CS 214: Introduction to computing systems
Winter Term, 2026

Lab 4[1]

This lab will introduce you to the debugger `gdb`. Debuggers are an important alternative to so-called "printf debugging", where you debug by adding print statements to your code.

Debugging with `gdb`

Begin by logging into euclid and copying the `lab4` subdirectory of the course directory and going into it:

```
cp -r /home/courses/cs214/lab4 .
cd lab4
```

We begin specifically on the files `main.c` and `prime.c`, which will be used for a tutorial on gdb. Once the code in these files is debugged, the resulting program will read an integer from standard input and print the primes less than or equal to this value. (Recall that primes are integers only divisible by 1 and themselves; they have many applications, most notably in cryptography.) Compile the copied files with the `-g` option:

```
gcc -Wall -std=gnu99 -o prime main.c prime.c -g
```

The `-g` option causes `gcc` to store information like line numbers in the code to facilitate debugging. Note that we are compiling two files of source code into a single executable. The files will compile with a warning, which actually comes from the first error we will find while debugging. (If you compile without `-Wall`, you don't get this warning.)

Run the program (`./prime`) and enter **20** at the prompt. The result will be a segmentation fault, which we will use `gdb` to find and fix.

The basic strategy

A typical usage of `gdb` runs as follows: After starting it, we set *breakpoints*, which are places in the code where we wish execution to pause. Each time `gdb` encounters a breakpoint, it suspends program at that point, giving us a chance to check the values of various variables.

In some cases, when we reach a breakpoint, we will single step for a while from that point onward, which means that `gdb` will pause after every line of source code. This may be important, either to pinpoint the location where a certain variable changes value, or to observe the flow of execution (like seeing which part of an if-then-else construct is executed).

It is wise to have two windows open when debugging: one in which to edit source, and one in which to run `gdb`. Now open that second window in which to run `gdb` and change into the directory containing your copies of `main.c` and `prime.c`. Then start `gdb` with the command `gdb prime`.

The h (help) command

The commands are not case sensitive. Command abbreviations are allowed as long as they are unambiguous. If you have questions about `gdb`, the help command should be your first resort. Try it by typing **h**.

The r (run) command

The run command begins execution of your program. If it takes command-line arguments (the prime program does not), then you must include them with the run command. For example, if you would ordinarily run the program with `prime param param2`, then within `gdb` you would type `r param param2`. If you use the

[1] The `gdb` part of this is based on a tutorial by by Norman Matloff.

run command more than once in the same debugging session, you do not have to type the command-line arguments after the first time; the old ones will be repeated by default. Now type **r**.

You now see the program's prompt. Enter **20**. You should then see something like the following:

```
Enter upper bound:
20

Program received signal SIGSEGV, Segmentation fault.
0x00007ffff7de3149 in __vfscanf_internal (s=<optimized out>, format=<optimized out>, argptr=argptr@entry
1896 ./stdio-common/vfscanf-internal.c: No such file or directory.
```

(On different operating systems and configurations, you'll get different messages. They should have the same gist in any case.) So, the error occurred in the function `__vfscanf_internal` This is not a function within the program, so it must have been called by a C library function such as `printf` or `scanf`. Given the name, the latter sounds more likely. We can use the Backtrace command to see where `__vfscanf_internal` is called.

The bt (Backtrace) Command

If you have an execution error with a mysterious message like "bus error" or "segmentation fault," the Backtrace command will at least tell you where in your program this occurred, and if in a function, where the function was called from. Since locating the error is often the key to solving a bug, this can be extremely valuable information. Now type **bt**. You should see something similar to the following:

```
#0  0x00007ffff7de5149 in __vfscanf_internal (s=<optimized out>,
    format=<optimized out>, argptr=argptr@entry=0x7fffffffe860,
    mode_flags=mode_flags@entry=2) at ./stdio-common/vfscanf-internal.c:1896
#1  0x00007ffff7de0142 in __isoc99_scanf (format=<optimized out>)
    at ./stdio-common/isoc99_scanf.c:30
#2  0x00005555555551c0 in main () at main.c:15
```

This shows that `__vfscanf_internal` was called from the function `__isoc99_scanf` in, which was called from `main.c` line 15. If you look at the code in `main.c`, you see that line 15 is

```
scanf("%d", UpperBound);
```

Aha! So it was indeed called from `scanf`, which in turn was called from `main`, at line 15. Since `scanf` is a C library function, it is presumably well-debugged already, so the error was probably not in `scanf`. So, the error was probably in how we called to `scanf` on line 15 of `main.c`.

