**21CY682 – Secure Coding lab – I**

**Assignment Topic: Debug a program with logical error**

**Register Number: CYS22005**

**Date: 27/9/2022**

**Name: B.Shebu**

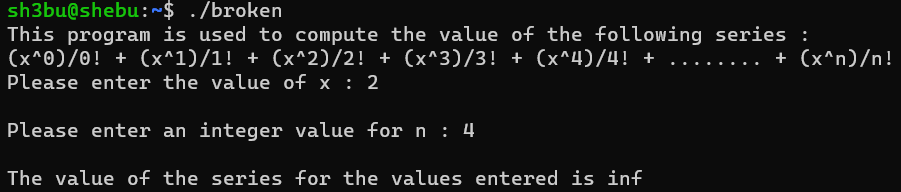
1. **Broken.cpp** - The program is supposed to output the summation of (X^0)/0! + (X^1)/1! + (X^2)/2! + (X^3)/3! + (X^4)/4! + ... + (X^n)/n!, given x and n as inputs. However the program outputs a value of infinity, regardless of the inputs. Trace the errors

**1.** Download the sample program broken.cpp

**2.** Compile the program and execute the program.

$ g++ -g broken.cpp -o broken

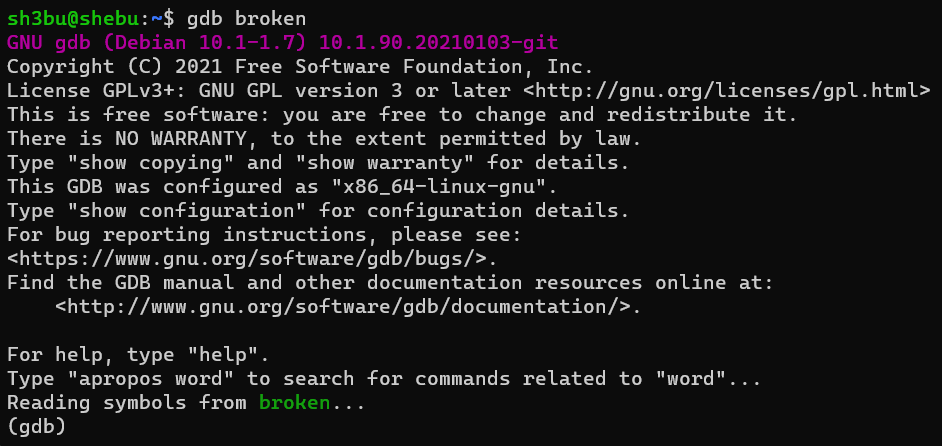
$ ./broken



Whatever the input, the output will be inf. The -g option is important because it enables meaningful GDB debugging.

**3.** Start the debugger

$ gdb broken

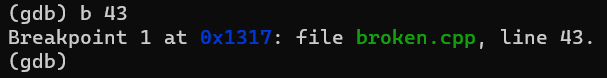


This only starts the debugger; it does not start running the program in the debugger.

**4.** Look at the source code and set a breakpoint at line 43

(gdb) b 43

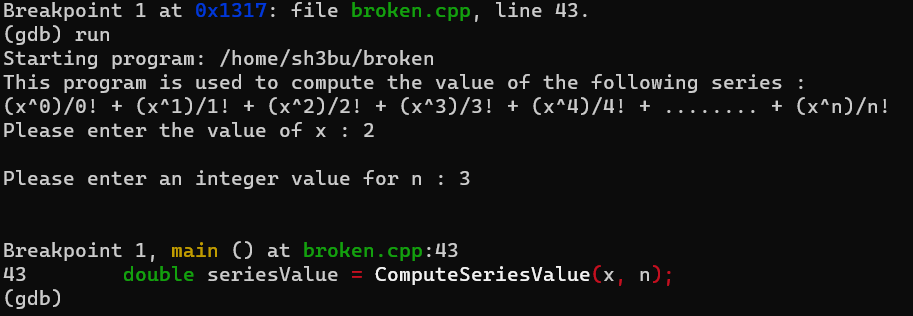
which is double seriesValue = ComputeSeriesValue(x, n);



**5.** Now, we start to run the program in the debugger.

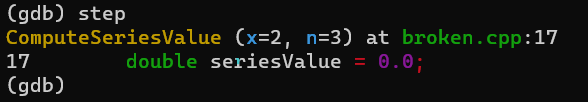
(gdb) run

**6.** The program starts running and asks us for the input. Let's enter the values as x=2 and n=3. The expected output value is 5. The following is a snapshot of the program running in the debugger:



**7.** Step into the ComputeSeriesValue() function .To step into a function call, we use the following command:

(gdb) step



**8.** Next let's step through the program until we get into ComputeFactorial.

(gdb) next

18 double xpow=1;

(gdb) n

20 for (int k = 0; k <= n; k++) {

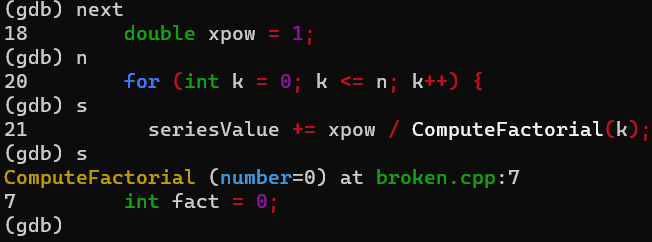
(gdb)

