A Brief Introduction to GDB

Nat McIntosh, Rice Computer Science Department
Kevin Skadron, Princeton Computer Science Department

Spring 1995

Programmers, like everyone else, make mistakes. Even though you may create a syntactically correct program, there may be logical errors or design flaws which cause your program to crash or not perform its intended function when run. One way to find such mistakes is just to look carefully at the program until you see the mistake. With tiny programs, this is a reasonable thing to do. When you start working on larger programs, code inspection starts to become difficult and time-consuming. A symbolic debugger allows you to have a closer look at what happens when your program runs so that you can get a better idea of where the problems lie. For example, if you suspect that there is a bug in one of the functions in your program, you can set a "breakpoint" in that function, so that when the program runs, the debugger will stop the execution at that point. Once the program is stopped the debugger allows you to poke around inside it to see if it is doing the right things. Breakpoints and many other features are described in the pages to come.

This document is a tutorial for the GNU symbolic debugger, GDB. This tutorial assumes that you know the basics of using Emacs, that you know how to compile and run simple C programs, and that you know how to use "make". If you aren't familiar with these things, this tutorial will be of marginal use to you.

1 Getting started

1.1 Compiling your program for use with GDB

In order to use GDB on your program, you need to compile your program files using the "-g" option to `lcc`. Modify the lines in your makefile which invoke `lcc` on source files to use the "-g" option. Example:

```
lcc -g foobar.c
```

Remove `a.out` and any .o files for your program and remake it using the new makefile.

1.2 Loading GDB

Once you have your program compiled correctly, you should load it into the debugger.

GDB is designed to be run from within an `emacs` buffer. By splitting the screen, this allows you to see the debugger and your source file at the same time, a particularly important feature.

Bring up `emacs` and load in your source files, then split the window using `^X-2`. Invoke GDB in one of the windows by using the command `M-x gdb`; GDB will prompt you for the name of your program. The current version of `emacs` has a funny bug; it calls the GDB buffer `*gud-whatever*`, instead of `*gdb-whatever*` as it should. This window with the `(gdb)` prompt will be referred to as the GDB I/O buffer.

2 Basic Features

2.1 Setting breakpoints

The most important feature provided by GDB is that of the *breakpoint*. The idea is that you select a particular point in your program (usually a particular source line), and when you run the program under GDB, it will halt the program's execution there and allow you to see what has happened up to that point. Sometimes just being able to see whether your program has reached a certain point is important (if you are trying to locate the source of an infinite loop).

To set a breakpoint, move to the source file line where you want the program to stop, and then type the Emacs command `^X-spc`. This will set a breakpoint at that line, and program execution will halt before executing it. Note that when you type `^X-spc`, a "break" command automatically appears within the GDB I/O buffer.

It's best to put the cursor at the beginning of the line that you want to stop at, so as to make sure that GDB interprets your request correctly. Sometimes if your cursor is at the very end of a line, GDB will set the breakpoint at the next line instead of the current line.

Setting breakpoints in the middle of multi-line statements is also a situation which can introduce some ambiguity. For example, consider the following section of code:

```
x = y          /* line N */
  +            /* line N+1 */
    z;         /* line N+2 */
```

Conceptually you can't set a breakpoint on line `N+1`, since it is not possible to stop in a state where line `N` has been executed but line `N+1` has not been executed. In situations like this GDB will select the nearest line number which makes sense (in this case it will stop at `N+2`).

You can stay out of trouble simply by always putting the cursor at the beginning of the line containing the statement which you want to stop at.

2.2 Running your program with breakpoints

To run your program under GDB, type the `run` command in the GDB I/O buffer. If your program requires command-line arguments, they should follow the `run` just like they would follow `a.out` on the command line. When you run your program, GDB will halt it if it reaches a breakpoint. When the program halts, GDB will display where it halted by displaying a `=>` on the left hand side of the line where execution has stopped. Again, when you set a breakpoint on a particular line, the program will run all the way up to that line but not including it.

2.3 Printing the contents of variables

Just being able to stop your program at a particular point is of limited usefulness if you can't see what's happening to the program's variables as it runs. GDB allows you to examine the contents of a variable when the program is halted at a breakpoint.

To print the contents of a variable, type the command

```
print <variable-name>
```

in the GDB I/O buffer. You can abbreviate this by `p`. In general, you can print the value of any local or global variable when the program is stopped at a breakpoint.

2.4 Continuing execution

The command `cont`, or just `c`, will cause GDB to resume execution of your program if it has halted at a breakpoint.

