GDB Information

Chris Smith

SDSMT ACM/LUG

January 30, 2020

Overview

- What is gdb?
- Compiling with debugging Flags
- Running gdb
- Interactive Shell
- Running the Program
- Breakpoints
- Stepping Through
- Viewing Variables
- More Commands
- Conditional Breakpoints

What is gdb?

- The GNU Debugger
- Debugs C/C++, Go, Objective-C, and more
- Allows for viewing what the program is executing
- www.gnu.org/softwoare/gdb/documentation

Compiling

Typically programs are compiled by running:

```
$ g++ [flags] prog1.cpp -o prog1
```

To enable debugging flags which gdb needs, add the -g flag like so:

```
$ g++ [flags] -g prog1.cpp -o prog1
```

Running gdb

To start gdb run the following:

```
$ gdb
```

To load a file on start up just add the file name as a commandline argument.
After the above command the following prompt should be shown:

```
(gdb)
```

Interactive Shell

gdb is like the interactive shells you get when first logging into Linux.

- Similar to bash, python, and lisp
- History of commands
- Tab completion
- Execution of shell commands with the **shell** command

If there ever is any confusion, the built in help menu gives a description of what a command does and what arguments it takes.

Running the program

To run the the program:

```
(gdb) run
```

If there are no serious problems it should execute normally.

```
[inferior 1 (process 1234) exited normally]
```

If there is a problem it will return information regarding where it crashed, what function, and the function variables that caused the error.

```
Program received signal SIGSEGV, Segmentation fault.
0x0000000000400633 in main () at fib.cpp:27
27 x[c] = next;
```

Debugging

The program is running successfully without any segmentation faults. However, the result isn't what was expected.

Downside to using console output:

- Input and output is slow (cout, cin, printf, and scanf)
- Information can scroll by too fast
- Memory allocation
- May not be thread safe
- Hinders timing sensitive programs

Breakpoints

Stepping through the code can be a more efficient approach since a segmentation fault could be happening in one function but the root of the cause was a couple of function calls ago.

Breakpoints in gdb are similar to what is used in Visual Studio. They allow for you to stop the program from executing once it reaches a designated spot in the code.

Breakpoints

To use a break just enter the command **break** followed by the file and line number pair like so:

```
(gdb) break fib.cpp:27
```

This will set a break point in file fib.cpp at line 27. Anytime this line is reached the program will stop executing until another command is entered.

Breakpoints

Breakpoints can also be used to break at functions instead of line numbers. If there existed a function called sum_array in fib.cpp the following could be done:

```
(gdb) break sum_array
```

gdb will stop executing once this function is reached.

Stepping Through

Now that the breakpoints are set we can use the **run** command again to start executing the code. It will stop at these breakpoints unless an error occurs before reaching one of these points.

Once a point is reached you can proceed to the next breakpoint by using the command **continue**.

Note: run will always restart execution from the beginning of the program.

Stepping Through

Instead of continuing on to the next breakpoint one can go to the next line of code by using the command **step**.

Be careful with step because it will go into the sub-routine like printf or scanf. If you don't want to go into the sub-routine use the command **next** and it will treat it as a instruction.

Retyping the same command over and over can get tedious so gdb allows the user to just hit enter and will execute the previously entered command.

Viewing Variables

Breakpoints allow you to see if a function gets called or if the interior of a conditional statement gets executed.

What if a variable is the cause of the problem?

GDB will allow the user to view the values of variables and registers.

Viewing Variables

If you want to see all the variables in the local function that the program has stopped executing in due to a breakpoint type the command **info** followed by the scope that is to be referenced.

For example to view local variables type the following:

```
(gdb) info locals
```

Viewing Variables

Other variables and registers can also be viewed at any time by using the command info. Some examples include:

- args - Allows viewing function arguments
- variables - Allows viewing of global variables
- registers - Prints the names of all registers excluding floating-point and vector
- all-registers - Prints information for all registers including floating-point and vector
- registers regname - Prints information for a specific register

Viewing Variables

In order to view one variable the **print** command followed by a variable name will print out the value of the variable. If hex output is desired use **print/x**

```
(gdb) print my_var
(gdb) print/x my_var
```

More Commands

Useful Commands:

- **backtrace** - summary of how your program got where it is (Done when a seg fault occurs)
- **where** - gives a function call trace of how you got to this point and shows line numbers inside files (Can be done anywhere)
- **finish** - runs until the current function has completed executing
- **delete** - deletes a specific breakpoint
- **info breakpoints** - shows information about all breakpoints

More on Breakpoints

Use the **tbreak** command instead of break. A temporary breakpoint only stops the program once, and is then removed.

Use the **disable** command followed by the breakpoint number to turn it off.

Use the **enable** command followed by the breakpoint number to turn it on.

Conditional Breakpoints

Breakpoints are useful to figure out what the problem is or the general issue with a variable.

We don't want to keep stepping through a function if the issue only happens towards the end of an array.

So for this reason we would like to stop executing based on a conditional requirement.

Conditional Breakpoints

Conditional breakpoints are just like normal breakpoints except for the extra conditional:

```
(gdb) break fib.cpp:27 if c >= SIZE
(gdb) break fib.cpp:27 if(c >= SIZE || c < 0)
(gdb) set $count = 0
(gdb) break funMethod: if ++$count == 1000
```

Useful Tools

GDB alone will not give you all the information you need. Other tools that can work well with GDB are:

- Valgrind - Tools that can automatically detect many memory management and threading bugs
- Electric Fence - Helps detect 2 common programming bugs: software that overruns the boundaries of a malloc() memory allocation, and software that touches a memory allocation that has been released by free()
- DUMA - Similar to Electric Fence
- Data Display Debugger (DDD) - Graphical Front-End for GDB

GNU PROJECT DEBUGGER