CS102: Debugging

Errors in your programs are commonly referred to as **bugs**. The process of fixing bugs is known as **debugging**. Debugging can be time-consuming and annoying!

Let's say that you've written your code and it doesn't compile. The compiler is usually pretty good at telling you which lines contain errors, and they are usually fairly obvious, so you can go back and fix them. If you try to call a function that doesn't exist, the compiler doesn't give you line numbers, but it does tell you the name of the function that you are trying to call, so it is still usually obvious what to do about it. With any common text editor or integrated development environment (IDE), you can see (or figure out) which line number you are currently on, and there will be some way to search for text. So it is generally not too difficult to find the code causing a problem with compilation, and more often than not you will realize quickly what to do about it.

The bigger problem comes next. Your code compiles, you run it, and it doesn't do what you expect it (or at least want it) to do. There are two general possibilities. There might be one or more lines in you program that do not do what you think they do; or it is possible that each individual line in your program does what you think, but strung together the net effect isn't what you intended. So what do you do?

We are going to look at a specific example. Let's say you are trying to write a program that allows the user to enter a number and then checks to see whether or not the number is prime. You decide to write a function that will take an integer as a parameter and return the number of factors of that integer. If there are exactly two factors, the number is prime. You write the following code. THIS CODE HAS MULTIPLE ERRORS! Still, it compiles just fine.

```
#include <stdio.h>

int num_factors(int);

int main(void)
{
  int x, y;

  printf("Enter a number: ");
  scanf("%d", x);

  y = num_factors(x);
  if (y = 2)
    printf("That number is prime.\n");
  else
    printf("That number is NOT prime.\n");

  return 0;
}
```

```
int num_factors(int n)
{
  int x;
  int count = 0;

  for (x = 1; x <= n; x++)
  {
    if (x % n == 0)
      count++;
  }

  return count;
}
```

You compile and run the program. You try it out a few times. The program will either crash, seem to run forever, or tell you that everything is prime. There is actually a bit of chance here! In any of these cases, you need to figure out what is wrong.

There are two methods we are going to discuss to debug code. This first method is simple, but often it is enough. The method is sometimes called tracing, and it involves the use of tracers, a.k.a. trace statements. A **tracer** is a statement that you temporarily add to your program to display information indicating the current location in your program or the values of variables.

The first question you might be asking yourself is whether or not we are correctly reading in the number form the user. So let's say we add the following "printf" statement after the "scanf":

```
#include <stdio.h>

int num_factors(int);

int main(void)
{
  int x, y;

  printf("Enter a number: ");
  scanf("%d", x);

  printf("In main, x = %d.\n", x);
  ...
}
```

Don't forget to save the program and re-compile after adding the new statement! You will probably sometimes forget, and this can actually cause a lot of confusion.

One of two things will happen now. You will either crash before you get to the added tracer, or you see an arbitrary value displayed for "x" that is different than the one you type in. Either way, hopefully that is enough for you to recognize your common mistake. You fix the bug:

```
  …
  printf("Enter a number: ");
  scanf("%d", &x);
  …
```

You save, re-compile, and run the program. You type in 10. You see:

```
Enter a number: 10
In main, x = 10.
That number is prime.
```

After running the program a few more times, you see that it tells you that everything is prime. A good question that you might be asking yourself at this point is whether or not we are calling the function correctly. To test this, let's say we add the following "printf" statement at the beginning of "num_factors":

```
int num_factors(int n)
{
  int x;
  int count;

  printf("In num_factors, n = %d.\n", n);
  …
}
```

Let's say you are now confident you are reading the number from the user correctly, so at the same time you add this tracer, you remove the previous tracer. Once again, you must remember to save your changes and recompile the program.

Running the current version, you type in 10 and see:

```
Enter a number: 10
In num_factors, n = 10.
That number is prime.
```

The good news is that you are calling the function and seem to be passing the correct value, but things are still not working, because 10 is not prime. Time to ask another question; what is the function actually returning? You therefore add another "printf" statement in you main function:

```
  …
  y = num_factors(x);
  printf("num_factors returned %d.\n", y);
  …
```

You might want to get rid of the second tracer, since you may be confident that you are calling "num_factors" correctly, but let's say we leave it in.

You re-compile, run the program, type in 10 and see something like:

```
In num_factors, n = 10.
num_factors returned 1.
That number is prime.
```

Now, you are confused about two things. One, why did "num_factors" return the wrong number, and two, given that this number was returned, why are we saying that the original number is prime?

