Exercise 1

Take a look at the file `powers.c`. The program prompts the user for two numbers and then computes the value of `base^exp` where `^` means the exponent (not the C xor operator).

We compile this code as:

```
$ gcc -Wall powers.c -o powers
```

with no errors.

The sample run of this program may look as follows:

```
$ ./powers
Enter the base (integer): 2
Enter the exponent (positive integer): 3
 2^3  is 131056
```

Well, this does not make much sense.

If we try to run ./powers in the debugger

```
gdb ./powers
```

the `gdb` will report a potential problem

```
Reading symbols from ./powers...(no debugging symbols found)...done.
```

This is because the code was not compiled to allow for easy debugging. Let's go back to the compilation stage and add `-g` flag. The manual page for `gcc` describes this option as

Produce debugging information in the operating system's native format (stabs, COFF, XCOFF, or DWARF 2). GDB can work with this debugging information.

This is exactly what we need.

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

and let's start the debugger with the newly compiled program

```
gdb ./powers
```

`gdb` provides an interactive shell that allows us to control what is happening during the debugging process. When you start the debugger, it is waiting for the first instruction. Your console should look similar to:

```
$ gdb ./powers
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 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".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./powers...done.
(gdb)
```

We will set a first **breakpoint** - this is where the program should stop and wait for the next instructions.

```
break 1
```

sets the breakpoint on the fist line. Since this program has only one file and only one function, the debugger knows which line 1 we mean. If this was not the case, the `break` instruction can be given additional arguments
`break [file_name]:line_number` or `break [file_name]:func_name`

The program is still not running. To start it we need to execute the `run` instruction:

`run`

The program will run all the way to the first breakpoint and display the content of the line. At this point you should see something similar to:

```
(gdb) break 1
Breakpoint 1 at 0x4005fe: file powers.c, line 1.
(gdb) run
Starting program: /home/asia/Data/NYU_Teaching/csci201/recitations/rec5/ex1/powers

Breakpoint 1, main () at powers.c:4
4   {
(gdb)
```

To tell `gdb` to advance to the next line of the code, we can use either `step` or `next` instructions. The difference between them is what happens when the line that we are about to execute contains a function call:

`step` goes into the function and allows us to execute that function one line at a time `next` goes over the function and moves to the next line after the function returns.

Let's first enter `step` and for the next line enter `next`. You should now see the following:

```
(gdb) run
Starting program: /home/asia/Data/NYU_Teaching/csci201/recitations/rec5/ex1/powers

Breakpoint 1, main () at powers.c:4
4   {
(gdb) step
6       printf ("Enter the base (integer): ");
(gdb) next
7       scanf ("%d", &base );
```

There is a function call to `printf()` on line 6 - there is no reason (and no way) to examine content of that function, so we just stepped over it. Now we are about to execute line 7, which is another function call. We will use `next` to step over the call to `scanf()` but since this function expects input, the program will prompt for the value to be entered. Let's enter `2`.

This is a good place to examine the value of the `base` variable. If there is any problem with how `scanf` is used, then the value of the `base` may be not equal to 2 as it should be. The following instruction tells `gdb` to figure out the value of `base`:

```
print base
```

This should produce the following output (unless, of course, you entered a value other than 2 for the value of the base.

```
8       printf ("Enter the exponent (positive integer): ") ;
(gdb) print base
$1 = 2
```

This tells us that the value of `base` variable is correct.

We will now execute the content of lines 8 and 9, enter the value for the exponent and check the value of that variable.

```
(gdb) next
9       scanf ("%d", &exp );
(gdb) next
Enter the exponent (positive integer): 3
12      for (i=1; i<exp; i++)
(gdb) print exp
$2 = 3
```

Again, the value of `exp` is correct. So we can continue with the program trying to find where the problem happens.

Line 12 is the one to be executed next - this starts a for loop that will calculate the final answer. We will use the `next` instruction to start the loop.

Line 13 is about to multiply `base` for the first time by `value`. Before we execute that line, let's make sure that the values of all the variables still make sense: we will use `print` to print the values of `i`, `base` and `value`.

You should now be looking at something like this:

```
13          value = value * base;
(gdb) print i
$3 = 1
(gdb) print base
$4 = 2
(gdb) print value
$5 = 32767
```

This result should suggest where the problem is: why does the current value of `value` equal to some large number?

Hopefully, you realized by now that the variable `value` should have been initialized to `1` before the loop. If that's the case, then there is not much point in continuing with the debugger run. We can let the program finish by telling it to `continue` And then exiting the debugger by telling it to `quit`.

In summary, we executed the following `gdb` instructions:

```
break 1
run
step
next
next
print base
next
next
print exp
next
print i
print base
print value
continue
quit
```

`gdb` actually allows you to reduce the amount of typing: you can just use the first letters of the instructions (or first two or three letters) instead of their full names:

```
 b 1
r
s
n
n
p base
n
n
p exp
n
p i
p base
p value
c
q
```