GDB Debugging: A quick introduction

Ferry Boender

Revision History

Revision 0.4 April 2004 Revised by: FB

1. About

This document contains a quick introduction on debugging using the GDB (GNU Debugger) commandline client. GDB, although not a graphical debugging client, is a very powerfull debugger. Like most Unix tools it has a pretty steep learning curve. This short tutorial like article will hopefully put you on the right track in order to debug your programs using GDB.

This article focusses on debugging within a Unix enviroment, using program examples in the C language. However, GDB is also supplied with the DJGPP package, which is a port of the GNU compiler-set for DOS and Windows. Using GDB under Dos is not much different from using it under Unix.

2. Preperation

Before you will be able to debug your programs using GDB, you must first make sure that GDB knows how your program works. It must know the names and places of functions, variables and various other things. The info required is usually called the *Debugging symbols*. These debugging symbols are contained within your executable, and GDB can read them so it can provide you with an altogether better debugging experience.

Getting debugging symbols in your program is quite easy. If you are using a GNU compiler, for instance GCC, G77 or G++, you can simply supply the `-g` parameter during the compile run. This will tell the compiler to include debugging symbols in your program. I'm not sure if GDB will work with debugging symbols generated by other compilers. If you've got more than one source file which you compile into libs and then link them together, you should supply the `-g` parameter to each compile commandline.

An example. Suppose you have the following commandline to compile your program:

```
$ gcc -Wall -o myprogram myprogram.c
```

Then you should change it to read the following:

```
$ gcc -g -Wall -o myprogram myprogram.c
```

3. Starting the debugger

Starting the debugging session using GDB is as easy as:

```
$ gdb ./myprogram
```

This will place you at the GDB commandline and you are now ready to start debugging your program. The screen will look something like this:

```
GNU gdb 6.0-debian
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or 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.
This GDB was configured as "i386-linux"...
(gdb)
```

3.1. GDB options

GDB also contains a semi-ncurses interface. This interface will show you the current source code line/file which you are debugging. When the scope changes to another file, the display will automatically change too. Most people prefer some kind of visual feedback, and if you are one of them you can use the ncurses interface. Be advised though that it is quite buggy and will garble your display on more than one occasion. (Yay GNU). To start gdb with the ncurses interface you must specify the `--tui` commandline parameter when starting GDB.

Some other usefull options for GDB are:

`--args`

The `--args` parameter allows you to directly specify parameters which will not be used by GDB himself but which will be passed along to the program you are debugging. The syntax for using it is: `gdb --args myprogram param1 param2`

`--commands=FILE`

If you want to run automated tests than `--commands`+ is for you. You can simply supply a text-file which contains the commands you wish to run automatically once GDB has been started. You might first want to get to learn GDB though. Also check out the `--batch` parameter in the GDB help documentation.

4. Basic debugging

4.1. GDB usage

As you might have seen in earlier examples, GDB is a command line interface. This means you will be provided with a prompt at which you can type commands. You might think this is arcane, and I won't tell you you're wrong. If you're not comfortable with a command-line you should perhaps check out another debugging tool. But make no mistake: GDB is a very powerful tool. It is also available on most development machines and Operating Systems. So perhaps it is a good idea if you at least know how to perform basic debugging with it.

The GDB commandline looks like this:

```
(gdb)
```

Type your commands at this line. During debugging, GDB will provide you with basic knowledge about where you are debugging at the moment and the things that happen. When, for instance, I start the program, and it hits a breakpoint, GDB will display:

```
Breakpoint 1, main () at myprogram.c:3
3       int main (void) {
```

The first line shows what happened. A breakpoint was triggered in the executable at the position which corresponds to line 3 in the source of myprogram.c. The second line shows your current location in the source file. GDB has now stopped due to the breakpoint and is awaiting your every command.

Sometimes, GDB's output may seem somewhat cryptic. This is probably because sometimes it simply is. You'll get used to it.

A helpful tip might be that when you typed in a command once, you can then simply press the return/enter key to execute the last command you entered.

