Programming Assignment #1: Glowworm

COP 3502, Fall 2018

Due: Sunday, September 2, *before* 11:59 PM

Table of Contents

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 intended to rekindle your knowledge of C programming and stoke your creative problem solving skills.

Deliverables

Glowworm.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 in Notepad, they will appear to contain one long line of text. That's because Notepad handles 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 program, you will print an adorable little glowworm as it eats its way through an input string passed to your program. Your program will receive two input parameters to start with: an integer indicating the maximum length of your glowworm, and a string of characters for the glowworm to eat. Those input parameters will be passed to you as command line arguments. The process for that is described below in Appendix B ("Processing Command Line Arguments") (pg. 17).

The glowworm always starts with three characters: a tail ('~'), a head ('G'), and a big segment between the tail and the head ('O'). Overall, a new glowworm always starts out looking like this: "~OG"

Your program should always start by printing the following:

1. The input string passed to your program for the glowworm to eat, followed by two newline characters ('\n'), followed by "Glowworm appears! Hooray!" and another newline character.

2. The initial glowworm ("~OG"), followed by a single newline character ('\n').

3. A ledge of equal signs ('='), followed by two newline characters ('\n'). The number of equal signs printed should always match the number passed to your program to indicate the maximum length of your glowworm.

For example, if someone runs your program with a maximum glowworm length of 7 and "oOo@x" as the input string to be consumed by the glowworm, you should start out by printing the following:

```
oOo@x

Glowworm appears! Hooray!
~OG
=======
```

Next, process each character in the input string. The table below shows how your glowworm should react to each possible character in an input string. Following the table is a description of the output that your program should produce for each of these glowworm actions/behaviors.

Character	Description
'o' (lowercase letter o) 'O' (uppercase letter o) '@' (at symbol)	Any of these characters will cause your glowworm to grow a new segment.
's' (lowercase letter s) 'S' (uppercase letter s)	Any of these characters will cause your glowworm to shrink by one segment.
'-' (minus sign) '=' (equal symbol)	Any of these characters will cause your glowworm to inch forward.
'x' (lowercase letter x) 'X' (uppercase letter x) '%' (percent sign)	Any one of these characters will cause your glowworm to die. SAD.

Any character not listed in the table above should cause your glowworm to chill.

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

Here’s how each of the glowworm behaviors listed above should work:

2.1 Growing

If your glowworm eats a character that causes it to grow, it should gain a single ‘o’ segment between its tail and its capital ‘O’ segment, causing all other segments to move forward. For example, the following output starts with an existing glowworm and shows exactly what the output should be when it grows by one segment:

```
~OG
=======

Glowworm grows:
~oOG
=======
```

Note: In this diagram, I have omitted all the output leading up to the “~OG” line. I similarly omit all the output leading up to the first line in all the other glowworm diagrams in the examples below. Note also that the output always has a blank line beneath the equal symbols.

A similar thing happens even if the glowworm has inched forward to another position on the platform:

```
  ~OG
=======

Glowworm grows:
  ~oOG
=======
```

If the glowworm is already maximally long, then consuming a growth character should cause it to chill instead:

```
~ooooOG
=======

Glowworm chills:
~ooooOG
=======
```

2.2 Shrinking

If your glowworm eats a character that causes it to shrink, it should lose an ‘o’ segment, causing all segments after that one to pull backward. For example, the following output starts with an existing glowworm and shows exactly what the output should be when it shrinks by one segment:

```
~ooOG
=======

Glowworm shrinks:
~oOG
=======
```

If the glowworm has no lowercase ‘o’ segments, then it should chill instead:

```
   ~OG
=======

Glowworm chills:
   ~OG
=======
```

2.3 Inching Forward

If your glowworm eats a character that causes it to inch forward, all segments should move forward in tandem. For example, the following output starts with an existing glowworm and shows exactly what the output should be when it inches forward:

```
~ooOG
=======

Glowworm inches forward:
 ~ooOG
=======
```

2.4 Dying

If your glowworm eats a character that causes it to die, the “OG” in the glowworm string should be replaced with “Xx” (an uppercase ‘X’ followed by a lowercase ‘x’), and the program should then terminate without processing any remaining letters in the string that the glowworm was consuming:

```
  ~ooOG
=======

Glowworm meets its demise. SAD.
  ~ooXx
=======
```

2.5 Chillin’

If the glowworm consumes a character not associated with one of the behaviors/actions above, the glowworm should not move or change in any way, and you should indicate that the glowworm is chillin’, like so:

```
 ~ooOG
=======

Glowworm chills:
 ~ooOG
=======
```

2.6 Magical, Translocational Glowworm Shenanigans

PLOT TWIST! This is some sort of magical, translocational glowworm we're dealing with. If it reaches the end of the platform it's on and then grows or inches forward, it wraps back around to the beginning of the platform. The following output examples illustrate how to handle wrap-around situations with the glowworm.