The answer to the second question might be something you can figure out right away, since you are printing out the value of "y" right before the "if" statement. You forgot to use a double equal sign! So in effect, you were assigning "y" to equal 2, and remember, assignment statements are also given the value that they are assigning. Since 2 is non-zero, it is considered true, and every number is considered prime. You fix this bug:

```
  …
  if (y == 2)
    printf("That number is prime.\n");
  …
```

You re-compile, run the program, type in 10 and see:

```
In num_factors, n = 10.
num_factors returned 1.
That number is NOT prime.
```

So at least the "if" statement seems to be working but we still have the major problem that "num_factors" is not returning the right value! 10 is not prime, but there is more than 1 factor. You try some other numbers, including some that are prime, and see that "num_factors" is always returning 1. There's still at least one more bug!

There are actually several tracers you could use to help you figure out what's going on, but let's say that you can't think of any. Maybe you don't want to put a tracer inside the loop because it would print out multiple times and be confusing, or maybe you just can't think of anything. What you really want to do is walk through the code one line at a time as it is running with the ability to check out values of variables as you go along. A program that allows you to do this is called a **debugger**!

One debugger that is available on most Linux/Unix/Cygwin systems is "gdb", a very common and popular debugger for C programs. In order to use it effectively, you must compile your program with a special debugging option turned on. If you are compiling from the command line using gcc, that option is turned on by including the "-g" specification when you run the compiler. For example, let's say the program above is saved in a file called "prime.c", and you want the executable to be called "prime.exe". You would compile it as follows:

```
gcc -g -o prime.exe prime.c
```

The "-g" specification is actually causing the compiler to add extra information to your executable that will be recognized by the debugger. This information links the machine code of the executable to specific lines of your source code so that you can execute machine code equivalent to one line of source code at a time, print out values of variables, and more. As a result of this extra information, your executable is larger and slower, but you are now able to step through it effectively with a debugger!

Once the program is compiled with debug information, you run the debugger by typing "gdb" followed by the name of the executable. For example:

```
gdb prime
```

This runs the debugger, and the debugger displays its own command prompt:

```
(gdb) _
```

Here, the '_' represents the cursor (it may appear differently). There are now different commands that the debugger will recognize. One of these commands is "run", which will run the program under the debugger. Before running the program, however, you usually want to add one or more breakpoints. A **breakpoint** is a specified location in your program that, when reached, will cause the debugger to halt execution of the program. You can add a breakpoint by typing:

```
break <location>
```

at the "gdb" prompt. The location can be either a line number or the name of a function. If it is a line number, the debugger will halt execution of the program when that line is reached. If it is a function, the debugger will halt execution of the program at the beginning of that function.

So let's say we want to start stepping through at the beginning by adding a breakpoint at "main". We type:

```
break main
```

You will get a fairly cryptic message indicating that the breakpoint has been set.

Now type "run" at the "gdb" command prompt. This sets the program in motion, and will stop as soon as the first breakpoint is hit (which happens right away, in this case). You see the current line displayed in the debugger. This is the "printf" line, the first line executed in main.

To step over the current line of source code to the next line, you type "next" or "n" at the "gdb" command prompt. If you type this now, you step over the "printf" statement to the "scanf" statement. Now type "next" or "n" again. Under the command prompt, you see the message "Enter a number: " printed, and the computer waits for you to type. The program is actually running, just stopping after each line of source code! When the program prints to the screen or awaits input, you still see the output or get to type the input. After you type a number, the program pauses again at the next line. For example, type "10" and hit enter. You are now at the line that will call the function "num_factors".

This time, you don't want to use the "next" command. That would step over the current line and go right to the "if" statement. You want to step into the function "num_factors". (If you wanted to, you could have just added a breakpoint at the start of this function instead of "main".) You can step into a function by using the "step" command. You type either "step" or "s" at the "gdb" command prompt. You will then be at the line that initializes count.

You can also type "list" to see the code around the current line. If you type "list" multiple times in a row, you will see more lines of code. (If you want to list the code around some specific line number, you can type "list <line_number>".)

Now type "next" or "n" twice. You step over the "printf", see the message from the tracer printed, and are at the beginning of the "for" loop. Now use the "next" command again. You are inside the "for" loop at the "if" statement.

