COMP 2401 -- Tutorial #8
Debugging with GDB

Learning Objectives

After this tutorial, you will be able to:

- Use GDB to help debug your programs by
  - Setting breakpoints
  - Printing variable values at points during execution
  - Navigating the function call stack at points during execution

Tutorial – Introduction to GDB

GDB (the GNU Project Debugger) is a debugger that allows you to see what is happening inside a program while it executes. It can be used for debugging programs written in different languages including C and C++.

GDB can start a program, specifying information might be needed for running the program. It can also stop the program at a specific line and on specific conditions. When the program stops, GDB lets you examine the running program by printing the value of the variables inside the program.

In this tutorial, you will see how to do the above tasks. Then you are asked to use GDB and find errors in some examples and fix them.

1. Download the file `T08.tar.bz2` from the tutorial page in *cuLearn*. Extract and read through the tutorial files.

2. Create the `example1` executable using the Makefile and then run `gdb` with this executable:

   ```
   make example1
   gdb ./example1
   ```

   The `gdb` program is now running and it should look something like:

   ```
   ~/T08$ gdb ./example1
   GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
   Copyright (C) 2016 Free Software Foundation, Inc.
   License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
   This is free software: you are free to change and redistribute it.
   There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
   and "show warranty" for details.
   This GDB was configured as "x86_64-linux-gnu".
   Type "show configuration" for configuration details.
   For bug reporting instructions, please see:
   <http://www.gnu.org/software/gdb/bugs/>.
   Find the GDB manual and other documentation resources online at:
   ```

```
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./example1...done.
(gdb)
```

3. Your program is now loaded into `gdb` and ready to be run (executed). The `list` command in `gdb` will give you a partial listing of the source code. Try typing:

   ```
   (gdb) list
   ```

   The `(gdb)` above denotes the `gdb` prompt (you do not type this). It should look something like:

   ```
   (gdb) list
   1    #include <stdio.h>
   2    #include <stdlib.h>
   3
   4    void printBinary(int);
   5    int power(int, int);
   6
   7    int main() {
   8        int n;
   9        printf("Enter an integer number:");
   10        scanf("%d", &n);
   (gdb)
   ```

   The `list` command shows about 10 consecutive lines of code. Notice that it didn't start at the beginning of the code. You can specify where in the code you want to list by giving a line number or a function name (only if you used `-g` when compiling).

4. Try the following commands:

   ```
   (gdb) list 20
   (gdb) list printBinary
   ```

   When you use the `list` command, it shows about 10 lines of code centred around the part you specify.

   Note: `gdb` *lets you be lazy. It allows you to use prefixes of commands, as long as the prefix uniquely specifies that command. For example, since there is only one command starting with the letter L, which is* `list`*, you can use* `l`, `li`, `lis` *and* `list` *to invoke the* `list` *command. So, all of the following are the same:*

   ```
   (gdb) l
   (gdb) list
   (gdb) lis
   ```

5. Now let's execute the program by giving `gdb` the command `run`. It executes your program as if it were being executed from the shell. When asked for input, you type the input in. When there is output printed out, it shows up on the screen. For example, running `example1` looks like:

   ```
   (gdb) run
   Starting program: /home/student/T08/example1
   Enter an integer number:
   ```

   You can also pass command line arguments and do input/output redirection when executing a program in `gdb` . This is especially useful when (re)typing in input is tedious...

   For example, (using the `in.txt` file)

   ```
   (gdb) run < in.txt
   ```

   Ok, you now know how to list your code and how to run it. Let's look at our first bug in the code. Try :

   ```
   (gdb) run
   Starting program: /home/student/T08/example1
   Enter an integer number: 123
   ```

   After entering `123` we would see:

   ```
   Enter an integer number:123
   Decimal = 123

   Program received signal SIGFPE, Arithmetic exception.
   0x0000000000400733 in printBinary (n=123) at example1.c:31
   31                      printf("%d", n % pow);
   (gdb)
   ```

6. Now, what does this mean? It tells us that we have an arithmetic exception. It tells us that it occurred inside the function `printBinary` and it tells us the value of the input parameter (`n=123`). It also tells us what line the code crashed on (line 31 of the file `example1.c`) and then shows us that line. To see the code around it, try:

   ```
   (gdb) list 31
   ```

7. We can also probe the "state" of the program when the code crashed. The `print` command will display the current value of a variable. For example, try:

   ```
   (gdb) print pow
   (gdb) print n
   ```

   The print command also lets us print expressions. For example, we can look inside arrays, ask for memory locations, and dereference pointers. Try the following:

   ```
   (gdb) print n*pow
   (gdb) print (n+10)/(pow+1)
   (gdb) print &n
   ```

Notice that whenever you print something, the line starts with `$n` for some integer `n`. This is a label for the value of the expression that was printed and can be used to access this value later. For example, try:

```
(gdb) print $4
```

This is the value displayed the 5th time you called `print` (counting starts at zero).

