Programming Assignment #3: Pointers

COP 3223H, Fall 2019

Due: Wednesday, October 23, *before* 11:59 PM

Table of Contents

Deliverables

assignment03.c

Note! The capitalization and spelling of your filename matter!

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

1. Super Important: Initial Setup (*Read This Carefully!*)

At the very top of your *assignment03.c* 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 "assignment03.h"
```

Yes, that needs to be in "double quotes" instead of <angled brackets>, because *assignment03.h* is a local header file that we've included with this assignment, rather than a standard system header file.

The *assignment03.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 <stdio.h>* as usual.

In your code, writing *main()* is optional. If you do write *main()* (which is a good idea so you can call and test your other functions) it doesn't matter too much what it does (as long as it's not coded up to do anything naughty or malicious), because we won't actually run your *main()* function. When we grade your program, we will write a script that will seek out your *main()* function, destroy it, and replace it with our own *main()* function for each test case we want to run. This is savage, but effective. If you don't have that *#include "assignment03.h"* line in your code, our script will be unable to inject our own *main()* functions into your code, and all the test cases we use to grade your program will fail. SAD.

From this point forward, you will always have to have *assignment03.h* (provided in the ZIP file for this assignment) in the same folder as your *assignment03.c* file any time you want to compile your code.

2. Note: Test Case Files Might 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 older versions of Notepad, 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.

3. Assignment Overview

This assignment has two primary purposes: Firstly, it has a few exercises that will help solidify your understanding of pointers. Secondly, it has a fairly sophisticated printing function that will stretch your critical thinking and creative problem solving skills.

In this assignment, you will write several functions (listed below). Some will be more straightforward than others. They are designed to ramp up in difficulty and challenge your understanding of the material covered in class. Before you start working on this assignment, your best bet is to read and comprehend *all* the notes posted in Webcourses.

4. Overview: Glowie Returns! (You're Going to Want to Read This)

In this assignment, you'll once again be working with the adorable glowworm that was introduced in Assignment #2 (So Many Loops). This time, there will be pointers in the mix, and the glowworm will be eating snacks that cause it to grow, shrink, inch forward, and even die. This section describes, in detail, five glowworm-related functions you have to write in this assignment.

4.1 Glowworm Actions

The following three functions cause the glowworm to grow, shrink, or move forward on its platform:

- ***void grow_glowie(int *head)***
  This function takes a pointer to a glowworm's *head* coordinate and increments (i.e. increases) that coordinate by one, thereby causing the glowworm to grow. Hooray, glowworm!
- ***void shrink_glowie(int *head)***
  This function takes a pointer to a glowworm's *head* coordinate and decrements (i.e., decreases) that coordinate by one, thereby causing the glowworm to shrink.
- ***void move_glowie_forward(int *tail, int *head)***
  This function takes pointers to the glowworm's *tail* and *head* coordinates and increments both of them by one, thereby causing the glowworm to move forward by one space on its platform. Go, glowworm, go!

4.2 Feeding Your Glowworm

The next glowworm function in this assignment is a *feed_glowworm()* function that takes pointers to the glowworm's *tail* coordinate, *head* coordinate, *is_alive* status (1 if the glowworm is alive, 0 if the glowworm is dead), and a *snack* character that the glowworm is going to eat. This function must perform one of the following actions based on what character the *snack* variable is pointing to:

Character	Description

`'o' (lowercase letter o)`
`'O' (uppercase letter o)`
`'@' (at symbol)`
	Any one of these characters will cause your glowworm to grow. In this case, you must call the *grow_glowie()* function from within the *feed_glowworm()* function.

`'s' (lowercase letter s)` `'S' (uppercase letter s)`	Any one of these characters will cause your glowworm to shrink. In this case, you must call the *shrink_glowie()* function from within the *feed_glowworm()* function.
`'-' (minus sign)` `'=' (equal symbol)`	Any one of these characters will cause your glowworm to move forward. In this case, you must call the *move_glowie_forward()* function from within the *feed_glowworm()* function.

`'x' (lowercase letter x)`
`'X' (uppercase letter x)`
`'%' (percent sign)`
	Any one of these characters will cause your glowworm to die. In this case, you must use the *is_alive* pointer to set the glowworm's *is_alive* status to 0 (zero).

For reference, the function signature for *feed_glowworm()* is as follows:

int feed_glowworm(int *tail, int *head, int *is_alive, char *snack)

In addition to performing the actions listed above, there are three other things you must do from within your *feed_glowworm()* function:

1. Print a string indicating what happens to the glowworm when it eats the *snack* passed to this function. (See the table below.)
2. The value that *snack* is pointing to must be set to a space character (‘ ’) before the function returns to indicate that the glowworm ate the snack. Note that if the character is **not** one of the 10 characters listed in the table above, the glowworm does **not** eat the snack, and so you should **not** change the character pointed to by *snack* in that case.
3. Return an integer. If the glowworm consumes a snack without dying, return 1. If the glowworm dies after consuming the snack, or if the glowworm refuses to consume the snack, return 0.

The strings printed by the *feed_glowworm()* function are as follows (and each of them should be followed by a newline character):

Action	**Output**
`Grow`	Glowworm munches on a snack that causes it to grow! Om nom nom.
`Shrink`	Glowworm munches on a snack that causes it to shrink!
`Inch Forward`	Glowworm feels energetic after its snack and inches forward!
`Death`	That snack poisoned the glowworm. SAD.
`(any other snack)`	The glowworm looks at the snack skeptically.

For detailed examples on how this function should work, be sure to consult the test cases and sample output files for these functions.

4.3 Printing the Glowworm: Magical, Translocational Glowworm Shenanigans

In Assignment #2, we saw a *print_glowie_from_coordinates()* function that took two parameters: the coordinate of the glowworm’s tail, and the coordinate of its head. Recall that when dealing with coordinates, 0 corresponds to the first character in the output line, 1 corresponds to the second character on the output line, 2 corresponds to the third character on the output line, and so on. In general, the number *n* corresponds to character (*n* + 1) on an output line. (This kind of thing where we start counting from zero is quite common in computer science.)

So, if we called *print_glowie_from_coordinates(2, 5)*, our tail (‘~’) would be at coordinate 2 (the third character on the output line), and our head (‘G’) would be at coordinate 5 (the sixth character on the output line), immediately preceded by a large segment (denoted with a capital ‘O’ character). All other characters between the

tail and head would be small segments (lowercase ‘o’) characters, and our output would look like this, with a platform beneath the glowie, followed by two newline characters:

```
  ~oOG
======

```

Recall that the platform in Assignment #2 was always just long enough to extend beneath the head of the glowworm – never longer, and never shorter. In the new printing function you have to write for this assignment, you’ll be given four parameters: the tail coordinate, the head coordinate, the length of the platform, and an integer indicating whether the glowworm is alive. The platform is guaranteed to be long enough to support the entire glowworm, but it might also be longer. For example, *print_magical_translocational_glowworm(2, 5, 10, 1)* would print the above glowworm, but the platform beneath would have ten equal signs (‘=’), like so:

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

```

Note that there are no spaces after the ‘G’ in that first line of output. Note also that the last parameter passed to the function simply indicates that the glowworm is alive.

PLOT TWIST! The glowworms we’re dealing with in this assignment are some sort of magical, translocational glowworms. Whereas Assignment #2 guaranteed the head coordinate would always be at *least* two spaces after the tail coordinate, the printing function you’re writing for this assignment has no such restriction and needs to handle situations where a tail appears *after* a glowworm’s head.

If the tail coordinate comes before the head coordinate, then, as before, the glowworm is printed from the tail coordinate through the head coordinate (following by a newline character and a platform of length *platform_length*). See, for example, the *print_magical_translocational_glowworm(2, 5, 10, 1)* call referenced above.