If the glowworm has grown and/or inched forward to the end of the platform and then consumes a character that causes it to grow, its head should wrap back around to the beginning of the platform:

```
  ~ooOG
=======

Glowworm grows:
G ~oooO
=======
```

From there, if the glowworm grows again, the behavior is as follows:

```
G ~oooO
=======

Glowworm grows:
OG~oooo
=======
```

Here's how shrinking affects a glowworm that has already wrapped around to the start of the platform:

```
OG  ~oo
=======

Glowworm shrinks:
G   ~oO
=======
```

Here's another example. Notice that the tail always remains stationary when the glowworm shrinks:

```
G   ~oO
=======

Glowworm shrinks:
    ~OG
=======
```

Here's an example in which a wrapped-around glowworm inches forward:

```
OG ~ooo
=======

Glowworm inches forward:
oOG ~oo
=======
```

A wrapped-around glowworm can inch forward even if it is taking up the entire platform:

```
oOG~ooo
=======

Glowworm inches forward:
ooOG~oo
=======
```

We have included some test cases with this program so you can see exactly what your output should look like in a variety of scenarios. We have also included a script (`test-all.sh`) for testing your code on Eustis. Some of the test cases are designed to test the functionality of your program when you run it with command line arguments, and others are source files that you can run by compiling them alongside your own code for this program (i.e., by compiling multiple source files into a single program). Descriptions of these different types of test cases and detailed instructions on how to work with them are included in the sections that follow.

Although we have included a variety of test cases to get you started with testing the functionality of your code, we encourage you to develop your own test cases, as well. Ours are by no means comprehensive. We will use much more elaborate test cases when grading your submission.

For a full list of the functions you're required to write for this assignment, see Section 6, "Function Requirements."

3. Special Restrictions (*Important!*)

In this assignment, you cannot call any of C's built-in library functions other than *printf()*, *strlen()*, and *atoi()*. You cannot use global variables, and you cannot read or write to any files.

Most importantly, you also cannot create any arrays (statically or dynamically) anywhere in the code you submit, other than the *argv* array (described in Appendix B, "Processing Command Line Arguments"). That means you cannot create a string to hold your glowworm (since a string is just a `char` array), and you cannot create copies of any of the command line arguments (because that would involve the creation of new strings). You also should not modify the contents of the *argv* array. For a big hint on how to solve this problem without arrays, see Section 7 ("Suggested Function").

4. Glowworm.h

Super Important: We will not be able to test your code unless you #include "Glowworm.h" correctly.

