Homework 7
Debugging

Assigned: Friday, February 19

Due: Friday, February 26, 11:00AM (Hard Deadline)

Submission Instructions

Submit this assignment on Gradescope. You must submit every page of this PDF. We recommend using the free online tool PDFescape to edit and fill out this PDF. You may also print, handwrite, and scan this assignment.

There may multiple answers for each question. If you are unsure, state your assumptions and your reasoning for why you think your answer makes sense.

1 Gitting started

When we first introduced git, we focused on the basics and how to use it locally. Next lecture, when we revisit git, we'll talk about sharing a repository with multiple collaborators and all that that entails. As a small preview, we'll use a few of the features now. As we walk through this, we'll introduce a few new terms and concepts. *You are not expected to understand all of this yet*; however we wanted to show you a real-world example and give you some experience before the next git lecture.

First, we `clone` a remote repository:

```
> git clone https://github.com/c4cs/debugging-basics
```

Unlike `init`, which creates a new repository from scratch, the `clone` command tells git to go copy an existing repository from the supplied location. The location that you cloned from will automatically be set up as the `origin`. Much like in Makefiles, where there is nothing actually special about the "`all`" target, there is nothing magical about the `origin` in git. It is simply an arbitrary name used by convention.

First things first, let's check out what status has to say about this new repository.

```
> cd debugging-basics
> git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

We have a new line of information! Git is a *distributed* version control system. This means that if you make changes (add new commits) on your machine, they will not automatically show up on any other machine. Being "up-to-date" means that there are no changes on your machine that you have not shared with the other machine, nor are there any changes on the other machine that your machine knows about but has not integrated—this can be a bit subtle and confusing, we'll talk about it next lecture.

Now let's take a quick look at what has happened in this repository:

```
> git log
```

Because we cloned a repository, it already has some history. You can also try `git log -p` to see the changes in more detail.

Next we introduce a new feature, `branches`:

```
> git checkout gdb-debug-1
Branch gdb-debug-1 set up to track remote branch gdb-debug-1 from origin.
Switched to a new branch 'gdb-debug-1'
```

(don't worry about the tracking part for now)

Branches allow us to take some code with a shared history and explore in two different directions. Try running `git log` again. Notice that the first two commits are the same, but the third commit is different.

We can use the `git branch` command, with no other arguments, to simply list all of our local branches:

```
> git branch
  master
* gdb-debug-1
```

We can see from the asterisk that we currently have the `gdb-debug-1` branch `checked out`. By default, every repository begins with a branch named `master`. If you run `git branch` in one of your other repositories, you will see that it has one branch, named master, that you have been using all along.

You may find a visual view of what is going on here helpful:

```
> sudo apt-get install gitk
...
> gitk --all
```

(don't worry about the remote/origin labels for now)

Notice how there is a shared history of two commits, that then diverges to two branches, `master` and `gdb-debug-1`.

Each part of this homework will take place on a different branch.

2 Finding primes

The repository you have cloned is an attempt at a program to find prime numbers. It will search from 3 up to a maximum number input by the user.

The program intends to follow the structure:

```
o prompt user for upper bound
o for N=3..upper bound:
  - check if N is prime
    * if no prime n between 1..sqrt(N) divides N, then prime
  - save whether N is prime for future loops to use
  - if N is prime, print N
```

2.1 Debugging with `gdb`

You may find it helpful to consult the gdb lecture notes or this quick command reference.

First, make sure you are on the `gdb-debug-1` branch:

```
> git branch
  master
* gdb-debug-1
```

Next, build and run the supplied program:

```
> make
> ./prime
Find all prime numbers between 3 and ?
10
Segmentation fault (core dumped)
> gdb -q ./prime
Reading symbols from ./prime...done.
(gdb) run
Starting program: /media/sf_prime/prime
Find all prime numbers between 3 and ?
10

