The GNU Debugger (gdb)

A symbolic debugger is a program that aids programmers in finding logical errors, by allowing them to execute their program in a controlled manner. More precisely, a debugger allows a programmer to step through a program one statement at a time, and examine the values of variables (and expressions) following the execution of each statement. This allows a programmer to find the statement responsible for a logical error involving a particular `variable` by stepping through the program a statement at a time and examining the value of `variable` after each statement executes.

Although there are several debuggers available on UNIX systems, we will look at the GNU symbolic debugger, **gdb**. In addition to allowing programmers to edit and compile their programs, emacs also interacts with gdb.

Getting Started

Whereas a high-level program operates on *variables*, a machine-level program operates on *addresses*. A compiler must therefore translate each variable into an address and must store (among other things) these associations of variables with addresses in a special dictionary, called a *symbol table* so that it can look up an identifier when it encounters a reference to it and find its address.

The symbol table is usually discarded when compilation is complete. Some way is needed to instruct the compiler to keep it for use by the debugger. For the GNU C++ compiler, this is done with the `-g` switch.

```
g++ -g -c prog.cc
g++ -g -c lib.cc
g++ -g prog.o lib.o -o prog
```

To use gdb to debug a binary file `prog` (including `lib`), one can invoke it from the command line:

```
gdb prog
```

But a more integrated, user-friendly way is to use it from within emacs.

1. Move the cursor to the `*compilation*` window (or create a new one) and
2. Invoke gdb with the emacs command `M-x gdb`

Gdb will then inform you how to run gdb:

```
run gdb (like this): gdb
```

Respond with the name of your program:

```
run gdb (like this): gdb prog
```

The `*compilation*` window will be replaced with a `*gud-driver*` window, in which gdb is running. After some introductory text, you will see the following gdb prompt, indicating that gdb is ready to go:

```
(gdb)
```

Gdb Commands

Quitting gdb

```
(gdb) quit
```

(or `q` or `Ctrl-D`)

Setting Breakpoints

Setting a breakpoint permits you to mark a particular line in your program (called a *breakpoint*) so that when execution reaches that line, program execution will be suspended, allowing you to enter a gdb command.

To set a breakpoint:

1. Move the cursor to the desired line in the source-code window.
2. Give the emacs command: `Ctrl-x` *`spacebar`* (Control-x followed by the spacebar)

A message like the following should appear in the gdb window:

```
(gdb) Breakpoint 1 at 0x22d0:  file prog.cc, line 12.
```

Removing Breakpoints

```
(gdb) clear line-number
```

removes all break points in the line with this number; for example,

```
(gdb) clear 12
```

Starting Execution of Program

```
            (gdb) run           (or r)
```

The program will execute until a break-point is encountered (or the program terminates). If your program has a logical error that causes it to crash and you use `run` without setting any breakpoints, the program will crash (inside gdb). However, the `(gdb)` prompt should reappear, allowing you to set breakpoints and then run the program again. While it is running, gdb displays a small arrow in the left margin of the window pointing at a statement, indicating that it is to be executed next.

Resuming Execution at a Breakpoint

Once you have suspended execution at a particular statement, you can resume execution in several ways:

`continue` (or `c`)
Resumes execution and continues until the next breakpoint or until execution is completed.

`next` (or `n`)
`step` (or `s`)
You can execute just the next statement in two ways: the `next` command will execute the statement pointed to by the arrow in its entirety, and move the arrow to the following statement. This is particularly important when the statement contains a function call, because `next` will execute a function in the current statement in its entirety.

By contrast, the `step` command will execute the next statement, but will step into function calls. That is, if the statement pointed to by the arrow contains a function call, then the next statement to be executed will be the first line of that function. The `step` command enables you to execute a called function one statement at a time, whereas the `next` command confines you to your main (or currently executing) function. When execution passes from one function to another, gdb displays the name of the function, plus the value of each argument passed to its parameters, allowing you to examine what is happening when the function is called and to check if its parameters are receiving the arguments you were expecting them to get.

Displaying Values

`print` *`expression`* (or `p` *`expression`*)
Displays the value of the expression (usually a variable) once — at the current point of execution. This is especially useful when a program crashes at a particular statement or we suspect that the statement is where there is a logical error. We can use `print` to display the value of each of the variables in that statement.

`display` *`expression`*
Displays the value of the expression (usually a variable) each time execution is suspended (e.g., at breakpoints, when `next` or `step` command is used, etc.) in a format like

```
1: Counter = 0
```

This is useful when we suspect one or more variables as causing the program to work incorrectly, but we don't know where they are receiving their bad values.

`info display`
Displays a numbered list of all expressions currently being displayed.

`undisplay` *`num`*
Stop displaying expression number *num*.

More?

You can learn about these gdb commands and others using the help command:

```
(gdb) help
```

The on-line manual entry for gdb also has additional information, and can be accessed with the `man` command:

```
man gdb
```