CS262 Lab 4 - Debugging I

This lab will teach you various methods for debugging. It will be different from earlier labs, in that you will only have a small amount of programming to do. This lab, and the narrative with it, is designed to help you learn techniques you can use to find out what is happening inside your programs and discover the errors that can cause seemingly correct programs to give incorrect results. This lab consists of a narrative that will take you step by step through a debugging process. To complete the lab, you will need to read the narrative, following the instructions for changing various lines of code in an existing program (lab4.c) and answering short questions about the results. You will submit a separate answer sheet that is provided to record your results as well as the modified lab4.c file.

It is important that you follow this narrative carefully, and avoid any shortcuts in this lab. The answers to many of the questions you will be given depend upon properly completing the lab in the sequence presented.

Remember that programs can contain three different types of errors: (1) syntax errors, which normally are caught by the compiler; (2) runtime errors, which occur when your program compiles correctly but when your program encounters an error while running, and therefore does not finish; and (3) logic errors, which mean that even though a program compiles and runs, it may produce incorrect *results* due to a logical inconsistency or improper algorithm/code within the program. This lab will give you the opportunity to address all three types of errors.

Note: This lab must be completed on zeus using the gcc compiler. Compilers on other systems (mason.gmu.edu, your personal linux machine, Microsoft Visual C++ or XCode) may not give the same results. The TA will be looking for specific answers for the grading of this lab and the use of other compilers may result in incorrect answers.

Step #0: Create an electronic version of the Lab4_QA_Sheet.html. You can either keep it as html, copy it to a text file, or a Word compatible document. Fill out the answer sheet with the responses you recorded as you go through the lab.

Step #1: Copy and paste the file lab4.c found on Blackboard to your lab4 directory on zeus and save it as file lab4.c. This file contains a short, easy program that calculates the sum of the integers from 0 up to (and including) an integer that is input by the user.

Attempt to compile this program using the following command.

```
gcc lab4.c -Wall -O2
```

(The -Wall portion of the compile statement is a compiler switch that “turns on” all warnings. **Use this option for all compilation performed for this lab.** The -O2 is an optimize flag that makes your code run more efficiently.)

How many errors appear? ___
How many warnings appear?____

As you have probably experienced by now, when an error occurs during compilation, you will not get an executable file, such as a.out. When a warning occurs without any errors, you *will* get an executable file, as you will find out below.

When you encounter errors during a compilation, it is important to correct the errors that occur at the *top* of the file first. Many times, an error at the top of your program can produce a “domino” effect, causing the compiler to produce additional “phantom” error statements based on code that would actually be correct, if it were not for the original error. Often, such errors arebased on common mistakes such as non-matching curly brackets, missing semi-colons, etc. In such cases, correcting the first few compiler errors may cause many of the others to disappear.

How To View Your Error Messages:

When you compile your program, particularly in the early debugging stages, it may happen that there are so many errors that the earliest errors (the ones most likely to be creating “phantom errors”) scroll off the top of the screen. Since it is important to correct the errors at the top of the program first, you will need to see what they are. You can do this by redirecting the output of the compilation to a source that is easy to view, such as a file or the screen. However, since compilation errors are output to “Standard Error” instead of “Standard Output,” you cannot see them using the typical more command. Instead, you must include an & after the pipe symbol (|).

Thus, in order to compile your program and to send the error messages to the screen, you would compile the file “myfile.c” as follows:

```
gcc myfile.c |& more     // to place the executable in the file a.out;
```

or

```
gcc -o myfile.out myfile.c |& more    // to place the executable in the file myfile.out
```

(In either case, the error messages will be sent to the screen one screenful at a time.)
Note: On some systems, the "|&" pipe does not work in the bash shell correctly. If this is the case, try using the c-shell (csh).

Step #2: Correct only the compilation errors in the code lab4.c. (Do **not** correct the statements labeled "Warnings" yet ... we will address these later.)

Notes:
1) Throughout this lab, when correcting errors found in the code, do *not* delete or edit the incorrect lines! Instead, use // to "comment out" incorrect lines, and copy or rewrite the correct version of the line immediately below the incorrect line.

