ENGR170 Lab
Debugging by using ddd
Dr. Michael Marefat, and Alex Mitev

1. Introduction

The purpose of this lab is to introduce you to the idea of a debugger. The compiler is a useful tool to help you get rid of the syntactic errors. These include such mistakes as typos, missing semicolons, badly placed parenthesis, or misspelled variable names. However the compiler cannot help you trace down problems once your program is running but producing the wrong output. These kinds of problems often manifest themselves as infinite loops, wrong calculations, or normally segmentation faults. The debugger is a tool that can help you trace down logical errors.

What do debuggers let you do?

- You can step through your code line by line.
- You can step into functions or over them.
- You can print the value of variables and parameters.
- You can monitor variables.
- You can insert "breakpoints" in your code to suspend execution at certain points

2. Instruction to enable tunneling of your session (if you get the message: “cannot open display”)

1. Start SSH and connect to one of the following addresses:
   - ece2.ece.arizona.edu
   - ece3.ece.arizona.edu
2. After successful login add the session in the profile manager (Profiles-Add). Assign session name ex. “ece3”
3. Set the tunneling option (Profiles-Edit-ece3) Click on Tunneling and make sure the option “Tunnel X11 connection” is enable
4. Click Apply and then confirm saving the above changes.
5. Exit SSH client
6. Run Xming
7. Run SSH . Click Profiles-ece3. Type your password
8. To check that your session is tunneled type “xclock”. As far as you can see the clock in separate window you are done
3. Background and getting started

I tried to write a program, code1.c, and it compiled fine. But it just did not do what I wanted it to do (i.e. it has logical bugs). You should copy the files code1.c into your own account so you can test it and fix it for me. To copy the program, first download it locally to the Windows machine (for example to the desktop). Next login into your account through SSH shell client and open the File transfer protocol window. Copy and paste the required files in the appropriate subfolder in your account.

4. What this program is expected to do (when debugged)?

It takes numbers from the standard input one by one and calculates their product. You may input as many numbers as you want. However, a zero input "0" is reserved for halting the process of input and printing the then final result for the product. Therefore, a typical run can be as follows:

```
> Enter number: 1
> Enter number: 4
> Enter number: 3
> Enter number: 10
> Enter number: 0
> The product is 120
```

The program has two procedures – `Main()` and `GetFloat()`. `Main()` provides the initial startup, where two variables are initialized: `product` to keep the current product, and `numout` which holds the current assigned number. The second function `GetFloat()` is simply supposed to read the input for a new number and return this number using the `numout` variable. The variable `done` in the while loop is a flag which should indicate when the process of reading in new inputs/numbers should be halted (in the current program, `done=1` indicates that the user has input zero).

5. LAB ASSIGNMENT and What you should turn in for this Lab.

The provided code1.c program has bugs and you are supposed to correct them. After you have found the bugs, corrected them, and tested the revised program for correctness, then you will use the following to turn in the source code for the correctly working program.

```
turnin engr170 lab2 code1.c
```

6. How to debug

Look over the program and let us see what it currently does. Compile the program, and run it:

```
  > gcc code1.c
  > a.out
```

Using debugger:

The debugger we will cover is `ddd`. In order to get the debugger going, we must recompile the code with a new option:

```
  > gcc -g -o code1 code1.c
```

(The `-g` places special symbol information into code1 so that `ddd` can be more informative. This extra information will make the executable larger and slower. Note: when all debugging is complete, the program should be re-compiled without the `-g` option to create a final, optimized program.).
Before running `ddd`, as a requirement you need to start X-window client, which for the machines in Rm250 this is done using the `Xming` program. You run Xming in background using the steps mentioned on the first page of this handout. Now type the following:

```bash
ddd code1
```

Notice that we start `ddd` using the executable, not the source code. After executing `ddd` you will see the `ddd` main window (Figure 1).

![Figure 1 An example window showing the ddd debugger running](image)

Type the following in the Debugger Console sub window:

```bash
(gdb) list 1,20
```

and you will see the top of the program. (Hint: The debugger will only show you the next line it is about to execute. However if you want to see more, use `list <current line number>` to see 5 lines before the current line to 5 lines after the current line.

