ECE220: Computer Systems & Programming **Spring 2019 Honors Section**
Machine Problem 5 **due: Monday 8 April, 11:59:59 p.m.**

Testing and Debugging C Programs

Your task this week is to learn how to debug C programs. Specifically, you will learn how to use a debugger (the Gnu debugger, GDB) to examine a C program and design test cases that expose bugs inside programs. You will then debug the programs we provide and write a report on the debugging process.

The objective for this week is for you to gain some experience with debugging programs in C, particularly with GDB. This skill is critical for testing software.

Background

Testing is an important part of software development. Software needs to be tested to make sure that it works as intended. Bugs may hide in not-well-tested code for years and cause disastrous events. Some of the most costly bugs and their causes are listed below:

Ariane 5 Flight 501: https://en.wikipedia.org/wiki/Cluster_(spacecraft)#Launch_failure

The Ariane 5 rocket, Flight 501, launched on Tuesday, 4 June 1996, ended in failure due to an error in the software design caused by assertions having been turned off, which in turn caused inadequate protection from integer overflow. In essence, the software had tried to cram a 64-bit number into a 16-bit space. The resulting overflow conditions crashed both the primary and backup computers (which were both running the exact same software). The Ariane 5 cost nearly $8 billion to develop, and was carrying a $500 million satellite payload when it exploded.

Morris worm: https://en.wikipedia.org/wiki/Morris_worm

The first internet worm (the so-called Morris Worm), developed by a Cornell University student for what he said was supposed to be a harmless experiment, infected between 2,000 and 6,000 computers in less than a day by taking advantage of a buffer overflow.

The specific code is a function in the standard input/output library routine called gets() designed to get a line of text over the network. Unfortunately, gets() has no provision to limit its input, and an overly large input allows the worm to take over any machine to which it can connect.

The graduate student, Robert Tappan Morris, was convicted of a criminal hacking offense and fined $10,000. Costs for cleaning up the mess may have gone as high as $100 Million.

Ping of death: https://en.wikipedia.org/wiki/Ping_of_death

A lack of sanity checks and error handling in the IP fragmentation reassembly code makes it possible to crash a wide variety of operating systems by sending a malformed "ping" packet from anywhere on the internet. Most obviously affected are computers running Windows, which lock up and display the so-called "blue screen of death" when they receive these packets.

GDB Tutorial

To use GDB effectively, you need to enable the built-in debugging support for the compiled executable. To do so, include the `-g` option when compiling, as with the factorial code in this week's SVN directory:

```
gcc -g -Wall factorial.c -o factorial
```

You can start GDB by typing:

```
gdb
```

You can load the executable to debug by using the file command (the first part, "`(gdb)`", is the GDB prompt:

```
(gdb) file factorial
```

Alternatively, you can pass the executable name as command-line argument when starting:

```
gdb factorial
```

GDB has an interactive shell, much like the one you use when you log into the Linux machines. It can recall history with the arrow keys, auto-complete words (most of the time) with the TAB key, and has other nice features.

If you're ever confused about a command or just want more information, use the "help" command, with or without an argument:

```
(gdb) help [command]
```

To run the program, type:

```
(gdb) run
```

Or, if the program needs command line arguments:

```
(gdb) run [arg1] [arg2] …
```

This command runs the program. If the program has no serious problems, the program should run fine under GDB, too. If the program crashes, GDB can usually tell you some useful information, such as the line number at which the program crashed and parameters to the function that caused the error.

However you don't need to use GDB to run the program when the program is working. When the program isn't working, you probably want to pause execution and check the current status of program. To pause the program, you can use breakpoints.

Breakpoints can be used to stop the program run in the middle, at a designated point. The simplest way is the command "break". This sets a breakpoint at a specified file-line pair:

```
(gdb) break factorial.c:18
```

This sets a breakpoint at line 18 of file `factorial.c`. You can set multiple breakpoints. The execution stops whenever it reaches any breakpoint.

You can look at your source code using the "list" command in GDB. You can list functions by name, for example:

```
(gdb) list factorial
```

If you just type "list" after starting GDB, the debugger will show you the start of main. "List" without arguments moves further along in the file. GDB remembers the last file that you examined, so if you see a line in the code you're looking at, you can omit the file name when setting a breakpoint.