For example, if you noticed the following line of code:
```
int value = "Zero";     // Initialize value to 0
```
you should *not* delete this line. Instead, place // at the beginning of the line, and insert the correct line immediately beneath it, as follows:
```
// int value = "Zero";        // Initialize value to 0
int value = 0;                // Initialize value to 0 <corrected version>
```

2) Address each Error message from first to last, and recompile the code until all "compilation errors" have been eliminated. There should still be at least one "warning." (It is possible to run a program that includes a "warning," but it is not possible to run a program that includes an "error.")

Step #3: You have now corrected all "Errors," but you have not addressed Warnings. Run the program, and at the prompt, enter the value 10.

What happens? ____________________________________

Notice that even though warnings appeared during compilation, an executable file was created that you were able to run. As you gain experience, you will begin to realize that although some warnings can be safely ignored (such as "Line 385: Variable x is declared but not used"), most need to be addressed. In general, you should strive to create code that will generate NO warnings whatsoever (e.g. if you encounter the above warning "Variable x is declared but not used," you should remove or "comment out" the unused variable x). This will help to make your code more portable, so that if you work on your programming assignments on another compiler at home such as

Microsoft Visual Studio (which for this class, you shouldn't be doing), you will be less likely to have errors compiling the same code when you upload it to GMU.

One warning that should appear in your compilation of lab4.c states that:

warning: format ‘%d’ expects argument of type ‘int *’, but argument 3 has type ‘int’

By now, you should have enough experience with sscanf() to see what the problem with the statement is. Fix ***only*** this problem, recompile and run the program again (enter the value 10).

What value is printed after “The sum of integers 0 to 10 is: “ ? ______
What value *should* have been printed? ______

Another warning that should appear in your initial compilation of lab4.c refers to "statement with no effect [-Wunused-value] . In this case, the "i == 0" in the loop construct will do nothing (why not?). This warning appears because of the -Wall option you placed on the command line. To see this, compile your code without the -Wall option (i.e. gcc lab4.c -O2).

How many warnings appear without the -Wall option? ________

Now recompile with the -Wall option.

How many warnings appear with the -Wall option?   _________

These types of warnings should be corrected, and ignoring them can cause bugs that are very difficult to track down. In order to understand this concept more clearly, for this lab, do *not* fix this "statement with no effect" warning until you are asked to do so.

Step #4: This stage of the lab involves correcting the logic errors contained in the code. Again, for this portion of the lab, be sure to follow the sequence set forth below; otherwise, the results you submit will not be those expected by your TA.

One way to debug logic errors is to use a system-provided debugger. Debuggers can be extremely valuable, and you are encouraged to learn to use debugging tools. However, since debuggers are not always available, it is important to understand that a useful way to obtain information about what is going on in your program is to insert statements that print informative messages at various places in your code.

You will now add debugging statements to your code in order to learn what may be happening during the summation loop to cause the incorrect result obtained above.

Place the following statement inside the for loop of the code:
printf("i = %d sum = %d\n", i, sum); // Added for Debugging
(It should be inserted immediately after the statement: “sum = count;”)

Compile and run the code, using the value 10 as your input.

Answer the following questions:

How many times does the printf statement you added get executed? _____
List in order the numerical values of *sum* output by the printf statement you added:
____

Examine how the variable sum is set within the loop. Sum should not be set to be equal to the variable count, but instead to the variable i. Comment out the statement sum = count, and add the following statement immediately below it:

sum += i;

(Do you see that this statement was an example of a logic error within the program??)

Compile and run the code again, using the value 10 as input.
Answer the following questions again, based on the most recent code execution:

How many times does the printf statement you added get executed? ___________
List in order the numerical values of *sum* output by the printf statement you added:
____

What is the value of sum output by the program in the final statement? ________

Obviously, the loop is still not working correctly. Now, look at the “statement with no effect” warning message that appears when the code is compiled. The variable i is supposed to be set, but instead the variable is somehow being used (hence the warning statement). The reason is that the statement i == 0 is actually a boolean statement, not an assignment statement. Using == instead of = (and vice-versa) is a very common source of bugs, and causes weird program behavior.