The l (List) Command

You can use this to list parts of your source file(s). Type **l 15**. This asks for a display of line 15 of the current file, but doesn't work because we're in an internal file. If you want to look at lines in a different file, precede the line number by the file name and a colon. Try this out by typing **l prime.c:15**.

You can also specify a function name after list, in which case the listing will display the lines surrounding the first line of the function. Now type **l main**. You will see the first few lines of the `main` function.

Now, take a closer look at line 15 of `main.c`. What you see is a typical beginning C programmer's error; there should be an ampersand before `UpperBound`. So, in another window, open `emacs` and fix line 15 of `main.c`, and then recompile the program.

Note that we do not leave `gdb` while doing this, since `gdb` takes a long time to load. When we give `gdb` the run command after recompiling, it automatically loads the newly-recompiled executable for our program (it notices that we recompiled because the timestamp on our .c source file is newer than the executable file). Now run the program in the `gdb` window by typing the command **r**. You will be asked if you want to restart the program; type **y**. Enter **20** at the prompt again. You should see something like

```
Program received signal SIGSEGV, Segmentation fault.
0x0000555555555267 in CheckPrime (K=3, Prime=0x555555558040 <Prime>) at prime.c:14
14     if (Prime[J] == 1)
```

This is another seg fault. One of the most common causes of a seg fault is a wildly-erroneous array index. The program tells us that the problem occurred on line 14 of `prime.c` in the `CheckPrime` function. Looking at that line we see an array `Prime` being accessed. Thus we should be highly suspicious of `J` in this case, and should check what its value is, using `gdb`'s Print command.

The p (Print) Command

The print command displays the value of the indicated variable or expression. If we have `gdb` print out a struct variable, the individual fields of the struct will be printed out. If we specify an array name, the entire array will be printed. Keep in mind the difference between global and local variables. If for example, you have a local variable `L` within the function `F`, then if you type `p L` when you are not in `F`, you will get an error message like "No variable L in the present context." Take a look at the value of J by typing **p J**.

You should see something like (your value may be different, but will be large):

```
$1 = 34800
```

Wow! The array `Prime` was created to contain 50 integers, and yet here we are trying to access `Prime[34800]`! So, `gdb` has pinpointed the exact source of our error; the value of `J` is too large on this line. Now we have to determine why `J` is so big. Let's take a look at the `CheckPrime` function by typing **l CheckPrime**. Then just press return to display the next 10 lines of the function. Your display should be something like the following:

```
(gdb) l CheckPrime
1      void CheckPrime(int K, int Prime[]) {
2
3      int J;
4
5      /* the plan:  see if J divides K, for all values J which are
6      themselves prime (no need to try J if it is nonprime), and
7      less than or equal to sqrt(K) (if K has a divisor larger
8      than this square root, it must also have a smaller one,
9      so no need to check for larger ones) */
10
(gdb)
11
12      J = 1;
13      while (1)  {
14        if (Prime[J] == 1)
15          if (J % K == 0)  {
16            Prime[K] = 0;
17            return;
18          } /* if */
19        J++;
20      } /* while */
```

Look at the comment starting on line 5. We are supposed to be dividing `K` by `J`. Looking at our line 15, we see that we are dividing `J` by `K` instead. In your text editor, change line 15 to read

```
if (K % J == 0) {
```

Recompile and run the program again using **r**. (You'll be asked if you want to restart the program; say **y**.) Enter **20** at the prompt again. The output should be similar to (the process number may differ)

```
Enter upper bound:
20
[Inferior 1 (process 10915) exited normally]
```

This is saying that the program finished and exited normally (as opposed to with an error). That isn't right because it didn't report any primes up to the value 20. Let's use `gdb` to step through the program.

The b (Breakpoint) Command

The Breakpoint command says that you wish execution of the program to pause at the specified line. For example, `b 30` means that you wish to stop every time the program gets to line 30. As with the List command, if you have more than one source file, precede the line number by the file name and a colon, e.g. `b prime.c: 9`. You can also use a function name to specify a breakpoint, meaning the first executable line in the function, e.g., `b CheckPrime`. We want to pause at the beginning of `main`, and take a look around. So type **b main**. You should see something like

```
Breakpoint 1 at 0x555555555195: file main.c, line 14.
```

Now, `gdb` will pause our program whenever it comes to line 14 of the file `main.c`. This is Breakpoint 1; we might (and will) set other breakpoints later, so we need numbers to distinguish them, e.g., in order to specify which one we want to cancel. Now let's run the program by typing **r**.

