GDB Tutorial for CS 263

This file contains pretty much all of the **gdb** commands you need to be successful in this course. Documentation for **gdb** is very strong, so just remember that more advanced functionality is always a Google away.

Table of Contents:

Links to helpful cheatsheets to bookmark are also included below.

1) Start

To start using **gdb** on a program called `program.c`, run:

```
gdb program.c
```

To quit out of **gdb**, type `quit`.

```
(gdb) quit
```

The `(gdb)` indicator will automatically appear on the console when you're using **gdb**. For the below examples, don't actually type `(gdb)` into the console when copying commands.

2) Control Program Flow

You'll find it useful to pause your program's execution on certain line numbers and/or function calls. This will allow you to inspect the contents of memory addresses and registers, change the values of variables, and track the flow of the program as it executes.

A) Pause execution

In order to pause a program using **gdb**, you need to set a *breakpoint* on the line where you want it to pause.

- Create a breakpoint

To pause the execution of the program at a specific line or function, use `b <function name or filename:line# or *memory address>` (or `break`) to insert a *[b]reakpoint* at the specified function, line #, or memory address.

Whenever **gdb** reaches a breakpoint, it will pause before executing that line.

For example, if we set `b func_1`, then **gdb** will stop the execution of your program whenever the function `func_1` is called.

```
(gdb) b func_1
```

- Delete a breakpoint

To remove a breakpoint, use the command `d <function name or filename:line# or *memory address>` (or `delete`) to *[d]elete* the breakpoint.

For example, if we had previously set `b func_1` but didn't want **gdb** to pause everytime the function `func_1` was called, then we should run `d func_1` to remove the breakpoint at `func_1`.

```
(gdb) d func_1
```

- Delete all breakpoints

To remove all breakpoints at once, use the command `clear`.

```
(gdb) clear
```

B) Resume execution

Suppose that **gdb** has stopped at a breakpoint, we've had time to check the memory addresses we needed, and now we want **gdb** to resume executing our program (until it finishes the program or hits another breakpoint).

- Continue

We can run `c` (or `continue`) to tell **gdb** to *[c]ontinue* execution. The continue command, `c`, will continue running our program until it encounters another breakpoint.

```
(gdb) c
```

- Step (into)

What if, however, we wanted to run our program **line-by-line**, and don't want to have to set hundreds of breakpoints in our file?

We can use the `s` (or `step`) command, which tells **gdb** to *[s]tep* into a single line of code. Note that if the current line calls a subroutine, then **gdb** will step **into** that subroutine. Usage:

```
(gdb) s
```

A more fleshed out example of `step` is below. Suppose that **gdb** is currently paused at a breakpoint on Line 1 (current control flow is marked with a `->`).

```
-> int a = 10;
int b = function();
int c = b + a;
...
function() {
        return 5;
}
```

We run `(gdb) s` to step past Line 1. Now, `a = 10` and we are paused on Line 2.

```
int a = 10; // a = 10
-> int b = function();
int c = b + a;
...
function() {
        return 5;
}
```

We run `(gdb) s` to step into `function()` on Line 2. We have now entered the `function()` subroutine and are paused on the first line of this subroutine.

```
int a = 10; // a = 10
int b = function();
int c = b + a;
...
function() {
        -> return 5;
}
```

We run `(gdb) s` once again to step past the `return 5` and, having now completed execution of Line 2, end up paused on Line 3. Now, `b = 5`.

```
int a = 10; // a = 10
int b = function(); // b = 5
-> int c = b + a;
...
function() {
        return 5;
}
```

Running `(gdb) s` one last time sends us past Line 3. Now, `c = 15`.

```
int a = 10; // a = 10
int b = function(); // b = 5
int c = b + a; // c = 15
...
function() {
        return 5;
}
```

****IMPORTANT NOTE:** In order to do everything listed above but at the **instruction level** instead of **line-by-line**, use `stepi` in place of `step`.

Next

The `n` (or `next`) command tells **gdb** to go to the *[n]ext* line in the program. In contrast to the `step` command, `n` **will not go into subroutines** and will instead evaluate each line as a single piece of execution.

```
(gdb) n
```

An example illustrating the difference between `n` and `s` is shown below. Suppose that **gdb** is currently paused at a breakpoint set on Line 1 (marked with a `->`).

```
-> int a = 10;
int b = function();
int c = b + a;
...
function() {
        return 5;
}
```

We run `(gdb) s` (or `(gdb) n`, both give the same result) to step past Line 1. Now, `a = 10` and we are paused on Line 2.

```
int a = 10; // a = 10
-> int b = function();
int c = b + a;
...
function() {
        return 5;
}
```

We run `(gdb) n` to step **over** the `function()` call on Line 2. Now, `b = 5` and we are paused on Line 3.

```
int a = 10; // a = 10
int b = function(); // b = 5
-> int c = b + a;
...
function() {
        return 5;
}
```

Running `(gdb) s` (or `(gdb) n`) one last time sends us past Line 3. Now, `c = 15`.

```
int a = 10; // a = 10
int b = function(); // b = 5
int c = b + a; // c = 15
...
function() {
        return 5;
}
```

Note that by using `n`, we didn't have to manually step through the `function()` subroutine. This can be highly useful is you want to step over a line that calls a standard C function like `snprintf` or `signal`.

