Getting to Know GDB

by Mike Loukides and Andy Oram

September, 1996

There are many reasons you might need a debugger— the most obvious being that you're a programmer and you've written an application that doesn't work right. Beyond this, Linux depends heavily both on sharing source code and on porting code from other Unix systems. For both types of code, you may turn up problems that the original authors didn't have on their platform. So it's worth making friends with a good C debugger.

gdb is free software and you are welcome to distribute copies of it under certain conditions.

Luckily, the free Software Foundation has come through with an excellent debugger named **gdb**, which works with both C and C++ code. gdb lets you stop execution within the program, examine and change variables during execution, and trace how the program executes. It also has command-line editing and history features similara to those used with **bash** (the GNU shell) and Emacs. In fact, it now has a graphical interface. But since we've grown up using the command-line interface (and it's easier to show in print) we'll stick to that in this article. To get full documentation on all gdb commands, read the *Debugging with gdb* manual on-line or order it from the Free Software Foundation.

Compilation for gdb

Before you can use gdb to debug a program, compile and link your code with the -g option. This causes the compiler to generate an augmented symbol table. For example, the command:

```
$ gcc -g file1.c file2.c file3.o
```

compiles the C source files file1.c, generating an expanded symbol table for use with gdb. These files are linked with file3.o, an object file that has already been compiled.

The compiler's `-g` and `-O` are not incompatible; you can optimized and compile for debugging at the same time. Furthermore, unlike many otherr debuggers, gdb will even give you somewhat intelligible results. However, debugging optimized code is difficult since, by nature, optimization makes the machine code diverge from what the source code says to do.

Starting gdb

To debug a compiled program with gdb, use the command:

$ gdb *program* `[` *core-dump* `]`

where *program* is the filename of the executable file you want to debug, and *core-dump* is the name of a core dump file left from an earlier attempt to run your program. By examining the core dump with gdb, you can discover where the program failed and the reason for its failure. For example, the following command tells gdb to read the executable file qsort2 and the core dump core.2957:

```
$ gdb qsort2 core.2957
```

```
gdb is free software and you are welcome to
distribute copies of it under certain conditions;
type ``show copying'' to see the conditions.
There is absolutely no warranty for gdb;
type ``show warranty'' for details.
gdb 4.13 (sparc-sun-sunos4.1.3),
Copyright 1993 Free Software Foundation, Inc...
Core was generated by 'qsort2'.
Program terminated with signal 7, emulator trap.
#0 0x2734 in qsort2 (l=93643, u=93864, strat=1)
at qsort2.c:118
118             do i++; while (i <= u && x[i] < t);
```

```
(gdb) quit
$
```

The startup is fairly verbose; it tells you which version of gdb your're using. Then it tells you how the core file was generated (by the program **qsort2**, which received signal 7, an "emulator trap"), and what the program was doing (executing line 118). The prompt "(gdb)" tells you that gdb is ready for a command. In this case, we'll just quit.

Both the executable file and the core file arguments are optional. You can supply a core file at a later date with the core command.

Basic gdb COMMANDS

With just a few commands, you can get most of your work done in gdb. The basic things you have to do are: look at your source code, set breakpoints, run programs, and check variables.

If you forget which command to use (or want to check for obscure features) use the built-in help facility, you can request a particular command (like `help print`) or help on a number of special topics.

Listing a File

To see the contents of the source file from which the executable program was compiled, use the command list:

```
$ gdb qsort2
(gdb) list
13          void qsort2();
14          void swap();
15          void gen_and_sort();
16          void init_orgar.();
17          void init_random();
18          void print_array();
19
20          main()
21          {
22                  int power=1;
(gdb)
```

To print specific lines from the file you are currently debugging, use a list command:

```
(gdb) list line, line2
```

To list the first 10 lines from a particular function, use a list command:

```
(gdb) list routine-name
```

Executing a Program

To run the program you are debugging, use the **run** command. This may be followed by any arguments you want to pass to the program, including the standard input and output specifiers `<` and `>`, and shell wildcards (`*, ? [, ]`). You can't use C-shell history (`!`) or pipes (`|`).

