Using GDB Within Emacs

Ali Erkan

January 7, 2002

This document has been prepared to help you get started with GDB (GNU DeBugger). Please remember that items given in brackets (e.g. the "[N]" in **'enable [N]'**) are optional.

1 Introduction

The purpose of a debugger is to show you what goes on "inside" a program at run-time. The debugger that we use in CS21, GDB, can do four main kinds of operations towards this goal:

- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what has happened, when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

GDB is used on executable files. That is, if you have a source file called "project.c", you must first create the corresponding executable file to be able to run it through GDB. This, of course, is accomplished by running gcc (or "make" which internally calls gcc as specified by the build-rules you put in your "Makefile") to translate "project.c" into "project". However, GDB requires a special compilation option "-g" which forces the compiler to insert special debugging information in the executable file. It is through these symbols placed in the executable code that GDB associates specific points in the source code with specific points in the executable.

2 Using GDB

GDB is a stand-alone program and is not an add-on to the gcc compiler or to the emacs editor. Therefore, like other stand-alone programs (such as gcc, emacs, pine, elm, mutt, netscape, etc) you can start it by typing its name "gdb" from the command line. However, since it is an important programming tool and since emacs is a programmers' editor, it is possible to use "gdb" from within emacs, just like we've been using the compiler from within emacs. This has two main advantages:

- All the important program development tools are available under one program (i.e. emacs) which means you will not have to jump between windows as you go through the "edit, compile, debug, edit, compile, debug, ..." cycle.
- Emacs tightly couples the current state of the debugger with the source code of your program. This simplifies many of the debugging operations such as setting breakpoints, inspecting variables, etc (all explained below in detail). If your source code is made up of multiple files, emacs automatically loads the right file as you single step through your program.

When GDB is started from within emacs, it is contained in a single buffer[1]. This means that you can switch to/from that buffer, cut and paste between that buffer and other open buffers, and close that buffer just like you would with other ordinary buffers.

3 A Few Emacs Commands...

You must know the following emacs commands to use gdb through emacs:

C-x C-f: Load a buffer
C-x k: Kill a buffer
C-x b: Select a different buffer in current window
C-x o: Move cursor to "next" window
C-x 2: Split current window horizontally into 2
C-x 3: Split current window vertically into 2
C-x 1: Maximize current window
C-x 0: Close current window
C-c c: Compile
C-c n: Move to next error
C-c g: Invoke gdb

In addition, it helps to know that you can type "Esc-x shell" to open a shell to the system to run your programs without having to switch to a different window with the mouse.

[1] Although you probably already know the definition of a buffer, we would like to reiterate it with the following paragraph from the online emacs tutorial:

> "If you find a second file with C-x C-f, the first file remains inside Emacs. You can switch back to it by finding it again with C-x C-f. This way you can get quite a number of files inside Emacs. Emacs stores each file's text inside an object called a "buffer". Finding a file makes a new buffer inside Emacs. To see a list of the buffers that currently exist in your Emacs job, type "C-x C-b". Some buffers do not correspond to files. For example, the buffer named "*Buffer List*" does not have any file. It is the buffer which contains the buffer list that was made by C-x C-b. ANY text you see in an Emacs window is always part of some buffer."

So far in the semester, we've seen two examples of buffers that do not correspond to plain files: shells (which we created by typing "Esc-x shell") and compiler messages (which we created by typing "Esc-x compile", or simply C-c C-c"). A GDB session in emacs is yet another example of a buffer that does not correspond to a file.

4 Starting GDB and Quitting GDB

You start a GDB session in emacs by typing "ESC-x, gdb". However, the key sequence "C-c, g" has also been defined for the same purpose in your default CS21 emacs configuration. At this point, emacs will ask you "Run gdb (like this): gdb". This is where you specify the name of the program you want to debug. Enter the name of an executable program (e.g. "p5") and press enter. Of course, we assume that p5 exists in the current directory. If not, you'll have to build "p5" from "p5.c" by running the "make" command prior to invoking GDB.

GDB has a **'quit'** command that terminates it. However, since we are using GDB through emacs, there is still the matter of closing the GDB buffer after GDB itself quits. The easiest thing to do to is to kill the GDB buffer with **'C-x, k'** which also eliminates the need to enter an explicit **'quit'** command. If you want to reopen a GDB session, you'll need to retype **'C-c, g'** .