However, if the head coordinate comes *before* the tail coordinate, we start by printing segments of the glowworm from the beginning of the line through the *head* coordinate, then stop and print spaces until we reach the tail, and then print segments after the tail until we reach the end of the platform below the glowworm. In this way, it’s as if the glowworm’s body starts part way down the platform, and when it reaches the end of the platform, the remaining segments (along with the head) are magically teleported back to the beginning of the platform.

For example, *print_magical_translocational_glowworm(7, 5, 10, 1)* would print the output at the top of the following page. Notice that the platform has 10 equal signs (‘=’), the tail is at coordinate 7, and then we have enough segments to fill the rest of the platform (which has a length of 10), but we *also* have segments resting on the beginning of the platform, from coordinate 0 through coordinate 5 (where the *head* is located). As always, a big ‘O’ segment comes right before the head:

```
oooo0G ~oo
==========
```

So, it's as if part of the glowworm has gone through a portal at the end of the platform and teleported back to the beginning of the platform, so that its head is looking forward and seeing its own tail.

Note that if the head happens to be at index 0, then the big 'O' segment that comes before it must be at the opposite end of the platform. For example, *print_magical_translocational_glowworm(4, 0, 6, 1)*, should print the output shown below. Notice there are 6 equal signs ('=') for the platform, the tail begins at coordinate 6, there are segments after the tail all the way until the end of the platform, the head has teleported back around to coordinate 0, and the segment that comes right before the 'G' is a capital 'O' (as opposed to a lowercase 'o'):

```
G   ~O
======
```

Similarly, *print_magical_translocational_glowworm(3, 0, 6, 1)* would print the following:

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

And finally, *print_magical_translocational_glowworm(3, 1, 6, 1)* would print the following:

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

There are additional examples of this in the test cases included with this assignment.

Important Note: In the first four functions listed in this section, you can assume the tail coordinate comes before the head. As the glowworm grows or inches forward in those functions, it will never fall off its platform or have to wrap around and become a magical, translocational glowworm.

Furthermore, recall from Assignment #2 that the head coordinate of a glowworm always needs to be at least two greater than its *tail* coordinate, because every glowworm always has at least three components: a tail ('~'), a head ('G'), and a big segment between the tail and head ('O'). You may assume that if we have a test case where the glowworm eats a snack that makes it shrink, the head will still be at least two greater than the *tail* after that operation. Of these five functions, only the printing function will ever have to handle a case where the head coordinate is less than the tail coordinate.

5. Function Requirements

You must implement the following functions in a file named *assignment03.c*. Please be sure the spelling, capitalization, return types, and function parameters match the ones given below. Even the most minor deviation could cause a huge loss of points. The order in which you write these functions in your file does not matter, as long as it compiles *without any warnings* (or errors, for that matter). Your functions are allowed to call one another, and you can write additional functions ("helper functions") if you find that doing so will make it easier for you to write some of these required functions.

You may assume that will never pass invalid pointers (such as NULL) to these functions.

```
void sort_ascending(int *a, int *b, int *c);
```

Description: This function takes three integer pointers (*a*, *b*, and *c*), and changes the values pointed to by those variables so that they are sorted in increasing order (with *a* pointing to the smallest value, *b* pointing to the middle value, and *c* pointing to the largest value).

For example, suppose we execute the following code in *main()*:

```
int x = 9, y = 2, z = 3;
sort_ascending(&x, &y, &z);
```

After executing *sort_ascending(&x, &y, &z)*, *x* should be 2, *y* should be 3, and *z* should be 9.

For this function, you may assume the values pointed to by *a*, *b*, and *c* are all distinct (i.e., there won't be any duplicate integers in the mix).

Output: This function should not print anything to the screen.

Return Value: This is a *void* function and therefore should not return a value.

Related Test Cases: *testcase01a.c*, *testcase01b.c*, *testcase01c.c*

```
void sort_ascending_with_repeats_allowed(int *a, int *b, int *c);
```

Description: This function takes three integer pointers (*a*, *b*, and *c*), and changes the values pointed to by those variables so that they are sorted in non-decreasing order (with *a* pointing to the smallest value, *b* pointing to the middle value, and *c* pointing to the largest value).