We see that, as planned, `gdb` did stop at the first line of `main` and the following is displayed:

```
Breakpoint 1, main () at main.c:14
14        printf("Enter upper bound:\n");
```

Now we would like to execute the program one line at a time. The Next and Step commands permit us to do this.

The n (Next) and s (Step) Commands

Both the Next and Step commands tell `gdb` to execute the next line of the program, and then pause again. If that line happens to be a function call, then Next and Step will give different results. If you use Step, the next pause will be at the first line of the function. If you use Next, the next pause will be at the line following the function call (the function will be executed, but there will be no pauses within it). This is very important, and can save you a lot of time: If you think the bug does not lie within the function, then use Next, so that you don't waste a lot of time single-stepping within the function itself. When you use Step at a function call, `gdb` will also tell you the values of the parameters, which is useful to confirm that the correct arguments are being passed to the function.

Now use the Next command by typing **n**. You will see

```
Enter upper bound:
15        scanf("%d", &UpperBound);
```

What happened was that `gdb` executed line 14 of `main.c` (the call to `printf`) and then paused at the next line, the `scanf`.

Ok, let's execute line 15; use the Next command again (type **n**). Since this executes `scanf`, you will have to enter an integer before the program can complete line 15. Type **20**. Your screen should look like

```
16        Prime[1] = 1;
```

As expected, `gdb` paused at the next executable line, line 16. Now let's check that `UpperBound` was read correctly. We think it was, but remember, the basic principle of debugging is to check anyway. To do this, we will use the Print command, so type **p UpperBound** Your screen should look like:

```
(gdb) p UpperBound
$2 = 20
```

OK, that's fine. So, let's continue to execute the program one line at a time. Since we are not calling any functions, both the Next and Step commands will do exactly the same thing. Let's get some experience with the Step command, type **s**.

As expected, this takes us to the next line, number 17. Use the step command twice more.

```
(gdb) s
19        for (i = 3; i <= UpperBound; i += 2) {
(gdb) s
20          CheckPrime(i, Prime);
```

Now we have paused at the call to `CheckPrime`. Since we think we found all the bugs in `CheckPrime`, let's just use Next to run it without pausing. Type **n**. Your screen should look like the following:

```
21          if (Prime[i])
```

Step once more.

```
19        for (i = 3; i <= UpperBound; i += 2) {
```

Hey! We didn't execute the `printf` statement, even though we know that 3 is prime! Let's take a quick look at the Prime array using Print. Type **p Prime[3]**. This gives a 0, but according to our comments at the top of `main.c`, the Prime array should be set to 1 when a number is prime. Looks like `CheckPrime` still has a bug in it. To find it, we should re-run the program and pause at `CheckPrime` this time. Set a new breakpoint at the beginning of `CheckPrime` by typing **b CheckPrime**.

```
Breakpoint 2 at 0x55555555524c: file prime.c, line 12.
```

Then restart the program with **r** and say that you want to restart from the beginning.

```
Breakpoint 1, main () at main.c:14
14        printf("Enter upper bound:\n");
```

The c (Continue) Command

Since we are fairly sure there are no bugs at the beginning of the program, we would prefer to not single-step through the lines of code before the next breakpoint. The Continue command causes the program to run at full speed until it reaches another breakpoint. Now type **c**, followed by **20** (as input to the program).

```
Continuing.
Enter upper bound:
20
Breakpoint 2, CheckPrime (K=3, Prime=0x555555558040 <Prime>) at prime.c:12
12        J = 1;
```

Note that we automatically see the values of the parameters passed to `CheckPrime`. We can easily confirm that `K` has the value 3.

Hit **n** to execute this line.

The disp (Display) Command

Since we are about to enter a for loop, it would be wise to keep track of the value of loop control variable, `J`. If we use the Print command, we have type a command after each step. The Display command displays the values of variables automatically at each pause. To watch the `J` variable type **disp J**.

The screen should look like

```
1: J = 1
```

As expected, J has been initialized to 1. Let's take another two Steps:

```
(gdb) s
15            if (K % J == 0)  {
1: J = 1
(gdb) s
16              Prime[K] = 0;
1: J = 1
```

Wait a minute! `K` is three and the comment in `main.c` says that `Prime[K]` is set to 0 when `K` is not prime! Looking at `if K % J == 0`, we can see that it is always true if `J` starts at 1. We want `J` to start at 2 instead; make this change.