Please note that if all you really want is just to restart the debugging of your program after it has already been started, you do not need to kill GDB. Instead, you should reenter the **'run'** command (explained below) and answer "yes" to GDB's "restart?" prompt.

5 Breakpoints

A "breakpoint" makes your program stop whenever a certain point in the program is reached. For each breakpoint, you can add conditions to control, in finer detail, whether your program stops. You can set breakpoints with the **'break'** command and its variants to specify the place where your program should stop (e.g. by line number, function name or exact address in the program). While a program is stopped at a breakpoint, you can examine or modify variables, execute functions, or execute any other gdb command.

GDB's assigns id numbers to breakpoints. After definition, a breakpoint is referred to with its id number. For example, see the **'info'** command below.

GDB also has "watchpoints"[2] and "catchpoints" [3]. Neither will be necessary nor used in CS21.

- **'break FUNCTION'**
  'break FILE:FUNCTION'
  Set a breakpoint at entry to function FUNCTION. The second form is to set a breakpoint in the specified source file.
- **'break LINENUM'**
  'break FILE:LINENUM'
  Set a breakpoint at line LINENUM in the current source file. The breakpoint will stop your program just before it executes any of the code on that line. The second form is to set a breakpoint in the specified source file.
- **'break ... if COND'**
  Set a breakpoint but stop only if COND evaluates to TRUE. COND can be expressed as a C Boolean expression such as "$(x < y)\&\&(z == 3)$", "(Array[10] >= -2)", "(Count == 100)", etc.
- **'info breakpoints [N]'**
  Print info on all or the specified (i.e. Nth) breakpoint.
- **'disable [N]'**
  Disable the specified breakpoint–or all breakpoints, if none are listed. A disabled breakpoint has no effect but is not forgotten in case it is enabled again in the future. You may abbreviate 'disable' as 'dis'.
- **'enable [N]'**
  Enable the specified breakpoint (or all defined breakpoints). They become effective once again in stopping your program.
- **'clear FUNCTION'**
  'clear FILENAME:FUNCTION'
  Delete any breakpoints set at entry to the function FUNCTION.
- **'clear LINENUM'**
  'clear FILENAME:LINENUM'
  Delete any breakpoints set at or within the code of the specified line.
- **'delete [N]'**
  Delete all breakpoints or just the specified one. You can abbreviate this command as 'd'.

6 Execution Control

- **'run'**
  Starts the execution of the program that was named when GDB was started. If no breakpoints were specified, the program behaves almost exactly as though it was started from the command line.
- **'run arg1 arg2 ...'**
  Starts the execution of the program as though it was started from the command line with the given command line arguments.
- **'run < infile'**
  'run > outfile'
  'run < infile > outfile'
  Run the program with input and/or output redirection.

[2] You can use a watchpoint to stop execution whenever the value of an expression changes, without having to predict a particular place where this may happen. Depending on your system, watchpoints may be implemented in software or hardware. GDB does software watchpointing by single-stepping your program and testing the variable's value each time, which is hundreds of times slower than normal execution. But this may still be worth it, to catch errors where you have no clue what part of your program is the culprit. Watchpoints should be treated as a last resort. In fact, it can even be argued they encourage you to be lazy and prevent you from understanding the dynamics of your own program. For these reasons, we will not be using them. We mention their existence just in case you run out of all options during some project outside of CS21.

[3] The C language has no use for catchpoints. Catchpoints are used in C++.

Depending on the example, file "infile" (instead of the keyboard) will be used to read the input from. Likewise, the file "outfile" (instead of the screen) will be used to send the output to.

- **'continue [IGNORE-COUNT]'**
  "Continuing" means resuming program execution until your program completes normally or encounters another breakpoint. The optional argument IGNORE-COUNT allows you to specify a further number of times to ignore any breakpoints at the statement that you're at while you are about to enter the "continue" command. This command is abbreviated as 'c'.

- **'step'**
  'step [NUM-OF-STEPS]'
  Execute one statement of the source code. If the current statement is a function call, then **'step'** steps into that function. If the optional argument NUM-OF-STEPS is given, then that many C statements are to be treated as a single block to step over. This command is abbreviated as 's'.

