ECE 3220 Lab 6 Report

John Walter

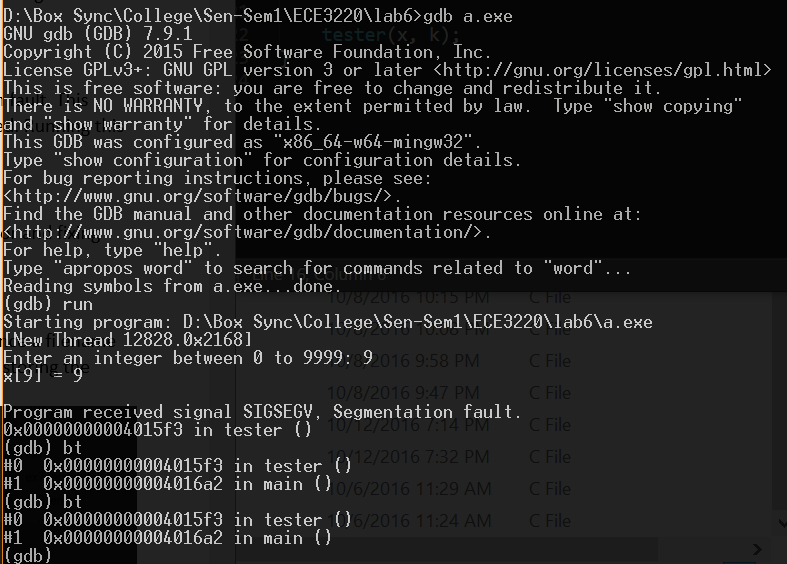
10/10/2016

**Objective**

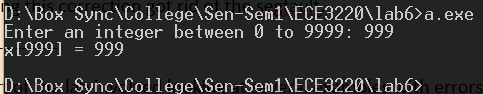
The objective of this lab was to become familiar with how to go about fixing errors and issues in your code. This process introduced the GDB debugger to help assist and aid in the location and correction of segmentation faults and other errors within the code. The GDB debugger is especially useful when dealing with pointers and memory allocation within the source code. This lab also discussed some important uses in the GDB debugger including running a program, backtracking, setting breakpoints, and stepping through the code line by line.

**Results**

The first part of the lab was looking at some code that produced a segmentation fault. This example compiled without any issues or warnings, however, when ran the program failed. Running this is GDB revealed that a segfault was being encountered within the tester() function.

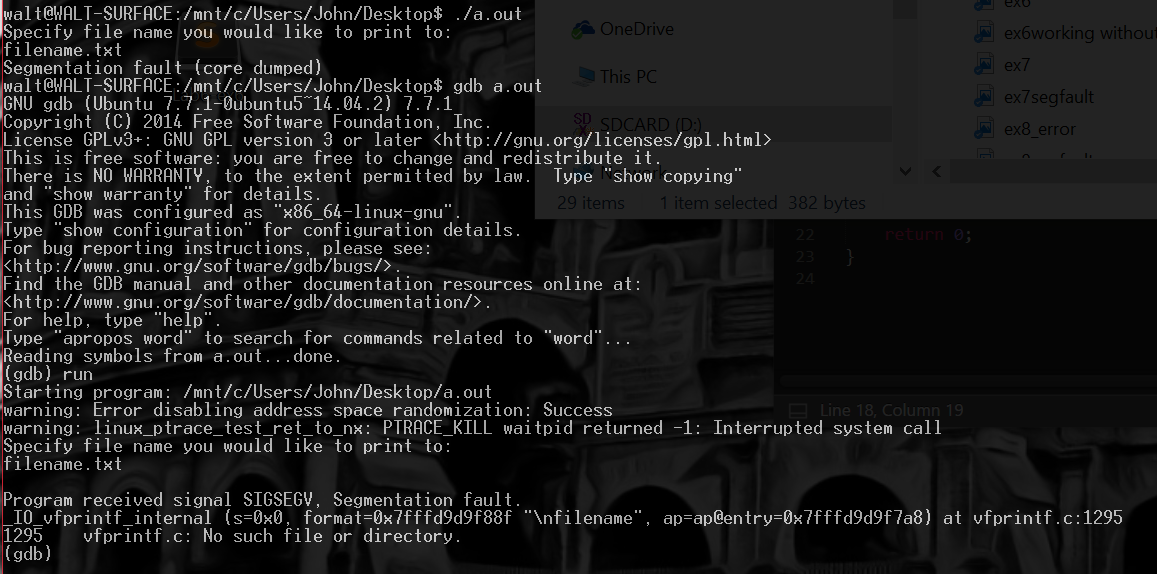


Analyzing the code revealed that a position within an array that was bigger than the array was trying to be assigned a value. This caused a segfault because the program is trying to address memory locations that it does not have access too. To correct this, a location within the array was set to be assigned a value. Notice that that location had to be between 0 and 999 inclusive because the array has a size of 1000. Applying this correction got rid of the segfault.