For example, suppose we execute the following code in *main()*:

```
int x = 9, y = 2, z = 2;
sort_ascending_with_repeats_allowed(&x, &y, &z);
```

After executing *sort_ascending_with_repeats_allowed(&x, &y, &z)*, *x* should be 2, *y* should be 2, and *z* should be 9.

The difference between this function and the *sort_ascending()* function is that the values pointed to by *a*,

b, and *c* in this function might contain duplicates.

Output: This function should not print anything to the screen.

Return Value: This is a *void* function and therefore should not return a value.

Related Test Cases: *testcase02a.c*, *testcase02b.c*, *testcase02c.c*

```
void sort_descending(int *a, int *b, int *c);
```

Description: This function takes three integer pointers (*a*, *b*, and *c*), and changes the values pointed to by those variables so that they are sorted in decreasing order (with *a* pointing to the largest value, *b* pointing to the middle value, and *c* pointing to the smallest value).

For example, suppose we execute the following code in *main()*:

```
int x = 9, y = 2, z = 3;
sort_descending(&x, &y, &z);
```

After executing *sort_descending(&x, &y, &z)*, *x* should be 9, *y* should be 3, and *z* should be 2.

For this function, you may assume the values pointed to by *a*, *b*, and *c* are all distinct (i.e., there won't be any duplicate integers in the mix).

Output: This function should not print anything to the screen.

Return Value: This is a *void* function and therefore should not return a value.

Related Test Cases: *testcase03a.c*, *testcase03b.c*, *testcase03c.c*

```
void sort_descending_with_repeats_allowed(int *a, int *b, int *c);
```

Description: This function takes three integer pointers (*a*, *b*, and *c*), and changes the values pointed to by those variables so that they are sorted in non-increasing order (with *a* pointing to the largest value, *b* pointing to the middle value, and *c* pointing to the smallest value).

For example, suppose we execute the following code in *main()*:

```
int x = 2, y = 9, z = 2;
sort_descending_with_repeats_allowed(&x, &y, &z);
```

After executing *sort_descending_with_repeats_allowed(&x, &y, &z)*, *x* should be 9, *y* should be 2, and *z* should be 2.

The difference between this function and the *sort_descending()* function is that the values pointed to by *a*, *b*, and *c* in this function might contain duplicates.

Output: This function should not print anything to the screen.

Return Value: This is a *void* function and therefore should not return a value.

Related Test Cases: *testcase04a.c*, *testcase04b.c*, *testcase04c.c*

```
int *get_pointer_to_median(int *a, int *b, int *c);
```

Description: This function takes three integer pointers (*a*, *b*, and *c*), and returns a pointer to the median of the three values pointed to by those variables.

For example, suppose we execute the following code in *main()*:

```
int x = 2, y = 9, z = 3;
get_pointer_to_median(&x, &y, &z);
```

That function call should return a pointer to *z*, since *z* contains the median of those three integers.

Note that the pointers passed to this function might contain duplicate values. If there are multiple pointers to the median, the function may return any one of those pointers. For example, suppose we execute the following code in *main()*:

```
int x = 2, y = 9, z = 2;
get_pointer_to_median(&x, &y, &z);
```

That function call can return a pointer to *x* or *z*, since both *x* and *z* contain the median of the three values passed to the function.

Output: This function should not print anything to the screen.

Return Value: Return a pointer to the median of the three values passed to the function.

Related Test Cases: *testcase05a.c*, *testcase05b.c*, *testcase05c.c*

```
void grow_glowie(int *head);
```

Description: This function takes a pointer to the head coordinate of some glowworm and increments (i.e., increases) the value of that head coordinate by one, thereby causing the glowworm to grow by one segment. You must call this function from the *feed_glowworm()* function described below. (See the bottom of pg. 6 of this PDF for additional notes about the input to this function.)

Output: This function should not print anything to the screen.

Return Value: This is a *void* function and therefore should not return a value.

