GDB and Debugging

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

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

In CS106A and CS106B, you may have used a graphical debugger, like the ones built into Eclipse and Qt Creator. 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`.

Getting Started

Before you start using `gdb`, you'll want to configure it to use the CS107 default preferences; this sets up the debugger to know things like our work will be in 64-bit systems. To do this, execute the following command after logging into `myth`:

```
wget https://web.stanford.edu/class/archive/cs/cs107/cs107.1194/resources/sample_gdbinit -O ~/.gdbinit
```

`gdb` looks for a special `.gdbinit` file on your system to know what preferences you would like, and this command downloads our pre-made `.gdbinit` file and puts it on your system. If you don't do this, some behavior may not match guides and examples in CS107.

Compiling for `gdb`: `gcc` does not automatically put debugging information into the executable program, but our Makefiles (make) 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:

```
$ 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 hello world
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 stop 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!

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

Examining Program State

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 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 reassem
ble.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 rea
ssemble.c:51
#1  0x0000000000400bd7 in read_all_frags (fp=fp@entry=0x603010, arr=arr@entry=0x7fffff
ff4cb0, maxfrags=maxfrags@entry=5000) at reassemble.c:69
#2  0x00000000004010ed in main (argc=<optimized out>, argv=<optimized out>) at reassem
ble.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.

Variables and Expressions

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 'a':

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

Print result of function call:

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

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

`info args` prints out the arguments 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=0x7fffff
ff4cb0, 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 rea
ssemble.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.).

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.

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

Along with `gdb`, it's important to use good debugging strategies when working on your programs. Many students have up to now done all their debugging via print statements. That works for simple cases, but becomes unwieldy as programs get larger and have more complex interactions. Now is the time to invest in mastering the powerful tools provided by a debugger like `gdb`, and using a careful and systematic debugging approach:

1. **Observe the bug**. If you never see the bug, you'll likely never fix it. Another reason you want comprehensive testing (/class/archive/cs/cs107/cs107.1194/testing.html)!
2. **Create a reproducible input**. Creating a trivial input that reliably induces the failure is a huge help.
3. **Narrow the search space**. Studying the entire program or tracing the execution line-by-line is generally not feasible. Some suggestions for how to narrow down your focus :
    - Start where your intuition believes is the likely culprit, such as a function that recently changed or one you find suspicious.
    - Use binary search to dissect. Set a breakpoint at the midpoint and poke around to determine whether the program state is already corrupt (which indicates the problem is in the front half) or looks good (so you need to focus your attention on the back half). Repeat to further narrow down.
    - Run under Valgrind (valgrind) to identify the root cause of any lurking memory errors.
4. **Analyze**. With only a small amount of code under scrutiny, execution tracing becomes feasible. Use `gdb` to see what the facts (values of variables and flow of control) are telling you. Drawing pictures may help.

5. **Devise and run experiments**. Make inferences about the root cause and run experiments to validate your hypothesis. Iterate until you identify the root cause.
6. **Modify code to squash bug**. The fix should be validated by your experiments and passing the original failed test case. You should be able explain the series of facts, tests, and deductions which match the observed symptom to the root cause and the corrected code.

Do not change your code haphazardly. This is like a scientist who changes more than one variable at a time. It makes the observed behavior much more difficult to interpret, and tends to introduce new bugs. That said, if you find buggy code, even if it is not obviously related to the bug you are tracking, you still might want to make a detour to fix it, using a reproducible input to trigger that bug and validate its fix. That bug might be related to or obscuring the original bug and it's good to remove any source of potential interface.

`gdb` Commands Reference

Here is a full list of the commands you'll want to be familiar with. For even more, check out this `gdb` Quick Reference (/class/archive/cs/cs107/cs107.1194/resources/gdb_refcard.pdf) document.

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.

Command	Abbreviation	Description
`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.
`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 what 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

The following capture is a typical `gdb` session. 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;
}
```