The next part of the lab involved looking at several examples with errors, finding the error and fixing them with the help of the GDB debugger.

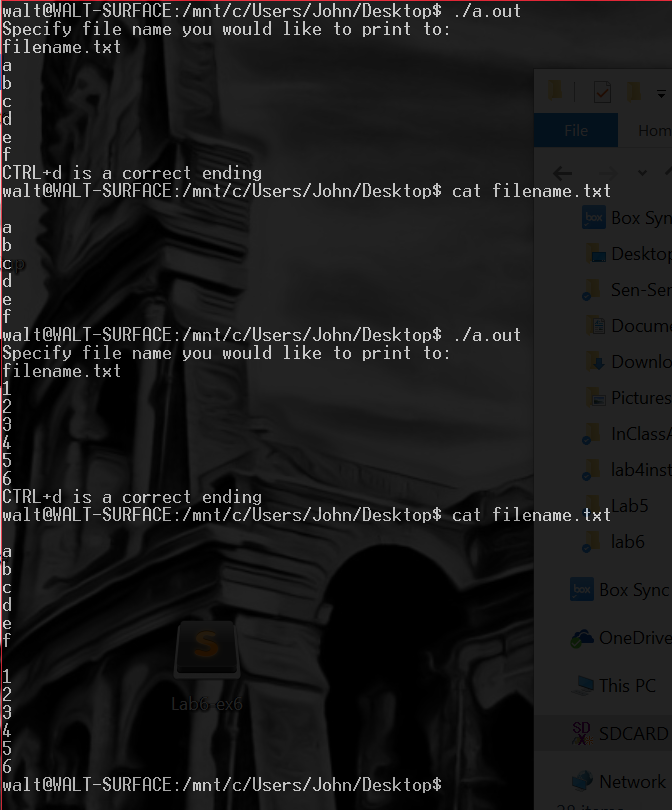
Lab6-ex1.c

This example did not produce any compilation errors and would initially run, however once a filename was entered a segfault occurred. Running the example in GDB declared a segfault when storing the filename. 

Looking back in the code where this process is done, it was found that no memory had been allocated to store the inputted filename. Also, when using scanf() with strings (%s) you do not use the address operator (&) when scanning a string. This is because you want to point at the first character in the array of characters. Applying those fixes got rid of the segfault, however another issue arose when writing to the file. Random characters were being written to the file.

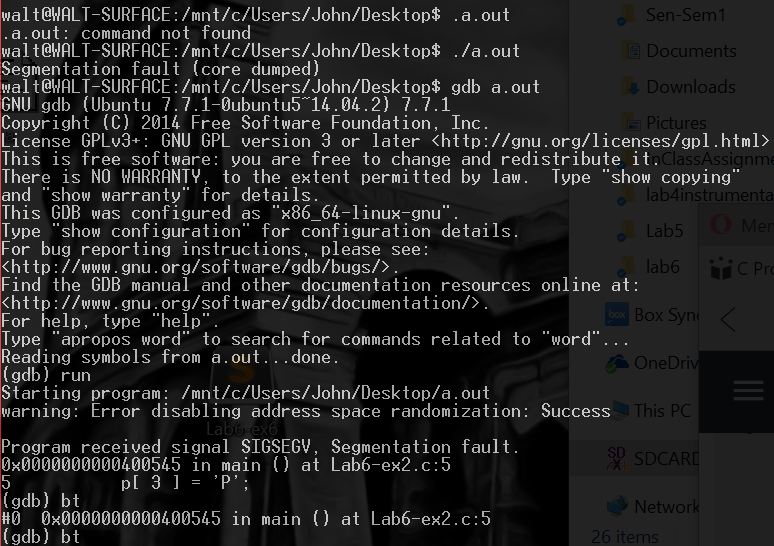


This was due to the (%c) not being present in the fprintf() function and also the address operator being present in front of the variable c. Fixing these issues resolved all errors in the example.

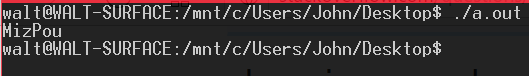


Lab6-ex2.c

When compiling and running this example, a segfault was encountered. Running the program in GDB showed the segfault occurring within main() on line 5 where ‘P’ is being assigned to an element in an array.