Related Test Cases: *testcase06a.c*, *testcase06b.c*, *testcase06c.c*

```
void shrink_glowie(int *head);
```

Description: This function takes a pointer to the head coordinate of some glowworm and decrements (i.e., decreases) the value of that head coordinate by one, thereby causing the glowworm to shrink by one segment. You must call this function from the *feed_glowworm()* function described below. (See the bottom of pg. 6 of this PDF for additional notes about the input to this function.)

Output: This function should not print anything to the screen.

Return Value: This is a *void* function and therefore should not return a value.

Related Test Cases: *testcase07a.c, testcase07b.c, testcase07c.c*

```
void move_glowie_forward(int *tail, int *head);
```

Description: This function takes pointers to the tail and head coordinates of some glowworm and increments the values of both coordinates by one, thereby causing the glowworm to move forward by one space on its platform. You must call this function from the *feed_glowworm()* function described below. (See the bottom of pg. 6 of this PDF for additional notes about the input to this function.)

Output: This function should not print anything to the screen.

Return Value: This is a *void* function and therefore should not return a value.

Related Test Cases: *testcase08a.c, testcase08b.c, testcase08c.c*

```
int feed_glowworm(int *tail, int *head, int *is_alive, char *snack);
```

Description: This function takes four arguments: pointers to the tail and head coordinates of some glowworm, a pointer to a variable indicating whether the glowworm is alive (1) or not (0), and a pointer to a single character that represents a snack for the glowworm.

The behaviors here are governed by the snack passed into the function. This function performs actions based on the table on pg. 3 of this PDF, and produces output based on the table on pg. 4. Please refer to those pages for additional details on the expected behavior of this function, and please be sure to refer to the test case files and sample output files for concrete examples of how this function should work.

Special Restriction: This function is not allowed to modify the value pointed to by *head* or *tail*. It **must** call the *grow_glowie()*, *shrink_glowie()*, and *move_glowie_forward()* functions to modify those values.

Output: This function should print one of the strings given in the table on pg. 4, followed by a newline character.

Testing Note: In creating your own test cases for this assignment, you might want to write one that prints a glowworm to the screen, then makes repeated calls to the *feed_glowworm()*, followed each time by calls to the printing function so you can see the glowworm growing, shrinking, and moving around as it eats various snack characters.

Return Value: If the glowworm consumes a snack without dying, return 1. If the glowworm dies after consuming the snack, or if the glowworm refuses to consume the snack, return 0.

Related Test Cases: *testcase09a.c, testcase09b.c, testcase09c.c, testcase09d.c, testcase09e.c*

Continued on the following page…

```
void print_magical_translocational_glowworm(int tail, int head, int platform_length, int is_alive);
```

Description: This function takes four arguments: the *tail* and *head* coordinates of some glowworm, the length of the platform beneath the glowworm (*platform_length*), and an integer (*is_alive*) indicating whether the glowworm is alive (1) or not (0).

This function then prints the corresponding glowworm, followed by the platform it's on, followed by two newline characters. Further details of the expected output for this function are described above on pgs. 4-6. As with the *print_glowie_dead_or_alive()* function in Assignment #2, this should replace the 'O' and 'G' characters in the output with 'X' and 'x' (respectively) if the glowworm is not alive.

You can assume that the parameters passed to this function will always be valid. So, the *tail* and *head* coordinates will always be on the range 0 through (*platform_length* - 1); they will never be negative, and they will never be larger than (*platform_length* - 1), because that would cause them to fall off the platform.

Furthermore, recall that a glowworm always has at least three components: a tail ('~'), a head ('G'), and a large segment ('O') immediately preceding the head. We guarantee the coordinates passed to this function will always allow for those three components to be represented. (It follows, then, that the value of *platform_length* passed to this function will always be greater than or equal to 3). There may or may not also be small segments ('o') that need to be printed in some cases.