For example, consider running the program **exp** through gdb. The following gdb command runs exp with the argument `-b`, taking the standard input to exp from *invalues* and redirecting standard output to the file *outtable*:

```
$ gdb exp
(gdb) run -b < invalues > outtable
```

That is, this command runs `exp -b < invalues > outtable`. If you have not set any breakpoints or used any other gdb debugging features, exp will run until it terminates, either correctly or incorrectly.

If the program you're debugging terminates abnormally, control returns to gdb. Your can then use gdb commands to find out why the program terminated. The **backtrace** command gives a stack backtrace showing exactly what the program was doing when it bombed out:

```
$ gdb badref
(gdb) run
Starting program:  /home/los/mike1/cuser/badref


0x22c8 in march_to_infinity () at badref.c:16
16                h |= *p;
(gdb) backtrace
#0   00x22c8 in march_to_infinity (1) at badref.c:16
#1   0x2324 in setup () at badref.c:25
```

```
#2   0x2340 in main () at badref.c:30
(gdb)
```

backtrace (which is usually abbreviated **back**) produces a list of all active procedures and the arguments with which they were called, starting with the most recent. So this display shows that the program died in a function named `march_to_infinity()`; this function was called by the function `setup()`, which in turn was called by the function `main()`. The only thing left is to figure out exactly what went wrong in `march_to_infinity()`.

Printing Data

You can inspect the variable values by using the **print** command. Let's use it to see exactly what happened in the previous program. First, we'll list some code to see what we're dealing with:

```
(gdb)list
8
9                 p=&j;
10                /* march off the end of the world*/
11              for ( i = 0; i < VERYBIG; i++)
12                {
13                      h |= *p;
14                      p++;
15                }
16        printf("h:  %d\en",h);
17
```

It should already be pretty clear what's happening. `p` in some kind of a pointer; we can test that by using the **whatis** command, which shows us its declaration:

```
(gdb) whatis p
type = int *
(gdb) print p
$1 = (int *) 0xf800000
(gdb) print *p
$2 = Cannot access memory at address 0xf800000.
(gdb) print h
$3 = -1
(gdb)
```

When we look at `p`, we see that it's pointing somewhere up in the stratosphere. Of course, there's no *ad hoc* way to know whether this value for p is legitimate or not. But we can see if we can read the data p points to, just as our program did—and when we give the command `print *p`, we see that it's pointing to inaccessible data.

print is one of gdb's true power features. You can use it to print the value of any expression that's valid in the language you're debugging. In addition to variables from your program, expressions may include:

- Calls to functions within your program; these function calls may have "side-effects" (i.e., they can do things like modify global variables that will be visible when you continue program execution).

```
(gdb) print find_entry(1.0)
$1 = 3
```

- Data structures and other complex objects.

```
(dgb) print *table_start
$8 = {e_reference = '\e000' <repeats 79
times>,
location = 0x0, next = 0x0}
```

Breakpoints

Breakpoints let you stop a program temporarily while it is executing. While the program is stopped at a breakpoint, you can examine or modify variables, execute functions, or execute any other gdb command. This lets you examine the program's state to determine whether execution is proceeding correctly. You can then resume program execution at the point where it left off.

The **break** command (which you can abbreviate to **b**) sets breakpoints in the program you are debugging. This command has the following forms:

break `line-number`

Stop the program just before executing the given line.

break `function-name`

Stop the program just before entering the named function.

break `line-or-function` **if** `condition`

Stop the program if the following `condition` is true when the program reaches the given line or function.

The command `break function-name` sets a breakpoint at the entrance to the specified function. When the program is executing, gdb will temporarily halt the program at the first executable line of the given function. For example, the **break** command below sets a breakpoint at the entrance to the function `init_random()`. The **run** command then executes the program until it reaches the beginning of this function. Execution stops at the first executable line within *init_random ()*, which is a *for* loop beginning on line 155 of the source file:

```
$ gdb qsort2
(gdb) break init_random
Breakpoint 1 at 0x28bc:  file qsort2.c, line 155.
(gdb) run
Starting program:  /home/los/mike1/cuser/qsort2
Tests with RANDOM inputs and FIXED pivot


Breakpoint 1, init_random (number=10) at
qsort2.c:155
155                for (i = 0; < number; i++) {
(gdb)
```

When you set the breakpoint, gdb assigns a unique identification number (in this case, 1) and prints some essential information about the breakpoint. Whenever it reaches a breakpoint, gdb prints the breakpoint's identification number, the description, and the current line number. If you have several breakpoints set in the program, the identification number tells you which one cased the program to stop. gdb then shows you the line at which the program has stopped.

