Debugging with GDB

A debugger lets you pause a program, examine and change variables, and step through a few hours to learn one so you can avoid dozens of hours of frustration in the future. T guide, more information here:

- Official Page – Documentation
- Sample session – Short Tutorial – Long Tutorial

Getting Started: Starting and Stopping

- `gcc -g myprogram.c`
  - Compiles myprogram.c with the debugging option (-g). You still get an a.out, but it contains debugging information that lets you use variables and function names inside GDB, rather than raw memory locations (not fun).
- `gdb a.out`
  - Opens GDB with file a.out, but does not run the program. You'll see a prompt (gdb) – all examples are from this prompt.
- `r`
- `r arg1 arg2`
- `r < file1`
  - Three ways to run "a.out", loaded previously. You can run it directly (r), pass arguments (r arg1 arg2), or feed in a file. You will usually set breakpoints before running.
- `help`
- `h breakpoints`
  - List help topics (help) or get help on a specific topic (h breakpoints). GDB is well-documented.
- `q` – Quit GDB

Stepping Through Code

Stepping lets you trace the path of your program, and zero in on the code that is crashing or returning invalid input.

- `l`
- `l 50`
- `l myfunction`
  - List 10 lines of source code for current line (l), a specific line (l 50), or for a function (l myfunction).
- `next`
  - Run program until next line, then pause. If the current line is a function, execute the entire function, then pause. Next is good for walking through your code quickly.
- `step`
  - Run the next instruction, not line. If the current instructions is setting a variable, it is the same as next. If it's a function, it will jump into the function, execute the first statement,

then pause. Step is good for diving into the details of your code.

- `finish`
  - Finish executing the current function, then pause (also called step out). Useful if you accidentally stepped into a function.

Breakpoints and Watchpoints

Breakpoints are one of the keys to debugging. They pause (break) a program when it reaches a certain location. You can examine and change variables, then resume execution. This is helpful when seeing why certain inputs fail, or testing inputs.

- `break 45`
- `break myfunction`
  - Set a breakpoint at line 45, or at myfunction. The program will pause when it reaches the breakpoint.
- `watch x == 3`
  - Set a watchpoint, which pauses the program when a condition changes (when x == 3 changes). Watchpoints are great for certain inputs (myPtr != NULL) without having to break on *every* function call.
- `continue`
  - Resume execution after being paused by a breakpoint/watchpoint. The program will continue until it hits the next breakpoint/watchpoint.
- `delete N`
  - Delete breakpoint N (breakpoints are numbered when created).

Setting Variables and Calling Functions

Viewing and changing variables at run-time is a huge part of debugging. Try giving functions invalid inputs or running other test cases to find the root of problems. Typically, you will view/set variables when the program is paused.

- `print x`
  - Print current value of variable x. Being able to use the original variable names is why the (-g) flag is needed; programs compiled regularly have this information removed.
- `set x = 3`
- `set x = y`
  - Set x to a set value (3) or to another variable (y)
- `call myfunction()`
- `call myotherfunction(x)`
- `call strlen(mystring)`
  - Call user-defined or system functions. This is extremely useful, but beware calling buggy functions.
- `display x`
- `undisplay x`
  - Constantly display value of variable x, which is shown after every step or pause. Useful if you are constantly checking for a certain value. Use undisplay to remove the constant display.

Backtrace and Changing Frames

The *stack* is a list of the current function calls – it shows you where you are in the program. A *frame* stores the details of a single function call, such as the arguments.

- `bt`
  - Backtrace, aka print the current function stack to show where you are in the current program. If `main` calls function `a()`, which calls `b()`, which calls `c()`, the backtrace is

```
c <= current location
b
a
main
```

- `up`
- `down`
  - Move to the next frame up or down in the function stack. If you are in `c`, you can move to `b` or `a` to examine local variables.
- `return`
  - Return from current function.

Crashes and Core Dumps

A "core dump" is a snapshot of memory at the instant the program crashed, typically saved in a file called "core". GDB can read the core dump and give you the line number of the crash, the arguments that were passed, and more. This is very helpful, but remember to compile with (-g) or the core dump will be difficult to debug.

- `gdb myprogram core`
  - Debug myprogram with "core" as the core dump file.
- `bt`
  - Print the backtrace (function stack) at the point of the crash. Examine variables using the techniques above.

Handling Signals

Signals are messages thrown after certain events, such as a timer or error. GDB may pause when it encounters a signal; you may wish to ignore them instead.

- `handle [signalname] [action]`
- `handle SIGUSR1 nostop`
- `handle SIGUSR1 noprint`
- `handle SIGUSR1 ignore`
  - Tell GDB to ignore a certain signal (`SIGUSR1`) when it occurs. There are varying levels of ignoring.

Integration with Emacs

The Emacs text editor integrates well with GDB. Debugging directly inside the editor is great because you can see an entire screen of code at a time. Use `M-x gdb` to start a new window with GDB and learn more here.

Tips

- I often prefer watchpoints to breakpoints. Rather than breaking on every loop and checking a variable, set a watchpoint for when the variable gets to the value you need (i == 25, ptr != null, etc.).
- `printf` works well for tracing. But wrap `printf` in a `log` function for flexibility.
- Try passing a log level with your message (1 is most important, 3 is least). You can tweak your log function to send email on critical errors, log to a file, etc.
- Code speaks, so here it is. Use `#define LOG_LEVEL LOG_WARN` to display warnings and above. Use `#define LOG_LEVEL LOG_NONE` to turn off debugging.

```
#include <stdio.h>

#define LOG_NONE 0
#define LOG_ERROR 1
#define LOG_WARN 2
#define LOG_INFO 3
#define LOG_LEVEL LOG_WARN


// shows msg if allowed by LOG_LEVEL
int log(char *msg, int level){
  if (LOG_LEVEL >= level){
    printf("LOG %d: %s\n", level, msg);
    // could also log to file
  }

  return 0;
}

int main(int argc, char** argv){
  printf("Hi there!\n");

  log("Really bad error!", LOG_ERROR);
  log("Warning, not so serious.", LOG_WARN);
  log("Just some info, not that important.", LOG_INFO);

  return 0;
}
```

- Spend the time to learn GDB (or another debugging tool)! I know, it's like telling people to eat