Comment out the for loop statement, and copy the statement exactly as it is written to the line below, with the exception that the i == 0 becomes i = 0. (You may notice that there is still a bug in the for loop statement. Bear with us ... we will soon fix it as well.)

Compile and run the code again (using 10 as input). Did the "statement with no effect" warning message go away? _______

Now answer the following questions:

How many times does the printf statement you added get executed? _____
List in order the numerical values of *sum* output by the printf statement you added:
____

What is the value of sum output by the program in the final statement? ________

The for loop still does not execute the appropriate number of times. Look at the for statement carefully. Ask yourself: "Why does the for loop execute the number of times it does? Why doesn't the code within the for loop get executed the proper number of times?" If you still haven't noticed the bug, we will give you the answer (this time). Remember that the semi-colon at the end of the for statement actually ends the loop. Therefore, the code within the brackets on the lines after the for statement aren't in any loop at all, so it only gets executed one time! Placing a semi-colon immediately after a loop statement is a very common programming error, and it is very difficult to notice without putting additional output statements within the loop body.

Comment out the for loop, and copy it to the line below, this time without the semicolon.

Compile and run the program again (using 10 as input). Now answer the questions:

How many times does the printf statement you added get executed? _____
List in order the numerical values of *sum* output by the printf statement you added:
____

What is the value of sum output by the program in the final statement? ________
Is the final value of sum correct? (Look at your answer to step #3 above) _________

Another common logic error that causes incorrect results in loops is having an incorrect **loop termination condition**. That is the problem here. Notice the values that get output within the loop. You are supposed to add the integers from 0 to 10. Which integer is missing? ________

The exit condition given in the loop is i < count. This means that when i is equal to count, the program stops executing the statements in the loop. In this case, when i is equal to 10, the loop exits without adding the value 10 to sum. Errors such as these are often called "off by one" errors, or "boundary condition" errors. It is important, when debugging your code, to check these conditions very carefully to make sure the code will do what is intended at such points.

Comment out the for loop (the last time, we promise!), and write the correct statement on the next line, using the proper loop termination condition.

Compile and run the code again (using 10 as input):

Answer the following questions:

How many times does the printf statement you added get executed? _____
List in order the numerical values of *sum* output by the printf statement you added:
____

What is the value of sum output by the program in the final statement? ________

Step #5: The warning statement that appeared during the earlier compilations referred to using a value before it was set. However, uninitialized variable errors sometimes occur without any warnings to tip you off. The code you have been working with had initialized the variable sum to 0 during its declaration. Often, uninitialized variables can cause errors in your program results. It is important to make sure that variables are set correctly before they are used.

Comment out the statement: int sum = 0;
and replace it with the statement: int sum;

Recompile, and run the code. Has the output changed? ______________________
(This may or may not cause output to change, depending on various factors, such as the system and/or compiler flags.)

One of the first statements you changed in the code was:

```
sum = count;
```

This was necessary because count was not the proper value for sum. However, the statement was proper in the sense that count already had a value before this assignment statement took place. However, your replacement: sum += i is attempting to use the value of sum, which has not yet been set! (If this is not immediately apparent to you, remember that sum += i is equivalent to sum = sum + i. Notice that the value of sum on the right side of the statement cannot be determined because it hasn't been set anywhere yet!) Therefore, the results you get may be unexpected. Different compilers handle such variable initialization differently. Some compilers automatically set all declared variables to 0 (unless otherwise initialized when declared); others do nothing, leaving garbage in the memory location corresponding to the uninitialized variable. Therefore, **do not depend upon the compiler to initialize variables for you!** It is important to check to see that the values of variables are what you expect them to be. In this case, another printf statement printing the value of sum before the loop begins can help find problems like this one.

Commenting and Short-Circuiting Code:

Another useful strategy (which was not used in the preceding steps, but which you may want touse for the next step) has to do with commenting out code that may be causing problems. Often, you will need to narrow down the exact lines of code that are causing a problem, such as a segmentation fault.

