GDB and Debugging

Written by Chris Gregg and Nate Hardison, with modifications by Nick Troccoli and Lisa Yan

Click here (https://youtu.be/uhIt8YqtmuQ) for a walkthrough video.

gdb Reference Card (gdb_refcard.pdf).

In CS106A and CS106B, you may have used a graphical debugger; these debuggers were built into the program you used to write your code, and allowed you to set breakpoints, step through your code, and see variable values, among other features. In CS107, the debugger we are using is a separate program from your text editor, called `gdb` (the "GNU Debugger"). It is a command-line debugger, meaning that you interact with it on the command line using text-based commands. But it shares many similarities with debuggers you might have already used; it also allows you to set breakpoints, step through your code, and see variable values. We recommend familiarizing yourself with how to use `gdb` as soon as possible.

This page will list the basic `gdb` commands you will need to use as you progress through CS107. However, it takes practice to become proficient in using the tool, and `gdb` is a large program that has a tremendous number of features. See the bottom of the page for more resources to help you master `gdb`.

Compiling for `gdb`: `gcc` does not automatically put debugging information into the executable program, but our Makefiles all include the `-g -Og` flags that give `gdb` information about our programs so we can use the debugger efficiently.

Running `gdb`

`gdb` takes as its argument the executable file that you want to debug. This is not the .c file or the .o file, instead it is the name of the compiled program:

```
myth$ gdb myprogram
GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.3) 7.7.1
Copyright (C) 2014 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 square...done.
(gdb)
```

The `(gdb)` prompt is where you start typing your commands. Note that nothing has happened yet - your program has not started running to debug - `gdb` is simply awaiting further instructions.

Once you've got the (gdb) prompt, the `run` command (shorthand: `r`) starts the executable running. If the program you are debugging requires any command-line arguments, you specify them to the `run` command. To run `myprogram` with the arguments "hi" and "there", for instance, you would type the following:

```
(gdb) run hi there
Starting program: /cs107/myprogram hi there
```

This starts the program running. When the program stops, you'll get your (gdb) prompt back.

Breakpoints

Normally, your program only stops when it exits. Breakpoints allow you to pause your program's execution wherever you want, be it at a function call or a particular line of code, and examine the program state.

Before you start your program running, you want to set up your breakpoints. The `break` command (shorthand: `b`) allows you to do so.

To set a breakpoint at the beginning of the function named `main`:

```
(gdb) break main
Breakpoint 1 at 0x400a6e: file myprogram.c, line 44.
```

To set a breakpoint at line 47 in `myprogram.c`:

```
(gdb) break myprogram.c:47
Breakpoint 2 at 0x400a8c: file myprogram.c, line 47.
```

If there is only once source file, you do not need to include the filename.

Each breakpoint you create is assigned a sequentially increasing number (the first breakpoint is 1, the second 2, etc.).

If you want to delete a breakpoint, just use the `delete` command (shorthand: `d`) and specify the breakpoint number to delete.

To delete the breakpoint numbered 2:

```
(gdb) delete 2
```

If you lose track of your breakpoints, or you want to see their numbers again, the `info break` command lets you know the breakpoint numbers:

```
(gdb) info break
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000400a6e in main at myprogram.c:44
```

Finally, notice that it's much easier to remember function names than line numbers (and line numbers change from run to run when you're changing your code), so ideally you will set breakpoints by name. If you have decomposed your code into small, tight functions, setting breakpoints will be easy. On the other hand, wading through a 50-line function to find the right place for a breakpoint is unpleasant, so yet another reason to decompose your code cleanly from the start!

Extra: Conditional Breakpoints

You can set breakpoints to only trigger when certain conditions in your code are true. For instance, say you have the following loop in your code:

```
1    for (int i = 0; i < count; i++) {
2        ...
3    }
```

If you wanted to step through the code inside the loop just the last time the loop executed, with a normal loop you may have to skip over many program breaks before you get to the part you want to examine. However, `gdb` lets you add an optional condition (in C code syntax) for when the breakpoint should be stopped at:

```
(gdb) break 2 if i == count - 1
```

The format is `[BREAKPOINT] if [CONDITION]`. Now this breakpoint will only be hit the last time around the loop! You can even use local variables in your expression, as shown above with `i` and `count`. Experiment to see what other useful conditions you might use.