To stop execution when the program reaches a particular source line, use the **break** *line-number* command. For example, the following **break** command sets a breakpoint at line 155 of the program:

```
(gdb) break 155
Note:  breakpoint 1 also set at pc 0x28bc.
Breakpoint 2 at 0x28bc; file qsort2.c, line 155.
(gdb)
```

When stopped at a breakpoint, you can continue execution with the **continue** command (which you can abbreviate as **c**):

```
$ gdb qsort2
(gdb) break init_random
Breakpoint 1 at 0x28bc:file qsort2.c, line 155.
(gdb) run
Starting program:  /home/los/mike1/curser/qsort2
Tests with RANDOM inputs and FIXED pivot

Breakpoint 1, init_random (number=10) at
qsort2.c:155
155                for (i = 0; i < number; i++){
(gdb) continue
Continuing.
test of 10 elements:  user + sys time, ticks:  0

Breakpoint 1, init_random (number=100) at
qsort2.c:155
155                for (i = 0; i < number; i++) {
(gdb)
```

Execution will continue until the program ends, you reach another breakpoint, or an error occurs.

gdb supports another kind of breakpoint, called a "watchpoint". Watchpoints are sort of like the "break-if" breakpoints we just discussed, except they aren't attached to a particular line or function entry. A watchpoint stops the program whenever an expression is true: for example, the command below stops the program whenever the variable `testsize` is greater that 100000.

```
(gdb) watch testsize > 100000
```

Watchpoints are a great idea, but they're hard to use effectively. You're exactly what you want if something is randomly trashing an important variable, and you can't figure out what: the program bombs out, you discover that `mungus` is set to some screwy value, but you know that the code that's supposed to set `mungus` works; it's clearly being corrupted by something else. The problem is that without special hardware support (which exists on only a few workstations), setting a watchpoint slows your program down by a factor of 100 or so. Therefore, if you're really desperate, you can use regular breakpoints to get your program as close as possible to the point of failure; set a watchpoint; let the program continue execution with the **continue** command; and let your program cook overnight.

Single-step Execution

gdb provides two forms of single-step execution. The **next** command executes an entire function when it encounters a call, while the **step** command enters the function and keeps going one statement at a time. To understand the difference between these two commands, look at their behavior in the context of debugging a simple program. Consider the following example:

```
$ gdb qsort2
(gdb) break main
```

```
Breakpoint 6 at 0x235c:  file qsort2.c, line 40.
(gdb) run
Breakpoint 6, at main () at qsort2.c:40
40            int power=1;
(gdb) step
43            printf(``Tests with RANDOM inputs
and FIXED pivot \n'');
(gdb) step
Tests with RANDOM inputs and FIXED pivot
45            for (testsize = 10; testsize <=
MAXSIZE; testsize *= 10){
(gdb) step
46                gen_and_sort(testsize,RANDOM,FIXED);
(gdb) step gen_and_sort (numels=10, genstyle=0,
strat=1) at
qsort2.c:79
79            s = &start_time;
(gdb)
```

We set a breakpoint at the entry to the `main ()` function, and started single-stepping. After a few steps, we reach the call to `gen_and_sort()`. At this point, the **step** command takes us into the function `gen_and_sort ()`; all of a sudden, we're executing at line 79, rather that 46. Rather than executing `gen_and_sort()` in its entirety, it stepped "into" the function. In contrast, next would execute line 46 entirely, including the call to `gen_and_sort()`.

Moving Up and Down the Call Stock

A number of informational commands vary according to where you are in the program; their arguments and output depend on the current frame. Usually, the current frame is the function where you are stopped. Occasionally, however, you want to change this default so you can do something like display a number of variables from another function.

The commands **up** and **down** move you up and down one level in the current call stack. the commands **up** *n* and **down** *n* move you up of down n levels in the stack. Down the stack means farther away from the program's `main()` function; up means closer to `main()`. By using **up** and **down**, you can investigate local variables in any function that's on the stack, including recursive invocations. Naturally, you can't move down until you've moved up first—by default you're in the currently executing function, which is as far down in the stack as you can go.

