

Basic GDB Commands¹

Making a GDB Compatible Program

In order to get the most out of using a debugger such as GDB, the compiler and linker must be told to format the resulting executable with extra information to help the debugger know how the machine language instructions relate to the higher level language expressions. This means your resulting program will be larger than without the extra debugging information, except that debugging without that extra information will make the debugger extremely difficult to use effectively.

The option for `gcc` is `-g`. This will tell `gcc` that you want it to prepare the extra information necessary for debugging using GDB². An example of using this option would be:

```
gcc -g -o cprog cprog.c
```

This produces a program named `cprog` from the code provided in `cprog.c` which includes extra information useful when debugging using GDB.

Running GDB

There are two ways of executing GDB, either from within emacs or at the shell prompt. If GDB is run within Emacs, you can often see the code as you step through the program. Emacs has two windows, one running the GDB interface, and the other has the current file being stepped through and highlights the current line being evaluated. The shell prompt doesn't have this feature of having both the file available to view simultaneously with GDB's interactive interface, so you may have to jump back and forth between your favorite editor (not Emacs in this case) and GDB constantly (unless you are in X Windows and have multiple windows open).

In Emacs the command is: `M-x gdb`
`gdb program [core]`

At the Shell prompt type: `gdb program [core]`

¹ I'm grabbing most of the commands from the "GDB Quick Reference" that comes along with the book *Debugging with GDB* by Richard M. Stallman and Roland H. Pesch, copyright 1995 by the Free Software Foundation, Inc.. The short descriptions of the commands, when available, are grabbed from it as well.

² Using non-GNU compilers could force you to use different debuggers. I recall using compilers requiring the use of special debuggers and were incompatible with GDB. Most good debuggers have the same flexible commands as GDB, so it's still a good idea to learn how to make the most of your debugger.

Running a Program Under GDB

Running a program with GDB means it'll be running with a great deal of overhead. It won't run as fast, and it may act slightly different than without using GDB. You even have a chance when running a program with GDB to run only a couple of lines of your program at a time with pauses in-between.

The magical command to actually run a program under GDB is:

```
run [arglist] - Start your program [with arglist]
```

Keep in mind that without breakpoints or watches, this will cause GDB to run the program without stopping.

Stopping a Runaway Program

In order to test a program for bugs and errors, you will want to stop it before it actually crashes or errors to find out what is causing the problem. This is done by using devices called *breakpoints* and *watches*. Breakpoints are used to define particular places to stop in your program. A watch is similar except it stops the program if the value of a given expression or variable simply changes at anytime during the execution of the program.

Simple Breakpoints

Simple breakpoints are placed at specific locations in the program. The location is either the beginning of a function or a line in your code. The commands allow you to specify a filename because you often have multiple files. If a function name is unique (not duplicated in any other files), then you do not have to specify the file it's defined within.

```
break or b - set breakpoint at next instruction
break [file:]line - set breakpoint at line number [in file]
break [file:]function - set breakpoint at function [in file]
info break - show defined breakpoints
```

Controlling Annoying Breakpoints

Sometimes you don't want breakpoints to always be triggered. Maybe just every time a certain expression is equal to a particular value. Or maybe you want a function to be evaluated everytime a particular location of your program is reached. This is done by setting a *condition* to particular break points. If the expression evaluates to a non-zero value, the breakpoint is triggered and execution stops.

```
cond n [expr] - new conditional expression on breakpoint n; make
unconditional if no expr
```

An example of how to setup a breakpoint to not stop, but run your own procedure all the time would be to use the `cond` command in the following manner:

```
cond n (proc(...)) || 1
```

Watching Values Carefully

Sometimes a bug could be attributed to a variable changing to a particular value. Let's say you don't know where the variable is changing to a bad value! It could be in a time dependant interrupt, or an obscure function! Have no fear, because GDB can *watch* these variables for you! Watches significantly reduce a process's speed because they are continuously being checked. If the value of a watch's expression changes, it's triggered and the program stops.

```
watch expr - set a watchpoint for expression expr
info watch - show defined watchpoints
```

Growing Tired of Old Breaks and Watches

You won't always need breaks and watches. Sometimes they outlive their usefulness and just seem to get in the way. You can either remove them or disable them to stop their bothersome interruptions of your running program.

```
delete [n] - delete breakpoints [or breakpoint n]
disable [n] - disable breakpoints [or breakpoint n]
enable [n] - enable breakpoints [or breakpoint n]
```

Running a Program in Steps

When you don't know where a bug is in your program, but have centralized it to a particular region in your code, it's nice to slowly move through the program's execution and watch how the variables change with the evaluation of expressions. This can be done with particular GDB commands that slowly move through a program, stopping at the end of each line evaluated.

Both the *step* and *next* commands will evaluate each line one at a time and stop after each line is evaluated. The difference between the two is the way they treat lines containing function calls. During a function call, *step* will enter the function and start evaluating the function's lines one by one. *Next* will just skip entering a function and go on to the next line.

```
step or s [count] - execute until another line reached; repeat count times if
specified
next or n [count] - execute next line, including any function calls; repeat
count times if specified
```

Continuing on Through a Program

There are times when you just need to start running the program again until the next breakpoint, or finish evaluating the procedure you stepped into. In this case there is the *continue* command to continue evaluation, and *finish* command to evaluate until the selected frame is once again the current frame.

```
continue or c [count] - continue running; if count specified, ignore this breakpoint
next count times
finish - run until selected stack frame returns
```

Finding Out Where "Here" Is

Very often, you'll find it useful to know what series of functions were called to finally arrive at the currently stopped position. You may have created a watch or a breakpoint, and now wonder what functions were called to cause GDB to stop your program. These commands in GDB examine the program's current stack. The stack is separated into frames which define all of a procedure call's arguments and local variables.

```
backtrace or bt [n] - print trace of all frames in stack; or of n frames—
innermost if n > 0, outermost if n < 0
frame [n] - select frame number n or frame at address n; if no n,
display current frame
info frame [addr] - describe selected frame, or frame at addr
info args - arguments of selected frame
info locals - local variables of selected frame
```

Printing Values of Expressions or Variables

During a debugging session, it's useful to see the values of variables. In this way you can discover whether or not a variable is obtaining incorrect values. For this purpose are two commands in GDB, *print* and *display*. *Print* is for displaying the value of an expression once. *Display* is for displaying the value of an expression each time GDB stops the program. There is even a formatting parameter that allows GDB to print the information in a reasonable format.

```
print or p [ /f ] [expr] - show value of expr [or last value $] [according to format
f]
display [ /f ] [expr] - show value of expr each time program stops [according to
format f]
undisplay n - remove number(s) n from list of automatically displayed
expressions
disable disp n - disable display for expression(s) number n
enable disp n - enable display for expression(s) number n
info display - numbered list of display expressions
```

Where to Look for Segmentation Faults

Segmentation faults are caused by your program accessing some portion of memory that does not belong to it. This means it is a *memory access bug*. What should you look for?

1. All pointers must point to valid locations in memory. They must be given a value before they are used!
2. All array and pointer indices must stay within the bounds of the allocated memory when in use.
3. A pointer should never be used to access memory if its value is zero.
4. Do not use a pointer after the memory it points to has been freed or unallocated.

If you find a segmentation fault, you can run GDB until it faults, then use *bt* to find out where it has the problem. A segmentation fault does not erase the program in GDB's memory.