Using gdb

1. Compile foo.s with

```
gcc foo.s -g -o foo
```

2. Run either with

```
gdb foo
```

   or with

```
gdb -tui foo
```

Vanilla-mode gdb allows you to set breakpoints, run or step through the program, and examine both registers and memory.

If you are using this mode keep a window open with the code file, preferably one that shows line numbers. If your code file has 10 lines of comments at the top, gdb will number the first line of executable code as 11, so gdb's line numbers will be the same as your editor's.

Graphical-mode gdb, which you get with the -tui option, give you a code "window" in addition to a command window, so with this you don't need a separate editor with the program.

It is also possible to get the graphical mode to give a third "window" which displays the values of the main registers

The graphical mode uses the *curses* library for text-based graphics, so it runs over the network, but it is both fairly clunky and fairly slow.

If you are going to use it make your terminal window quite large before you enter gdb.

The biggest advantage of the graphical mode is being able to see the registers without specifically querying them, so you might as well make use of this feature.

Bring up the register window with the command

```
        layout regs
```

The commands for executing the program are

run -- runs the program to the next breakpoint, or to the end. At any point you can restart the program with the *run* command.

continue -- resume executing the program to the next breakpoint.

step -- Does the next instruction. If this is a call it steps into the called function.

next -- Also does the next instruction. If this is a call it steps over the called function.

The breakpoint commands are

```
b <n> -- sets a breakpoint at line <n>
info break -- gives a numbered list of the
          breakpoints you have set
delete <n> -- deletes the nth breakpoint
delete -- deletes all breakpoints
```

The *print* command is useful for displaying registers (if you aren't using the register layout). The format is

```
print expression
```

such as

```
print $rax
```

or

```
print/d expression (as a decimal integer)
print/x expression (as a hexidecimal)
```

The *x* command is for examining memory, which for us primarily means the stack. The full format is

```
x/nfu <address>   where registers are used in
                  indirect mode, so
x $rsp  prints the top value on the stack
```

The three arguments are

n: number of values to print

f: the format (d for base-10, x for hex)

u: the size of the value (w for 32 bits, h for 16 bits, and b for 8 bits) There is no way to ask for 64-bit values.

For example, if you have just pushed to integers onto the stack you can see them with

```
x/4dw $rsp
```

This might print 5 0 3 0.

The stack is then

```
0  5   <-- $rsp
0  3
```

In the command x/nfu the f argument defaults to d (for base-10) and the u argument to w (for 32-bits) so you can usually use it as

```
x/n  <address>
```

or even just

```
x   <address>
```

(n defaults to 1)

We use %rbx as our frame pointer. During a call the arguments are at 16(%rbx), 24(%rbx) and so forth; the local variables are at -8(%rbx), -16(%rbx) etc. If there are two arguments we might print them as

```
x/4 16+$rbx
```

The first three local variables will be

```
x/6  -24+$rbx
```

but these will print in reverse order: the higher ones on the stack first.

help <command> gives you a brief summary of what the command does and what its options are.