The following sections deal with things you can do when you're stopped at a breakpoint, or when you've encountered a segfault.

Backtrace

Easily one of the most immediately useful things about `gdb` is its ability to give you a backtrace (or a "stack trace") of your program's execution at any given point. This works especially well for locating things like crashes ("segfaults"). If a program named `reassemble` segfaults while running in GDB during execution of a function named `read_frag`, `gdb` will print the following information:

```
Program received signal SIGSEGV, Segmentation fault.
0x0000000000400ac1 in read_frag (fp=fp@entry=0x603010, nread=nread@entry=0) at reassemble.c:51
51      if (strlen(unusedptr) == MAX_FRAG_LEN)
```

Not only is this information vastly more useful than the terse "Segmentation fault" error that you get outside of `gdb`, you can use the `backtrace` command to get a full stack trace of the program's execution when the error occurred:

```
(gdb) backtrace
#0  0x0000000000400ac1 in read_frag (fp=fp@entry=0x603010, nread=nread@entry=0) at reassemble.c:51
#1  0x0000000000400bd7 in read_all_frags (fp=fp@entry=0x603010, arr=arr@entry=0x7fffffff4cb0,
maxfrags=maxfrags@entry=5000) at reassemble.c:69
#2  0x00000000004010ed in main (argc=<optimized out>, argv=<optimized out>) at reassemble.c:211
```

Each line represents a stack frame (ie. a function call). Frame #0 is where the error occurred, during the call to `read_frag`. The hexadecimal number 0x0000000000400ac1 is the address of the instruction that caused the segfault (don't worry if you don't understand this, we'll get to it later in the quarter). Finally, you see that the error occurred from the code in `reassemble.c`, on line 51. All of this is helpful information to go on if you're trying to debug the segfault.

Controlling Execution

`run` will start (or restart) the program from the beginning and continue execution until a breakpoint is hit or the program exits. `start` will start (or restart) the program and stop execution at the beginning of the `main` function.

Once stopped at a breakpoint, you have choices for how to resume execution. `continue` (shorthand: `c`) will resume execution until the next breakpoint is hit or the program exits. `finish` will run until the current function call completes and stop there. You can single-step through the C source using the `next` (shorthand: `n`) or `step` (shorthand: `s`) commands, both of which execute a line and stop. The difference between these two is that if the line to be executed is a function call, `next` executes the *entire function*, but `step` goes into the function implementation and stops at the first line.

Variables and Expressions

The `print` command

You're likely to want to check into the values of certain key variables at the time of the problem. The `print` command (shorthand: `p`) is perfect for this. To print out the value of variables such as `nread`, `fp` and `start`:

```
(gdb) print nread
$1 = 0
(gdb) print fp
$2 = (FILE *) 0x603010
(gdb) print start
$3 = 123 '{'
```

You can also use `print` to evaluate expressions, make function calls, reassign variables, and more.

Sets the first character of buffer to be 'Z':

```
(gdb) print buffer[0] = 'Z'
$4 = 90 'Z'
```

Print result of function call:

```
(gdb) print strlen(buffer)
$5 = 10
```

`print` format codes

Since `print` can pretty much evaluate any expression you throw at it, it's very useful for doing arithmetic and converting values between hex, decimal and binary. You can achieve this using `p/format` with different format codes. Check it out:

```
(gdb) p -7
$1 = -7
(gdb) p/t -7
$2 = 11111111111111111111111111111001
(gdb) p/x -7
$3 = 0xfffffff9
(gdb) p/u -7
$4 = 4294967289
(gdb) p -7 + 4
$5 = -3
(gdb) p (unsigned) (-7 + 4)
$6 = 4294967293
```

Format letters for `print` :

- `x` for hex
- `t` for binary
- `c` for char
- `u` for unsigned decimal
- `d` for signed decimal

The `info` command

The following commands are handy for quickly printing out a group of variables in a particular function:

`info args` prints out the arguments (parameters) to the current function you're in:

```
(gdb) info args
fp = 0x603010
nread = 0
```

`info locals` prints out the local variables of the current function:

```
(gdb) info locals
start = 123 '{'
end = 125 '}'
nscanned = 3
```

Stack Frames

If you're stopped at a breakpoint or at an error, you may also want to examine the state of stack frames further back in the calling sequence. You can use the up and down commands for this.