The undisp (Undisplay) Command

Since we no longer want to watch the value of `J` repeatedly again, let's use the undisplay command. Each time a variable is displayed, it is preceded by a number and a colon. You use this number to turn off its display. The variable `J` is preceded by `1:`, so you can undisplay it by typing **undisp 1** Nothing will appear on the screen, except the next prompt. Now recompile.

The d (Delete) Command

Since we are sure the beginning of the program contains no bugs, there is no need for the breakpoint at its start. We can use the Delete command to remove breakpoints. We follow the command with the breakpoint number(s) that we wish to remove. If we do not supply any numbers, then all breakpoints are deleted. As we noted at the time, our breakpoint at the beginning of `main` is number 1, so type **d 1**. Nothing will appear on the screen, except the next `(gdb)` prompt. In fact, let's delete all the breakpoints. Use **d** to do this. (Since only number 2 remains, you could also delete it individually with **d 2**.)

Now run the program and enter 20 at the prompt again. You should see

```
Program received signal SIGSEGV, Segmentation fault.
0x0000555555555267 in CheckPrime (K=3, Prime=0x555555558040 <Prime>) at prime.c:14
14          if (Prime[J] == 1)
```

Another segmentation fault! Since a line with an array is again suspect we should check the value of `J` by using the print command: **p J**. The value of `J` is huge. The array `Prime` only has space for fifty integers. If you read the comments on lines 5–9 of `prime.c`, then you see that `J` should stop when it reaches $\sqrt{K}$. Yet we never made this check, so `J` just kept growing and growing, eventually triggering a seg fault. In your editor, delete the `J++` on line 19 and the `J = 2` on line 11. Then change the while statement on line 12 into

```
for (J = 2; J * J <= K; J++) {
```

When you are done, that part of your code should look like the following:

```
for(J = 2; J * J <= K; J++)
  if (Prime[J] == 1)
    if (K % J == 0)  {
      Prime[K] = 0;
      return;
    } /* if */
```

Then recompile and run the program with input 20. Now you should get the correct output:

```
3 is a prime
5 is a prime
7 is a prime
11 is a prime
13 is a prime
17 is a prime
19 is a prime
```

Looking at buffer overflows

Now that you have a handle on `gdb`, we will use it to briefly examine a buffer overflow. We're not going to actually take control of the program since euclid has OS-level defensive measures in place, but we'll see the main ideas.

For this part of the lab, turn your attention to `overflow.c`. Open this file with `emacs` and you'll see that it has minimal code, just a `main` method that calls a function `fun`, each only a few lines long. The overflow can happen in `fun`, which has a stack-allocated string `s` with 5 chars of memory reserved. It uses `scanf` to read a string into this buffer. If the user gives too many characters for this string, they will overflow the buffer.

Compile this program with the `-g` flag and open it in `gdb`. Use `b fun` to set a breakpoint at the beginning of the `fun` function. Then run the program, continue it through the call to `scanf` (use `n`) and enter a short string (like "hi" (without the quotes and ending with a newline). After the string is read, you can print `s` to see that it got your input. You can also see that the stack is still in good shape with `bt`; recall that this function prints the different levels of the stack so you can see that you're currently in `fun`, which was called from `main`. Use `c` to finish the program and it will print out the length of the input string.

Now run the program again but give it a string that is too long (10ish characters is enough). Now if you run `bt` after the call to `scanf`, you can see that the stack is messed up. Here is what I saw:

```
#0  fun () at overflow.c:9
#1  0x0000555555555217 in main () at overflow.c:13
Backtrace stopped: previous frame inner to this frame (corrupt stack?)
```

The stack frame is broken so tracing back through the stack quickly leads to garbage. The program will die with an error (likely a segmentation fault) if you continue it.

To see what is happening, run the program again, giving it a bunch of the character 'a', and stop right before `fun` returns. If you print the value of `s`, it will show "aaaaa", but this is using the length of the variable. If you print `s[4]`, which should be the string-terminating 0, you get 'a'. Same with `s[5]`, `s[6]`, etc.

If you have extra time, write another C program with a buffer overflow, this time using `strcpy` to perform the overflow. See if you can get the overflow to change the value of antoher string rather than crashing.