

gdb Cheatsheet

Fall 2013

Contents

1	Introduction	1
2	Program Execution	1
3	Viewing Variables, Registers and Memory	4
4	More Information	5
4.1	Official Documentation	5
4.2	Tutorials	5
5	Tips	6

1 Introduction

This document contains several `gdb` commands which you will find useful throughout your x86- and C-programming career.

The commands contained within this document are by no means exhaustive; `gdb` contains many features which are not documented here. Consult the man pages (`man gdb`) or the internet if you require further information.

Throughout this document, commands shall be listed in the form

```
[c]ommand <required arg> (optional arg)
```

This is what the command does.

This is an example use of this command.

where the character(s) in brackets are the abbreviated version of the command.

2 Program Execution

- `[b]reak <function name or filename:line# or *memory address>`

Sets a breakpoint on either a function, a line given by a line number, or the instruction located at a particular address.

If you do not have access to the source code of a function and wish to set a breakpoint on a particular instruction, call `disassemble function_name` (where `function_name` is the name

of the procedure); this command will allow you to see the memory address of each instruction. See section 3 for further information.

```
(gdb) break main
Breakpoint 1 at 0x80488f6: file main.c, line 67.
```

- **[d]elete <breakpoint #>**

Removes the indicated breakpoint. To see breakpoint numbers, run `info break`, or `i b`.

```
(gdb) delete 4
```

- **[condition] <breakpoint #> <condition>**

Updates the breakpoint indicated by the given number so that execution of the program stops at that point only if `condition` is true. `condition` is expressed in C syntax, and can only use variables and functions that are available in the scope of the breakpoint location.

```
(gdb) break main
Breakpoint 1 at 0x80488f6: file main.c, line 48
(gdb) condition 1 argc <= 2 || !strcmp(argv[1], "jasmine")
```

- **[i]nfo (about)**

Lists information about the argument (`about`), or lists what possible arguments are if none is provided.

`about` can be one of the following¹:

- **[f]rame** - list information about the current stack frame, including the address of the current frame, the address of the previous frame, locations of saved registers, function arguments, and local variables.
- **[s]tack** - list the stack backtrace, showing what function calls have been made, and their arguments. You can also use the commands `backtrace` or `where` to do the same.
- **[r]egisters** - lists the contents of each register. `[all-r]egisters` lists even more registers.
- **[b]reak** - lists the number and address of each breakpoint, and what function the breakpoint is in.
- **[f]unctions** - lists all of the function signatures, if the program was compiled with the `gcc` flag `-g`. This is useful for setting breakpoints in functions.

- **[file] <filename of executable>**

Loads the specified file into `gdb`.

- **[r]un (arg1 arg2 ... argn)**

Runs the loaded executable program with program arguments `arg1 ... argn`.

- **[c]ontinue**

Resumes execution of a stopped program, stopping again at the next breakpoint.

¹Note that this list is *not* exclusive.

- [s]tep

Steps through a single line of code. Steps into function calls.

```
(gdb) break main
Breakpoint 1 at 0x8049377: file main.c, line 34.
(gdb) r
Breakpoint 1, main (argc=2, argv=0xbffff704) at main.c:34
35  int val = foo(argv[1]);
(gdb) s
foo (word=0xbffff8b3) at main.c:11
12  char bar = word[0];
```

- [s]tep[i]

Steps through a single x86 instruction. Steps into calls.

```
(gdb) 0x080484d5 in main ()
1: x/i $pc
=> 0x080484d5 <main + 113>:      call      0x80484ec<do_something>
(gdb) si
0x080484ec in do_something()
1: x/i $pc
=> 0x080484ec <main + 113>:      push     %ebp
```

- [n]ext

Steps through a single line of code. Steps over function calls.

```
(gdb) break main
Breakpoint 1 at 0x8049377: file main.c, line 34.
(gdb) r
Breakpoint 1, main (argc=2, argv=0xbffff704) at main.c:34
35  int val = foo(argv[1]);
(gdb) n
36  bar(val);
```

- [n]ext[i]

Steps through a single x86 instruction. Steps over calls.

```
(gdb) 0x080484d5 in main ()
1: x/i $pc
=> 0x080484d5 <main + 113>:      call      0x80484ec<do_something>
(gdb) ni
0x080484da in main ()
1: x/i $pc
=> 0x080484da <main + 113>:      mov     $0x0,%eax
```

- [k]ill

Kills the current debugging session.