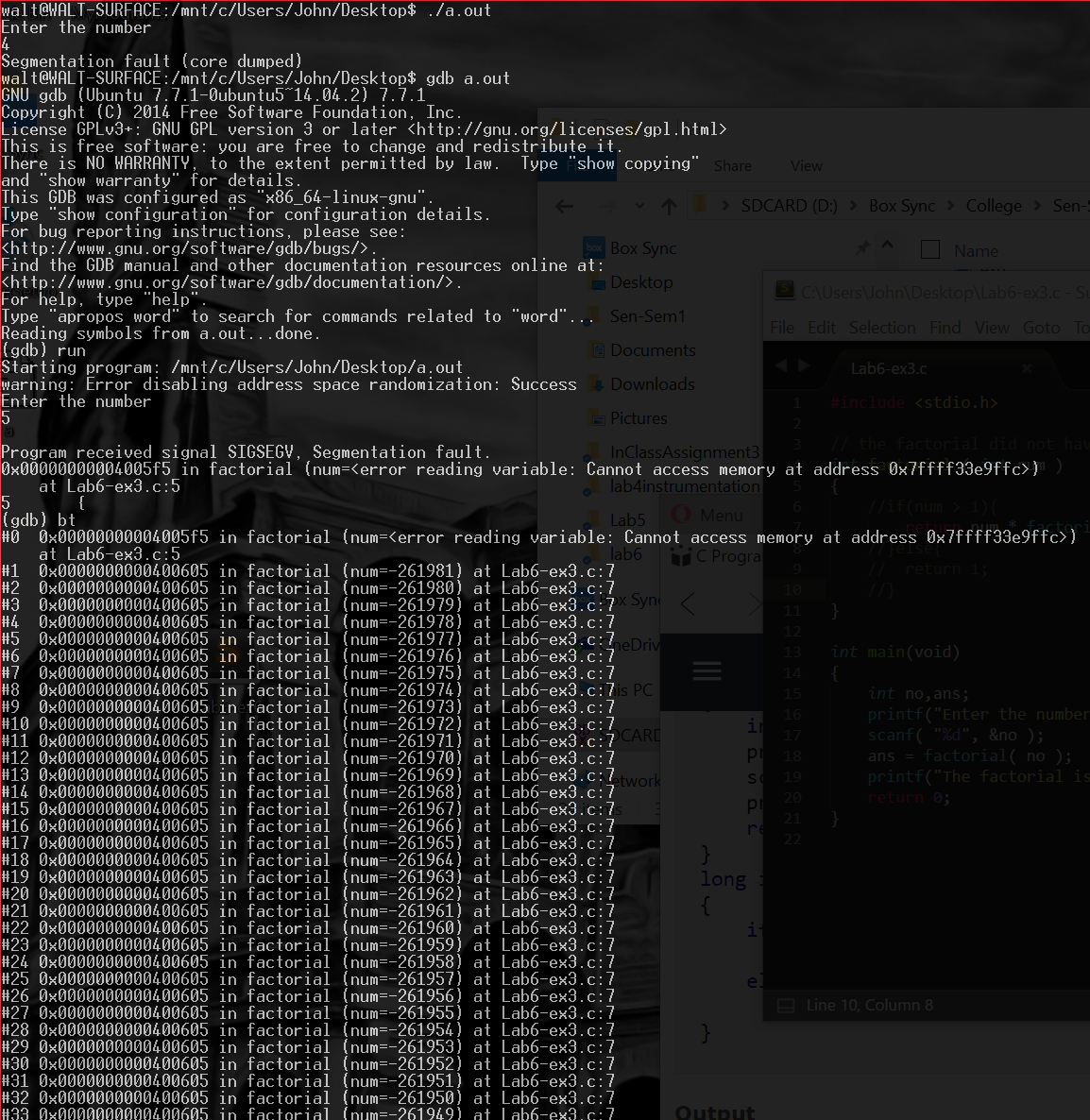


After analyzing the code, it should be noted that when a character pointer is assigned a string at initialization this char pointer becomes constant and therefore whatever it is pointing to cannot be altered. So during execution, this example was trying to write to memory that was being held constant and therefore did not have access too. To fix this issue, the string can be initialized as an array. This then allows for alterations to the string.