4.2. Basic commands

Some of the basic commands with which you control the GDB program are:

`run [args]`

This starts the program which you want to debug. You may specify some parameters which will be transfered to current program being debugged. (also see the `set args` command).

`quit`

Quess what? It quits GDB.

`cd [path]`

Set the current directory in which GDB and the file you are debugging operate in.

`list [file]:[position]`

The list command allows you to change the focus (note: NOT the scope of the current debugging run) to another file. This allows you to set breakpoints and other things in another file which is part of your total executable, for instance A library. It works by specifying the filename and / or the position where you wish to list. The position can be either a linenumber, function name or an address. Each time you issue list, the next 10 lines will be displayed. If you want to list the previous 10 lines instead, you may use the `list -` command. Some examples:

```
(gdb) list myexample.c:0
(gdb) list myexample.c:main
(gdb) list 10
(gdb) list myexampleslib.c:myfunction
```

`set args [arg1] [arg2] [..]`

Using the `set args` command you can specify the commandline arguments which must be passed on to your program. Each time you run the program they will be passed on.

4.3. Program execution

We've already seen how we can start the program being debugged. If you wish to stop it, you can simply switch to the debugging window (if you are debugging a graphical program) and press `CTRL-C`. This will break off the execution of the current program and show you some information about where you currently are.

If you wish to continue the program from the point where you pressed `CTRL-C` (or any other point where the program was interrupted), simply type cont.

4.4. Backtracing

Whenever the program is interrupted, you can use the backtrace command to see what functions have been called in order to bring you to the current location. This is especially neat when your program segfaults. A small example:

Suppose we have the following stupid program:

```
#include <stdio.h>

int main (void) {
        char *s;

        strcpy (s, "Kaboooooooooooooooooom!");
        return (0);
}
```

This program will produce a segmentation fault because you are trying to copy a string into a memory region (identified by 's') which has not been allocated. When the program is run, GDB will produce the following output:

```
Starting program: /home/todsah/dev/myprogram/myprogram a b c

Program received signal SIGSEGV, Segmentation fault.
0x4009bbc6 in strcpy () from /lib/tls/libc.so.6
When we perform a backtrace of this program, we'll see the following output:

(gdb)backtrace
#0  0x4009bbc6 in strcpy () from /lib/tls/libc.so.6
#1  0x08048367 in main () at myprogram.c:6
```

This output is called a stack frame. The nice thing about this is that you can walk back through the functions that have been called (`strcpy()` and `main()`) and view variables in the scope of that function. Suppose the `strcpy` function was actually in a function named `myfunc()`, but the memory to which the `strcpy` function copies the string was (not) initialised in `main()`. You can now use the up and down to walk up and down the stack frame. When you enter a different stack frame you can view the contents of it's variables by using print.

You can also supply the full parameter to the backtrace command. This will make backtrace automatically display all variable names and values in the scope of the stack frames.

4.5. Breakpoints

One of the most important aspects of a debugger are breakpoints. In GDB you can set breakpoints with the appropriate break command. Breakpoints can be set on lines, functions or an address. Some examples:

```
(gdb)break panel_list_jump
        Breakpoint 1 at 0x8051d95: file panel.c, line 1093.
(gdb)break 100
        Breakpoint 2 at 0x804da54: file nimf.c, line 100.
```

An overview of breakpoint related commands:

`break [linenumber|function|address]`

Set a breakpoint. For more info see the above paragraphs.

`info break`

Show all breakpoints which are currently set.

`delete [nr] [nr] [..]`

Delete the breakpoints identified by NR. If you specify no arguments all breakpoints will be deleted.

`ignore [nr] [times]`

Tells GDB to ignore a breakpoint identified by NR for TIMES times. Suppose you have a for loop which will run 20 times, but on the 10th time your program segfaults. You can issue the command ignore 1 9 to ignore breakpoint 1 for 9 times.

`disable [nr] [nr] [..]`

Temporary disable breakpoints identified by NR's.

`enable [nr] [nr] [..]`

Enable the disabled breakpoints identified by NR's.

