CSE 3100 Systems Programming | Spring 2023

Lab #1

Welcome to Lab 1! We will learn a debug tool GDB in this lab and practice how to use it. Before we get to that, you will be setting up your *personal virtual machine* or VM hosted by the university. This is where you will be doing your coding, debugging, and testing for the remainder of the semester.

Setting Up Your VM

Before you can access your virtual machine, there is a short setup to complete.

1. **Ensure you are connected to university Wi-Fi and that you have a mobile device for 2FA**. If you are not connected to university Wi-Fi, you will need to use the UConn AnyConnect VPN available here. (This applies to when you are using the virtual machine for the duration of the semester to work on assignments. If you are not connected to university Wi-Fi, you need to be connected to the VPN.)
2. Open a terminal on your local machine and SSH (Secure Shell Protocol; a way of connecting remotely to the terminal of another machine in a network.) into your VM at the address **[NETID]-vm.cse.uconn.edu/**. If you are on Linux, you can run

   ```
   ssh [NETID]@[NETID]-vm.cse.uconn.edu
   ```

   to connect. If you are having issues with SSH or are unsure how to do so on Mac/Windows, ask your TA.
3. Enter your NetId password when prompted. You will not be able to see what you are typing as it is hidden to not reveal your password. Press enter when finished.
4. When prompted for 2FA, enter either 1 or 2 and press enter to select your preferred 2FA method.
5. You are (hopefully) now connected to your VM by SSH! Now, every command you type is run by your VM and relayed back to you in the terminal you have open! To quit the connection, you can run the command `exit`.
6. Run the command `/opt/scripts/codefix.sh`, which sets up your VSCode environment for browser usage.

You should now have access to your VM in browser at the address `[NETID]-vm.cse.uconn.edu`! You can just enter this into your browser of choice and begin working. Two *very* important things to remember for the VM:

- **BACK UP YOUR WORK! VMs are NOT backed up. Losing significant work to data loss on the VM is not an excuse. If you are done working on a file in your VM, make sure you download it to your local machine just in case.**
- **VMs undergo maintenance from 5:30AM to 7:30AM. There is a possibility that machines are down during this time, so do not plan on having access during this block of time.**

GDB is GNU Project Debugger. It will be the main tool you will use to find bugs in your code, especially those difficult to find. The official GDB website is here. The full GDB guide is here. There are also many GDB tutorials and cheat sheets on the Internet, for example, here and here.

Part 1. parity. The *even parity* of a binary number is defined as 1 if the number of 1's in the number is odd, and 0 if the number of 1's is even. For example, 9 written as a binary number is:

$$9 = 1001_2 \tag{1}$$

So, the even parity of this number is 0, since there are 2 1's in its binary representation. The following code computes the even parity of integer `v`:

```
unsigned int v = 19;
char parity = 0;
while (v) {
    parity = !parity;
    v = v & (v - 1);
}
```

Kernighan's Algorithm

An algorithm for computing the number of 1 bits in an integer is attributed to Brian Kernighan, coauthor of the C Programming Language. [a] Here I'll explain why it works, because it's not trivial. First, consider a non-zero binary number; call it v. We'll refer to bit j as the bit j positions to the left of the least significant bit. For example, in 101_2, bits 0 and 2 are set to 1, and bit 1 is set to 0.

Now, suppose bit k is the rightmost bit in v that is to 1. For example, in 10100_2, this would be bit 2. If we subtract 1 from v, then bit k flips to a zero, and bits 0 through $k-1$ flip to 1. You can see this with 10100_2:

```
  10100
-     1
-------
  10011
```

Then, by bitwise anding v with $v-1$, we flip bits 0 through $k-1$ back to 0:

```
  10100
& 10011
-------
  10000
```

Essentially, each time we do `v & (v - 1)`, we flip the rightmost 1-bit to a 0-bit.

[a] It was discovered independently by several people, see: https://graphics.stanford.edu/~seander/bithacks.html but is frequently attributed to B.K.

With GDB, you can see the value of variables at runtime. It is more convenient if you ask the compiler to keep useful information in the executable by specifying `-g` option. For example,

```
gcc -o parity -g parity.c
```

Then, you load the program you would like to debug into GDB.

```
gdb ./parity
```

Now, put a breakpoint on the `v = v & (v - 1)` line. In the following example, the `list` command lists the source code around `main()`, the `break` command sets a breakpoint, and the `run` command starts the program.

```
(gdb) list main
1       #include <stdio.h>
2
3       int main(void)
4       {
5         unsigned int v = 19;
6         char parity = 0;
7         while(v){
8           parity = !parity;
9           v = v & (v - 1);
10        }
(gdb) break 8
Breakpoint 1 at 0x4004e7: file parity.c, line 8.
(gdb) run
```

The program should have stopped at the specified line. You can examine the value of `v` in binary format before the execution of the statement as follows:

```
(gdb) p/t v
$10 = 10011
(gdb) p/t v-1
$11 = 10010
```

After the execution of the statement (with an 'n' command), check the updated value in `v`.

```
(gdb) p/t v
$12 = 10010
```

Part 2. Running average. Write a C program `average.c` that reads floating-point numbers (double) from the standard input and, after reading each number, prints the running total and average of the numbers that have been read so far. The program must terminate when there is an error or the end of file (EOF) is detected at the standard input (e.g., when the user presses Ctrl-D).

The block below shows a sample execution of the program in which the user provided input consists, in this order, of numbers 1, 2, 1e10 (meaning 1×10^{10}), and 0:

```
$  ./average
1
Total=1.000000 Average=1.000000
2
Total=3.000000 Average=1.500000
1e10
Total=10000000003.000000 Average=3333333334.333333
0
Total=10000000003.000000 Average=2500000000.750000
```

Notes:

The loop needed to read the input is as follows.

```
while (scanf("%lf", &x) == 1) {  // pay attention to %lf
    ...
};
```

Here `x` is a `double` variable used to store the number read in each iteration. Note the character in the middle of `%lf` is the letter el, not the digit 1. The `while` loop continues as long as `scanf()` returns 1, indicating it has read one double from the standard input successfully. Otherwise, the loop is terminated.

Printing the running total and average formatted as in the example above can be done using

```
printf("Total=%f Average=%f\n", total, average); // pay attention to %f
```

where `total` and `average` are variables of type `double`.

Also, remember to initialize variables before using them.