Program received signal SIGSEGV, Segmentation fault.
0x000000000040068f in check_prime (k=3) at check.c:15
15      if (is_prime[j] == 1)
```

Explain in what case(s) executing this line of code could cause a segmentation fault?

> This question is asking only about this specific line of code, not all of the things that could possibly cause segmentation faults.
>
> This is an array access, which is a fancy way of dereferencing a pointer. Since we are in C, `is_prime[j]` is the exact same as `*(is_prime + j)`.
>
> This means that this line of code could cause a segmentation fault if the value `is_prime + j` does not point to a valid memory address. This is most likely to happen if `is_prime` is not a valid pointer or `abs(j)` is an enormous value.
>
> *Worth Noting: While a value of `j=-1` is very likely incorrect, it is unlikely to cause a segmentation fault. It will simply point to a different, nearby variable in the program's memory space. The fact that bugs like this do not cause crashes (at least not immediately) can make them hard to track down.*

What gdb commands could you run next to prove your hypothesis right or wrong?

> Either `print is_prime` or `ptype is_prime` to learn the size of the array, then `print j` to learn whether `j` is valid.
>
> Asking gdb to `print is_prime[j]` will directly show that the memory pointed to is inaccessible.

Now checkout gdb-debug-2:

```
> git checkout gdb-debug-2
> make
> ./prime
Find all prime numbers between 3 and ?
10
3 is a prime
5 is a prime
7 is a prime
9 is a prime
```

Copy your debugging session and add notes explaining your thought process as you track down why this program thinks 9 is a prime.

Hint: It looks like things go well up until 9. A good place to start then may be to break in and observe how the code determines whether 9 is a prime number.

```
$ gdb -q ./prime
Reading symbols from ./prime...Reading symbols from [...] ...done.
done.

Things go well up until checking if 9 is prime, so jump there
(gdb) break check_prime if k==9
Breakpoint 1 at 0x100000e87: file check.c, line 13.
(gdb) run
Starting program: /private/tmp/debugging-basics/prime
Find all prime numbers between 3 and ?
10
3 is a prime
5 is a prime
7 is a prime

Breakpoint 1, check_prime (k=9) at check.c:13
13    for (j=2; j*j <= k; j++) { expect to run this loop for j=2 and j=3

There's not a whole lot going on in this loop. As we move through it, the
only variable that changes is j. (We can infer the values of things like
is_prime[j] based on the path the code takes, though you could certainly
print those values as well if you like)
(gdb) display j
1: j = 0
(gdb) step
14      if (is_prime[j] == 1) expect this to be true, as 2 is prime
1: j = 2
(gdb) <enter> repeats last command, in this case step
15        if (k % j == 0)  { 9 % 2 isn't 0
1: j = 2
(gdb) <enter>
19      j++; okay, onto the next iteration of the loop
1: j = 2
(gdb) <enter>
13    for (j=2; j*j <= k; j++) { starting off the loop for j=3
1: j = 3
(gdb) <enter>
23    is_prime[k] = 1; wait, we just dropped out of the loop, and j is 4
1: j = 4
(gdb)
Key thing to notice: The loop body never executed for j=3. Why? j was
incremented twice, once at the end of the body of the for loop, and once
by the declaration of the for loop. This double-increment is the bug.
```

This homework is based off of a simplified and updated version of this gdb tutorial: http://heather.cs.ucdavis.edu/~matloff/UnixAndC/CLanguage/Debug.html If you would like some more practice, or simply a slightly different explanation of the material, this may be a good source.

3 Valgrind

Valgrind is a different kind of debugging tool. It is essentially a type of virtual machine (like a simpler VirtualBox). Valgrind actually recompiles every assembly instruction to allow it to intercept and monitor all of the hardware calls made by the child process. While this is very powerful and gives valgrind a lot of insight, it is also very slow, which can be problematic for debugging large pieces of software.

Valgrind also can be challenging when libraries (such as the STL, or Boost) do funny things with memory that result in a large number of false warnings. We will look at valgrind in more depth during the second week on debugging tools. For today, let's just explore the basic functionality and check out the kind of things that valgrind can catch that gdb can't.

First, we'll need to install valgrind:

```
> sudo apt-get install valgrind
```

Now, checkout the valgrind branch:

```
> git checkout valgrind-debug
```

Notice (perhaps `gitk --all`) that the valgrind branch builds on the master branch, with all the compiler warnings turned on. It has also cherry-picked the fix "Stop searching for primes once sqrt(k) is reached", that is the same commit object that managed to end up on two different histories. This is one example of how git can be both very powerful, and very confusing.

Finally, this branch adds a commit that consolidates they two source code files into one and gets rid of all of the global variables in the process. Unfortunately, this refactoring may also have introduced a bug.

We run valgrind very similarly to gdb:

```
> make
> valgrind ./prime
==11959== Memcheck, a memory error detector
==11959== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==11959== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==11959== Command: ./prime
==11959==
Find all prime numbers between 3 and ?
10
3 is a prime
==11959== Conditional jump or move depends on uninitialised value(s)
==11959==    at 0x400683: check_prime (prime.c:32)
==11959==    by 0x400739: main (prime.c:52)
==11959==
5 is a prime
7 is a prime
==11959==
==11959== HEAP SUMMARY:
==11959==     in use at exit: 0 bytes in 0 blocks
==11959==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==11959==
==11959== All heap blocks were freed -- no leaks are possible
==11959==
==11959== For counts of detected and suppressed errors, rerun with: -v
==11959== Use --track-origins=yes to see where uninitialised values come from
==11959== ERROR SUMMARY: 3 errors from 1 contexts (suppressed: 0 from 0)
```

(The numbers in the left column are the process ID. They will be different for every person)

Notice that "3 is a prime" printed before this warning. Why was this warning not emitted when running `check_prime(3, ...)`?

> Valgrind detects errors at run time. For `check_prime(3, ...)`, the for loop check `j*j <= k` (`2*2 <= 3`) fails and the body of the loop is never entered. This means `is_prime[j]` (`is_prime[2]`) never runs, which means valgrind does not detect the error.
>
> The key takeaway here is that because of how valgrind works, it can only catch an error when it actually happens.

These tools really get powerful when you combine them. We can run valgrind in a way that lets gdb connect to it, and allows us to debug/inspect whenever valgrind detects a problem:

```
> valgrind --vgdb=yes --vgdb-error=0 ./prime
```

In another terminal, follow the directions that valgrind prints to connect gdb. In this case, valgrind has already 'run' the program for you and inserted a breakpoint at the very beginning of the program, we just need to 'continue' it. The program will run until valgrind encounters an issue, at which point valgrind will automatically break for you.

```
(gdb) continue

valgrind terminal:
==23589== Conditional jump or move depends on uninitialised value(s)
==23589==    at 0x400623: check_prime (prime.c:32)
==23589==    by 0x4006C4: main (prime.c:52)
==23589==
==23589== (action on error) vgdb me ...  # this is where valgrind waits for gdb

gdb terminal:
Program received signal SIGTRAP, Trace/breakpoint trap.
0x0000000000400623 in check_prime (k=5, is_prime=0xffefff910) at prime.c:32
32      if (is_prime[j] == 1)   # this is the problematic line of code
```

What is the value of `j` at this point?

```
(gdb) print j
j = 2
```

What is the value of `is_prime[j]` at the point?

```
(gdb) print is_prime[j]
= -16777744 ← This will be a garbage value and vary person to person
```

Use git to look at the most recent commit. What line of code was deleted that should not have been?

```
$ git log -p

[... trim some output ...]

  int main() {
+       int is_prime[MAX_PRIMES];
+       int upper_bound;
        int N;

        printf("Find all prime numbers between 3 and ?\n");
        scanf("%d", &upper_bound);

-       is_prime[2] = 1;
-
        for (N = 3; N <= upper_bound; N += 2) {
-               check_prime(N);
+               check_prime(N, is_prime);
                if (is_prime[N])
                        printf("%d is a prime\n",N);
  }
```