Included with this assignment is a customer header file that includes functional prototypes for all the functions you will be implementing. You **must** #include this file from your `Glowworm.c` file, like so:

```
#include "Glowworm.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 "glowworm.h"` (with a lowercase 'g'), your program might compile on Windows, but it won't compile when we test it. You must use an uppercase 'G'.

You should not send `Glowworm.h` when you submit your assignment, and you should be very careful about modifying `Glowworm.h`. We will use our own copy of `Glowworm.h` when compiling your code.

If you write auxiliary functions ("helper functions") in your `Glowworm.c` file (which should not be necessary for this particular assignment), you should **not** add those functional prototypes to `Glowworm.h`. Our test case programs will not call your helper functions directly, so they do not need any awareness of the fact that your helper functions even exist. (We only list functional prototypes in a header file if we want multiple source files to be able to call those functions.) So, just put the functional prototypes for any helper functions you write at the top of your `Glowworm.c` file.

Think of `Glowworm.h` as a bridge between source files. It contains functional prototypes for functions that might be defined in one source file (such as your `Glowworm.c` file) and called from a different source file (such as the `UnitTestXX.c` files we have provided with this assignment).

5. Test Cases and the test-all.sh Script

The multiple 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.

You can run the script on Eustis by placing it in a directory with `Glowworm.c`, `Glowworm.h`, the `sample_output` directory, and all the test case files, and then typing:

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

Please note that these test cases are not comprehensive. You should also create your own test cases if you want to test your code comprehensively. In creating your own test cases, you should always ask yourself, "How could these functions be called in ways that don't violate the function descriptions, but which haven't already been covered in the test cases included with the assignment?"

6. Function Requirements

In the source file you submit, `Glowworm.c`, you must implement the following functions. You may implement auxiliary functions (helper functions) to make these work, as well, although that is probably unnecessary for this assignment. Please be sure the spelling, capitalization, and return types of your functions match these prototypes exactly.

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

> **Description:** Process the command line arguments as described above in Section 2 ("Overview"). The first argument, `argv[0]`, is the name of the executable file you're running, and can safely be ignored. The second argument, `argv[1]`, will be the string version of an integer indicating the maximum length of the glowworm (which is also the number of '=' symbols in the glowworm's platform). That integer can range from 3 through 50 (inclusively). The third argument, `argv[2]`, is the string that the glowworm will consume. It will have at least one character, but there is no upper limit on the number of characters it might contain. That string will conform to all the restrictions listed above in Section 2 ("Overview").
>
> As you process the string the glowworm is eating, be sure to print out the results according to the output specifications given above and illustrated in the test cases included with this assignment. Note that you'll always print a blank line after the platform the glowworm is on, so the output always ends with a totally blank line. Note also that you'll never print spaces after you've printed all the glowworm segments. (I.e., none of your output lines should have trailing spaces at the end of them.)
>
> Please be sure to follow all the restrictions listed above in Section 3, "Special Restrictions (Important!)". Information about processing command line arguments can be found in Appendix B (pg. 17). That appendix also includes information about converting strings to integers, which will be necessary for processing `argv[1]` (see pg. 19).
>
> **Return Value:** You must return zero (0) from this function. Returning any value other than zero from `main()` could result in catastrophic test case failure when we grade your program.

```
double difficultyRating(void);
```

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

```
double hoursSpent(void);
```

> **Output:** This function should not print anything to the screen.
>
> **Return Value:** A reasonable estimate (greater than zero) of the number of hours you spent on this assignment.

7. Suggested Function

This function is not required, but I think it will simplify your task tremendously if you implement it properly and call it as you process the input string that your glowworm is consuming. Think of this function description as a huge hint at how to proceed. If you want, you're more than welcome to implement this function with different parameters, a different return type, and so on.

```
void printGlowworm(int tail, int head, int maxLength);
```

Description: Receives the maximum length of the glowworm (which is also the number of equal symbols ('=') in the ledge the glowworm is on), and the position of the tail and head of the glowworm, as integers on the range 0 through (*maxLength* – 1). This function then prints out the glowworm and the ledge it's on.

For example, if this function receives *tail* = 2, *head* = 6, and *maxLength* = 8, it would print the following (including two '\n' characters after the last '=' symbol in the ledge):

```
  ~ooOG
========

```

Notice that the *tail* and *head* variables are effectively referring to the indices where the tail and head would be stored if we were using a character array to solve this problem. (Of course, using an array is forbidden by the special restrictions in Section 3 on pg. 8, above.)

Return Value: Nothing; void function don't return anything.

The fun continues on the following page!

8. Deliverables (Submitted via Webcourses, Not Eustis)

Submit a single source file, named `Glowworm.c`, via Webcourses. The source file should contain definitions for all the required functions (listed above), as well as any auxiliary functions you need to make them work. Don't forget to `#include "Glowworm.h"` in your source code (with a capital 'G').

Do not submit additional source files, and do not submit a modified `Glowworm.h` header file. Your source file must work with the `test-all.sh` script, and it must be able to compile and run like so:

```
gcc Glowworm.c
./a.out 7 soso
```

Be sure to include your name and NID as a comment at the top of your source file.

9. Grading

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

50%	Passing test cases with 100% correct output formatting. This portion of the grade includes tests of the *difficultyRating()* and *hoursSpent()* functions.
20%	Comments and whitespace. To earn these points, you must adhere to the style guidelines set forth in the Eustis Screenshot assignment. 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.

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 using any arrays.

Start early. Work hard. Good luck!

Appendix A:

Getting Started: A Guide for the Overwhelmed

Getting Started: A Guide for the Overwhelmed

Okay, so, this might all be overwhelming, and you might be thinking, “Where do I even start with this assignment?! I’m in way over my head!” First and foremost, let me say this:

There are plenty of TA office hours where you can get help, and here’s my general advice on starting the assignment:

1. First and foremost, start working on this assignment early. Nothing will be more frustrating than running into unexpected errors or not being able to figure out what the assignment is asking you to do on the day that it is due.

2. Secondly, glance through all the section headings in this PDF to get an idea of what kind of information it contains.

3. Thirdly, read over the PDF to get an idea of what the assignment is asking you to do. Section 2, “Overview,” will be the most important part to start out. Some sections might not make sense right away, and it’s okay to skim them if that’s the case, but don’t blow them off entirely.

4. After you’ve read through the PDF, but before you start programming, open up the test cases and sample output files included with this assignment and trace through a few of them to be sure you have an accurate understanding of what you’re supposed to be doing. Refer back to relevant sections of the PDF (the ones you might have skimmed in your first pass through the PDF) to clarify any questions you might have about those test cases.

5. Once you’re ready to begin coding, open up the `Glowworm.c` file included with this assignment. Add a header comment with your name and NID, and add one or two standard `#include`

directives. I've already added the `#include "Glowworm.h"` line for you and set up a skeleton structure with the required function signatures.

6. Set up your `main()` function to take command line arguments. For details on that, see Appendix B ("Processing Command Line Arguments") on pg. 17 of this PDF. Be sure your `main()` function returns zero.

7. Test that your `Glowworm.c` source file compiles. Do this before you've even taken a stab at making the functions work! If you're at the command line, your source file will need to be in the same directory as `Glowworm.h`, and you can test its ability to compile like so:

   ```
   gcc Glowworm.c
   ```

   Alternatively, you can try compiling it with one of the unit test source files using the instructions set forth in "Compiling and Running Unit Tests at the Command Line (Linux/Mac)" (Appendix D, pg. 24).

8. Once you have your project compiling, see if you can set up a basic for-loop that prints all command line arguments to the screen, one by one. Then see if you can successfully convert `argv[1]` from a string to an integer, and print that value to the screen. You'll want to be sure that you're processing that argument correctly before you start passing it to functions and trying to determine what's going on with the glowworm.

9. Once you have that working, go back to Sections 6 and 7 ("Function Requirements" and "Suggested Function"), and read through the function descriptions. From there, I think you have two solid choices as to how to proceed:

   a. You could do some brainstorming on how you would solve the problem if you already had a `printGlowworm()` function working. If you followed through with Step #4 in this section, then you've already programmed your brain to solve this part of the problem – neat! Now you just need to take that brain code and turn it into C code. At this point, you might want to trace through some test cases again (and revisit Section 2, "Overview," if necessary). As you work through test cases on paper, pay attention to what step-by-step process your brain is using to solve this problem. The process you're following is going to involve a loop (to traverse the input string) and some if-else logic, and if you pay attention to it, it'll start to look a lot like code.

   b. You could dive straight into writing the `printGlowworm()` function. What's neat about this is that you can work on this totally separately from the rest of the code; you can write a `main()` function that calls `printGlowworm()` a bunch of times with a bunch of different parameters, and see if you can get that function working properly in a variety of input scenarios. Once that's written, you can start fleshing out the rest of the program.

   Regardless of which path you choose above, once you have an idea of how you want to write a function (even if it feels a bit vague at first), dive in fearlessly. Be bold! Don't hesitate to run code that you think might not work; you won't break anything. You also don't have to finish everything before running your code. You should run your code frequently to see if each individual part you code up is working as intended.

General Tips for Working on Programming Assignments

Here's some general advice that might serve you well in this and all remaining assignments for the course:

1. In general, you'll want to stop to check whether your code compiles after every one or two significant blocks or chunks of code you throw down. Throughout the semester, if you frequently stop to check whether your code is compiling (rather than writing 30-100 lines of code before checking whether any of it is even syntactically correct), you will save yourself a lot of headaches.

2. If you get stuck while working on an assignment, draw diagrams on paper or a whiteboard. Make boxes for all the variables in your program. Trace through your code carefully, step by step, using these diagrams.

3. You're bound to encounter errors in your code at some point. Use `printf()` statements liberally to verify that your code is producing the results you think it should be producing (rather than making assumptions that certain components are working as intended). You should get in the habit of being immensely skeptical of your own code and using `printf()` to provide yourself with evidence that your code does what you think it does.

4. If you encounter a segmentation fault, you should always be able to use `printf()` and `fflush()` to track down the *exact* line you're crashing on.

5. When you find a bug, or if your program is crashing on a huge test case, don't trace through hundreds of iterations of some for-loop to track down the error. Instead, try to cook up a very small, new test case (as few lines as possible) that causes your code to crash. The smaller the test case that you have to trace through, the easier your debugging task will be.

6. You will eventually want to read up on how to set break points and use a debugger. One helpful Google query might be: gdb debugging tutorial.

Appendix B:

Processing Command Line Arguments

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 Glowworm.c
```

In this example, the string "*Glowworm.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. The following sections show you how to get that set up.