****IMPORTANT NOTE:** In order to do everything listed above but at the **instruction level** instead of **line-by-line**, use `nexti` in place of `next`.

C) Speed up execution

Finish

The `finish` command tells **gdb** to finish executing the current function until it returns. **gdb** then prints the returned value to the console.

```
(gdb) finish
```

Suppose that **gdb** is currently paused at a breakpoint set in `func1` (marked with a `->`) after `func1` was called from Line 1, `int d = func1();`.

```
int d = func1();
int e = 1;
...
func1() {
        -> int a = 10;
        int b = func2();
        int c = b + a;
        return 5;
}
```

We run `(gdb) finish` to finish executing `func1`. Now, `d = 5` and we are paused on Line 2, having executed every line in `func1`.

```
int d = func1(); // d = 5
-> int e = 1;
...
func1() {
        int a = 10;
        int b = func2();
        int c = b + a;
        return 5;
}
```

3) Inspect Memory

In addition to controlling the flow of your program, **gdb** also provides several useful tools for inspecting the contents of a program's memory as it executes.

A) Specific Memory

- Print

The `print <exp>` command prints the contents of whatever the expression `<exp>` evaluates to. The expression `<exp>` is a C expression that can contain variable names, memory addresses, registers, constants, and operators like arithmetic, casting, and dereferencing operations.

```
(gdb) print ($eax + 4) + *some_ptr
```

Examine

The `x/<num><format><size> <address>` command *e[x]amines* the memory at the address `<address>`.

The `<num>` parameter specifies how many units of memory (each unit of size `<size>`) to display. Defaults to 1.

The `<format>` parameter specifies how the memory contents will be displayed. Defaults to `x` for hexadecimal. This value can be any of:

- `a` - pointer
- `s` - String
- `c` - Read as integer, print as character
- `f` - Float
- `d` - Integer, signed decimal

- `o` - Integer, octal
- `t` - Integer, binary
- `u` - Integer, unsigned decimal
- `x` - Integer, hexadecimal

The `<size>` parameter specifies how large each unit is. If you don't specify a `<size>`, **gdb** will simply re-use the last size value used. This value can be any of:

- `b` - 1 byte (*[b]yte*)
- `h` - 2 bytes (*[h]alfword*)
- `w` - 4 bytes (*[w]ord*)
- `g` - 8 bytes (*[g]iant word*)

```
(gdb) x/5xb 0x303030
```

B) General Info

- Info

The `info <arg>` command lists information about the argument specified in `<arg>`. The value for `<arg>` can be the following:

- `frame` - info on current stack frame (current/previous frame address, saved regsiters, function args, local vars)
- `args` - info on the arguments of the function of the current stack frame
- `locals` - info on local variables
- `stack` - info on everything on the stack (previous function calls along with their arguments)
- `registers` - info on the content of every register
- `breakpoints` - info on the location of every breakpoint (number, address, function)
- `functions` - info on every function signature (NOTE: Only works if program was initially compiled with `gcc -g`)

```
(gdb) info frame
```

- Backtrace/Where

The `backtrace` and `where` commands do the exact same thing as `info stack`. They print a stack trace listing every function and its arguments.

```
(gdb) backtrace
```

4) Manipulate the Program

- Set variables

The `set var <var_name>=<value>` command allows you to set the contents of the named variable `<var_name>` to `<value>`.

```
(gdb) set var a=10
```

- Force function return

The `return <expression>` command will force the currently executing function to immediately return with the value given by `<expression>`.

```
(gdb) return -1
```

5) If you're lost...

If you're ever lost in the middle of an intense debugging session and don't know/forgot where you are, use the commands below to view the context around what you're doing.

- Disassemble

The `disassemble <function>` command shows you the **assembly code** for the function `<function>`. If `<function>` is not specified, then `disassemble` will default to showing you the assembly code for the function that you are currently debugging.

```
(gdb) disassemble
```

- List

The `list` command shows you the **C source code** surrounding the line that you are currently debugging.

```
(gdb) list
```

6) Helpful Cheatsheets

A more comprehensive list of relevant **gdb** commands can be found at the following easy-to-read resources:

1. https://darkdust.net/files/GDB%20Cheat%20Sheet.pdf
2. https://cs.brown.edu/courses/cs033/docs/guides/gdb.pdf