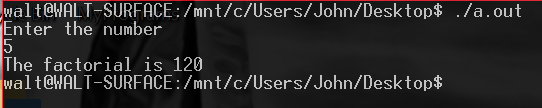


Lab6-ex3.c

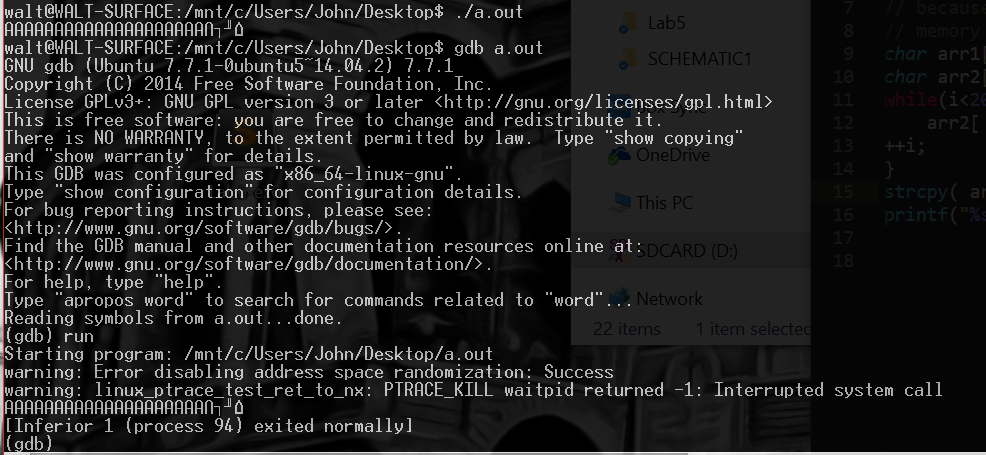
When compiling and running this example, a segfault occurred after inputting a number. Running the program in GDB showed that the segfault was occurring within the factorial() function. It also showed that several memory locations were trying to be accessed but failing.



Analyzing the code resulted in the factorial function being written as a recursive function but with no base case that would stop the recursion. This results in the factorial() function calling on memory locations that it does not have access to. To fix this, a base case of 1 or lower had to be added to stop the recursion once the total factorial had been calculated.



Lab6-ex4.c

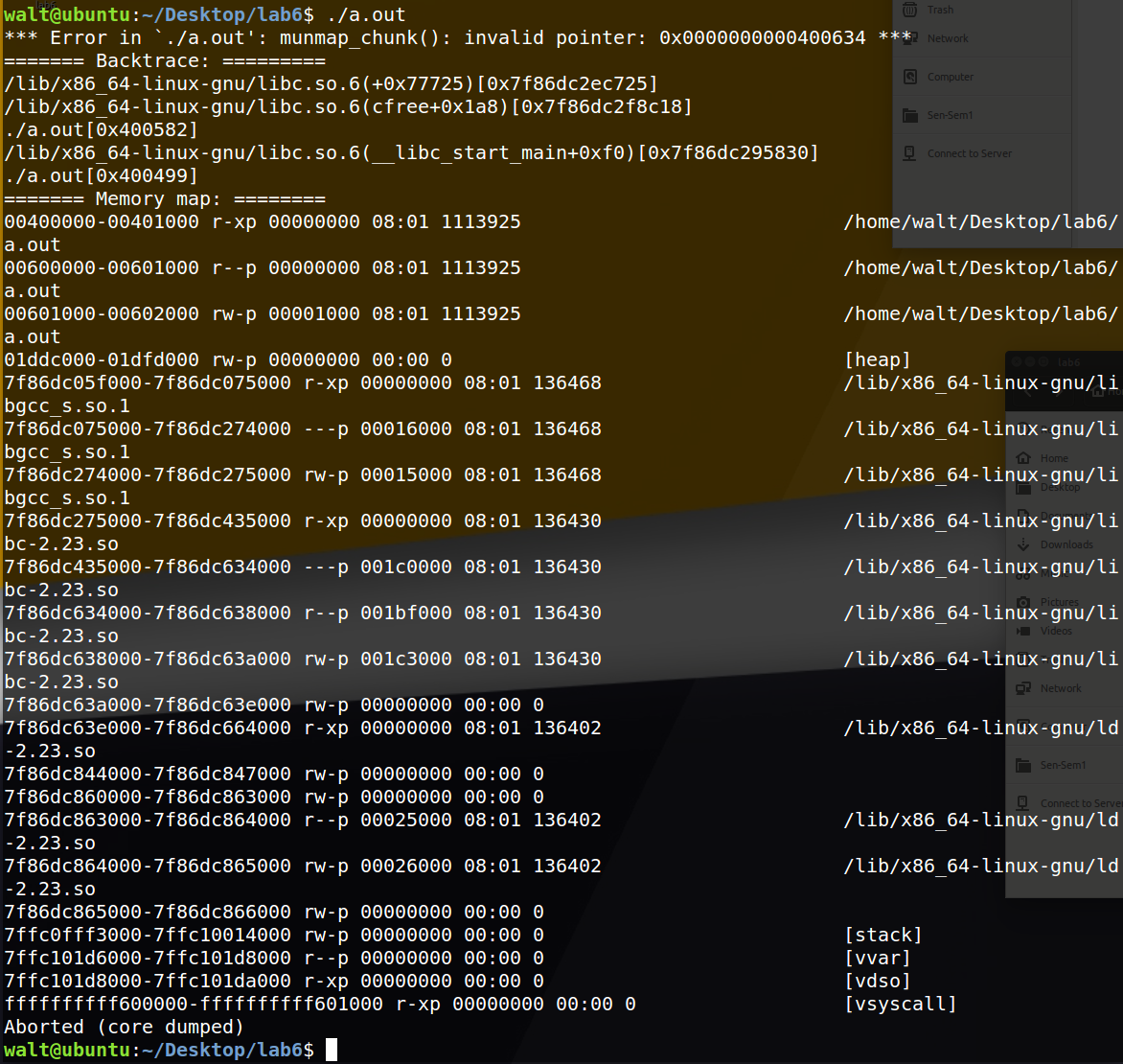
When compiling and running this example, no segfault was encountered but the output produce extraneous characters that were not expected at the end of the string. Running the example in GDB showed that the program exited normally. 

