Debugging C Programs

The purpose of this lab is to introduce you to the use of "**gdb**," the GNU debugger program. While one commonly used simple debugging technique is simply to put print statements at various points in your program to debug logical errors, a more elegant method is to use a debugging program. Such a program makes it possible not only to trace values at various points in a program without altering and recompiling your source code, but you are also able to pause and step execution and trace parameter passing and subprogram calls.

This tutorial only introduces you to the bare essentials of `gdb`. For more information check out links obtained by googling "Introduction to gdb".

1. The following subprogram is intended to convert a 4-hexadecimal digit value into a 16-bit binary sequence, stored as a string. Type the source code exactly as shown into a file called `lab3subprog.c`:

```
typedef unsigned int bit16;
void cvt2bitstr(bit16 x, char* strng){
  int i;
  char* str;
  for( i = 0; i < 16; i++){
    str[i] = (char) ((x & 0x8000) >> 15) | 0x30;
    x = (x << 1) & 0xffff;
  }
  str[16] = '\0';
  return;
}
```

   WARNING: This subprogram has at least one runtime error. Do not correct them yet.

2. Compile the subprogram to obtain an object program as follows but do not attempt to execute the program:

   `gcc -O0 -g -c lab3subprog.c`

   NOTE: It is important that the `-O0` and `-g` flags be included in the compile as these insure that no optimization of your code occurs and that breakpoints are provided for gdb. Since you will almost always need to debug your program, this should be your usual method for compiling if you want to use gdb.

   The code as shown should compile successfully. Correct any copying errors and recompile if necessary until an object file, `lab3subprog.o`, is successfully obtained.

3. To test the subprogram, define a source file, `lab3prog.c`, exactly as shown:

```
#include <stdio.h>
typedef unsigned int bit16;
void cvt2bitstr(bit16, char*);
void main(){
  bit16 x;
  char bitstr[16];
  x = 0x1234;
  cvt2bitstr(x,bitstr);
  printf("%s \n", bitstr);
}
```

4. Compile `lab3prog.c` using the flags `-O0` and `-g` as shown previously:

   ```
   gcc -O0 -g -c lab3prog.c
   ```

   Again the file as shown should compile successfully.

5. After all copying errors have been corrected, link it with `lab3subprog.o` to obtain an executable file `run_test`. Do not proceed to the next step until the files are compiled and linked successfully (i.e., without warnings or errors).

6. Run the executable file and observe the result. You should obtain the error message:

   ```
   Segmentation Fault
   ```

   This is one of the most common logical errors in both C and assembly language programs and often occurs when an invalid address is used to retrieve either an instruction or some data during the performance of the instruction execution algorithm. The main thing to observe is that not much information is provided to determine where the error has occurred.

7. Instead of running the executable file, this time launch the gdb debugger program with the command line:

   ```
   gdb run_test
   ```

   A preamble should be displayed followed by the prompt:

   ```
   (gdb)
   ```

   This indicates the debugger is now running and a "gdb command" can be entered.

8. Enter the gdb command "run" following the prompt. This will run your executable file, `run_test` and should produce something like the following:

   ```
   (gdb) run
   Starting program: /home/dixon/sfuhome/CMPT295/LAB3/run_test

   Program received signal SIGSEGV, Segmentation fault.
   0x0000000000400627 in cvt2bitstr (x=4660, strng=0x7fffffffe620 "P\006@")
       at lab3subprog.c:7
   7     str[i] = (char) ((x & 0x8000) >> 15) | 0x30;
   ```

   Notice the information that is provided:

   - the subprogram where the error occurred is identified
   - the formal parameters of the subprogram and their values are shown.
   - the line within the subprogram where the program was terminated because of the error is displayed
   - the name of the file where the subprogram with the segmentation fault is located.

9. The values of local variables in the subprogram can be displayed. For example enter the following gdb command:

   ```
   (gdb) print i

   $1 = 0
   ```

   This will display the value of the subscript `i` in the line of source code where the program was terminated. In this case the value is 0 so we can infer that this is the first time this line, which is inside a for loop, is executed.