`commands [nr]`

Any commands you enter here will be automatically run once a breakpoint hits. GDB will query you for the commands you wish to run when you press enter. Example:

```
commands 1
Type commands for when breakpoint 1 is hit, one per line.
End with a line saying just "end".
> print *string
cont
end
```

Remember that the commands must work within the scope of the place where the breakpoint is set.

4.6. Stepping

Sometimes it can be beneficial to watch the program executing step by step. This can help you pinpoint where exactly the program is producing errors or how it got to the part which is buggy. When stepping through a program, GDB will execute one line in the current scope of the source-code each time. There are two ways of stepping through a program:

`step [TIMES]`

This will execute the next line or, if `TIMES` is specified, the next `TIMES` lines. When the scope of the program will change due to the step, GDB will follow the focus to the new scope and stop the execution of the program there. This means that the step command always executes only one line of code. For example, suppose you are debugging a program and the next line in the source-code is a call to a function. When using step, GDB will step into the function and break off execution there.

```
next [TIMES]
```

This will execute the next line or, if `TIMES` is specified, the next `TIMES` lines. When the scope of the program will change due to the step, GDB will NOT follow the focus to the new scope. Instead, GDB will treat the line as if it was a source-code line which didn't change the focus. For instance, when the next line is a function call, and you use next to execute the line, the function will be called by GDB, but execution will continue until the program returns to where the function was called and display the next line there for execution.

5. Data

Displaying data is a key aspect of debugging. The more you work with GDB, the more you will learn about displaying data and the better you will become at easier tracking down problems. GDB provides a number of ways in which you can view variable contents and other data.

5.1. Displaying data

5.1.1. print

You can use the print command to view the contents of variables and expressions. It's usage is quite simply:

```
print [expr]
```

The expression can be a variable name, a calculation and even a function call. Remember that you can only access variables within the scope of the current stack frame. (See also the paragraph on 'Backtracing') When printing a value, GDB will show the address of the variable and it's value. GDB is intelligent in that it automatically knows how to represent different variable types like string, integers, structs, arrays and floating point values. It is possible to use typecasting to show variables in different ways. For instance, to view the first character of a array of characters (otherwise known as a string), you can issue the command: `print (char)*string`, or more easily: `print s[0]`. If you would like to know the length of the string 's', you can issue a command like: `print strlen(s)`.

You can specify formats in which the specified variables' value should be shown. Formats are available to more than just the `print` command. A format should be specified in the form of `/[count][fmt]` and should go between the `print` (or any other command which supports formats, like `x`) command and the variable you wish to print. `count` tells GDB to print the following count number of memory elements too. You can't use count with the print command, but you CAN use it with `x`. `FMT` can be any of these formats:

- `o`, Octal
- `x`, Hexidecimal

- d, Decimal
- u, Usigned decimal
- t, Binary
- f, Float
- a, Address
- i, Instruction (processor)
- c, Character
- s, String

Some examples

```
(gdb) print s
$1 = 0x804a008 "Kab", 'o' <repeats 17 times>, "m!"
(gdb) print s[0]
$2 = 75 'K'
(gdb) print /x s[0]
$3 = 0x4b
```

5.1.2. x (examine memory)

A more powerful command than print is the x command, which examines memory. It, too, supports formats (see print command). With x, the COUNT parameter of formats is very important. It allows you to view a block of memory in some kind of format. Here are some examples of 'x' usage:

```
(gdb) x /20c s
0x804a008:      75 'K'  97 'a'  98 'b'  111 'o' 111 'o' 111 'o' 111 'o' 111 'o'
0x804a010:      111 'o' 111 'o' 111 'o' 111 'o' 111 'o' 111 'o' 111 'o' 111 'o'
0x804a018:      111 'o' 111 'o' 111 'o' 111 'o'
(gdb) x /20x s
0x804a008:      0x4b    0x61    0x62    0x6f    0x6f    0x6f    0x6f    0x6f
0x804a010:      0x6f    0x6f    0x6f    0x6f    0x6f    0x6f    0x6f    0x6f
0x804a018:      0x6f    0x6f    0x6f    0x6f
```

In the above example we are displaying the memory region occupied by a string. The string has been initiated in the corresponding C code by allocating 20 characters to the 's' string. As we can see when we display the first 20 characters in this string that there isn't a `\0` character anywhere within the first 20 characters. This means that we have a buffer overrun, and thus a possible segmentation fault.

