GDB - QUICK GUIDE

WHAT IS GNU DEBUGGER?

A debugger is a program that runs other programs, allowing the user to exercise control over these programs, and to examine variables when problems arise.

GNU Debugger, which is also called **gdb,** is the most popular debugger for UNIX systems to debug C and C++ programs.

GNU Debugger helps you in getting information about the following:

- If a core dump happened, then what statement or expression did the program crash on?
- If an error occurs while executing a function, what line of the program contains the call to that function, and what are the parameters?
- What are the values of program variables at a particular point during execution of the program?
- What is the result of a particular expression in a program?

How GDB Debugs?

GDB allows you to run the program up to a certain point, then stop and print out the values of certain variables at that point, or step through the program one line at a time and print out the values of each variable after executing each line.

GDB uses a simple command line interface.

Points to Note

- Even though GDB can help you in finding out memory leakage related bugs, but it is not a tool to detect memory leakages.
- GDB cannot be used for programs that compile with errors and it does not help in fixing those errors.

GDB - INSTALLATION

Before you go for installation, check if you already have gdb installed on your Unix system by issuing the following command:

```
$gdb -help
```

If GDB is installed, then it will display all the available options within your GDB. If GDB is not installed, then proceed for a fresh installation.

You can install GDB on your system by following the simple steps discussed below.

step 1: Make sure you have the prerequisites for installing gdb:

- An ANSI-compliant C compiler *gccisrecommended − notethatgdbcandebugcodesgeneratedbyothercompilers*
- 115 MB of free disk space is required on the partition on which you're going to build gdb.
- 20 MB of free disk space is required on the partition on which you're going to install gdb.
- GNU's decompression program, **gzip**
- The **make** utility - the GNU version is known to work without a problem, others probably do as well.

step 2: Download the gdb source distribution from **ftp.gnu.org/gnu/gdb.** (We used **gdb-6.6.tar.gz** for these instructions.) Place the distribution files in your build directory.

step 3: In your build directory, decompress gdb-6.6.tar.gz and extract the source files from the archive. Once the files have finished extracting, change your working directory to the gdb-6.6 directory that was automatically created in your build directory.

```
$ build> gzip -d gdb-6.6.tar.gz
$ build> tar xfv gdb-6.6.tar
$ build> cd gdb-6.6
```

step 4: Run the configure script to configure the source tree for your platform.

```
$ gdb-6.6> ./configure
```

step 5: Build gdb using the **make** utility.

```
$ gdb-6.6> make
```

step 6: Login as root and install gdb using the following command.

```
$ gdb-6.6> make install
```

step 7: If required, disk space can be reclaimed by deleting the gdb build directory and the archive file after the installation is complete.

```
$ gdb-6.6> cd ..
$ build> rm -r gdb-6.6
$ build> rm gdb-6.6.tar
```

You now have gdb installed on your system and it is ready to use.

GDB - DEBUGGING SYMBOLS

A **Debugging Symbol Table** maps instructions in the compiled binary program to their corresponding variable, function, or line in the source code. This mapping could be something like:

- Program instruction ⇒ item name, item type, original file, line number defined.

Symbol tables may be embedded into the program or stored as a separate file. So if you plan to debug your program, then it is required to create a symbol table which will have the required information to debug the program.

We can infer the following facts about symbol tables:

- A symbol table works for a particular version of the program – if the program changes, a new table must be created.
- Debug builds are often larger and slower than retail $non-debug$ builds; debug builds contain the symbol table and other ancillary information.
- If you wish to debug a binary program you did not compile yourself, you must get the symbol tables from the author.

To let GDB be able to read all that information line by line from the symbol table, we need to compile it a bit differently. Normally we compile our programs as:

```
gcc hello.cc -o hello
```

Instead of doing this, we need to compile with the -g flag as shown below:

```
gcc -g hello.cc -o hello
```

GDB - COMMANDS

GDB offers a big list of commands, however the following commands are the ones used most frequently:

- **b main** - Puts a breakpoint at the beginning of the program
- **b** - Puts a breakpoint at the current line
- **b N** - Puts a breakpoint at line N
- **b +N** - Puts a breakpoint N lines down from the current line
- **b fn** - Puts a breakpoint at the beginning of function "fn"
- **d N** - Deletes breakpoint number N
- **info break** - list breakpoints
- **r** - Runs the program until a breakpoint or error
- **c** - Continues running the program until the next breakpoint or error
- **f** - Runs until the current function is finished
- **s** - Runs the next line of the program
- **s N** - Runs the next N lines of the program
- **n** - Like s, but it does not step into functions
- **u N** - Runs until you get N lines in front of the current line
- **p var** - Prints the current value of the variable "var"
- **bt** - Prints a stack trace
- **u** - Goes up a level in the stack
- **d** - Goes down a level in the stack
- **q** - Quits gdb

GDB - DEBUGGING PROGRAMS

Getting Started: Starting and Stopping

- gcc -g myprogram.c
    - Compiles myprogram.c with the debugging option $-g$. You still get an a.out, but it contains debugging information that lets you use variables and function names inside GDB, rather than raw memory locations $notfun$.
- gdb a.out
    - Opens GDB with file a.out, but does not run the program. You'll see a prompt gdb - all examples are from this prompt.
- r
- r arg1 arg2
- r < file1
    - Three ways to run "a.out", loaded previously. You can run it directly r, pass arguments $rarg1arg2$, or feed in a file. You will usually set breakpoints before running.