Back to breakpoints. You can also tell GDB to stop when execution reaches a particular function:

```
(gdb) break factorial
```

The execution stops whenever the `factorial` function is called.

You can also stop execution only when a particular requirement (or a set of requirements) is satisfied. Using conditional breakpoints allows us to accomplish this goal. Conditional breakpoints are identical to regular breakpoints, except that you get to specify some criterion that must be met for the breakpoint to trigger:

```
(gdb) break factorial if n == 2
```

You can add a condition to an existing breakpoint using cond. Each breakpoint has a number, which GDB prints when you create the breakpoint. Let's say that the breakpoint in function factorial was #2. In that case, the above command is equivalent to:

```
(gdb) cond 2 n == 2
```

You can also change the condition on a breakpoint, or disable or enable each breakpoint after hitting it. You can also tell GDB to skip the first N times a breakpoint is hit. Use "help" to learn more—these variants are not used as frequently as basic breakpoints.

To resume execution, you can use the "continue" command:

```
(gdb) continue
```

The execution will continue until the next breakpoint, unless a fatal error occurs before reaching that point.

You can also single-step (execute just the next line of code) with the "step" command:

```
(gdb) step
```

Similar to "step," the "next" command executes the next line, but does not enter the source of any functions. Instead, "next" silently finishes execution of any called instructions, treating the whole line one instruction.

```
(gdb) next
```

You may need to repeat the "continue," "step." or "next" commands multiple times. In GDB, you repeat the last command by just pressing ENTER.

When executing through loops, you may want to continue execution until the loop terminates. To do so, use the "until" command:

```
(gdb) until
```

Similarly, you may want to continue execution until the current function returns:

```
(gdb) finish
```

So far, you've learned how to interrupt program flow at fixed, specified points, and how to continue stepping line-by-line. When the execution pauses, you may want to see values of variables or even modify variables' values. You can do so with the "print" and "set" commands:

```
(gdb) print n
(gdb) set n = 2
```

The "set" command sometimes requires you to specify that you want to change a variable: "`set var n = 2`". You can call functions in your program as part of a "print" command. GDB uses the stack provided by the hardware and executes the called function in your program. Be careful: those functions can also crash if they are given bad values. If you just want to call a function rather than printing or calculating something based on its return value, use the "call" command instead. Sometimes, it's useful to add functions to your program solely for use from within GDB.

If you want to print an expression or a variable's value every time GDB stops execution, use the "display" command instead. This command fails when variables in the expression go out of scope, and may change meaning if variables with the same names come into scope (remember that the program has a current point of execution that defines which variables are in scope—that is, usable by name).

You can use the "backtrace" command to view a trace of the function calls that brought execution to the current point (the current function, the function that called it, and so forth, all the way back to `main`). The "backtrace" command (also known as "where") is particularly useful when debugging with seg faults (illegal memory accesses, as sometimes arise from array bounds errors).

```
(gdb) backtrace
```

You can also use the "up" and "down" commands to change the current scope to the caller/callee function. You can specify how many levels you want to change with these two commands:

```
(gdb) up 2
(gdb) down 1
```

Other useful commands include "info," "delete," and "clear."

You can use the "info" command to list the current breakpoints ("`info breakpoints`"), local variables ("`info locals`"), file scope/global variables ("`info variables`") register values ("`info registers`"), and so forth.

You can use the "delete" command to delete a breakpoint (or disable it temporarily with the "disable" command, then re-enable it later with the "enable" command).

```
(gdb) delete breakpoint 2
```

You can use the "clear" command to delete all breakpoints set inside a function.

```
(gdb) clear factorial
```

The Task

You need to identify bugs within several buggy programs by using GDB. You need to document the GDB commands that you use and the results that GDB produces in the file `report.txt`.

1. reverse

The first program that you are required to debug is the print reverse program. This buggy program is in the `printRev` folder in your SVN directory. You may look at all of the code for this particular program.

Your task is to document what the code does, how it works, and what arguments it takes. After doing so, you must run the code repeatedly and identify at least three inputs for which the program does not work. You must then fix the bug and document how you fixed it. You must document everything in `report.txt`.

To get started, you can make the program (type "`make`"), and then run it in GDB by typing

```
gdb prev
```

Then type

```
r <your favorite word>
```

This command will run the program with your favorite word as a command-line argument. Use the skills you learned in reading and trying the GDB tutorial (the first part of this document) to debug the program.