Hint: I've included a copy of my *print_glowie_dead_or_alive()* function with this assignment. (See *print_glowie.c*.) You are welcome to dissect, re-use, and modify that code in order to get this particular function working for this assignment, but if you re-use any of my code directly, you should include a comment attributing it to me. (Recall that you should not be reading through or using bits of anyone else's code in this class.) One thing you might want to do is set up this function so if *tail* < *head*, you simply re-use my logic from *print_glowie.c*. Then, you'll have to write similar (albeit slightly more complex) logic for the case where *tail* > *head*.

Output: For concrete examples of the expected output for this function, please refer to pgs. 4-6 of this PDF, as well as the test case files and sample output files released with this assignment. This is one where you're going to want to write a lot of your own test cases, as well, to make sure you cover all your bases and thoroughly check whether this function is working as intended.

Return Value: This is a *void* function and therefore should not return a value.

Related Test Cases: *testcase10a.c, testcase10b.c, testcase10c.c, testcase10d.c, testcase10e.c*

```
double difficulty_rating(void)
```

Description: The body of this function should contain a single line of code that returns a value indicating how difficult you found this assignment on a scale from 1.0 (ridiculously easy) through 5.0 (insanely difficult).

Output: This function should not print anything to the screen.

Return Value: This function returns a real number (a double) as described above.

Related Test Case: *testcase11.c*

`double hours_invested(void)`

Description: The body of this function should contain a single line of code that returns a value (greater than zero) that is an estimate of the number of hours you invested in this assignment. Your return value must be a realistic and reasonable estimate. Unreasonably large values will result in loss of credit for this particular function.

Output: This function should not print anything to the screen.

Return Value: This function returns a real number (a double) as described above.

Related Test Case: *testcase12.c*

`double prior_experience(void)`

Description: This is the same function as in Programming Assignment #1. The body of this function should contain a single line of code that returns a value indicating how much prior programming experience you had coming into this course on a scale from 1.0 (never programmed before) through 5.0 (seasoned veteran who has worked in industry as a programmer).

Output: This function should not print anything to the screen.

Return Value: This function returns a real number (a double) as described above.

Related Test Case: *testcase13.c*

6. Special Restrictions (*Important!*)

You must abide by the following restrictions in the *assignment03.c* file you submit. Failure to abide by any one of these restrictions could result in a catastrophic loss of points.

- ★ (***Restriction Lifted!***) You can now use nested loops, but they shouldn't be necessary.
- ★ (***New!***) You cannot use typecasting in this assignment. (That's where you put a data type in parentheses in front of a variable in order to force C to view it as a particular data type – possibly a data type other than what it really is. I'm imposing this restriction so people can't use type casting to artificially suppress warnings.)
- ★ (***Important!***) You cannot use arrays in this assignment.
- ★ You cannot call any of C's built-in functions except for *printf()*.
- ★ Do not declare new variables part way through a function. All variable declarations must occur at the *top* of a function.
- ★ All variables must be declared inside your functions or declared as function parameters.
- ★ Do not make calls to C's *system()* function.
- ★ Do not use *goto* statements in your code.

- ★ Do not read or write to files (using, e.g., C's *fopen()*, *fprintf()*, or *fscanf()* functions).
- ★ Do not write malicious code. (I would hope this would go without saying.)
- ★ No crazy shenanigans.

7. Style Restrictions (*Important!*)

Please conform as closely as possible to the style I use while coding in class. In particular:

- ★ Any time you open a curly brace, that curly brace should start on a new line, and it should be indented to align properly with the line above it. See my code in Webcourses for examples.
- ★ 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 always use code blocks with if/else statements and loops, even if there's just one line of code within that code block.
- ★ 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*"
- ★ Any libraries you *#include* should be listed *after* the header comment at the top of your file that includes your name, course number, semester, NID, and so on.
- ★ 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 overly long lines of code. Lines must be fewer than 100 characters wide.
- ★ Avoid excessive consecutive blank lines. In general, you should never have more than one or two consecutive blank lines.
- ★ Please leave a space on both sides of any arithmetic operator you use in your code. For example, use *(a + b) - c* instead of *(a+b)-c*.
- ★ When defining a function that doesn't take any arguments, put *void* in its parentheses. For example, define a function using *int do_something(void)* instead of *int do_something()*.