5.1.3. display

Use the `display [EXPR]` command to tell GDB to show a particular variables contents each time the program is interrupted (breakpoints, `CTRL-C`, etc). `display` works in the same way as x does.

Expressions, once again, can be anything from a simple variable-name, an memory address and even a function. Use the `undisplay` command to stop displaying the data automatically. If you only want to temporary stop displaying, you may use the `disable display` and `enable display` commands. Both these commands require as a parameter the display rule. You can view all display rules by typing `display` without any parameters.

6. Changing data

Changing data in a running program can be a very powerful tool. For example, one can use it to see what happens when you assign some daft value to a variable in your program. Does it crash, or does it handle it graceful? Or if you think you've finally found that nasty bug in your program and are preparing to change the code accordingly. You could first alter some variables to see if the fix you thought would bring an end to the suffering actually is the fix. Changing variables is also a great way to test all the possibilities in a routine. (There's a name for this, but I forgot what it was. Something along the lines of touching every line of code?)

Changing data can be done with the `set variable [EXPR]` command. It's a little confusing since `set` is also used to change options in GDB itself. Adding to the confusion: you can, in most cases, abbreviate the command and leave out the variable bit, so that the command becomes `set [EXPR]`

`[EXPR]` is an expression which assigns a value to a variable. For example, `set my_linked_list = NULL`. The expression must be valid within the current scope. For some reason it does not seem possible to use #define'd values in expressions. (If anybody can point out why this is, please let me know. It's quite annoying) The expression should adhere to the syntax of the language the program you are debugging was written in. For C this would be `set foo=5`, for Pascal: `set foo := 5`.

You can also assign values to something called convenience variables. These variables are an extension of GDB and they are not part of the program that you are debugging. A convenience variable is prepended with a '$'. For example, `set $prev_i = i;`.

7. Settings

GDB has an enormous ammount of settings. Some of the more usefull ones are:

`set args [ARG] [ARG] [..]`

Sets the commandline arguments for the current program being debugged. set args has already been discussed in the 'Basic commands' paragraph.

`set follow-fork-mode [OPTION]`

This setting changes the way GDB behaves when it encounters a fork() inside your program. Useful for programs that spawn children. Possible options are: parent, child and ask. No explanation needed I assume?

`set listsize [NR]`

This lets GDB know how many lines it should display each time you run the list command.

`set logging [SUBOPTION]`

GDB has the ability to log all output to a log-file. Nice if you want to review a debugging session. Use set logging file `[FILE]` to specify to which file the log will be written. Next, set logging on will turn on logging.

`set print [SUBOPTION]`

The print settings specify how data is printed by default. It has lots of different sub-options, of which I personally only find the set print pretty on command useful. It makes print print structures somewhat prettier.

There are many more settings which you can change. Please see the 'Getting help' chapter for more information

8. Getting help

GDB has lots of documentation, both in the GDB program itself as in various info and man pages.

Using the internal help system is easy. Type `help` at the command-line to get a overview of top-level help topics. Issue `help [SUBTOPIC]` to view the commands in that subtopic. To get help on a specific command, use help `[COMMAND]`.

The info system on Unix systems holds the complete user manual for GDB, which is huge. Type `info gdb` at the commandline to view it. You might need a separate 400 page manual on how to use 'info' though. Some quick help: Press the enter key on anything which ends in '::' to follow that hyper-link. Use the 'l' key to return to the page which you were viewing before the current one.

You can also view the documentation online at http://sources.redhat.com/gdb/current/onlinedocs/gdb_toc.html.

9. Copyright / Author

Copyright (c) 2002-2004, Ferry Boender

Author:

```
Ferry Boender
<ferry (DOT) boender (AT) electricmonk (DOT) nl>
```