3 Intermediate GDB features

3.1 Single Stepping

GDB allows you to run your program "one line at a time" using the commands `step` and `next` (`s` and `n`). When your program is halted at a breakpoint and you type the `step` command, GDB will execute only the next line of your program, and then halt the execution again. The `next` command is essentially the same as `step`, except that when GDB encounters a call to a function, `step` will descend into the function, whereas `next` will treat the function like any other statement and execute it all at once. The `next` command can be thought of as "stepping over" function calls; the `step` command can be thought of as "stepping into" function calls.

3.2 Navigating through function calls

Most non-trivial programs have a fair number of nested function calls; as a result, when you stop at a breakpoint sometimes it isn't obvious from the location in the source just exactly how the program arrived at that point. Knowing the nesting level can be particularly important for programs which use recursion. GDB provides a number of commands which allow you to deal with nested function calls.

The `where` command will display the history of outstanding function calls up to the breakpoint. For example, suppose you set a breakpoint in the function `X()`, which is called by the function `Y()`, which is called by `main()`. When you set a breakpoint on `Y()`, run to the breakpoint, and then type `where`, GDB will display the call history (sometimes called a "stack trace") which led up to the breakpoint.

The `up` and `down` commands allow you to move your frame of reference up and down in the hierarchy of function calls. This is usually done for the purposes of printing variables. For example, if you are currently halted in function `S()`, and you want to look at the contents of a variable which is local to the function that called `S()`, you can use the `up` command to change the frame of reference back up to the caller, print a variable, and then move back down again using the `down` command. If you supply a number to `up` or `down`, GDB moves up or down that many levels of nested functions.

The `finish` command provides a way of completing the current function call. When you type `finish` while the program is halted during an invocation of a function, GDB will continue the execution and halt after the function returns.

3.3 Abbreviations and Accelerators

Most commands within GDB can be abbreviated. For example, you can type `s` instead of `step`, or `n` instead of `next`. GDB also provides a very simple way of repeating the last command you typed: hit return.

In addition, GDB provides a set of Emacs key bindings which allow you to execute certain commands even faster. In the GDB I/O buffer, you can use these special Emacs commands:

C-c C-s Execute to another source line,
like the GDB "step" command.

C-c C-n Execute to next source line in this function,
skipping all function calls,
like the GDB "next" command.

C-c C-f Execute until exit from the selected stack
frame, like the GDB "finish" command.

C-c C-r Continue execution of the program,
like the GDB "cont" command.

C-c < Go up the number of frames indicated by the numeric
argument, like the GDB "up" command.

C-c > Go down the number of frames indicated by the numeric
argument, like the GDB "down" command.

As you already know, the Emacs command `'C-x SPC'` (`'gdb-break'`) tells GDB to set a breakpoint on the source line point is on.

3.4 Displaying and manipulating breakpoints

Since GDB allows you to have more than one active breakpoint, it can occasionally be necessary to display the set of breakpoints or to delete breakpoints. Use the `info breakpoints` or `i b` command for a display of the currently active breakpoints. Each breakpoint (each with a numeric identifier) is printed, along with the source file and line number for it.

To remove a breakpoint, use the `delete` or `d` command. If given a numeric argument, it will delete the breakpoint with that number. If given no arguments, it will delete all breakpoints.

If you just want to turn a breakpoint off temporarily, find its number, `n`, using `info breakpoints` and then type `disable n`. To reenable it, type `enable n`

3.5 The GDB info command

The GDB `info` command provides a generic way of printing out status information about the debugging session. For example, we just saw that "info breakpoints" will provide you with status information about all of your breakpoints. Type "help info" for a summary of all of the options to "info".

3.6 Getting Help

Help is available both from GDB itself, and through the generic Emacs "info" facility (not the same as GDB's "info"). If you need quick-and-dirty help on a particular command, you can use the GDB `help` command from within the GDB I/O buffer; in general, this form of help is more limited.

The Emacs info facility provides much more detailed information on GDB. Type `C-h i` to get into Info, and select the menu item for GDB.

3.7 Recompiling your program

Of course, once you find a bug in your program, you'll want to make changes to your source files, recompile, and rerun your program to see if you have removed the bug. All you have to do, after recompiling, is type `run` to tell GDB to start the program over. GDB checks to see whether you have recompiled and loads the new version if so.