- help
- h breakpoints
    - Lists help topics *help* or gets help on a specific topic *hbreakpoints*. GDB is well-documented.
- q - Quit GDB

Stepping through Code

Stepping lets you trace the path of your program, and zero in on the code that is crashing or returning invalid input.

- l
- l 50
- l myfunction
    - Lists 10 lines of source code for current line *l*, a specific line *l*50, or for a function *lmyfunction*.
- next
    - Runs the program until next line, then pauses. If the current line is a function, it executes the entire function, then pauses. **next** is good for walking through your code quickly.
- step
    - Runs the next instruction, not line. If the current instruction is setting a variable, it is the same as **next**. If it's a function, it will jump into the function, execute the first statement, then pause. **step** is good for diving into the details of your code.
- finish
    - Finishes executing the current function, then pause *alsocalledstepout*. Useful if you accidentally stepped into a function.

Breakpoints or Watchpoints

Breakpoints play an important role in debugging. They pause *break* a program when it reaches a certain point. You can examine and change variables and resume execution. This is helpful when some input failure occurs, or inputs are to be tested.

- break 45
- break myfunction
    - Sets a breakpoint at line 45, or at myfunction. The program will pause when it reaches the breakpoint.
- watch x == 3
    - Sets a watchpoint, which pauses the program when a condition changes *whenx* == 3*changes*. Watchpoints are great for certain inputs *myPtr*! = *NULL* without having to break on *every* function call.
- continue
    - Resumes execution after being paused by a breakpoint/watchpoint. The program will continue until it hits the next breakpoint/watchpoint.
- delete N
    - Deletes breakpoint N *breakpointsarenumberedwhencreated*.

Setting Variables

Viewing and changing variables at runtime is a critical part of debugging. Try providing invalid inputs to functions or running other test cases to find the root cause of problems. Typically, you will view/set variables when the program is paused.

- print x
    - Prints current value of variable x. Being able to use the original variable names is why the $-g$ flag is needed; programs compiled regularly have this information removed.
- set x = 3
- set x = y
    - Sets x to a set value 3 or to another variable y
- call myfunction
- call myotherfunction$x$
- call strlen$mystring$
    - Calls user-defined or system functions. This is extremely useful, but beware of calling buggy functions.
- display x
    - Constantly displays the value of variable x, which is shown after every step or pause. Useful if you are constantly checking for a certain value.
- undisplay x
    - Removes the constant display of a variable displayed by display command.

Backtrace and Changing Frames

A stack is a list of the current function calls - it shows you where you are in the program. A *frame* stores the details of a single function call, such as the arguments.

- bt
    - **Backtraces** or prints the current function stack to show where you are in the current program. If main calls function a, which calls b, which calls c, the backtrace is

```
c <= current location
b
a
main
```

- up
- down
    - Move to the next frame up or down in the function stack. If you are in **c,** you can move to **b** or **a** to examine local variables.
- return
    - Returns from current function.

Handling Signals

Signals are messages thrown after certain events, such as a timer or error. GDB may pause when it encounters a signal; you may wish to ignore them instead.

- handle [signalname] [action]

- handle SIGUSR1 nostop
- handle SIGUSR1 noprint
- handle SIGUSR1 ignore
    - Instruct GDB to ignore a certain signal *SIGUSR1* when it occurs. There are varying levels of ignoring.

GDB - DEBUGGING EXAMPLES

Go through the following examples to understand the procedure of debugging a program and core dumped.

- Debugging Example 1

  This example demonstrates how you would capture an error that is happening due to an exception raised while dividing by zero.

- Debugging Example 2

  This example demonstrates a program that can dump a core due to non-initialized memory.

Both the programs are written in C++ and generate core dump due to different reasons. After going through these two examples, you should be in a position to debug your C or C++ programs generating core dumps.

GDB - SUMMARY

After going through this tutorial, you must have gained a good understanding of debugging a C or C++ program using GNU Debugger. Now it should be very easy for you to learn the functionality of other debuggers because they are very similar to GDB. It is highly recommended that you go through other debuggers as well to become familiar with their features.

There are quite a few good debuggers available in the market:

- **DBX Debugger** - This debugger ships along with Sun Solaris and you can get complete information about this debugger using the man page of dbx, i.e., *man dbx*.
- **DDD Debugger** - This is a graphical version of dbx and freely available on Linux. To have a complete detail, use the man page of ddd, i.e., *man ddd*.

You can get a comprehensive detail about GNU Debugger from the following link: Debugging with GDB