Debugging with GDB

A debugger lets you pause a program, examine and change variables, and step through code. Spend a few hours to learn one so you can avoid dozens of hours of frustration in the future. This is a quick 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 their vegetables, but it really is good for you – you’ll thank me later.

Category: **Guides**, **Programming**

Share what worked: Aha moments & FAQ

Let's create a living reference for how best to understand this topic.

No items yet -- add one below.

Aha! The insight that helped was:

Post feedback

26 THOUGHTS ON “DEBUGGING WITH GDB”

Pingback: GDB Debugging Tips : Debugging Tips - Program Debuggers
Pingback: Московский сельдерей » Blog Archive » Отладка с помощью Gdb
Pingback: links for 2009-06-12 | manicwave.com
Pingback: RealTime - Questions: "Help with debugging this c++ program?"
Pingback: Using GDB effectively to debug programs « xccxf4xc3
Pingback: Segmentation Fault correction with GNU Debugger | fortystones

1. **Nejd**
   on **May 17, 2007 at 2:44 am** said:

   thank you for this short tut, it’s very helpful.
   It would be good if you make another one to us how to integrate the debugger in the source code (#define …)

2. Frodo
   on **July 2, 2007 at 1:59 am** said:

   The article was clear and easy to follow. Thank you. But actually, the step command executes all the statements in a line (e.g. “a=5;a+=1;”), it’s a line step (“Run the next instruction, not line.”). For stepping on (machine) instructions the stepi command should be used. As far as I know, stepping on source level instructions/statements is not possible. Correct me, if I’m wrong. 🙂

3. **Kalid**
   on **July 2, 2007 at 1:12 pm** said:

   Thanks Frodo, didn’t know they had GNU tools in the shire 🙂

   Great catch, I’ll update the article. Nejd, I’ll put up some examples of the debugging too, thanks.

4. **technochakra**
on **July 27, 2008 at 8:45 pm** said:

Nice article. I just wrote a debugging article dedicated to hit breakpoints. Check it out and do give it a read if you get time.

http://www.technochakra.com/debugging-using-breakpoint-hit-count-for-fun-and-profit/

5. **Nejd**
on **October 16, 2008 at 4:42 pm** said:

Nice .. I will bookmark it to give it a read..

6. Juan Carlos
on **October 28, 2008 at 7:21 am** said:

How I can change the value of a local or global variable in gdb?

7. Mike
on **November 8, 2008 at 11:01 am** said:

Beautiful GDB tut, easy, precise, and to the point, def the best for quick no brainer questions!

8. **Kalid**
on **November 8, 2008 at 12:09 pm** said:

@Juan: I believe you can just do “set x = 3″

@Mike: Thanks, glad you enjoyed it!

9. Nimrod
on **December 15, 2009 at 2:43 am** said:

More tips that make gdb more bearable:

1. You can set conditions on breakpoints. E.g. “cond 3 (x==2)” will stop at breakpoint 3 only when x==2.

2. You can set ignore count on breakpoints. “ignore 3 1000″ ignores the next 1000 crossing of breakpoint 3. Then at some interesting point in time (when your program crashes…), “info break 3″ shows exactly how many times breakpoint 3 had been hit. Next time set the ignore count to one less than that number and gdb will stop one iteration before the crash…

3. When using “watch”, make sure gdb says “hardware watchpoint set”. Unlike software watchpoints, these do not slow down program execution.

4. It is possible to define macros (using define xxx … end) in .gdbrc

Other useful features I haven’t used:

1. gdb7 supports reverse debugging

2. gdb7 is scriptable using Python. Together with the new libstdc++ you get pretty printing of C++ STL collection classes.

10. Anonymous
on **September 16, 2010 at 4:02 am** said:

can we load more den one exe?..like d one we did above “gdb a.out”.

can it be like “gdb a.out b.out”?

11. Virender Kashyap
on **September 20, 2010 at 4:02 am** said:

I have been using gdb minimally. but some of the extra info here is really hepful. Thanks for the turorial.

12. Kalid
on **October 8, 2010 at 10:35 am** said:

@Anonymous: The gdb command line help indicates you can only load a single file.

13. **Tapas Mishra**
on **December 13, 2010 at 7:42 am** said:

Great article.

14. Paul
on **February 15, 2011 at 11:05 pm** said:

Good Intro.. Very helpful.

15. Kalid
on **April 17, 2011 at 9:30 pm** said:

@Paul: Thanks!

16. **Amit Sharma**
on **June 5, 2011 at 12:35 am** said:

thanks, gdb tutorial really helped.

17. rajesh
on **June 16, 2011 at 3:07 am** said:

thanks, gdb tutorial really helped.

18. Dhanunjay
on **December 1, 2011 at 11:25 pm** said:

Good one for learners and hunters!!!

19. kalid
on **December 2, 2011 at 10:43 am** said:

@Dhanunjay: Glad you liked it!

20. noname
on **March 16, 2012 at 12:53 pm** said:

#include

using namespace std ;

int main()
{
string stra , strb , strc ;
cin>>stra>>strb>>strc ;

int lena , lenb , lenc ;
lena = stra.length(); lenb = strb.length() ; lenc = strc.length() ;

int found =0 , j=0 , k=0 ;

if((lena + lenb) == lenc)
{
int a,b,c ;
for(int i =0 ; i<lenc ; i++)
{
found = 0 ;
if(j<lena && found ==0)

```
if(strc[i] == stra[j])
{
j++ ;
found = 1;
}

if(k<lenb && found ==0)
if( strc[i] == strb[k] )
{
k++;
found = 1;
}

if(found == 0)
{
cout<<"not interlieved"<<endl;
break ;

}
}
if(found == 1)
cout<<"interlieved"<<endl;
}
else
{
cout<<"not interlieved"<<endl;
}

}
```