GDB, unfortunately, isn't always good at detecting when you recompile your program. This means that if you remake your program while GDB is running, it occasionally gets very confused if it doesn't recognize all the changes, and still has stored information which is specific to your old program (prior to the recompile). If this happens, the easiest thing to do is just kill GDB using the `quit` command and restart it before running the new version of your program.

On rare occasions, GDB will crash due to an error of its own. This almost always goes away when you restart GDB.

Note, by the way, that GDB remembers any arguments to your program, so just typing `run` with no arguemnts will cause it to run your program with the same arguments you used last time.

3.8 Symbol Table Information

At this point it is probably worth throwing in some background information about how GDB works.

Normally when you compile a program, the compiler discards a large amount of information in the process of translating it to machine language. For example, variables (roughly speaking) are translated into memory addresses in the program, and source statements are translated into sequences of machine instructions. If you were to read the object file and look at the instructions, it would be impossible to determine things like which memory location corresponded to the variable "X" in your program, of what sequence of machine instructions corresponded to the statement "a = b".

When you compile a program using the "-g" option, the compiler places so-called "symbol-table information" into the object files created. This includes all of the names of the variables and functions, their size/location in the object file, line number information (such as which assembly language instructions correspond to the statement on line N of your source file). It is this information that allows GDB to perform its magic.

For power users, GDB provides the `exec-file` and `symbol-file`. Both commands normally take a filename as an argument. The first command allows you to tell GDB specifically to reread the symbol table information for the program being debugged from a particular file. The second command instructs GDB to reread the executable program from a particular file. Normally these two files would be the same. If you invoke these commands with no arguments, GDB will reset (clear) executable file and symbol table information it is using. Normally you would only do this if you wanted to debug another program.

The `exec-file` and `symbol-file` commands can be used to "reset" GDB after you have re-compiled your program (as an alternative to quitting and restarting GDB).

4 Miscellany

4.1 Program crashes and "Signals"

Occasionally you will be debugging a program which commits some sort of fatal error, such as trying to dereference a zero-valued pointer variable. When something of this nature happens, Unix takes it as an indication that your program is running wild, and it aborts the program by sending it a *fatal signal.* When your program receives a signal, it will halt as if it had hit a breakpoint, after printing a message to the effect of "Program received signal... ". Most commonly the signal will be 11, which is a segmentation fault, or 10, which is a bus error. Once the program is halted, you can print the values of variables, but you cannot continue execution.

This is actually very useful behavior, because it is often a fast way to find where in your program the error is being generated.

4.2 More sophisticated breakpoints

Setting a breakpoint in a heavily-called function can be difficult. Sometimes you can run into a bug in such a function which only happens on certain calls to the routine, under certain conditions. If you just set a breakpoint in the routine, you may have to wait a long time for the conditions to be right to reproduce the bug.

GDB provides *conditional breakpoints* to deal with this problem. A condition, in this case, is just a boolean expression in C. A breakpoint with a condition evaluates the expression each time the program reaches it, and the program stops only if the condition is true.

To set a conditional breakpoint, first set a normal breakpoint on the line in question, then (using the breakpoint number returned by the `C-x spc` command) type the command

```
condition <number> <expression>
```

(or `cond` for short) to stop when the specified expression is true (nonzero).

Another useful trick is ignoring breakpoints. The `ignore` command allows you to tell GDB to ignore the first N crossings of a particular breakpoint before halting there. It takes two arguments; the first is the breakpoint number and the second is the number of times to ignore the break. GDB maintains a counter for each breakpoint which has been modified in this way; “info breakpoints” will display the number of crossings remaining before the breakpoint actually causes a halt.

The following is an example (courtesy of Doug Moore) of a clever way to use “ignore”:

A program crashes, but you do not know under what circumstances. You desire to step into the crash to try to understand it and what led to it. You pick a point from which you want to begin this stepping and set a breakpoint there. You “ignore” the breakpoint 10000 times and run the program. When the program crashes, you can use “i b” to determine how many more times the breakpoint is to be ignored. Let x be that number of times. Now you choose to ignore that breakpoint 9999-x times and run the program again. The program stops at the last execution of the statement before the crash.

4.3 More on displaying variable contents

Often the variable which you need to look at is a complicated data structure or an array/pointer, rather than an elemental type such as `int` or `char`.

In general, GDB provides ways of printing just about any variable. GDB allows you to print arbitrary expressions, such as `x+y*z`, or `a[2][x]`. Simply give the the expression (in C) to GDB as the argument of the `print` command. For more information on printing, see the Emacs Info section on GDB.