You think that "x" should now be 1, "n" should now be 10, and "count" should now be 0. How do you make sure? You use the "print" command! For example, at the "gdb" command prompt, type "print x" and you will see something like "$1 = 1". The value to the right of the equal sign is the value of the variable (or expression) you are printing. Do not worry about the expression to the left of the equal sign. Using "print", you verify that all variables are what you expect. Now, since "x" is a factor of "n", you expect the "if" expression to be true, and that "count" will be incremented.

You use the "next" command. Something unexpected happens! You are back at the start of the "for" loop. The "if" condition was not satisfied. Use "next" several additional times. You will find that the program keeps trying out the "if" statement and skipping the following line. Why isn't the expression true? Perhaps you now realize you listed the operators of the '%' operator in the wrong order. You want to take the remainder when "n" is divided by "x", not the other way around! (If you didn't realize this, you could have evaluated the "if" statement's condition, or part of it, using the "print" command; for example, "print (x % n == 0)" will print 1 if the expression is true and 0 if the expression is false, and "print x % n" will print the remainder when "x" is divided by "n".)

Now that you've discovered a bug, you want to quit the debugger and fix it. You do this by typing "quit" at the "gdb" command prompt. The debugger asks if you really want to stop the program from running and quit. Say yes. (If you wanted to let the program finish first, you could have typed "continue" at the "gdb" command prompt before typing "quit".)

After quitting "gdb", start up your text editor and fix the bug:

```
    if (n % x == 0)
      count++;
```

Now re-compile (without the -g option) the program and run it. Try it several times with different numbers, and you should find that it works! All that's left is to take out your tracers; you don't want them in your final version.

Of course, there's much more that you can do with "gdb" than we've covered with this little example. Here are a few other commands you can use at the "gdb" command prompt:

"set" - Used to set the value of a variable. For example "set variable x=5" will set the value of the variable "x" (within your C program) to 5.

"clear" - Clears a breakpoint. The notation is "clear <location>", where location is a line number or the name of a function. This is helpful when a breakpoint becomes annoying because it is hit too often.

"backtrace" – Shows the chain of functions from main to the current function, including additional information. (The details of this would make more sense after you learn about the call stack, a.k.a. the stack, which stores information about every currently executing function.)

"help" - Perhaps the most important "gdb" command for now is the "help" command. If you just type "help" by itself, you get a list of help categories. The categories include "running", "stack", "data", "breakpoints", etc. Each category name is followed by a very brief description of the category.

To list the commands within a category, type "help <category>". For example, "help data". This gives you a list of actual commands that fall into the "data" category.

To get help with a specific command, type "help <command>". For example, "help break" will tell you how to use the "break" command. One thing you can learn from this example is that it is possible to set **conditional breakpoints**, which can be very useful!

The rest you need to learn on your own with experimentation! It's really the only way to do it. It is absolutely worth it though. Within a couple of hours, you should be semi-fluent, and you might save much more time with that on every program you every write!

If you use an IDE (e.g., Eclipse, Quincy, etc.), it will offer much of the same debugging functionality as "gdb", and it will probably be more user friendly. It is likely that breakpoints can be set by positioning the cursor on a line of code and clicking an appropriate button or using a keyboard shortcut. Other buttons will allow you to step over lines of code or step into function calls. Some IDEs might offer debugging functionality that is actually built on top of "gdb", while others might use other implementations. You will need to learn this on your own for the IDE of your choice. As with "gdb", it will be well worth the effort.

One more useful piece of advice which can not be overstressed is: MAKE FREQUENT BACKUPS OF YOUR CODE! A backup of your code is a copy of the source file(s) in case something happens to the original. You can make a backup of your code using the "cp" command at the Linux/Unix/Cygwin prompt. For example, "cp prime.c prime.bak" will create a file called "prime.bak" which is identical to the file "prime.c". Of course, you can also make a copy of the file(s) however your operating system lets you. You should make backups of your code often, whenever you have made a lot of changes since your last backup or you are pleased with the current state of your code.

There are at least two important reasons for frequently backing up your code. One is that if you program long enough, there will come a time when you accidentally erase your source file! If you've backed it up 10 minutes ago or even 30 minutes ago, it's not that big a deal, but if you've been working on your program for a week and you've never backed it up, you might have to start over from scratch. The second reason is that sometimes you will change your code a lot and then realize you were much closer to having your program work an hour earlier than you are now, but you don't remember exactly what you changed. This type of thing may happen more often than you would expect! If you backed up the program when it was close to working, you can go back to that version, and this may be very helpful. Some students may want to learn how to use a version control system which remembers older versions of source code files and tracks changes, also allowing multiple programmers to edit the same code.