- [b]ack[t]race

Prints a stack trace, listing each function and its arguments. This does the same thing as the commands `info stack` and `where`.

```
(gdb) bt
#0 fibonacci (n=1) at main.c:45
#1 fibonacci (n=2) at main.c:45
#2 fibonacci (n=3) at main.c:45
#3 main (argc=2, argv=0xbffff6e4) at main.c:34
```

- [where] Prints a stack trace, listing each function and its arguments. This is the same as the commands `info stack` and `backtrace`.
- [q]uit
Quits gdb.

3 Viewing Variables, Registers and Memory

- [p]rint <expression>

Prints the value which the indicated expression evaluates to. `expression` can contain variable names (from the current scope), memory addresses, registers, and constants as its operands to various operators. It is written in C syntax, which means that in addition to arithmetic operations, you can also use casting operations and dereferencing operations.

To access the value contained in a register, replace the `%` character prefix with `$`, e.g. `$eax` instead of `%eax`.

```
(gdb) print *(char *)($esp + $eax + my_ptr_array[13])
'e'
```

- [p]rint/x <expression>

Prints the value which the indicated expression evaluates to as a hexadecimal number. `expression` is evaluated the same way as it is in `print`.

```
(gdb) p/x my_var
$1 = 0x1b
```

- [x]/(number)(format)(unit_size) <address>

Examines the data located in memory at `address`.

`number` optionally indicates that several contiguous elements, beginning at `address`, should be examined. This is very useful for examining the contents of an array. By default, this argument is 1.

`format` indicates how data should be printed. In most cases, this is the same character that you would use in a call to `printf()`. One exception is the format `i`, which prints an instruction rather than a decimal integer.

`unit_size` indicates the size of the data to examine. It can be `[b]ytes`, `[h]alfwords` (2 bytes), `[w]ords`, or `[g]iant words`. By default, this is bytes, which is perfect for examining instructions.

A variation of this command is the `display` command. This command takes the same arguments, but repeats execution every time `gdb` waits for input. For example,

```
display/i $pc
```

would display the next instruction after each step.

```
(gdb) x/4x argv
0xbffff6e4:      0xbffff86b      0xbffff8b3      0xbffff8b6      0x00000000
(gdb) x/2c argv[1]
0xbffff86b      104 'h'          105 'i'
(gdb) x/3i foo
0x80485e6 <foo>:   push %ebp
0x80485e7 <foo+1>: mov  %esp, %ebp
0x80485e9 <foo+3>: push %ebx
```

- `[disassemble] <function name>`

Disassembles a function into assembly instructions, displaying the address, name, and operands of each instruction.

4 More Information

Below are some additional resources for all of your `gdb` needs.

4.1 Official Documentation

- Viewing the stack
- Running programs
- Stopping execution
- Viewing program source
- Viewing program data

4.2 Tutorials

- Using GNU's GDB Debugger
- Beej's Quick Guide to GDB

5 Tips

- If you edit your program while it is being run in `gdb`, open another terminal, recompile your program, and restart it in `gdb` by typing `run (args)`. `gdb` will load the new version of the program while maintaining all of your previous breakpoints.
- To view the next assembly instruction that will be executed, use the command
`display/i $pc`
- Type CTRL-Z to suspend execution of your program within `gdb`. You can then resume execution with the `[c]ontinue` command.