Upon analysis of the code, it was obvious that the error occurring was due to the varying size of the strings being copied. Because arr2 was declared bigger than arr1, and arr2 was then trying to be copied into arr1, this resulted in some conflicting memory issues. These issues caused arr1 to retain some extraneous characters at the end of its string. To fix this, the sizes of each array needed to be declared the same.

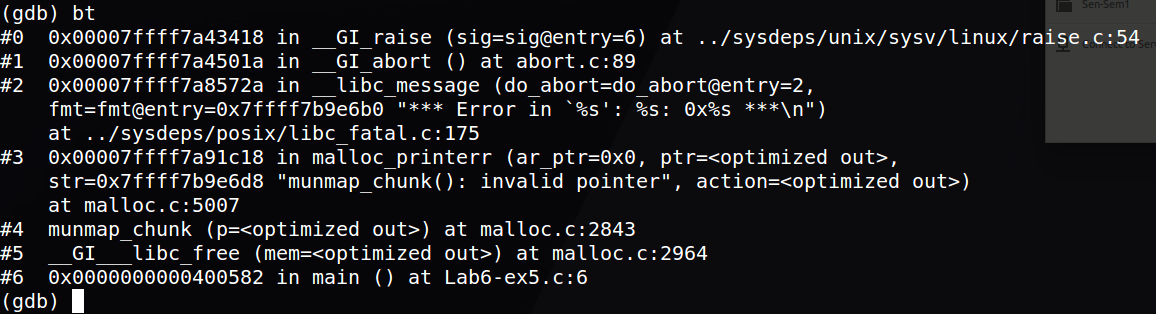
C:\Users\JW048238\AppData\Local\Microsoft\Windows\INetCache\IE\3T1GGB46\ex4working.PNG

Lab6-ex5.c

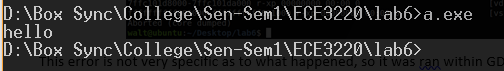
When compiling and running this example, an error is displayed dealing with an invalid pointer.



This error is not very specific as to what happened, so it was ran within GDB. Doing a backtrace showed that inside the main function an error occurred when the free() function was called.

