EE2160
Testing and Debugging Using the GNU Debugger (GDB)

Download the `Pointers.c` program from Blackboard and SFTP it into your COE account. You will use that program in this tutorial. Commands that you will issue on the command line terminal are indicated in bold e.g. **`ulimit`**

The tutorial guides you through debugging a program that has an error that generates a core dump. Please note that not all errors generate a core dump. The segmentation fault can occur when pointers are trying to access virtual memory that does not exist or not allowed. If you are working on a different system your program may not generate the segmentation fault or the core dump file. To use GDB without the core dump, just invoke it with the executable file (e.g `gdb PointExe`) compiled from `Pointers.c` program.

1. You will use your COE login and password in the following steps. On MobaXterm, or any terminal, login to the gateway system using the following command:
   `ssh yourUserName@gateway.coe.neu.edu`

2. List all the Linux machines available using the *linux-load* command:
   `linux-load`

   ergs: 3 users, load average: 0.12, 0.39, 0.33: last update Sep 4 10:39

   farads: 0 users, load average: 0.86, 0.31, 0.11: last update Sep 4 10:39

   etc.

3. Select the machine with the least number of users and issue the command:
   `ssh -p 27 machineName –l yourUserName`

   The ***machineName*** is one of the following machines: *ergs, farads, hertz, joules, quark, laminar, moles, nano, ohms, etc.*

   yourUserName is your COE account username

   For example: **`ssh -p 27 ergs -l jkimani`**

Respond with your password when requested. You should be able to login.

The following command enables the generation of core dumps on process errors. Issue the command:

```
-bash-3.00$ ulimit -c unlimited
```

You can issue this command from the shell to confirm the current setting:

```
-bash-3.00$ ulimit
-bash-3.00$ unlimited
```

For the GDB to produce useful output for debugging, enable debugging symbols in your binary by compiling with the `‘-g’` flag (`gcc –g Pointers.c –o PointExe`) as shown below:

```
-bash-3.00$ gcc -g Pointers.c -o PointExe
```

Run the executable program `PointExe` that was generated:

```
-bash-3.00$./PointerExe
-bash-3.00$ Segmentation Fault (core dumped)
```

The program intentionally produces a segmentation fault. GDB is invoked by the command `gdb` with the executable file followed by the core dump file. If your program does not generate any errors, just invoke the debugger with the executable created instead, and follow along (e.g **`gdb PointExe`**)

```
-bash-3.00$ gdb PointExe core    // Your core dump might have an extended name.
                Use the ls command to list the files and see the actual name.
```

GDB prints out banner lines (information about the gdb) followed by the reasons for termination starting with the line:

```
Core was generated by ‘./PointExe’.
Program terminated with signal 11, Segmentation fault. // look for these lines
```

GDB depends on some libraries. The binary files for those libraries are also loaded. The contents of the stack frame are loaded next indicating the function where the error occurred in stack frame `#0:`

```
#0  0x00010770 in printResults (chptr=0x10888 "My dog has fleas!") at
Pointers.c:31
31          printf("\n [%c],[%s], [%c]\n", *chptr, chptr, *ptr2);
```

The first line above shows that the error occurred in a function called `printResults` in stack frame `#0` at virtual address `0x00010770` (*`your address might be different`*) in the program `Pointers.c` at line `31.` The next line starting with `31` is the one with the error that caused the core dump.

We can display the function call chain all the way back to `main` with the `backtrace` (**`bt`**) command:

```
(gdb) bt
```

```
#0  0x00010770 in printResults (chptr=0x10888 "My dog has fleas!") at
Pointers.c:31
#1  0x00010738 in main () at Pointers.c:22
```

This shows that `printResults()` (stack frame `#0`) was called from `main` (stack frame `#1`) on line `22` of `main`. When a function is called, it creates a stack frame that tells the computer how to return control to its caller after it has finished executing. We can go to `main` where `printResults()` function was called by going to `frame #1:`

```
(gdb) frame 1
```

From here, we can list a few lines of code before and after the call to `PrintResults():`

```
(gdb) list
```

We can go back to `printResults()` function where the fault occurred by going to `frame #0:`

```
(gdb) frame 0
```

```
#0  0x00010770 in printResults (chptr=0x10888 "My dog has fleas!") at
Pointers.c:31
31          printf("\n [%c],[%s], [%c]\n", *chptr, chptr, *ptr2);
```

From here, we can list a few lines of code before and after the line (`31` above) that resulted in an error:

```
(gdb) list
```

Issuing another `list` command displays next block of code after the first `list.`

```
(gdb) list
```

To examine our debug session, we can also step though the code using breakpoints. For example, to put a breakpoint in main, we use the `break` (abbreviated `b`) command set at `main` by:

```
(gdb) b main
```

We can also set a breakpoint at a specific line by using the line numbers from the `list` command. Set the breakpoint at `line 22` just before the call to `printResults()`:

```
(gdb) b 22
```

To run the program from the debug environment, we use the `run` command. Run the program now:

```
(gdb) run
```

The program runs to the first breakpoint and displays the line to be executed next. Resume program execution by typing `cont` for continue:

```
(gdb) cont
```

The program hits the second breakpoint just before calling `printResults()`. Type `cont` again to resume execution:

```
(gdb) cont
```

Program crashes and `A Segmentation Fault` is generated. Put a third breakpoint at the start of `printResults` so that we can step through the function where the error occurred:

```
(gdb) b printResults
```

Run the program again. If the program was already running, you can agree to start from the beginning by typing `'y'` for yes if asked:

```
(gdb) run
```

When you hit breakpoint `1` continue to breakpoint `2` and `3`. Stop at breakpoint `3`.

```
(gdb) cont
(gdb) cont
Breakpoint 3, printResults (chptr=0x10888 "My dog has fleas!") at Pointers.c:28
28              char ch2 = 'J';
```

From here we can start stepping through the program one line at a time using the `next` command. Issue the `next` command until `A Segmentation Fault` is generated on line `31`:

```
(gdb) next      // issue next until you get the segmentation fault
Program received signal SIGSEGV, Segmentation fault.
#0  0x00010770 in printResults (chptr=0x10888 "My dog has fleas!") at
Pointers.c:31
31          printf("\n [%c],[%s], [%c]\n", *chptr, chptr, *ptr2);
```

Program crashes at line `31` above and `A Segmentation Fault` is generated. Here you can examine the contents of the pointers and variables in the `printf` statement by using the `print` command. Check if

the values conform the specifiers `%c, %s, %c` given in the `printf` statement and any memory access violation.

```
(gdb) print *chptr

$1 = 77 'M'
```

The first print statement shows that the pointer is pointing at a character `'M'` with ASCII decimal representation of `77.`

```
(gdb) print chptr

$2 = 0x10888 "My dog has fleas!"
```

The second print statement shows that the variable is a string `"My dog has fleas!"` starting at address `0x10888.` *// your address might be different*

```
(gdb) print *ptr2

Cannot access memory at address 0x0   // source of error
```

The third print statement shows that the pointer is pointing to an unauthorized memory location. THAT IS WHAT IS MAKING OUR PROGRAM TO CRASH!

From here you can practice with other debugging commands. You can use:

- `man gdb` in Linux environment for the GDB manual
- Online documentation and tutorials e.g. http://www.unknownroad.com/rtfm/gdbtut/

When you are done with debugging in GDB, exit the environment by using the `quit` command:

```
(gdb) quit
```