- **'next'**
  'next [NUM-OF-STEPS]'
  Similar to the **'step'** command, but a function call is treated as a single statement (i.e. we do not step into the function). If the optional argument NUM-OF-STEPS is given, then that many C statements are to be treated as a single block to step over. This command is abbreviated 'n'.

- **'until'**
  'until LOCATION'
  An accurate description of this command sounds confusing so we'll simply mention what it is most frequently used for. During the first iteration of a loop, **'until'** is the same as the **'step'** command. After the first iteration, it jumps to the very first statement after the loop. This is very convenient since in most cases, for a loop of n iterations, we want to single step through the first iteration but skip the remaining $n-1$ iterations. The **'until LOCATION'** variation allows you to specify a destination location in the same way breakpoint location are specified. This command is abbreviated 'u'.

- **'finish'**
  Continue running until just after the current function completes and returns. Print the returned value (if any). Therefore, this function has the effect of doing an **'until LOCATION'** command where LOCATION is set to the end of the of the current function.

- **'return'**
  'return EXPRESSION'
  You can cancel the execution of a function call with the 'return' command. The **'return EXPRESSION'** variant allows you to specify what the return value of the function should be. This is needed, of course, since you are really canceling the rest of the execution of the function, thereby potentially preventing the function from computing its return value.

- **'call FUNCTION'**
  The call command executes the named function. If the function definition has formal parameters, then the call should be made with appropriate actual parameters. This is a very useful feature for bottom-up design. As you should know at this point, the correct way to write a program is by coding/debugging one function at a time. The **'call FUNCTION'** command makes it possible for a function to be tested without having to write a complete program for testing purpose.

7 Data Inspection

While a program is stopped at a breakpoint, you can investigate its current state. For all commands given below, EXPR can be a variable name, a function name (with the appropriate actual arguments since our request is really an attempt to call the function), or any C expression such as "x+y".

The most common kind of expression is the name of a variable. Variables in expressions are understood in the context of the "current place" in the source code (i.e. where we have stopped the program with a breakpoint). Variables can be global or local[4]. A local variable can be examined and used whenever the program is executing within the function that contains that local variable. A global variable, however, can be used or examined anywhere, as long as it is not being shadowed by a local variable of the same name:

- **'whatis EXPR'**
  Print the data type of expression EXPR.

- **'print EXPR'**
  Print the value of expression EXPR. The result is printed in a format appropriate to its data type.

- **'print ARRAY-EXPR[START]@OFFSET'**
  The **'print'** command has a special form to print the contents of arrays. This command prints OFFSET elements of the array ARRAY-EXPR, starting from position START. So, for example, if we had an integer array Numbers[10], we could do "print Numbers[3]@2" to print the fourth and the fifth elements[5].

- **'display EXPR'**
  Same as **'print EXPR'** , but the output is redisplayed after each gdb command. Each displayed item is identified with an id number assigned by GDB. This id number is necessary to **'undisplay'** the displayed items.

- **'undisplay ID'**
  Undisplay the displayed item that corresponds to the specified id.

- **'set var NAME=EXPR'** Set the value of the named variable to the specified VALUE.

[4]The formal arguments of a function and the variables defined in that function are both considered local to that function.

[5]Remember, indexing always starts from 0 which is why Numbers[3] is the fourth element.

8 Modifying and Recompiling while using GDB

The point of debugging is to find bugs. When you find a bug, you change your source code. This turns out to be a problem if you continue with the same GDB session because the changes you make in your source code mis-aligns your source code with respect to the executable code that GDB is following. For example, if you delete a line, all breakpoints defined by line numbers are off by a line. If you terminate the ongoing GDB session (i.e. a **'C-x k'** command in emacs to terminate the GDB buffer) and restart with the newly generated executable, everything is OK. However, this also means that you lose your breakpoints, displayed variables, etc. If, on the other hand, the source code changes are not substantial, then you might get away with continuing the same session. For this purpose, you need to rebuild the executable (i.e. a **'C-c, c'**) and enter another **'run'** command in the GDB buffer. When GDB restarts, it notices that the object file of your program has changed (the product of the recompile) and refreshes its internal image of the object file by rereading it from the disk. If things look normal, then you're in good shape. If you notice any peculiarities (such as the "current statement" arrow pointing to the wrong lines in your source code) you may have to kill the GDB buffer and start a new GDB session.