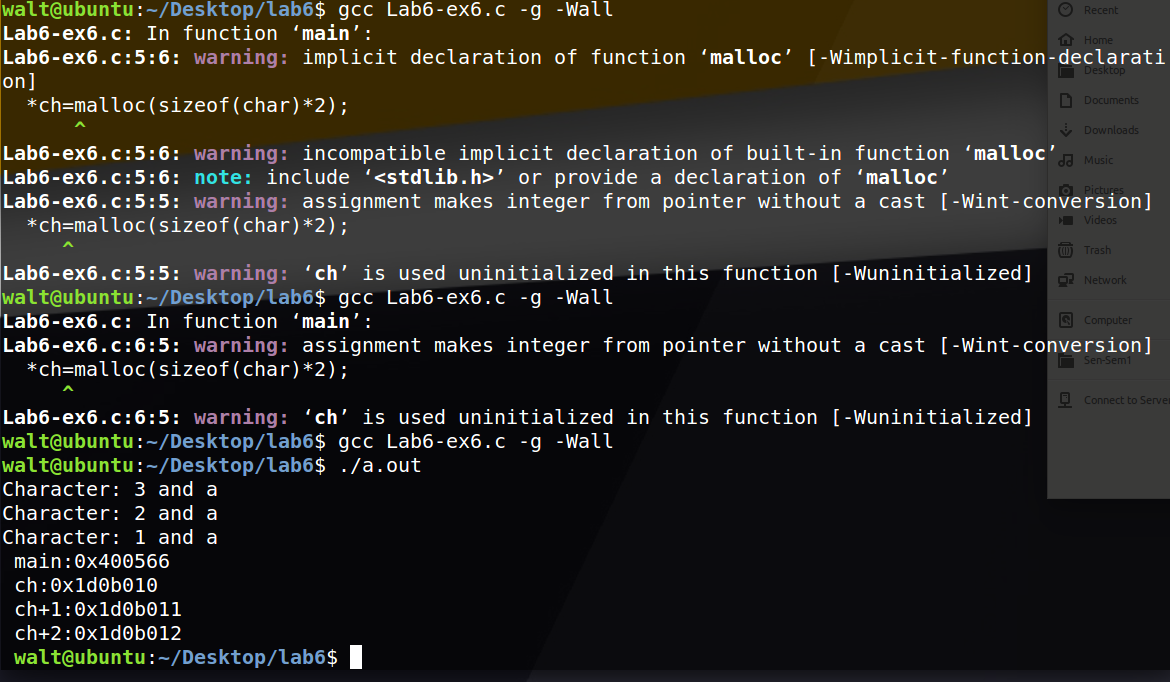


Examining the code reveals that a pointer pointing to a constant string was trying to be freed. This is an invalid operation because we did not malloc any memory for that pointer and therefore cannot free that pointer. Getting rid of that free statement allows the program to run without issues.

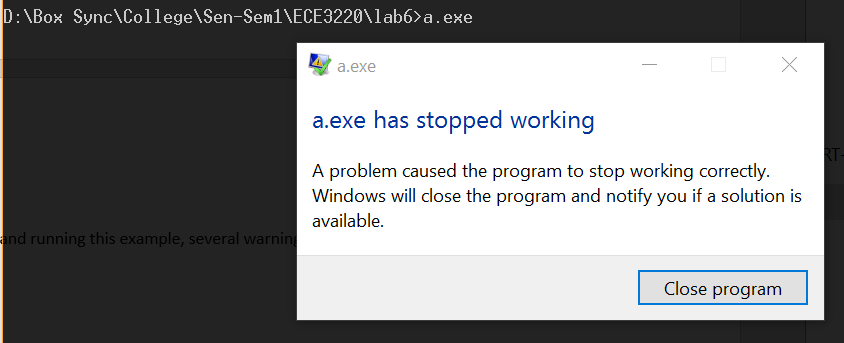


Lab6-ex6.c

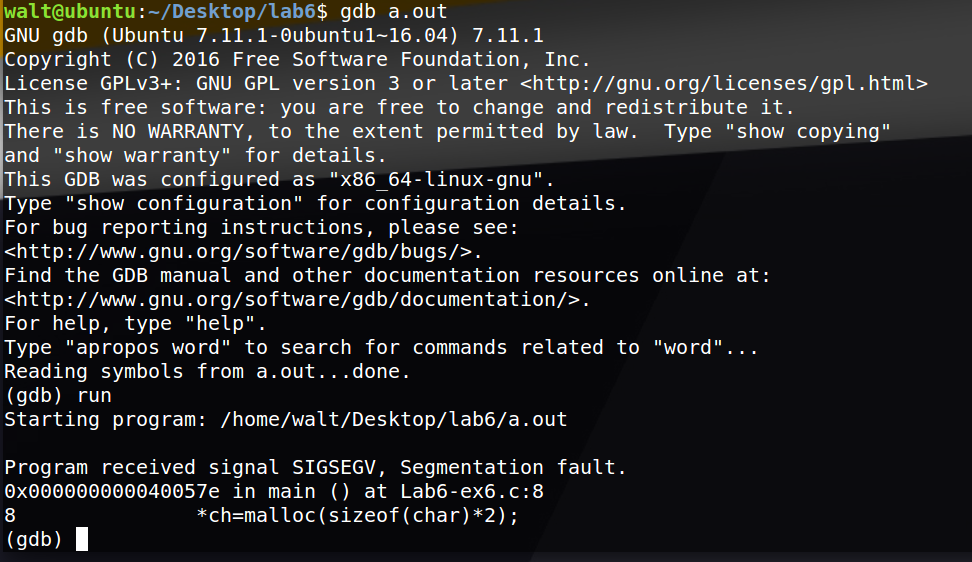
When compiling this example, several warnings are issued.



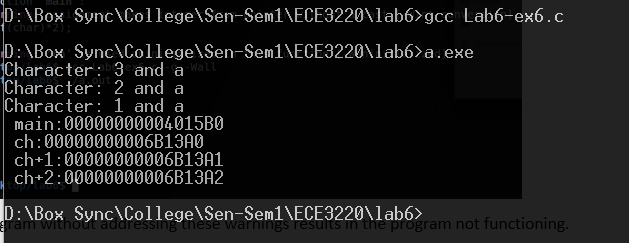
Running the program without addressing these warnings results in the program not functioning.



Running this example in GDB produced a segfault occurring when a “dereferenced” pointer is being assigned some malloced space.