Let us make the program stop in the main function so that we can then run it and step through it.

```bash
(gdb) break main
```

Breakpoint 2 at 0x10ef: file debug1.c, line 44.

```bash
(gdb) run
```
Starting program: a.out

Now you will see the very first line in the main program. This line has not executed, but it will if we type in n, or s.

Breakpoint 2, main() at debug1.c:44

```
44 int done = 0;  /* done tells us when a zero has been entered */
```

Keep typing n, until you get to the line `num = GetFloat()` of the program. Here is what should happen:

```
46 float product = 0,  /* product keeps the product of all the numbers */
```

So we see that variable `maincount` now has the value 1. Let’s for now assume that `GetFloat()` works correctly, so we do not want to step into it. Type “n”. Now you need to type in a number. Enter any number. Before we move on let us make sure that variable `num` has the correct value. try the following:

```
52 num = GetFloat( maincount) ; /* read in the number */
```

What happened? Obviously `GetFloat()` did not work correctly, thus assumption about its correctness needs to be tested. Restart the program, but this time we will step into `GetFloat()`. Type run to restart the program and keep typing n to get to the same `GetFloat()` line again. Now type:

```
step
```

We are now at the first line in the invocation of the GetFloat() function.

```
print functioncnt
```

We see that the argument `functioncnt` also holds the value 1.
Type n twice and then enter in a number. Now print num. Why isn’t num the correct value? **This is the first bug you should recognize and figure out how to correct.** Pay attention to the `GetFloat()` function about how it reads the input and how it assigns values to the variable(s). Exit `gdb` and fix it. Then Recompile the program and start the debugger again.

Now that you have successfully fixed `GetFloat()`, start debugging some more. (Don’t forget to set a break point in main again.) Use n to get past the `GetFloat()` line and make sure num now has the correct value.

Now use n to execute the line `product *= num;`

> 54  product *= num;  /* multiply the number by the running total */
> (gdb) n

> (gdb) print product
> $2 = 0

Print product. Why didn’t the variable product change? **This is your second bug that you should recognize and figure out how to fix.** What is the value of the product variable? Why does it take this value? Fix the bug, recompile the program and let’s try it again. Again use n to get past the `product *= num` line. Now use n to get all the way back to the `GetFloat()` line. But this time enter in 0 for the number so that we can stop the program.

Use n to get to the “if” statement. Before we execute the “if” statement make sure that the value of num is indeed 0.

> if (num = 0)
> (gdb) print num
> $1 = 0

Since num is 0, type n to execute the if statement. Why did we reenter the iterative loop? Why didn’t we go into the “if” statement? **Hence you discover the third bug in the code.** Check the “if” statement and especially the logical condition. Fix this bug, recompile the program and again get back into the debugger. Test out the “if” statement by entering a zero. It should now work correctly. So let’s test this program without the debugger and make sure it really does work.

Exit the debugger (quit). Type a.out

> a.out
> Enter number1: 42
> Enter number2: 3
> Enter number3: 0
> The product is 12.0000

Well done. The code was debugged.
APPENDIX

Program with bugs

/*Program with bugs
* ENGR170
* Revised Alex Mitev
* engr170
* Description:
* This program reads in floating point numbers until it encounters a zero.
* It keeps a product of all of the floating point numbers (not including the zero)
* and after the zero is entered, it prints out the product.*/
#include <stdio.h> /* for scanf, printf, etc... */

/*
* GetFloat (int functioncnt)
* This function reads in the next floating point number from standard
* input. It then returns that floating point number.
 *
* Ex: This function can be used like this:
* 
*  float x; *
*  x = GetFloat(maincount);*
*  < x will now have the value typed in from standard input
>
*/
float GetFloat(int functioncnt)
{
  float numin;
  /* read in the float */
  printf("Enter number%d : ",functioncnt);
  scanf("%d",&numin);
  /* return its value */
  return (numin) ;
}

int main(void)
{
  /* done tells us when a zero has to be entered */
  int done = 0;
  /* product keeps the product of all the the numbers */
  float product = 0;
  float numout ; /* num holds the current number */
  int maincount = 0;

  while ( !done)
  {

maincount ++;
numout = GetFloat(maincount); /* read in the number */
/* multiply the number by the running total */
if (numout==0) /* check if zero was typed */
{
    done = 1; /* now we are done */
    printf("The product is %f\n", product);
}
product *= numout;
}