`up` moves you up one stack frame (e.g. from a function to its caller)

```
(gdb) up
#1  0x0000000000400bd7 in read_all_frags (fp=fp@entry=0x603010, arr=arr@entry=0x7fffffff4cb0,
maxfrags=maxfrags@entry=5000) at reassemble.c:69
69          char *frag = read_frag(fp, i);
```

`down` moves you down one stack frame (e.g. from the function to its callee)

```
(gdb) down
#0  0x0000000000400ac1 in read_frag (fp=fp@entry=0x603010, nread=nread@entry=0) at reassemble.c:51
51      if (strlen(unusedptr) == MAX_FRAG_LEN)
```

The commands above are really helpful if you're stuck at a segfault and want to know the arguments and local vars of the faulting function's caller (or that function's caller, etc.).

Examining Memory

`examine`

The examine command, `x` (**click here for documentation (ftp://ftp.gnu.org/old-gnu/Manuals/gdb/html_chapter/gdb_9.html#SEC56)**) is a helpful command to **examine the contents of memory independent of the type of data at a memory location**. It's like `print` , but for generic memory rather than a specific type of variable. `x` instead prints out a certain number of bytes starting at a given address. To exzamine out the memory a pointer `ptr` points to:

```
(gdb) x/8bx ptr
0x7fffffffe870: 0x05    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

In other words, `ptr` stores the address `0x7fffffffe870` and has the following 8 bytes of information. The optional parameters after the slash specify what you would like to print:

- The first one (e.g. `8` or `2`) lets you specify how many you would like to examine.

- The second (e.g. `b`, `h`, `w`, or `g`) specifies whether you would like to print out bytes, half-words (2 bytes), words (4 bytes), giant words (8 bytes), etc.
- The third (e.g. `x`) specifies how you would like to print them out (e.g. `x` for hex, `d` for decimal).

Printing arrays

For example, `gdb` fully knows the type and number of elements in stack arrays in the context of a function for which they are declared, but it cannot automatically do the same in other contexts (for which the array of elements decays to a pointer to the first element). To print out arrays in other contexts:

```
(gdb) p argv[0]@argc
```

will print out the entire contents of the `argv` array. The syntax to learn is `p ELEM@COUNT`. Supposing you have a `void *ptr` that you know is the base address of an array of `int` elements, you can typecast as needed:

```
(gdb) p *(int *)ptr@2
```

will print out the first two elements as ints. Note the dereference; this syntax takes elements, and not pointers to elements.

Useful Tips

- If you're using a Makefile, you can recompile from within `gdb` so that you don't have to exit and lose all your breakpoints. Just type `make` at the (gdb) prompt, and it will rebuild the executable. The next time you `run`, it will reload the updated executable and reset your existing breakpoints.
- Use `gdb` inside Emacs! It's just another reason why Emacs is really cool. Use "Ctrl-x 3" to split your Emacs window in half, and then use "Esc-x gdb" or "Alt-x gdb" to start up `gdb` in your new window. If you are physically at one of the UNIX machines, or if you have X11 forwarding enabled, your breakpoints and current line show up in the window margin of your source code file.
- If you simply type the enter key, the last command will be re-run. This is nice if you are stepping through a program line-by-line: you can use the `next` command once, and then hitting the enter key will re-run the `next` command again without you having to type the command.

Debugging Strategies

Both learning GDB commands and how to apply GDB to fix bugs are essential. Check out our debugging guide for more advice.

View Debugging Guide (/class/cs107/resources/debugging)

`gdb` Commands Reference

Here is a full list of the commands you'll want to be familiar with. For even more, check out the reference card at the top of this page.

Command	Abbreviation	Description
`help [command]`	`h [command]`	Provides help (information) about a particular command or keyword.
`apropos [command]`	`appr [command]`	Same as `help`
`info [cmd]`	`i [cmd]`	Provides information about your program, such as the breakpoints (`info breakpoints`, local variables (`info locals`), parameters (`info args`), breakpoint numbers (`info break`), etc.
`run [args]`	`r [args]`	The `run` command runs your program. You can enter command line arguments after the command, if your program requires them. If you are already running your program and you want to re-run it, you should answer `y` to the prompt that says, "The program being debugged has been started already. // Start it from the beginning? (y or n)"
`list`	`l`	The `list` command lists your program code either from the beginning, or with a range around where you are currently stopped.
`next`	`n`	The `next` command steps to the next program line and *completely runs functions*. This is important: if you have a function (even one you didn't write) and you use `next`, the function will run to completion, and will stop at the next line after the function (unless there is a breakpoint inside the function).
`step`	`s`	The `step` command is similar to `next`, but it will step into functions. This means that it will attempt to go to the first line in a function if there is a function called on the current line. Importantly, it will also take you into functions you didn't write (such as `printf`), which can be annoying (you should use `next` instead). If you do accidentally step into a function, you can use the `finish` command to finish the function immediately and go back to the next line after the function.
`continue`	`c`	This will continue running the program until the next breakpoint or until the program ends.
`print [x]`	`p [x]`	This very important command lets you see the value of a variable [x] when you are stopped in a program. If you see the error "No symbol xxxx in current context", or the error, "optimized out", you probably aren't in a place in the program where you can read the variable.
`break [x]`	`b [x]`	This will put a breakpoint in the program at a specified function name or a particular line number. If you have multiple files, you should use `file:lineNum` when specifying the line number (e.g. `source.c:57`).
`clear [x]`		Removes the breakpoint at a specified line number or at the start of the specified function.

Command	Abbreviation	Description
`delete [x]`		Removes the breakpoint with the given number. If the number is omitted, deletes all breakpoints after confirming.
`backtrace`	`bt`	This will print a stack trace to let you know where in the program you are currently stopped. This is a useful command if your program has a segmentation fault: the `backtrace` command will tell you the last place in your program that had the problem (sometimes you need to look at the entire stack trace to go back to the last line your program tried to execute).
`up` and `down`		These commands allow you to go up and down the stack trace. For instance, if you are inside a function and want to see the status of variables from the calling function, you can use `up` to place the program into the calling function, and then use `p variable` to look at the state of the program in that function. You would then use `down` to go back to the function that was called.
`finish`		Runs a function until it completes, which is helpful if you accidentally step into a function.
`disassemble`	`disas`	Disassembles your program into assembly language. We will be discussing this in depth in cs107.
`ctrl-x`, `ctrl-a`		Go into or leave **"TUI" (https://sourceware.org/gdb/onlinedocs/gdb/TUI.html#TUI)** mode: `gdb` has a mode that shows you source code, or assembly output in a manner that allows you to scroll up and down. It is useful but sometimes can be a bit buggy. You will want to use the `ctrl-l` command to refresh the display.
`quit`	`q`	Quit `gdb`

Example `gdb` Session

Below, we've included a program and a sample usage of GDB - lines that start with `(gdb) # some text` are comments. Also see **this video (https://youtu.be/uhIt8YqtmuQ)** for a walkthrough demonstration of `gdb` .

We will be looking at this simple program below:

```
#include<stdlib.h>
#include<stdio.h>

int square(int x);

int main(int argc, char *argv[]) {
    printf("This program will square an integer.\n");

    // the program should have one number as an argument
    if (argc != 2) {
        printf("Usage:\n\t./square number\n");
        return 0;
    }