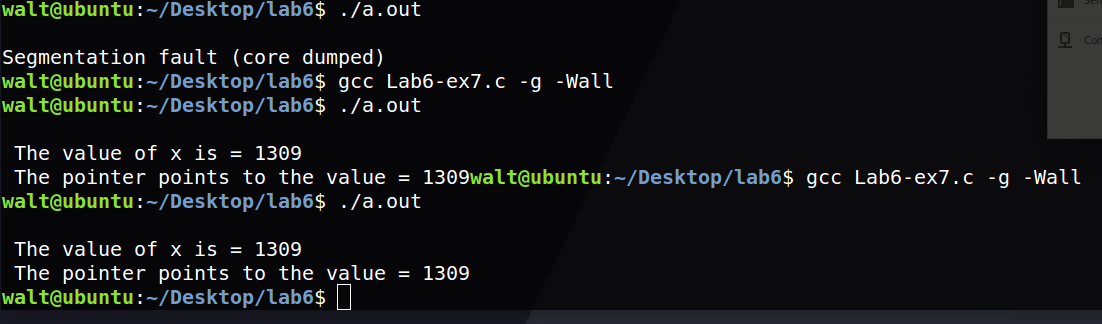


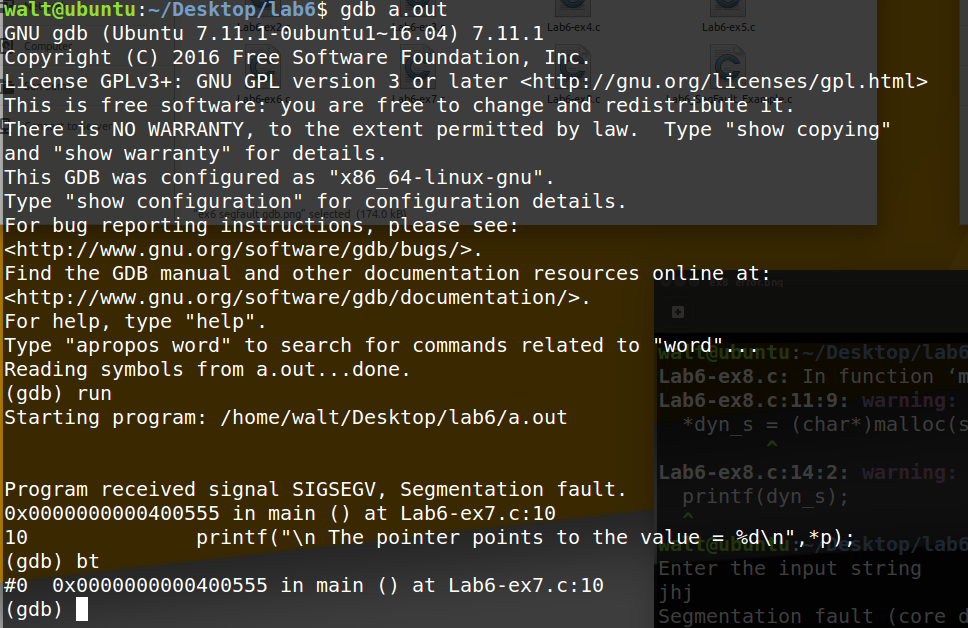
By definition of a pointer, this operation cannot happen. A variable cannot contain malloced space; a pointer has to point to it. In order to fix this error, the pointer should be pointing the malloced space and not trying to “hold” it. This is simply done by removing the (\*) before ch. Another issue present from the warnings it that malloc is being called implicitly. To fix this, a library containing the malloc function must be included. After addressing these issues, the program runs accordingly.



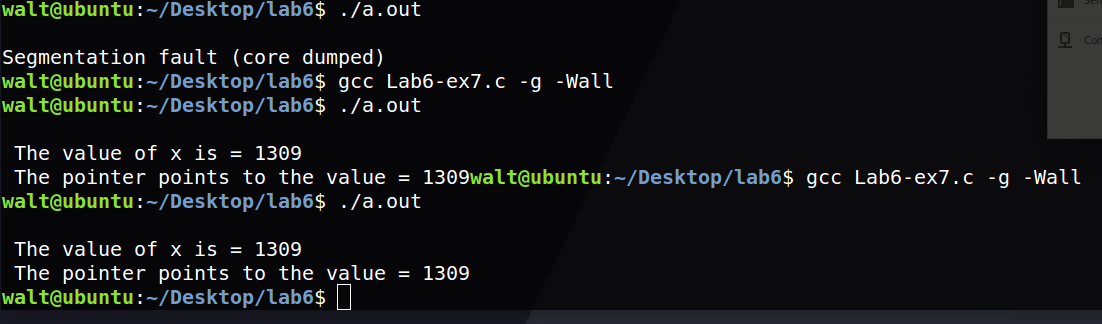
Lab6-ex7.c

When compiling and running this example, a segfault was encountered.