21 seriesValue += xpow / ComputeFactorial(k) ;

(gdb) s

ComputeFactorial (number=0) at broken.cpp:7

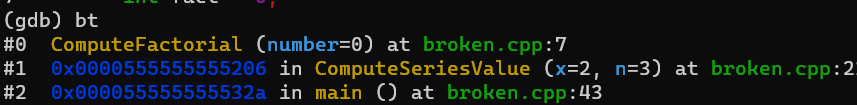
7 int fact=0;



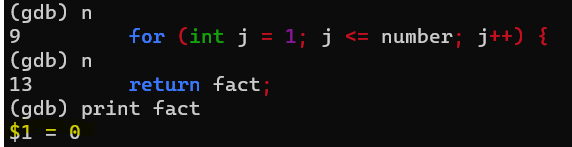
Here we use the next command, which is similar to step except it will step over (instead of into) functions. The distinction doesn't matter here since there are no functions. You may use the shortest, unambigious spelling of a GDB command to save some typing. Here we use n and s instead of next and step, respectively. If the command is simply a repeat of the previous command, you can just hit return, which will execute the last command. Finally, we step (with s) into ComputeFactorial(). (If we'd used next, it would have stepped over ComputeFactorial.)

**9.** To know where you are in the program's execution we can view the contents of the stack using the backtrace command as follows:

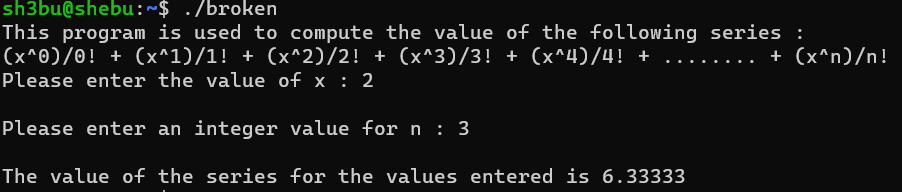
(gdb) bt



**10.** Watching changes We can step through the program and examine the values using the print command.



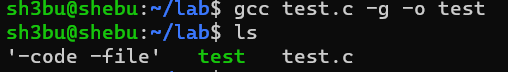
**11.** After changing the fact value to 1. We again run the program & when we given the input a=2 and b=3 we get the result as 6.3333



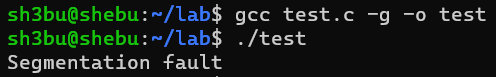
**2. TESTIT.c** - This program causes a core dump due to a segmentation fault. Try to trace the reason for this core dump.

  1. Compile the program using the following command.

gcc testit.c –g –o testit



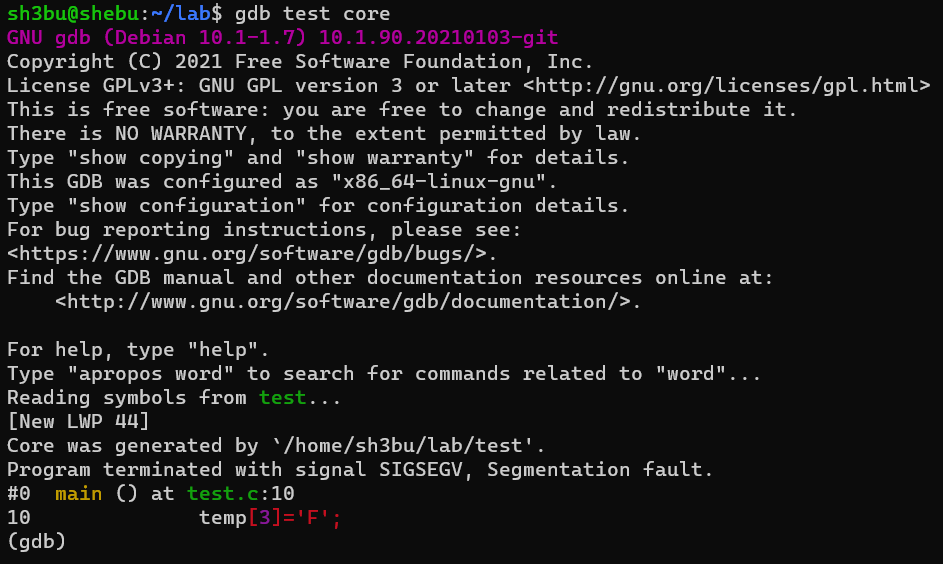
2. Run it normally, you should get the following result: **(SEGMENTATION FAULT)**

****

3. The core dump generates a file called core which can be used for debugging. Since, this program is really short, we will not need to set any breakpoints. Use the following command to start running the debugger to debug the core file produced by testit.

    gdb testit core

4. As we can see from the output above, the core dump was produced       as a result of execution of the statement on line 10: temp[3] =”PARAS”;



5.  Take a closer look at the declaration of temp on line 5

Line 5 char \*temp=”Paras”;



We find that temp is a char\* which has been assigned a string literal, and so we cannot modify the contents of the literal as on line 10. This is what is causing a core dump

6. So we need to make temp variable an array instead of an pointer (ie)**temp []=”PARAS”**. Make changes , compile and run the program to get the output as expected .