8. Hopefully you can see what the problem with the code is already. But most bugs will not be so simple. What can we do to try and find more difficult bugs. We can look at the function call stack and see what it looked like when the crash occurred. To view the function call stack, we use the "backtrace" command, which in our case should look something like:

```
(gdb) backtrace
#0  0x0000000000400733 in printBinary (n=123) at example1.c:31
#1  0x00000000004006af in main () at example1.c:13
(gdb)
```

9. We can move into any stack frame that we wish, with the `frame` command, to see what the state is. For example, we can move into stack frame 1 (which is the `main()` function).

```
(gdb) frame 1
#1  0x00000000004006af in main () at example1.c:13
13              printBinary(n);
(gdb) list
8        int n;
9        printf("Enter an integer number:");
10        scanf("%d", &n);
11        printf("Decimal = %d\n", n);
12        printf("Binary = 0b");
13        printBinary(n);
14        printf("\n");
15        return 0;
16    }
17
(gdb)
```

This tells us that we are currently (executing) at line 13 of the `main()` function. We can use the `print` command to view any variables in this current stack frame. That is, we can ask what the values of all the local variables in this function are (even though the execution of the program is not really in this stack frame).

To get back to where we started from, (the top of the function call stack) we use:

```
(gdb) frame 0
```

10. Now let's see how we can stop our program before it crashes (where the exception happens), look at the state, and then step through the program (to hopefully find our bug).

    The `break` command tells `gdb` to stop the program when it gets to a certain line (or reaches a function). It specifies a breakpoint for the code. For example, we know that our code crashed on line 31. We can set a breakpoint on this line as follows:

    ```
    (gdb) break 31
    Breakpoint 1 at 0x40072f: file example1.c, line 31.
    (gdb)
    ```

    Notice that it gives a number to the breakpoint (1 in this case). Now run the program and again give `123` as input integer number. You should see something like:

    ```
    (gdb) run
    Starting program: /home/student/T08/example1
    Enter an integer number:123
    Decimal = 123

    Breakpoint 1, printBinary (n=123) at example1.c:31
    31              printf("%d", n % pow);
    (gdb)
    ```

11. At this point, you can probe the state of the program. (use `print` to see what some values are, for example). After looking at the state of the program, we can now use `continue`, `step`, or `next` to move on and resume execution of the program. Each of these does something different.

    ```
    (gdb) continue
    ```

    will continue the execution of the code until it reaches another breakpoint, crashes, or the program ends.

    ```
    (gdb) step
    ```

    will execute the next line of code, and then stop. If the previous line of code was a function call, `step` takes you inside the new function and stops at the first line of code of the new function.

    ```
    (gdb) next
    ```

    will execute the next line of code inside the same function. If the previous line of code was a function call, it executes the entire function and brings you to the next line of code after the function returns.

12. Try these three different ways of continuing execution when your program is using `example1` with `123` as the input number. To see the difference between `step` and `next`, run `example1` with a breakpoint at line 13 (where `printBinary` has been called). When `gdb` stops at line 13, execute `step` and then use `list` to see what the current line is. Try the same again, but this time execute `next` when you reach line 13 and then again see the current line by `list`.

13. More about `break`. You can set a temporary breakpoint with `tbreak`.

```
(gdb) tbreak 31
```

will set a breakpoint at line 31 and will stop execution the first (and only the first) time the code reaches this line.

You can list all the breakpoints you have set with:

```
(gdb) info breakpoints
```

You can disable any breakpoint with the `disable` command. For example, to disable breakpoint 2, you would use:

```
(gdb) disable 2
```

For example, try:

```
(gdb) break 31
(gdb) break 10
(gdb) tbreak 33
(gdb) info breakpoints
(gdb) disable 2
(gdb) info breakpoints
```

Sometimes you might want to skip a breakpoint one more time.

```
(gdb) ignore 1 3
```

tells `gdb` to ignore breakpoint number 1 a total of 3 times.

Tutorial – Debugging

1. Make `example1`. Find and fix any bugs.
2. Make `example2`. Find and fix any bugs.
3. Make `example3`. Find and fix any bugs.
4. Make `example4`. Find and fix any bugs.
5. Make `example5`. Find and fix any bugs.
6. Make `example6`. Find and fix any bugs.

Exercises

1. Use the following structure and implement a binary search tree. Assume that the head of the list will be a `TreeNodeType` pointer.

   ```
   typedef struct TreeNode {
        int data;
        struct TreeNode *right, *left;
   } TreeNodeType;
   ```

   You should write insert, delete, search, and print functions to support the BST.