or *m* and *n* for the sizes of some inputs.)

9. Deliverable (Submitted via Webcourses, not Eustis)

Submit a single source file, named *DupeyDupe.c*, via Webcourses. The source file should contain definitions for all the required functions (listed above). Be sure to include your name and NID as a comment at the top of your source file. Also, don't forget `#include "DupeyDupe.h"` in your source code (with correct capitalization). Your source file must work on Eustis with the *test-all.sh* script, and it must also compile and run on Eustis like so:

```
gcc DupeyDupe.c
./a.out chocolate lava lava lava cake
```

10. Grading

Important Note: When grading your programs, we will use different test cases from the ones we've release with this assignment, to ensure that no one can game the system and earn credit by simply hard-coding the expected output for the test cases we've released to you. You should create additional test cases of your own in order to thoroughly test your code. In creating your own test cases, you should always ask yourself, "What kinds of inputs could be passed to this program that don't violate any of the input specifications, but which haven't already been covered in the test cases included with the assignment?"

The *tentative* scoring breakdown (not set in stone) for this programming assignment is:

50%	Passes test cases with 100% correct output formatting. This portion of the grade includes tests of the *difficultyRating()* and *hoursSpent()* methods.
20%	Adequate comments and whitespace. To earn these points, you must adhere to the style restrictions set forth above. We will likely impose huge penalties for small deviations, because we really want you to develop good style habits in this class. Please include a header comment with your name and NID.
20%	Implementation details and adherence to the special restrictions imposed on this assignment. This will likely involve some manual inspection of your code.
10%	Source file is named correctly. Spelling and capitalization count.

Note! Your program must be submitted via Webcourses, and it must compile and run on Eustis to receive credit. Programs that do not compile will receive an automatic zero.

Your grade will be based largely on your program's ability to compile and produce the *exact* output expected. Even minor deviations (such as capitalization or punctuation errors) in your output will cause your program's output to be marked as incorrect, resulting in severe point deductions. The same is true of how you name your functions and their parameters. Please be sure to follow all requirements carefully and test your program throughly. Your best bet is to submit your program in advance of the deadline, then download the source code from Webcourses, re-compile, and re-test your code in order to ensure that you uploaded the correct version of your source code.

Additional points will be awarded for style (appropriate commenting and whitespace) and adherence to implementation requirements. For example, the graders might inspect your code to make sure you're not creating any arrays or extra copies of command line argument strings.

Start early. Work hard. Good luck!

Appendix A:
Processing Command Line Arguments

1. Guide to Command Line Arguments

All your interactions with Eustis this semester will be at the command line, where you will use a text-based interface (rather than a graphical interface) to interact with the operating system and run programs.

When we type the name of a program to run at the command line, we often type additional parameters *after* the name of the program we want to run. Those parameters are called "command line arguments," and they are passed to the program's *main()* function upon execution.

For example, in class, you've seen that I run the program called *gcc* to compile source code, and after typing "*gcc*," I always type the name of the file I want to compile, like so:

```
gcc DupeyDupe.c
```

In this example, the string "*DupeyDupe.c*" is passed to the *gcc* program's *main()* function as a string, which tells the program which file it's supposed to open and compile.

In this assignment, your *main()* function will have to processes command line arguments. This appendix shows you how to get that set up.

2. Passing Command Line Arguments to *main()*

Your program must be able to process an arbitrary number of string arguments. For example:

```
seansz@eustis:~$ ./a.out chocolate lava lava lava cake
lava
```

To get command line arguments (such as the strings "chocolate", "lava", "lava", "lava", and "cake" in the above example) into your program, you just have to change the function signature for the *main()* function you're writing in *DupeyDupe.c*. Whereas we've typically seen *main()* defined using `int main(void)`, you will now use the following function signature instead:

```
int main(int argc, char **argv)
```

Within *main()*, *argc* is now an integer representing the number of command line arguments passed to the program, including the name of the executable itself. So, in the example above, *argc* would be equal to 6. *argv* is an array of strings that stores all those command line arguments. *argv[0]* always stores the name of the program being executed (`./a.out`), and in the example given above, *argv[1]* would be the string "chocolate", *argv[2]* would be the string "lava", and so on through *argv[5]*, which is the string "cake".

3. Example: A Program That Prints All Command Line Arguments

For example, here’s a small program that would print out all the command line arguments it receives (including the name of the program being executed). Note how we use *argc* to loop through the *argv* array:

```
int main(int argc, char **argv)
{
   int i;

   for (i = 0; i < argc; i++)
      printf("argv[%d]: %s\n", i, argv[i]);

   return 0;
}
```

If we compiled that code into an executable file called *a.out* and ran it from the command line by typing `./a.out lol hi there!`, we would see the following output:

```
argv[0]: ./a.out
argv[1]: lol
argv[2]: hi
argv[3]: there!
```

4. Side Note

There are certain characters that cause Linux to do wonky things with command line arguments. Accordingly, we guarantee that the following characters will never appear in the command line arguments passed to your program:

dollar sign (‘$’)	semi-colon (‘;’)
opening parenthesis (‘(‘)	tilde (‘~’)
closing parenthesis (‘)’)	asterisk (‘*’)
exclamation point (‘!’)	period (‘.’)
hash (‘#’)	space (‘ ’)
ampersand (‘&’)	single quotes (‘ and ’)
backslash (‘\’)	double quotes (“ and ”)
pipe (‘|’)	redirection symbols (‘<’ and ‘>’)

I think any other standard keyboard characters should be fair game, but if you have any questions about odd behaviors that you’re getting with non-alphabetic and non-numeric input characters, feel free to ask – although the answer is almost certainly, “Don’t worry about that.”