For example, in `qsort2(),main()` calls `gen_and_sort()`, which calls `qsort2()`, which calls `swap()`. If you're stopped at a breakpoint in `swap()`, a where command gives you a report like this:

```
(gdb) where
#0         swap (i=3, j=7) at qsort2.c:134
#1         0x278c in qsort2 (1=0, strat=1) at
qsort2.c:121
```

```
#2         0x25a8 in gen_and_sort (numels=10, genstyle=0,
strat=1) at qsort2.c:90
#3         0x23a8 in main () at qsort2.c:46
(gdb)
```

The **up** command directs gdb's attention at the stack frame for `qsort2()`, meaning that you can now examine `qsort2`'s local variables; previously, they were out of context. Another **up** gets you to the stack frame for `gen_and_sort()`; the command **down** moves you back towards `swap()`. If you forget where you are, the command frame summarizes the current stack frame:

```
(gdb) frame
#1         0x278c in qsort2 (1=0, u=9, strat=1) at
qqsort2.c:121
121                              swap(i, j);
```

In this case, it shows that we're looking at the stack frame for `qsort2()`, and currently executing the call to the function `swap()`. This should be no surprise, since we already know that we're stopped at a breakpoint in `swap`.

Machine Languae Facilities

gdb provides a few special commands for working with machine language. First, the **info line** command is used to tell you where the object code for a specific line of source code begins and ends. For example:

```
(gdb) info line 121
Line 121 of "qsort2.c" starts at pc 0x277c and
ends at 0x278c.
```

Your can then use the **disassemble** command to discover the machine code for this line:

```
(gdb) disassemble 0x260c 0x261c
Dump of assembler code from 0x260c to 0x261c:
0x260c <qsort2>:            save     %sp, -120, %sp
0x2610 <qsort2+4>:          st   %i0,   [ %fp + 0x44 ]
0x2614 <qsort2+8>:          st   %i1,   [ %fp + 0x48 ]
0x2618 <qsort2+12>:         st   %i2,   [ %fp + 0x4c ]
End of assembler dump.
```

The commands **stepi** and **nexti** are equivalent to **step** and **next** but work on the level of machine language instructions rather that source statements. The **stepi** command executes the next machine language instruction. The **nexti** command executes the next machine language instruction, unless that instruction calls a function, in which case **nexti** executes the entire function.

The memory inspection command **x** (for "examine") prints the contents of memory. It can be used in two ways:

(gdb) x/nfu addr (gdb) x addr

The first form provides explicit formatting information; the second form accepts the default (which is, generally, whatever format was used for the previous **x** or **print** command—or hexadecimal, if there hasn't been a previous command). addr is the address whose contents you want to display.

Formatting information is given by `nfu`, which is a sequence of three items:

- *n* is a repeat count tht specifies how many data items to print;
- *f* specifies what format to use for the output;
- *u* specifies the size of the data unit (e.g., byte, word, etc.).

```
(gdb) list 1,30
1         #include <fstream.h>
2         #include <strings.h>
3         #include <strings.h>
4
5         const unsigned int REF_SIZE = 80;
6
7         class entry {
8                char *e_text;
9                char.e_reference[REF_SIZE];
10        public:
11               entry(const char *text,
12                      const unsigned int length,
13                      const char *ref) {
14                   e_text = new char(length+1);
15                   strncpy(e_text, text, length+1);
16                   strncpy(e_reference, ref, REF_SIZE);
17               }
18         };
19
20         main(int argc, char *argv[])
21         {
22                char *text_1 = ``Finding errors in C++ programs'';
23                char *ref_1 = ``errc++'';
24                entry entry_1(text_1, strlen(text_1), ref_1);
25         }
```

Listing 1.Trivial C++Program

For example, let's investigate `s` in line 79 of our program. **print** shows that it's a pointer to a `struct tms`:

```
79                s = &start_time;
(gdb) print s
$1 = (struct tms *) 0xf7fffae8
```

The easy way to investigate further would be to use the command `print *s`, which displays the individual fields of the data structure.

```
(gdb() print *s
$2 = {tms_utime = 9, tms_stime = 14,
tms_cutime = 0, tms_cstime = 0}
```

For the sake of argument, let's use **x** to examine the data here. The `struct tms` (which is defined in the header file time.h) consists of four `int` fields; so we need to print four decimal words. We can do that with the command **x/4dw**, starting at location s:

```
(gdb) x/4dw s 0xf7fffae8 <_end+--138321592>:    9   14
0   0
```

The four words starting at location `s` are 9, 14, 0, and 0—which agrees with what **print** shows.

