Breakpoint

In debugging, sometimes we need to execute or halt a code line by line. That's where breakpoints come in handy.

We'll cover the following

- Inspecting a variable with the gdb print command
- Stepping
- More breakpoints

The `print` command will print out the value of a variable that is defined in the current stack.

Open the go.c file from the previous lesson. We'll use this as a reference for this lesson. Let's try to print the `s` variable (which is supposed to be a character buffer). Let's first however define a breakpoint.

A **breakpoint** is a flag that will stop the program at a specific line of code. Let's insert a breakpoint on line 9 of our code. This means the program will stop before executing that line of code and leave us in the gdb debugger. We use the `gdb` command `break` to insert a breakpoint, and then the `run` command to run the program:

```
plg@wildebeest:~/Desktop/CBootCamp/code/debug$ gdb go
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/plg/Desktop/CBootCamp/code/debug/go...done.
(gdb) break 9
Breakpoint 1 at 0x400666: file go.c, line 9.
(gdb) run
Starting program: /home/plg/Desktop/CBootCamp/code/debug/go

Breakpoint 1, getWord (maxsize=256) at go.c:9
9     scanf("%s", s);
```

```
(gdb)
```

Now `gdb` is waiting for input, and we are stopped just short of line 9. We can now inspect the value of some variables, specifically let's look at our character buffer `s`.

Inspecting a variable with the gdb print command

We can use the `gdb` `print` command to inspect the value of a variable on the stack:

```
(gdb) print s
$1 = 0x400865 "H\205\355t\034\061\333\017\037@"
(gdb)
```

What we see is a little bizarre. The code in quotes is a bunch of random looking stuff. It's instructive to ask, what did we expect to see here? Well, the `s` variable ought to be an empty string, not something filled with junk. Why is it not empty? The reason is, because although we have declared the pointer `char *s`, we have not actually allocated any memory. So when we ask `gdb` to `print s`, it is going to whatever the (uninitialized) pointer `s` points to (junk) and printing that out.

Let's fix our bug and re-run the program and see how it differs. Un-comment out the line that you commented out in the previous lesson, so that `calloc()` can allocate memory. Now let's re-run using `gdb`, insert a breakpoint again, and see what the value of `s` is:

```
plg@wildebeest:~/Desktop/CBootCamp/code/debug$ gcc -g -o go go.c
plg@wildebeest:~/Desktop/CBootCamp/code/debug$ gdb go
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/plg/Desktop/CBootCamp/code/debug/go...done.
(gdb) break 9
Breakpoint 1 at 0x4006bc: file go.c, line 9.
(gdb) run
Starting program: /home/plg/Desktop/CBootCamp/code/debug/go

Breakpoint 1, getWord (maxsize=256) at go.c:9
9     scanf("%s", s);
(gdb) print s
$1 = 0x602010 ""
(gdb)
```

Now we see what we expected, `s` is an empty string “”. Let’s now put another breakpoint at line 10, so that we can verify that once we enter a string, it’s actually in there:

```
(gdb) break 10
Breakpoint 2 at 0x4006d5: file go.c, line 10.
(gdb) continue
Continuing.
enter a string (max 256 chars): TheDudeAbides

Breakpoint 2, getWord (maxsize=256) at go.c:10
10    return s;
(gdb) print s
$2 = 0x602010 "TheDudeAbides"
(gdb)
```

We insert a new breakpoint at line 10, we issue the `continue` command, we enter a string, the breakpoint is hit, and now we ask to see the value of the `s` variable again, and indeed, it contains our string. Good.

Stepping

Above we used the `continue` command to continue execution of our program. One can also take individual steps, line by line, using the `step` command in `gdb`.

More breakpoints

We can list the current breakpoints with the `info breakpoints` command. Here’s what we get if we do that with the code that we have been working with above:

```
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x00000000004006bc in getWord at go.c:9
	breakpoint already hit 1 time
2       breakpoint     keep y   0x00000000004006d5 in getWord at go.c:10
	breakpoint already hit 1 time
(gdb)
```

We can use the `disable` command to disable individual breakpoints. They are named by the `Num` column in the info listing. So to disable the breakpoint on line 9 above we would issue the command `disable 1`.

We can delete breakpoints altogether using the `delete` or `clear` commands. We can issue the command `delete 2` to delete the line 10 breakpoint above. We can also use the `clear` command to delete breakpoints by location like this: `clear go.c:10`

Other than errors in the code itself, there are many logical mistakes which distort a programs functionality without the user knowing.

So we’ll highlight some tips to help you avoid logical errors related to C.