10. To see more of the subprogram , and confirm that `i` is indeed the loop counter inside the loop, type the following:

    ```
    (gdb) list
    ```

    This results in part of the subprogram in which the line of source code lies where the program terminated:

    ```
    1 typedef unsigned int bit16;
    2
    3 void cvt2bitstr(bit16 x, char* strng){
    4   int i;
    5   char* str;
    6   for( i = 0; i < 16; i++){
    7     str[i] = (char) ((x & 0x8000) >> 15) | 0x30;
    8     x = (x << 1) & 0xffff;
    9   }
    10   str[16] = '\0';
    ```

11. Segmentation faults are often the result of an undefined or faulty address. Since `str` is declared to be a pointer (Line 5), examine the value of this pointer in line 7 as follows:

    ```
    (gdb) print str
    ```

    The output should be as follows:

    ```
    $2 = 0x0
    ```

    Generally, an address of 0x0 is an invalid address. To confirm that this is the case, type the following:

    ```
    (gdb) print str[i]
    ```

    The following message should be displayed:

    ```
    Cannot access memory at address 0x0
    ```

    The problem is that the declaration of `str` does not allocate any space for `str`. Based on how `str` is used in line 7, `str` should be declared as an array. Further inspection of the function reveals a formal parameter, "`strng`", which does not play any role in the function. In fact, its purpose is to return the bit string after it has been calculated. Therefore, a typo has presumably occurred and the name of the formal parameter should be changed to "`str`" rather than "`strng`".

12. Exit from the gdb debugger with the following command:

    ```
    (gdb) quit
    ```

    You will be advised that a debugging session is active. Enter 'Y' to exit anyway.

13. Edit the subprogram in the file `lab3subprog.c` commenting out the declaration in line 5 (Don't delete it as this will affect the line numbers provided as references in the rest of this lab):

    ```
    char* str
    ```

    and replacing the formal parameter, "`strng`", with "`str`":

    ```
    void cvt2bitstr(bit16 x, char* str){
    ```

14. Recompile the subprogram file to obtain a new object file, `lab3subprog.o`, and then link it with the object file, `lab3prog.o`.

    NOTE: It is important that all modified files be recompiled and a new executable file be constructed to insure that all changes are reflected in the new executable file. Don't forget the -O0 and `-g` flags in the compile command.

15. Launch the debugger again with the command `gdb run_test`.

16. Do not type "run". Instead you are going to set up some breakpoints that will cause the program to pause. You will then be able to examine the values of variables that may help to detect any further logical problems.

    To set up a breakpoint each time the subprogram `cvt2bitstr` is called, enter:

    ```
    (gdb) break cvt2bitstr
    ```

17. Now type "`run`". The program will run until the subprogram `cvt2bitstr` is called. The program will pause and display:

    ```
    Breakpoint 1, cvt2bitstr (x=4660, str=0x7fffffffe620 "P\006@")
        at lab3subprog.c:6
    6   for( i = 0; i < 16; i++){
    (gdb)
    ```

18. Enter "list" to obtain a listing of the source code on either side of the break point.

19. In order to check the values being assigned to `str[i]` for each iteration of the loop, a breakpoint needs to be set immediately after the assignment of a value to `str[i]`. From the listing just produced, a breakpoint needs to be set on line 8. This is achieved with:

    ```
    (gdb) break 8
    ```

20. Now resume execution with:

    ```
    (gdb) continue
    ```

    Execution will continue until the breakpoint at line 8 is encountered again when the loop is repeated. The following is then displayed:

    ```
    Continuing.

    Breakpoint 2, cvt2bitstr (x=4660, str=0x7fffffffe620 "0\006@")
        at lab3subprog.c:8
    8     x = (x << 1) & 0xffff;
    (gdb)
    ```

    NOTE that the number 2 is assigned to this breakpoint. Each breakpoint that is set is given a unique number. For example, the breakpoint assigned to "`cvt2bitstr`" is breakpoint 1.

21. The `print` command can now be used to print out the value of `str[i]`. Since the purpose of the program is to convert a 16-bit integer to a string of one's and zero's, the value should be either 0 or 1:

    ```
    ((gdb) print str[i]
    $1 = 48 '0'
    (gdb)
    ```

    The result is expressed as the ASCII value (in decimal) and as a character. If the result looks promising, we resume execution with:

    ```
    (gdb) continue
    Continuing.

    Breakpoint 2, cvt2bitstr (x=9320, str=0x7fffffffe620 "00@") at lab3subprog.c:8
    8     x = (x << 1) & 0xffff;
    (gdb)
    ```

NOTE: It is also possible to step through the program, one instruction at a time, with the gdb command "next."

22. We are now paused with the second iteration of the loop. So the value of the next bit is obtained with:

```
(gdb) print str[i]
$2 = 48 '0'
(gdb)
```

23. Continue the execution for a few more iterations to see that the character values are '0' or '1'.

24. To resume execution without further interruption, we need to remove the breakpoint on line 8 first. This is achieved with:

```
(gdb) disable 2
(gdb) continue
```

NOTE that the breakpoint number originally assigned to the breakpoint is used to identify which breakpoint to remove.

The execution now resumes without further interruption, with the main program printing out the hex value `0x1234` as a string:

```
Continuing.
0001001000110100
[Inferior 1 (process 6520) exited normally]
(gdb)
```

25. Terminate execution of `gdb` with the "quit" command.