Signals

gdb normally traps most signals sent to it. By trapping signals, gdb gets to decide what to do with the process you are running. For example. pressing `CTRL-C` sends the interrupt signal to gdb, which would normally terminate it. But you probably don't want to interrupt gdb; you really want to interrupt the program that gdb is running. Therefore, gdb catches the signal and stops the program it is running; this lets you do some debugging.

The command **handle** controls signal handling. It takes two arguments: a signal name, and what should be done when the signal arrives. For example, let's say that you want to intercept the signal `SIGPIPE`, preventing the program you"re debugging from seeing it. Whenever it arrives, though, you want the program to stop, and you want some notification. To accomplish this, give the command:

```
(gdb) handle SIGPIPE stop print
```

Note that signal names are always capital letters! You may use signal numbers instead of signal names.

C++ Programs

If you write in C++ and compile with **g++**, you'll find gdb to be a wonderful environment. It completely understands the syntax of the language and how classes extend the concept of C structures. Let's look at a trivial program to see how gdb treats classes and constructors. Listing 1 contains a listing produced in gdb.

In order to see the program in action, we'll set a breakpoint at the `entry` statement on line 24. This declaration invokes a function, of course—the `entry` constructor.

```
(gdb) b 24
Breakpoint 1 at 0x23e4:  file ref.C, line 24.
(gdb) run
Starting program:  /home/los/mike1/crossref/ref

Breakpoint 1, main (argc=1, argv=0xeffffd8c) at
ref.C:24
24            entry entry_1(text_1, strlen(text_1),
ref_1);
```

Now we'll enter the function. We do this through the **step** command, just as when entering a function in C.

```
(gdb) step

entry::entry (this=0xefffcb8, text=0x2390
``Finding errors in C++ programs'',
        length=30, ref=0x23b0 ``errc++'') at ref.C:14
14                  e_text = new char(length+1);
```

gdb has moved to the first line of the `entry` constructor, showing us the arguments with which the function was invoked. When we return to the main program, we can print the variable `entry_1` just like any other data structure.

```
(gdb) print entry_1
$1 = {e+_text = 0x6128 ``Finding errors in C++ programs'',
e_reference = ``errc++'',
'\e000' <repeats 73 times>}
```

So C++ debugging is just as straightforward as C debugging.

Command Editing

Another useful feature is the ability to edit your commands in order to correct errors in typing. gdb provides a subset of the editing commands available in Emacs, letting you move back and forth along the line you're typing. For example, consider the command below:

```
(gdb) stop in gen_and_sort
```

If this doesn't look familiar to you, it shouldn't; it's a **dbx** command. We really meant to type **break gen_and_sort**. To fix this, we can type `ESC b` three times, to move back over the three words in `gen_and_sort` (spaces, underscores, and other punctuation define what's meant by a "word":). Then we type `ESC DEL` twice, to delete the erroneous command `stop in`. Finally, we type the correct command, `break`, followed by **RETURN** to execute it:

```
(gdb) break gen_and_sort
```

```
Breakpoint 1 at 0x2544:  file qsort2.c, line 79.
(gdb)
```

Emacs has a special mode that makes it particularly easy to use gdb. To start it, give the command `ESC x gdb`. Emans prompts you for a filename in the minibuffer:

```
Run gdb (like this) :  gdb
```

Add the executable's name and press **RETURN**; Emacs then starts a special window for running gdb, where you can give all regular gdb commands. When you stop at a breakpoint, gdb automatically creates a window displaying your source code, and marking the point where you stopped, like this:

```
        struct tms end_time, *e;
        int begin, end;

=>      s = &start_time;
        e = &end_time;
```

The mark `=>` shows the next line to be executed. The position is updated whenever gdb stops execution—that is, after every single-step, after every **continue**, etc. You amy never need to use the built-in **list** command again!