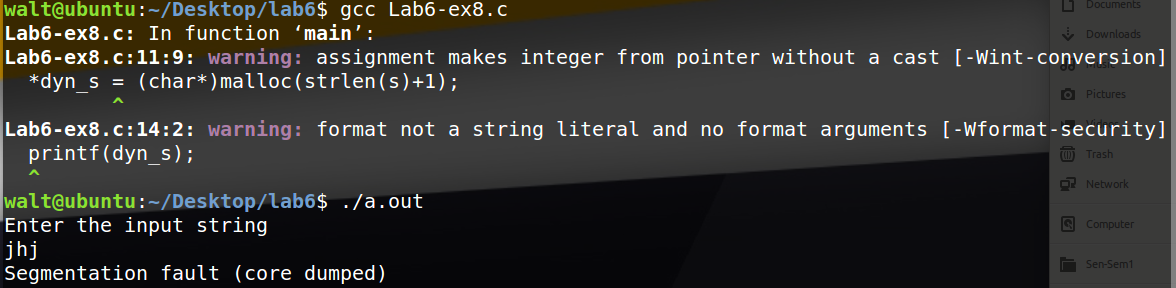
Running this example within GDB showed that the segfault occurred when trying to print the value the pointer was pointing to. 

Observing the code revealed that the pointer was not actually pointing to any memory, and was therefore addressing memory it did not have access too. This issue was easily fixed by setting the pointer to point to the variable x. This allowed the program to dereference the pointer to print the value in that memory location.

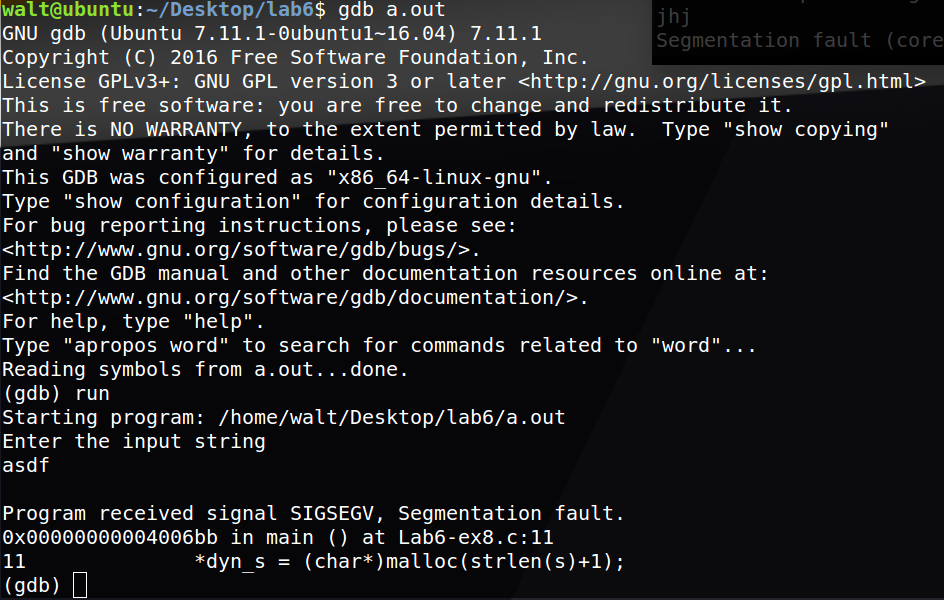


Lab6-ex8.c

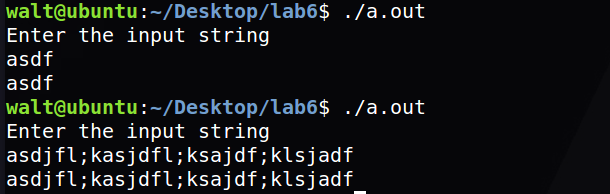
When compiling and running this program, we were presented with a few warnings. Upon running the program, we encountered a segfault.



Running this program in GDB revealed that the segfault was occurring when the dyn\_s was being dereferenced and assigned some allocated memory.



This is an obvious error because dereferenced pointers (or variables) cannot “hold” allocated memory; a pointer has to point to it. By simply removing the dereference (getting rid of the \*) fixes the issue.



**Discussion**

The results of this lab were as expected. Because each example was relatively short and simple, the use of GDB was probably not needed, however for bigger programs I can see it as being a very useful tool. GDB allows us to view exactly where an issue or segmentation fault is occurring so that it can be addressed accordingly. The greatly reduces the time and struggle of having to review every line in the program where the error could be occurring. Knowing how to properly debug and error check a program is a very important concept to know.

Because of previous class work, I encountered no problems or difficulties while debugging and addressing issues within the examples, however ex1 stood out more than the others. The program acts like it is encountering an infinite loop because nothing is being asked of the user when the getchar() function is called. However, once the code was reviewed it was clear what was to be expected from the program.