GDB will even go so far as to evaluate expressions which contain calls to functions defined in your program. There are limits, however, to the lengths which GDB will go in evaluating expressions, so it's best to be conservative when composing expressions which involve calls to your program's functions (particularly when they involve I/O or Unix system calls).

Expressions are evaluated in the scope of the function which the debugger is stopped at.

4.4 Array subranges using pointers

It is not uncommon to have a pointer variable which points into an array of items. If you just print the value of the pointer (Example: `print *p`), you will only get one value, which can be frustrating if you want to see a range of values.

GDB provides a convenient feature for doing just this. If "p" is a pointer variable, the the command `print x[i]@j` displays elements x[i],x[i+1],...,x[i+j-1].

4.5 Automatic display of variables

The `display` command instructs GDB to print the value of a variable each time the program stops (in other words, after each breakpoint, step, next, etc). This feature can be a useful way of tracking down what part of your program is responsible for "mangling" a variable. The usage for the command is `display <variablename>`

4.6 Assigning values to variables

GDB allows you to assign values to program variables when the program is stopped at a breakpoint. This feature can be useful to force some sort of exceptional condition or to correct the effects of one bug in order to expose another bug. To assign a value to a variable, use a command of the form

```
set <assignment-expression>
```

where `<assignment-expression>` is an assignment statement in C. For example, the command `set x=22` would set the value of the variable x to 22. If you want to see the value of the assignment, the command `print <assignment>` will perform the assignment expression and print the value assigned.

4.7 Interrupting execution

If GDB is happily running your program and you want to interrupt it—perhaps you're in an infinite loop—type `C-c C-c`.

5 Some general debugging suggestions

5.1 Extra warnings

We encourage you to compile with `lcc`'s "-A" option, which is more picky about your C and may find constructs that are technically legal but probably not what you intended. If you are a `gcc` user, the equivalent option is "-Wall". It's usually a good idea to get rid of all errors produced by "-A".

5.2 Debugging-only code

Sometimes it's useful to put `printf` statements in your code to print information that's useful for debugging, but shouldn't appear in the turn-in version. To avoid having to strip these statements out, only to put them back if you discover another problem, you can say something like

```
#ifdef DEBUG
        printf("function foo: n = %d, x = %6.3f\n'', n, x);
#endif
```

When compiling, this statement will be executed only if `DEBUG` is defined.

It's generally a good idea to have the `#ifdef` and `#endif` at the left margin, and the C statement(s) inside them indented as though they were a normal part of the function.

How do you define `DEBUG`? You should add the option "-DDEBUG" to your compiler command, giving you something like `lcc -A -g -DDEBUG foo.c`. The "-D" defines whatever symbol follows it.

How does this work? The first stage of the compiler is called the *preprocessor*, and reads the file, obeying all the `#` commands, and also stripping out comments. So the contents of any `#include` files get substituted into the file at that point; the values `#define`'ed get substituted where appropriate, and any code between a `#ifdef` and `#endif` that evaluates false (e.g., DEBUG isn't `#define`'ed) gets stripped out like a comment.

Please note that it's always a good idea to try your program with DEBUG undefined before turning it in—if you do anything more than `printf`'s, undefining DEBUG can break your program if you had accidentally put something important inside the `#ifdef DEBUG` section.

Please also note that you're perfectly welcome to turn in a program containing these kind of statements as long as they are turned off.

5.3 Asserting yourself

There is another very useful tool: `assert` statements. These should be used wherever you think some fact is always true. For example, if it is illegal for a function to receive a negative number as an argument, you might make the first statement in the function (after the variable declarations) be

```
assert(n >= 0);
```

To use `assert`'s, you must `#include <assert.h>` at the top of your file.

What happens if an assertion fails? The program comes to a screeching halt, with an announcement giving you the file and line number of the offending assertion. For example:

```
foo.c:121: failed assertion(n >= 0)
```

To turn off your assertions (which you should do before turning your program in), define NDEBUG. The preprocessor then strips out all assert statements.

Assertions are useful for making explicit any assumptions you're making. It takes a little practice to learn what to assert. `malloc`, for example, returns NULL if no memory is available. You might think you should assert that the return from a `malloc` is not NULL, but this would be wrong—if this happens, you don't want to crash with an "assertion failed" message, but instead want to print a message like "Out of memory: exiting...", and exit gracefully.

Use lots of assertions. Assert anything you think should always be true. In the best case, you'll catch bugs. In the worst case, you'll find you shouldn't have asserted something, but you'll also learn when and why your assumption is wrong.