8. Running All Test Cases on Eustis

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 the *test-all.sh* script on Eustis, first transfer it to Eustis along with with *assignment03.c*, *assignment03.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 (*assignment03.c*, *assignment03.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 $(pwd) YOUR_NID@eustis.eecs.ucf.edu:~
   ```

   Warning: Note that the `$(pwd)` 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 the script 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 assignment03
```

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.

Warning: When working at the command line, any spaces in file names or directory names either need to be escaped in the commands you type, or the entire name needs to be wrapped in double quotes. For example:

```
cd assignment\ files
```

```
cd "assignment files"
```

It's probably easiest to just avoid file and folder names with spaces.

9. Running Test Cases Individually

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. Here are two ways you can do that:

1. The ideal way:
    a. (*Optional*) Remove the *main()* function from your *assignment03.c* file.
    b. Compile *both* your *assignment03.c* file and the test case file you want to run into a single program. To compile multiple source files at the command line, simply type both filenames after *gcc*:

    ```
    gcc assignment03.c testcase01a.c
    ```

    c. Run the program as usual:

    ```
    ./a.out
    ```

    d. If you want to check whether the output of your program is an *exact* match of the expected output for that case, you can force your program to dump its output into a text file and then use the *diff* command to check whether the newly created text file has the exact same contents as one of the sample output test files. For example:

    ```
    ./a.out > my_output.txt
    diff my_output.txt sample_output/output01.txt
    ```

    Note: If two files have the exact same contents, *diff* does not produce any output. If it produces an output message, that means the files differ somehow.

2. Following is the less ideal way to run a single test case. However, this is what you'll most likely want to do if you want to write your own *main()* function to test your code:
    a. Comment out the *#include "assignment03.h"* line in your *assignment03.c* source file.
    b. Copy and paste the *main()* function from one of the test case files (such as *testcase01a.c*) into your *assignment03.c* source file, or write your own *main()* function for testing.
    c. Compile *assignment03.c* as usual:

    ```
    gcc assignment03.c
    ```

    d. Run the program as usual:

    ```
    ./a.out
    ```

    e. When you're finished, don't forget to un-comment the *#include "assignment03.h"* line in your *assignment03.c* file so that your code will be compatible with our grading infrastructure!

10. Deliverables (Submitted via Webcourses, Not Eustis)

Submit a single source file, named *assignment03.c*, via Webcourses. The source file should contain definitions for all the required functions (listed above), as well as any helper functions you've written to make them work. Don't forget to *#include "assignment03.h"* in your *assignment03.c* code.

Do not submit additional source files, and do not submit a modified *assignment03.h* file. Your source file must work with the *test-all.sh* script, and it must be able to compile and run with each individual test case, like so:

```
gcc assignment03.c testcase01a.c
./a.out
```

Be sure to include your name, the course number, the current semester, and your NID in a comment at the top of your source file.

11. Grading

> ***Important Note:*** When grading your programs, we will use different test cases from the ones we've released 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.

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

60%	Passing test cases with 100% correct output formatting. Points will be awarded for each individual test case you pass. (It's possible to pass some, but not others.)
10%	Adherence to all style restrictions listed above. We will likely impose significant penalties for small deviations, because we really want you to develop good coding style habits in this class.
10%	(***New!***) Code includes useful and appropriate comments. See my commenting guidelines in the notes in Webcourses from Monday, Sept. 30. Points may also be deducted from this category for missing required information in the header comment(s), or for incorrect placement of the header comment(s), or for naming the source file incorrectly.
10%	Code compiles without warnings (and without the use of typecasting to artificially suppress warnings).
10%	The *feed_glowworm()* function calls the *grow_glowie()*, *shrink_glowie()*, and *move_glowie_forward()* functions rather than directly modifying the values pointed to by *tail* and *head*.

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

Please also note that failure to abide by any the special restrictions listed above on pg. 12 (Section 6, “Special Restrictions”) could result in a catastrophic loss of points.

Start early. Work hard. Good luck!