A useful debugging method to find the troublesome portion of your code is to "comment out" a portion of the code that may have problems, so that you can check the accuracy of the rest of your code first. Commenting out a particular function or block of code "short-circuits" the actions performed in that section, so that you can determine what portion of code is causing the problem and address problems individually. For example, suppose you have a program that compiles just fine, but that generates a segmentation fault after the user inputs the integer used for the call to function Factorial():

```
printf("Please input an integer: ");
fgets(Buffer, 100, stdin);
sscanf(Buffer, "%d", &x);
value = Factorial(x);
printf("The factorial of %d is %d\n", x, value);
```

You suspect that the problem is in your Factorial() function, but you don’t want to tackle it until you are sure the rest of the program works. One way to do this is to comment out the call to Factorial(x), and replace the result with some value that you know will be correct given a specific input. For example, suppose the integer you are using to test your program is 6. You can comment out the function call and replace it with an assignment statement:

```
printf("Please input an integer: ");
fgets(Buffer, 100, stdin);
sscanf(Buffer, "%d", &x);
// value = Factorial(x);          <= This statement is now “commented out” and will not
execute
value = 720;                 // This simulates a call to Factorial(x) if x is equal to 6, since
6! = 720
printf(“The factorial of %d is %d\n”, x, value);
```

Now, if the program prints “The factorial of 6 is 720" and runs to completion, you know that the Factorial() function is very likely causing the segmentation fault. So, you can complete the rest of the program and correct any other problems before tackling the Factorial() function. Then, when the rest of your program works, you will know that the only remaining problems must be in the Factorial() function. (Of course, don’t forget to uncomment the function call and to remove the assignment line when working on the Factorial() function later!)

Sometimes it is difficult to determine which lines of code are causing segmentation faults. (This is especially true because sometimes the computer will “save up” printf statements in the buffer, so that the statements will not print even though the code containing them has already executed!) One way to discover where the problem lies is to stop the execution of your program before the segmentation fault occurs. This can be done by using the predefined exit() function (available to your program by #including stdlib.h) as a debugging technique. For example, suppose you have commented out (or fixed) the Factorial() function, but running your program still produces a “seg fault.” You now know that the segmentation fault is occurring elsewhere in your program but you don’t know exactly where. Assuming that your code works properly up through the call to Factorial(), you could place a call to the exit() function immediately after the last printf statement shown above, and the program should exit without reaching the incorrect code further down. By moving the placement of the exit() call progressively later in the flow of your program, you caneventually find the line (or combination of lines) that is causing the problem.

In order to do this, begin by placing a call to exit() at a place in your code where you know the program has worked correctly up to that point. (Sometimes this is the first line of code after your variable declarations!) Recompile the code, and run the program again. If the program doesn't perform a segmentation fault, move the exit() call down a few more lines until it does. When the program finally does generate a seg. fault, you will know that the line causing the fault must be between the current location of the call to exit() and the line where you previously placed the call. Usually, inspecting the specific line that is causing the seg. fault will reveal the problem.

One common cause of segmentation faults is division by 0. To check this possibility, add a statement that prints the denominator of a division calculation just before the calculation occurs. Is the value of the denominator 0?? If so, this is likely the source of your segmentation fault! Other causes of segmentation faults include attempting to access NULL pointers. Debugging code that uses pointers will be introduced in a future lab.

Step #6: If you haven't already, create an electronic version of this page. You can either keep it as html, or copy it to a text file. Fill out the answer sheet with the responses you recorded as you went through the lab. Then submit this document using Blackboard AND your revised file lab4.c containing the changes you made throughout the course of this lab.

***** YOU ARE DONE!!! *****

Strategies
To summarize, some common debugging strategies are:

1. Correct compiler errors that occur at the top of the file first.
2. Correct all compiler warnings.
3. Use printf statements liberally to discover what is happening.
    1. Find out the number of times loops are being executed.
    2. Find out what parts of if / else statements are being executed.
    3. Make sure any boolean statements (especially compound boolean statements) are written correctly.
    4. Determine the values of variables before and after they are to be changed, both within loops, within function calls, and anywhere else in your program that may be significant.
4. Make sure that all variables are properly initialized before being accessed or used (including pointer variables).
5. Comment out code and/or use calls to exit() to narrow down where problems are occurring.

We hope you will find that these exercises will be helpful for future program debugging!