Here is a sample `gdb` run:

```
(gdb) # the "run" command runs the program, and we need to remember to put a command l
ine argument if we need one. Let's try it without an argument:
(gdb) run
Starting program: /afs/.ir.stanford.edu/users/c/g/cgregg/cs107/assignments/assign5/squ
are
This program will square an integer.
Usage:
    ./square number
[Inferior 1 (process 14146) exited normally]
(gdb)
(gdb) # the program caught that we tried to run without a number.
(gdb) # let's run it again with 25 as the argument:
(gdb) run 25
Starting program: /afs/.ir.stanford.edu/users/c/g/cgregg/cs107/assignments/assign5/squ
are 25
This program will square an integer.
25 squared is 625
[Inferior 1 (process 14132) exited normally]
(gdb) # we successfully ran the program. To look at the program, you can
(gdb) # type 'list' or just 'l':
(gdb) l
1   #include<stdlib.h>
2   #include<stdio.h>
3
4   int square(int x);
5
6   int main(int argc, char *argv[]) {
7       printf("This program will square an integer.\n");
8
9       // the program should have one number as an argument
10      if (argc != 2) {
(gdb) # we type the return key to see the next lines, or just 'l' again:
(gdb) l
11          printf("Usage:\n\t./square number\n");
12          return 0;
13      }
14
15      // the first argument after the filename
16      int numToSquare = atoi(argv[1]);
17
18      int squaredNum = square(numToSquare);
19
20      printf("%d squared is %d\n",numToSquare,squaredNum);
(gdb)
21
22      return 0;
23  }
24
25
26  int square(int x) {
27      int sq = x * x;
28      return sq;
29  }
30
(gdb) # when debugging, we often want to stop at a particular place
(gdb) # in our code. We can set breakpoints at a line, or at a function.
(gdb) break main
Breakpoint 1 at 0x4005f3: file square.c, line 6.
(gdb) break 20
Breakpoint 2 at 0x40064e: file square.c, line 20.
```

```
(gdb) # now there are two breakpoints in the code. When we run the program
(gdb) # again, it will stop on the first line:
(gdb) run 25
Starting program: /afs/.ir.stanford.edu/users/c/g/cgregg/cs107/assignments/assign5/squ
are 25

Breakpoint 1, main (argc=2, argv=0x7fffffffeab8) at square.c:6
6   int main(int argc, char *argv[]) {
(gdb) # now we are stopped. A nice command is the "where" command to tell
(gdb) # you exactly where you stopped:
(gdb) where
#0  main (argc=2, argv=0x7fffffffeab8) at square.c:6
(gdb) # if we type "l" again, we will get a list surrounding that line:
(gdb) l
1   #include<stdlib.h>
2   #include<stdio.h>
3
4   int square(int x);
5
6   int main(int argc, char *argv[]) {
7       printf("This program will square an integer.\n");
8
9       // the program should have one number as an argument
10      if (argc != 2) {
(gdb) # there are three commands that will continue the program.
(gdb) # the first command is "next" (or just "n"), which is the
(gdb) # "step over" command. It will run one line in your program, and
(gdb) # it will step over functions -- i.e., it will run a function
(gdb) # and not go into it.
(gdb) n
7       printf("This program will square an integer.\n");
(gdb) n
This program will square an integer.
10      if (argc != 2) {
(gdb) # now we are on line 10. We can use a different command, "step" to
(gdb) # go forward, as well, but "step" will go into a function if there
(gdb) # is a function to go into:
(gdb) s
11          printf("Usage:\n\t./square number\n");
(gdb) s
printf (__fmt=<optimized out>)
    at /usr/include/x86_64-linux-gnu/bits/stdio2.h:104
104   return __printf_chk (__USE_FORTIFY_LEVEL - 1, __fmt, __va_arg_pack ());
(gdb) # oops! We stepped into the "printf" function! We can get back to our
(gdb) # program by typing the "finish" command:
(gdb) finish
Run till exit from #0  printf (__fmt=<optimized out>)
    at /usr/include/x86_64-linux-gnu/bits/stdio2.h:104
Usage:
    ./square number
12          return 0;
(gdb) # The final way to continue is to type "continue", which runs the
(gdb) # program to completion, or to the next breakpoint.
(gdb) continue
Continuing.

Breakpoint 2, main (argc=<optimized out>, argv=0x7fffffffeab8) at square.c:23
23  }
(gdb) continue
Continuing.
[Inferior 1 (process 14154) exited normally]
```

```
(gdb) # let's run it again with the proper argument
(gdb) run 25
Starting program: /afs/.ir.stanford.edu/users/c/g/cgregg/cs107/assignments/assign5/squ
are 25

Breakpoint 1, main (argc=2, argv=0x7fffffffeab8) at square.c:6
6   int main(int argc, char *argv[]) {
(gdb) n
7       printf("This program will square an integer.\n");
(gdb) n
This program will square an integer.
10      if (argc != 2) {
(gdb) n
16      int numToSquare = atoi(argv[1]);
(gdb) n
18      int squaredNum = square(numToSquare);
(gdb) # now if we want to step into the square() function, we can, with s
(gdb) s
square (x=25) at square.c:27
27      int sq = x * x;
(gdb) where
#0  square (x=25) at square.c:27
#1  0x0000000000400636 in main (argc=<optimized out>, argv=0x7fffffffeab8)
    at square.c:18
(gdb) # the where command gives us a stack trace, and we can see that
(gdb) # we are inside the square function, on line 27.
(gdb) # We can take a look at variables here, too, with the 'p' command:
(gdb) p x
$1 = 25
(gdb) # "p x" says "print x"
(gdb) # let's list the area around where we stopped
(gdb) l
22      return 0;
23  }
24
25
26  int square(int x) {
27      int sq = x * x;
28      return sq;
29  }
30
31
(gdb) where
#0  square (x=25) at square.c:27
#1  0x0000000000400636 in main (argc=<optimized out>, argv=0x7fffffffeab8)
    at square.c:18
(gdb) # we can go one step forward using next or step
(gdb) n
29  }
(gdb) p sq
$2 = 625
(gdb) # we can look at the value of sq
(gdb) finish
Run till exit from #0  square (x=25) at square.c:29
main (argc=<optimized out>, argv=0x7fffffffeab8) at square.c:20
20      printf("%d squared is %d\n",numToSquare,squaredNum);
Value returned is $3 = 625
(gdb) l
15      // the first argument after the filename
16      int numToSquare = atoi(argv[1]);
17
```

```
18      int squaredNum = square(numToSquare);
19
20      printf("%d squared is %d\n",numToSquare,squaredNum);
21
22      return 0;
23  }
24
(gdb) n
25 squared is 625

Breakpoint 2, main (argc=<optimized out>, argv=0x7fffffffeab8) at square.c:23
23  }
(gdb) # we just called printf to print the result
(gdb) cont
Continuing.
[Inferior 1 (process 15818) exited normally]
(gdb)
```

More Resources

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

- This gdb Reference Card (/class/archive/cs/cs107/cs107.1194/resources/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 (http://sourceware.org/gdb/current/onlinedocs/gdb/index.html)
- Two gdb articles written by Julie Zelenski, another Stanford Lecturer, for a programming journal: Breakpoint Tricks (/class/archive/cs/cs107/cs107.1194/resources/gdb_coredump1.pdf) and GDB's Greatest Hits (/class/archive/cs/cs107/cs107.1194/resources/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:1
02
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 b
een 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)
```