    // the first argument after the filename
    int numToSquare = atoi(argv[1]);

    int squaredNum = square(numToSquare);

    printf("%d squared is %d\n",numToSquare,squaredNum);

    return 0;
}


int square(int x) {
    int sq = x * x;
    return sq;
}
```

The sample GDB run output can be found **here (gdb_sample)**.

Viewing assembly code with `tui`

Post-midterm, you will be viewing assembly code, often in combination with C code. The `tui` (text user interface) splits your session into panes for simultaneously viewing the C source, assembly translation, and/or current register state.

Opening, refreshing, and closing `tui`

Tui mode is great for tracing execution and observing what is happening with code/registers as you `stepi` . Occasionally, tui trips over itself and garbles the display. The gdb command `refresh` sometimes works to clean it up, or you can try `ctrl-l` . If things get really out of hand, `ctrl-x a` will exit tui mode and return you to ordinary non-graphical gdb.

Command	Description
`layout split`	Starts `tui` mode with C file, assembly file, and gdb
`refresh` , `ctrl-l`	Refreshes the `tui` display.
`ctrl-x a`	Exits `tui` mode and return to ordinary non-graphical gdb.

Advanced use of `tui`

You may want to customize your `tui` layout---say, to only view the assembly code, or to view the assembly and register state simultaneously. In addition, when you open `tui` with `layout split`, by default your arrow keys map to the C source file, meaning that you can't use arrow keys as normal to navigate your gdb commands.

Command	Description
`focus cmd`	Changes active window to gdb command window to allow use arrow keys.
`layout <argument>`	Specifies which pane you want (`src`, `asm`, `regs`, `split`, or `next`).
`focus <argument>`	Uses same arguments as `layout` to change active window.
`winheight <name> + <count>`	Change the height of the window **name** by **count** lines. To reduce height by **count** lines, supply argument `-<count>`.
`ctrl-x 1`	Opens one window, default is `src` (C file).

Suppose you want to view assembly, registers, and your gdb command window at the same time, and you want your arrow keys to work in gdb:

```
(gdb) layout regs
(gdb) focus cmd
```

More Resources

If you're interested in even more information about `gdb`, check out the following resources:

- This CS107 **gdb Reference Card (gdb_refcard.pdf)**
- Section 3 of this **Stanford Unix Programming Tools (http://cslibrary.stanford.edu/107/UnixProgrammingTools.pdf)** document
- The **full gdb manual (http://www.gnu.org/software/gdb/)** (from GNU)
- This **extensive gdb guide (https://sourceware.org/gdb/current/onlinedocs/gdb)**
- Two gdb articles written by Julie Zelenski, another Stanford Lecturer, for a programming journal: **Breakpoint Tricks (gdb_coredump1.pdf)** and **GDB's Greatest Hits (gdb_coredump2.pdf)**

Frequently-Asked Questions

When I run my program under gdb, it complains about missing symbols. What is wrong?

```
gdb myprogram
Reading symbols from myprogram...(no debugging symbols found)...done.
(gdb)
```

This means the program was not compiled with debugging information. Without this information, gdb won't have full symbols for setting breakpoints or printing values or other program tracing features. There's a flag you need to pass to gcc to build debugging info into the executable. If you're using raw gcc, add the -g flag to your command. If you're using a Makefile, make sure the CFLAGS line includes the -g flag.

When I view my code from within in gdb, it warns that the source file is more recent. What does this mean?

```
(gdb) list
warning: Source file is more recent than executable.
```

This means that you have edited one or more of your .c source files, but have not recompiled those changes. The program being executed will not match the edited source code and gdb's efforts to try to match up the two will be hopelessly confused. You can quit out of gdb, make, and then restart gdb, or even more conveniently, `make` from with gdb will rebuild the program and reload it at next run, all without leaving gdb.

I step into a library function and gdb complains about a missing file. What is this and what should I do about it?

```
(gdb) s
_IO_new_fopen (filename=0xffffdc9f "samples/input", mode=0x804939a "r") at iofopen.c:102
102 iofopen.c: No such file or directory.
```

The `step` command usually executes the next single line of C code. If that line makes a function call, step will advance into that function and allow you trace inside the call. However, if the function is a library function, gdb will not be able to display/trace inside it because the necessary source files are not available on the system. Thus, if asked to step into a library call, gdb responds with this harmless complaint about "No such file". At that point, you can use `finish` to continue through the current function. As alternative to `step`, the `next` command will execute the entirety of the next line, completing all function calls rather than attempting to step into them.

My program crashes within a library function. It's not my fault the library is broken! What can I do?

