The Problem

Let's say you've been given an assignment. You then spend days thinking about it (maybe passively), go through the logic behind it on paper, and then code it. Finally, it's ready and you're prepared to see this beautiful program of yours in action. Always one step ahead, your Makefile is ready to go and you use `make` to compile your code. No syntax errors, and no compiler warnings. Fantastic.

You finally run your program, and see it throw some nice text at you in the terminal. It prompts you for your username, and you enter whatever alias you're used to. Suddenly, this happens:

```
Segmentation fault: 11
```

Worry not, for this has happened to many C programmers.

Debugging with GDB

There are different ways of debugging your code, but we will go over the GNU Debugger ("GDB") in particular. As its name suggests, GDB is a tool designed to help pinpoint errors in your code. To enable the usage of GDB on your program, you must first compile it with the `-g` flag, like so:

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

To use GDB, simply type `gdb <program-name>`. With the example above, we would execute the following:

```
gdb example
```

Assuming you have GDB installed, you should now see a wall of text detailing things such as the GDB version, where to find the manual, etc. To use GDB, you simply enter text commands like you would in your Unix shell. If you're new to the tool, you can:

- Type `help` for a list of classes of commands
- Type `help <class>` for a list of commands within the class
- Type `help <command>` for documentation regarding a particular command

At this point, you're free to figure GDB out on your own by using combinations of commands to figure out what's wrong with your code. That said, here are some useful commands which may help you pinpoint the problem (based off content provided by Marina Barsky):

`run`
Runs through your program and tells you when an error is encountered, or if the program terminates without error (i.e. the desired outcome). If you execute run when you have breakpoints set (see below), when you execute the "run" command, program execution will *pause* at wherever you set your breakpoint(s). From there, you can use `step` or `next` to go through your code.

`break`
Sets a breakpoint. You can set breakpoints at specific lines in your code, or even at function calls. Some examples:

- `break 3`: sets a breakpoint at line 3 of your code
- `break func_name`: sets a breakpoint at whenever the `func_name` function is called

`delete`
Removes a previously-set breakpoint. If not given any parameters, `delete` will remove all set breakpoints (after asking if that's okay with you).

`print`
Takes a variable as a parameter, and prints the value of the variable.

`print/x`
Similar to `print`, but prints the value in hexadecimal. This is particularly useful if you're dealing with pointers, or want to know the address of something.

`step    (shorthand: s)`
Executes the next instruction of your program.

`next    (shorthand: n)`
Similar to step, but treats function calls as single instructions (i.e. does not "step into" subroutines).

`continue`
Go to the next breakpoint.

`backtrace    (shorthand: bt)`
Generate a stack trace of the function call(s) which lead to a segmentation fault.

To illustrate the difference between step and next, consider the following program:

example.c

```c
#include <stdio.h>

void do_stuff() {
    for (int i = 0; i < 4; i++) {
        printf("%d\n", i);
    }
}

int main() {
    printf("Hello!\n");
    do_stuff();
    return 0;
}
```

Now compare the following:

step

```
(gdb) break main
Breakpoint 3 at 0x4005b2: file example.c,
line 10.
(gdb) run
Starting program: ...

Breakpoint 3, main () at example.c:10
10      printf("Hello!\n");
(gdb) n
Hello!
11  do_stuff();
(gdb) s
do_stuff () at example.c:4
4       for (int i = 0; i < 4; i++) {
(gdb) n
5           printf("%d\n", i);
(gdb) n
0
4       for (int i = 0; i < 4; i++) {
(gdb) n
5           printf("%d\n", i);
(gdb) n
1
4       for (int i = 0; i < 4; i++) {
(gdb) n
5           printf("%d\n", i);
(gdb) n
2
4       for (int i = 0; i < 4; i++) {
(gdb) n
5           printf("%d\n", i);
(gdb) n
3
4       for (int i = 0; i < 4; i++) {
(gdb) n
7   }
(gdb) n
main () at example.c:12
12      return 0;
(gdb) n
13  }
(gdb) n
(gdb)
[Inferior 1 (process 38097) exited
normally]
```

next

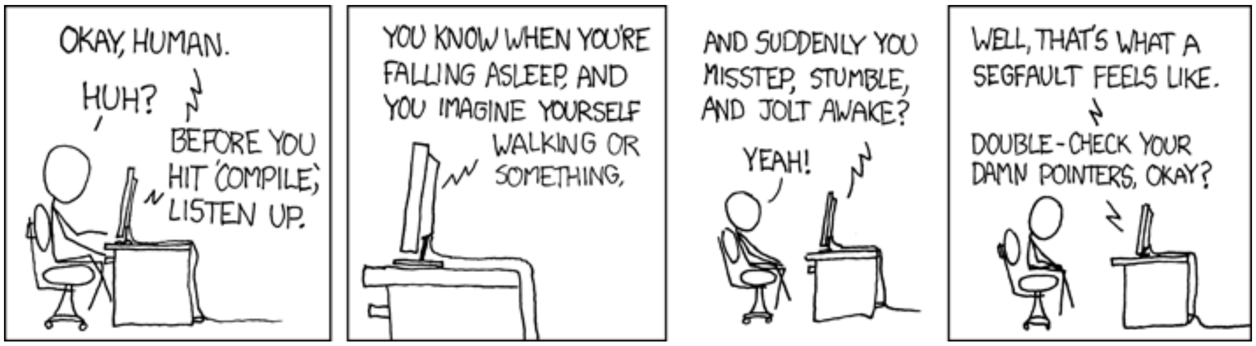
```
(gdb) break main
Breakpoint 4 at 0x4005b2: file example.c,
line 10.
(gdb) run
Starting program: ...

Breakpoint 4, main () at example.c:10
10      printf("Hello!\n");
(gdb) n
Hello!
11      do_stuff();
(gdb) n
0
1
2
3
12      return 0;
(gdb) n
13  }
(gdb) n
[Inferior 1 (process 38176) exited
normally]
```

As seen above, using `step (s)` allows us to *step* into `do_stuff()`, from which we can walk through each instruction of the function call (a subroutine). In contrast, using `next (n)` treats the entirety of `do_stuff()` as if it were one instruction, executing it without examining the instructions inside. (If you're familiar with Python debuggers, `next` acts like "step over".)

By stepping through the program with GDB, we can better locate exactly where our program went wrong, and under what circumstances. The next time you run into a segmentation fault, try using GDB to spot the error and then adjust your code accordingly.

Prepared by Elaine Huynh

Relevant xkcd


OKAY, HUMAN.
HUH?
BEFORE YOU HIT 'COMPILE,' LISTEN UP.
YOU KNOW WHEN YOU'RE FALLING ASLEEP, AND YOU IMAGINE YOURSELF WALKING OR SOMETHING,
AND SUDDENLY YOU MISSTEP, STUMBLE, AND JOLT AWAKE?
YEAH!
WELL, THAT'S WHAT A SEGFAULT FEELS LIKE.
DOUBLE-CHECK YOUR DAMN POINTERS, OKAY?

Source: https://xkcd.com/371/