Passing Command Line Arguments to *main()*

Your program must be able to process exactly two arguments: the maximum length of the glowworm, and the string the glowworm is going to consume. For example:

```
seansz@eustis:~$ ./a.out 5 xobf
xobf

Glowworm appears! Hooray!
~OG
=====

Glowworm meets its demise. SAD.
~Xx
=====

seansz@eustis:~$ _
```

To get command line arguments (like the strings "5" and "xobf" in the above example) into your program, you just have to change the function signature for the `main()` function you're writing in `Glowworm.c`. Whereas we have 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 where we ran `./a.out 5 xobf`, `argc` would be equal to 3. `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]` stores the string "5", and `argv[2]` contains "xobf".

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 array:

```
#include <stdio.h>

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!
```

Converting a String to an Integer in C: atoi()

Note that you will have to convert `argv[1]` from a string to an integer. C has a handy function for doing that: `atoi()`, which resides in `stdlib.h` (so, be sure to `#include <stdlib.h>` in your code). It takes a string as its only argument and returns the integer represented by that string (or zero if it gets gobbledygook and can't convert the input string to a function.) Here's how you'll use that function:

```
int max_glowworm_length;
max_glowworm_length = atoi(argv[1]);
```

Appendix C:

Compilation and Testing

Overview

This appendix has instructions for compiling and running test cases in a variety of different contexts.

Dissecting the Contents of the *testcases* Folder

This project includes a folder named "*testcases*." In that folder, you'll find the following goodies:

1. The `test-all.sh` script (discussed above in Section 5, "Test Cases and the test-all.sh Script"). This little beast is made with love and can compile and run your code on all the test cases.

2. Test cases named `arguments01.txt` through `arguments08.txt`. These files contain command line arguments to pass to your program for testing. Instructions on how to use these files are given on pg. 21: "Running Standard Test Cases (arguments01.txt through arguments08.txt)."

3. Test cases named `UnitTest09.c` and `UnitTest10.c`. A unit test is a test case that runs a specific segment of your code to check that it's producing the correct results. In this case, each unit test will call one of the functions you're required to write for this assignment. These unit tests are source files that you will have to compile into a program along with your `Glowworm.c` source file and the `UnitTestLauncher.c` source file included in this folder. Instructions for working with these unit tests are on pg. 22: "Running Unit Test Cases (UnitTest09.c and UnitTest10.c)."

4. A source file named `UnitTestLauncher.c`. This source file has to be compiled into your program if (and only if) you are running a unit test. This process is described on pg. 22: "Running Unit Test Cases (UnitTest09.c and UnitTest10.c)."

5. A folder named `sample_output`. This folder contains output files named `output01.txt` through `output12.txt`. These files show exactly what your output should look like for different test cases (provided that you do **not** open them in Notepad; see Section 1, "Important Note: Test Case Files Look Wonky in Notepad").

   The `arguments01.txt` through `arguments08.txt` test cases should produce the output given in `output01.txt` through `output08.txt`, and the `UnitTest09.c` and `UnitTest10.c` test cases should produce the output given in `output09.txt` and `output10.txt`. For details on how to ensure that your output is a 100% match for the expected output given in these text files, see Section 5 ("Test Cases and the test-all.sh Script").

Running Standard Test Cases (*arguments01.txt* through *arguments08.txt*)

There are three ways to run the standard test cases (`arguments01.txt` through `arguments08.txt`) included with this assignment:

1. Compile your program at the command line, and then copy and paste the arguments from one of those files into the command prompt after "./a.out." You can use *diff* to check the difference between your output and the expected output. For example, `arguments01.txt` contains the text "`5 soso`." You can run that test case and check your output against `output01.txt` like so:

```
seansz@eustis:~$ gcc Glowworm.c
seansz@eustis:~$ ./a.out 5 soso > myoutput.txt
seansz@eustis:~$ diff myoutput.txt sample_output/output01.txt
seansz@eustis:~$ _
```

The lack of response from the *diff* command indicates that your output was correct. Yay!