```
Program received signal SIGSEGV, Segmentation fault.
__strcmp_ssse3 () at ../sysdeps/i386/i686/multiarch/strcmp-ssse3.S:232
232 ../sysdeps/i386/i686/multiarch/strcmp-ssse3.S: No such file or directory.
```

The example crash shown above is occurring during a call to `strcmp`, a function from the standard library. The arguments to `strcmp` are expected to be valid `char*` s. If given an invalid address, the function will crash trying to read from that location. The library function does not have a bug, the mistake is that you are passing an invalid argument; look at your call to `strcmp` to resolve your bug. The complaint about missing files (discussed above) is a harmless warning you can safety ignore. On the other hand, the bug in your use of the library function needs to be investigated further! :-)

When I start gdb, it gives a long warning about auto-loading being declined. What's wrong?

```
Reading symbols from bomb...(no debugging symbols found)...done.
warning: File "/afs/ir.stanford.edu/users/z/e/zelenski/a5/.gdbinit" auto-loading has been declined by your `auto-
load safe-path' set to "$debugdir:$datadir/auto-load".
To enable execution of this file add
    add-auto-load-safe-path /afs/ir.stanford.edu/users/z/e/zelenski/a5/.gdbinit
line to your configuration file "/afs/ir/users/z/e/zelenski/.gdbinit".
... blah blah blah ...
```

A .gdbinit file is used to set a startup sequence for gdb. However, for security reasons, loading from a local .gdbinit is disabled by default. If there is a .gdbinit in the current directory and you have not configured gdb to allow loading it, gdb complains to alert you that this .gdbinit file is being ignored. To enable loading, you must edit your personal gdb configuration file to change your auto-load setting. Your personal gdb configuration is `~/.gdbinit` (a hidden file in your home directory). A .gdbinit file is a plain text file you can edit with your favorite editor. If file doesn't yet exist, you will need to create it. The line you need to add is `set auto-load safe-path /`. Alternatively, you can copy and paste the command below to append the proper setting to your personal configuration file, creating the file if it doesn't already exists. You will need to make this configuration change only once.

```
bash -c 'echo set auto-load safe-path / >> ~/.gdbinit'
```

You can check your current configuration by searching your personal configuration file for the setting. See command below and expected response:

```
myth> grep auto-load ~/.gdbinit
set auto-load safe-path /
```

Once your personal configuration is appropriately set, there will be no further complaints from gdb about declining auto-load and it will load any local .gdbinit file on start.

When my program finishes, gdb prints a message calling my program "inferior". What have I done to offend gdb?

```
[Inferior 1 (process 25178) exited normally]
                  or
[Inferior 1 (process 25609) exited with code 01]
```

Don't take it personally, gdb runs your program as a sub-process which it terms the "inferior". The message indicates the program has run to completion and exited in a controlled fashion-- there was no segmentation fault, abort, hang, or other catastrophic termination condition.

When I run gdb without a program, some things (like sizeof and typecasts) behave differently. What's happening?

```
myth$ gdb
(gdb) p sizeof(long)
$1 = 4
(gdb) p/x (long)-1
$2 = 0xffffffff
(gdb)
```

By default, gdb determines your CPU architecture based on the program you're debugging. If you don't specify a program when starting gdb, it defaults to assuming a 32-bit system, rather than the 64-bit system that all of the programs we write will use. Starting gdb on one of your CS107 programs (any one will do) will cause gdb to use the proper architecture. You can also manually put gdb into 64-bit mode with the following command:

```
set architecture i386:x86-64
```

After that, the above commands should now print correctly:

```
myth$ gdb
(gdb) set architecture i386:x86-64
The target architecture is assumed to be i386:x86-64
(gdb) p sizeof(long)
$1 = 8
(gdb) p/x (long)-1
$2 = 0xffffffffffffffff
(gdb)
```