You may find it useful to write the output of the program into a file using the redirection operator, ">":

```
./prev "argument" > out.txt
```

View `out.txt` to see that the output of the program has been written to that file.

2. primeNumber

The next program to debug is the `primeNumber` program. The correct version of the program can be found on the course website:

`http://lumetta.web.engr.illinois.edu/220-S18/Ccode/primes.c`

The buggy program is contained in the `primeNumber` folder in your SVN directory. The `is_prime` function has been rewritten in a more efficient way than in the original code. However, the implementation of `is_prime` function contains a bug. You may not see the source code for the new `is_prime` function.

Your task is to reason about how the `is_prime` function is implemented and the possible cause of the bug. To help you get started, you can use the following command to generate the executable:

```
make
```

Then start GDB by typing:

```
gdb primeNumber
```

Your task is to use GDB commands to figure out what the possible cause of the bug is inside the `is_prime` function. Append your findings to the `report.txt` file that you created in part 1.

3. sort

The last program that you are required to debug is the `sort` program. This buggy program is in the `sort` folder in your SVN directory. You are given some of the code to look at, but the buggy function is hidden. Your task is to first test this program, then describe the bug in your report.

The code implements a Heap Sort on an array of 32 bit integers. A Heap Sort is an in-place sorting algorithm that is generally efficient. To learn about Heap Sort, you may want to read the description of the algorithm at

```
https://en.wikipedia.org/wiki/Heapsort
```

To help you get started, use the following command to generate the executable:

```
make
```

You may then invoke GDB by:

```
gdb sort
```

and run the program by typing "`run out.txt`" within GDB.

Then you should use GDB to analyze the program using runtime information. Because the buggy function is hidden, you should use GDB to monitor the program when it calls into the functions that are visible to you (those in `sort.c`).

In the report, document the GDB commands that you use and the erroneous intermediate values that you find. Also, explain the possible cause of the bug.

Pieces

There are three buggy programs for this MP: printRev, primeNumber, and sort.

`printRev`*:* All source code for `printRev` is provided to you. You may play with the code, however we will only grade the `report.txt`. Changes to the source code will not count toward your grade.

`prmain.c`*:* The main file for `printRev`.

`pr_buggy.c`*:* This file contains the `print_reverse` function.

`primeNumber`*:* The buggy function `is_prime` is hidden from you. You may play with the source code provided, but we will only grade `report.txt`. Changes to the source code will not count toward your grade.

`isPrime.h`*:* This file contains the function definition for the `is_prime` function.

`primeNumber.c`*:* This file contains the `main` function and the helper function `divides_evenly` called by the `is_prime` function.

`sort`*:* The `main` function and the buggy function `heapSort` are both hidden from you. You are provided with the source code for helper functions that are called by the `heapSort` function.

`sort.c`*:* This file contains the helper functions.

Specifics

For this MP, you are required to submit a file named `report.txt`. For each of the three buggy problems, you must first describe the error that the program produces. The description should at least include test cases, the correct output, and the erroneous output.

After analyzing the program using test cases, you should analyze the program using GDB. In `report.txt`, document the GDB commands you use to analyze the program, and also GDB output that contains important information about the program status. Please note that you should not include all output produced by GDB.

Finally, please describe briefly what the bug might be, and in the case of the `printRev` program, how to fix it.

Compiling and Executing the Buggy Programs

You may use

```
make
```

to compile the buggy programs.

You may use

```
gdb {executable}
```

to use GDB to debug the buggy program.

Grading Rubric

We will only look at `report.txt` to grade your MP.

Identifying the bug (30%)

For each buggy program, you need to report the test cases used to test the program, the desired output, and the actual output of the program. You should report test cases for both correct executions and erroneous executions.

Trace the bug (30%)

For each buggy program, you need to use GDB to trace erroneous executions of the buggy program. You need to report the GDB commands you used and the results GDB produced. You should also analyze the results GDB produced and the relationship between the results and the erroneous behavior.

Analysis of the bug (30%)

For each buggy program, you need to report what might be the root cause of the bug. For `printRev`, you should also explain how to fix the bug.

Style (10%)

Your report should be formatted such that it is easy to read and follow.