gdb operation reference

Lecture can be a little free-form sometimes, which can make the terminal output a little difficult to read. In response to that, I'm trying something new with this section. It's something like annotated notes of what an uninterrupted lecture might look like.

A debugger is a program than runs another program. This lets the debugger (the parent) stop, start, or modify the program that is being debugged (the child, target, or inferior process).

The debug interface itself is remarkably simple. It only needs the ability to read/write registers/memory and trap on certain memory access.

To debug a program with gdb, simply put gdb in front of the program, i.e.:

```
> ./prime       # running normally
> gdb ./prime   # debugging the program
```

One annoying gotcha shows up if the program to debug takes any arguments. The simple prime program does not, but if it did:

```
> ./prime --imaginary-argument              # running normally
> gdb ./prime --imaginary-argument          # will not work
gdb: unrecognized option '--imaginary-argument'
> gdb --args ./prime --imaginary-argument   # gdb will ignore everything after --args
```

Once you start gdb, it presents you with a prompt (gdb), asking for commands:

- `(gdb) run`
  - When you first start gdb, it does not start running the child program by default. You must use the run command to run the child program.
  - Every time you call run, gdb will run the program with any arguments you specified on the original command line. You can also pass command line arguments here instead, e.g.:

    ```
    (gdb) run --different-argument
    ```

    will run the program with the new argument.
  - If you recompile the program being debugged, gdb will automatically reload the new version every 'run'.
    - *You should not ever quit gdb!* Have it open in another terminal. Otherwise you will have to set up new breakpoints every time you run gdb.
- `(gdb) backtrace, up, down, frame, print`
  - Recall that when you program is running, it has a function call stack. The idea here is that every time a function calls another function, you build up a list. Consider this program:

```c
#include <stdio.h>
int subtract (int a, int b) { return a - b; }
int divide (int a, int* b) { return a / *b; }
int do_math (int x, int y, int z) {
  int temp = subtract(x, y);
  temp = divide(z, &temp);
  return temp;
}
int main () {
  int temp;
  temp = do_math(10, 10, 20);
  printf("Result: %d\n", temp);
  return 0;
}
```

```
Function call stack over time:
main
main -> do_math
main -> do_math -> subtract
```

```
main -> do_math
main -> do_math -> divide
! divide by zero exception
```

```
(gdb) backtrace
#0  0x000000000040058d in divide (a=20, b=0x7fffffffdf54) at test.c:3
#1  0x00000000004005d4 in do_math (x=10, y=10, z=20) at test.c:6
#2  0x0000000000400608 in main () at test.c:10
```

The pointer makes it difficult to see what happened here. We can ask gdb to dereference it for us, however:

```
(gdb) print *b
$1 = 0
```

Alternatively, we could ask it to print the value of the variable temp:

```
(gdb) print temp
No symbol "temp" in current context.
```

But the "current context", aka the divide function, has nothing called temp inside of it, so we need to go *up* the call stack:

```
(gdb) up
#1  0x00000000004005d4 in do_math (x=10, y=10, z=20) at test.c:6
6   temp = divide(z, &temp);
(gdb) print temp
$2 = 0
```

You can also use the frame command to change your 'frame of reference'. Notice that the backtrace is numbered, to get back into the divide context, we could either call `down`, or:

```
(gdb) frame 0
#0  0x000000000040058d in divide (a=20, b=0x7fffffffdf54) at
test.c:3
3 int divide (int a, int *b) { return a / *b; }
```

If we try to move forward in the execution of the program, we'll find that it has died:

```
(gdb) continue
Continuing.

Program terminated with signal SIGFPE, Arithmetic exception.
The program no longer exists.
```

- (gdb) list, break, continue, step, next, set
  - We can stop, manipulate, and control the program a lot as well, affecting its behavoir:

```
(gdb) list
1
2 int subtract (int a, int b) { return a - b; }
3 int divide (int a, int *b) { return a / *b; }
4 int do_math (int x, int y, int z) {
5   int temp = subtract(x, y);
6   temp = divide(z, &temp);
7   return temp;
8 }
9 int main () {
```

Let's set a breakpoint right before anything bad happens.

```
(gdb) break 3
Breakpoint 1 at 0x400583: file test.c, line 3.

(gdb) run
Starting program: /tmp/a/a.out

Breakpoint 1, divide (a=20, b=0x7fffffffdf54) at test.c:3
3 int divide (int a, int *b) { return a / *b; }
```

We can see here that this is about to be a problem.

```
(gdb) print *b
$3 = 0
```

But we can simply overwrite the value and it will run!

```
(gdb) set *b=1
(gdb) continue
Continuing.
Result: 20
[Inferior 1 (process 5549) exited normally]

(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x00000000004005c3 in divide at test.c:3
        breakpoint already hit 1 time
```

Let's delete that breakpoint so we can try something else

```
(gdb) delete 1
(gdb) info breakpoints
No breakpoints or watchpoints.

(gdb) list main
4 int do_math (int x, int y, int z) {
5   int temp = subtract(x, y);
6   temp = divide(z, &temp);
7   return temp;
8 }
9 int main () {
10    int temp;
11    temp = do_math(10, 10, 20);
12    printf("Result: %d\n", temp);
13    return 0;
(gdb) break 11
Breakpoint 2 at 0x400638: file test.c, line 11.
(gdb) run
Starting program: /tmp/a/a.out

Breakpoint 2, main () at test.c:11
11    temp = do_math(10, 10, 20);
(gdb) next
```

Notice that next attempts to "jump over" do_math, running code until it finishes, which will fail

```
Program received signal SIGFPE, Arithmetic exception.
0x00000000004005cd in divide (a=20, b=0x7fffffffdf44) at test.c:3
3 int divide (int a, int* b) { return a / *b; }
(gdb) run
The program being debugged has been started already.
```

```
    Start it from the beginning? (y or n) y
    Starting program: /tmp/a/a.out

    Breakpoint 2, main () at test.c:11
    11    temp = do_math(10, 10, 20);
```

Instead here, we "step into" do_math

```
    (gdb) step
    do_math (x=10, y=10, z=20) at test.c:4
    4 int do_math (int x, int y, int z) {
    (gdb) <enter -- repeat last command>
    5   int temp = subtract(x, y);
    (gdb) next
```

This time next succeeds because we can run subtract without error

```
    6   temp = divide(z, &temp);
    (gdb) step
    divide (a=20, b=0x7fffffffdf44) at test.c:3
    3 int divide (int a, int* b) { return a / *b; }
```