How to Debug Using GDB

We are going to be using two programs to illustrate how GDB can be used to debug code.

Debugging a program with a logical error

The first sample program has some logical errors. 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. We will take you step by step through the debugging process and 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
   ```

   Note: If you need to supply the command-line arguments for the execution of the program, simply include them after the run command, just as normally done on the command line.
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:

   ```
   This program is used to compute the value of the following series :
   (x^0)/0! + (x^1)/1! + (x^2)/2! + (x^3)/3! + (x^4)/4! + ........ + (x^n)/n!
   Please enter the value of x : 2

   Please enter an integer value for n : 3


   Breakpoint 1, main () at broken.cpp:43
   43  double seriesValue = ComputeSeriesValue(x, n);
   ```

   Note that the program execution stopped at our first (and only) breakpoint.
7. Step into the `ComputeSeriesValue()` function

To step into a function call, we use the following command:

```
(gdb) step
ComputeSeriesValue (x=2, n=3) at broken.cpp:17
17  double seriesValue=0.0;
```

At this point, the program control is at the first statement of the function `ComputeSeriesValue (x=2, n=3)`

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. Where are we?
If you want to know where you are in the program's execution (and how, to some extent, you got there), you can view the contents of the stack using the `backtrace` command as follows:

```
(gdb) bt
#0  ComputeFactorial (number=0) at broken.cpp:7
#1  0x08048907 in ComputeSeriesValue (x=3, n=2) at broken.cpp:21
#2  0x08048a31 in main () at broken.cpp:43
```

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

```
(gdb) n
9  for (int j = 0; j <= number; j++) {
(gdb) n
10    fact = fact * j;
(gdb) n
9  for (int j = 0; j <= number; j++) {
(gdb) print fact
$2 = 0
(gdb) n
13  return fact;
(gdb) quit
```

The `print` command (abbreviated `p`) reveals that the value of `fact` never changes. Note that the function is returning a value of 0 for the function call `ComputeFactorial(number=0)`. This is an ERROR!

By taking a closer look at the values printed above, we realize that we are computing `fact=fact * j` where `fact` has been initialized to 0; `fact` should have been initialized to 1. We quit GDB with the `quit` command. Next we need to change the following line:

```
int fact = 1;
```

Recompile the code and run it, you will get the expected output.

Debugging a program that produces a core dump

This program causes a core dump due to a segmentation fault. We will try to trace the reason for this core dump.

Download the program, from here.

1. Compile the program using the following command.

   g++ testit.c -g -o testit

2. Run it normally, you should get the following result:

   Segmentation fault (core dumped)

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

   The output of the above command should look like this:

   bash$ gdb testit core

   GNU gdb 19991004

   Copyright 1998 Free Software ...

   Core was generated by `testit'.

   Program terminated with signal 11, Segmentation fault.

   Reading symbols from /usr/lib/libstdc++-libc6.1-1.so.2...done.

   Reading symbols from /lib/libm.so.6...done.

   Reading symbols from /lib/libc.so.6...done.

   Reading symbols from /lib/ld-linux.so.2...done.

   #0 0x804851a in main () at testit.c:10

   10   temp[3]='F';

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] =F;*

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