2. Instead of copying and pasting text from `arguments01.txt`, you can also use the following magical incantations to make the Linux command line pass the contents of `arguments01.txt` to your program as command line arguments for you. (This works with the Mac terminal, too!)

```
seansz@eustis:~$ gcc Glowworm.c
seansz@eustis:~$ ./a.out $(cat arguments01.txt) > myoutput.txt
seansz@eustis:~$ diff myoutput.txt sample_output/output01.txt
seansz@eustis:~$ _
```

Again, the lack of any grumbling from *diff* indicates that your output was correct.

3. Alternatively, at a Linux or Mac command line (or in the new bash shell in Windows), you can run the `test-all.sh` script and let it do all the work for you. It will compile and run your source code on all the test cases, and give you a report of the results. For details on how to do that, see Section 5, “Test Cases and the test-all.sh Script,” on pg. 9.

Running Unit Test Cases (*UnitTest09.c* and *UnitTest10.c*)

The unit tests are a bit more tricky to get running. Here are your options:

1. The easiest option is to simply let the `test-all.sh` script do all the work for you, but you’ll learn more if you try one of the following options as well. For details on how to use the `test-all.sh` script, see Section 5, “Test Cases and the test-all.sh Script.”

2. You can run each unit test individually at the command line (Linux, Mac, or the Linux bash shell in Windows 10) by following the detailed instructions on pg. 24 (“Compiling and Running Unit Tests at the Command Line (Linux/Mac)”).

Continued on the following page...

Compilation and Testing at the Command Line (Linux/Mac)

To compile your source file (`.c` file) at the command line:

```
gcc Glowworm.c
```

By default, this will produce an executable file called `a.out`, which you can run with command line arguments like so:

```
./a.out 5 soso
```

If you want to name the executable file something else, use:

```
gcc Glowworm.c -o Glowworm.exe
```

...and then run the program with command line arguments like so:

```
./Glowworm.exe 5 soso
```

Running the program could potentially dump a lot of text to the screen. If you want to redirect your program's output to a text file in Linux, it's easy. Just run the program using the following command, which will create a file called `whatever.txt` that contains the output from your program:

```
./Glowworm.exe 5 soso > whatever.txt
```

Linux has a helpful command called `diff` for comparing the contents of two files, which is really helpful here since we've provided several sample output files. You can see whether your output matches ours exactly by typing, e.g.:

```
diff whatever.txt sample_output/output01.txt
```

If the contents of `whatever.txt` and `output01.txt` are exactly the same, `diff` won't have any output. It will just look like this:

```
seansz@eustis:~$ diff whatever.txt sample_output/output01.txt
seansz@eustis:~$ _
```

If the files differ, it will spit out some information about the lines that aren't the same. For example:

```
seansz@eustis:~$ diff whatever.txt sample_output/output01.txt
4c4
< ~oG
---
> ~OG
seansz@eustis:~$ _
```

Compiling and Running Unit Tests at the Command Line (Linux/Mac)

Compiling a unit test at the command line requires you to compile three source files into one program. For example, if you want to compile `UnitTest09.c`, you must compile it along with your `Glowworm.c` file and the `UnitTestLauncher.c` file we have included with this project.

Important: In order for this to work, you **must** ensure the `#define main __hidden_main__` line in `Glowworm.h` is **not** commented out (i.e., remove the "`//`" from the beginning of that line).

Once you've done that, you can compile these three source files into a single program like so:

```
gcc Glowworm.c UnitTest09.c UnitTestLauncher.c
```

By default, that gcc command will produce an executable file called `a.out`. You can now run the unit test like so:

```
./a.out
```

You can redirect the output of the unit test to a text file like so:

```
./a.out > whatever.txt
```

Linux has a helpful command called `diff` for comparing the contents of two files, which is really helpful here since we've provided several sample output files. You can see whether your output matches ours exactly by typing, e.g.:

```
diff whatever.txt sample_output/output09.txt
```

If the contents of `whatever.txt` and `output09.txt` are exactly the same, `diff` won't have any output. It will just look like this:

```
seansz@eustis:~$ diff whatever.txt sample_output/output09.txt
seansz@eustis:~$ _
```

If the files differ, it will spit out some information about the lines that aren't the same, as we saw on the previous page.

An Important Note About Messing with Glowworm.h

Super Important: After running unit tests, before you can run standard test cases again, you will need to comment out the `#define main __hidden_main__` line in `Glowworm.h`. You **must** use "`//`" to comment out that line, with no space after the "`//`," like so: `//#define main __hidden_main__`.

Uncommenting that line essentially kills your `main()` function in `Glowworm.c` and allows one of the unit test functions to take over the program. Commenting out that line again brings your `main()` function back to life.