GNU-GDB

debugging tool

tutorialspoint

SIMPLYEASYLEARNING

About the Tutorial

GDB, short for GNU Debugger, is the most popular debugger for UNIX systems to debug C and C++ programs.

This tutorial provides a brief introduction on how to use GDB commands to ensure the programs are error-free.

Audience

A debugger is regarded as the best friend of a software programmer. Hence, this tutorial will be useful for all those programmers who would like to develop robust and error-free programs.

Prerequisites

This tutorial assumes that you already know how to program in C and C++ and that you can compile and execute programs.

Copyright & Disclaimer

Table of Contents

1. WHAT IS GDB?

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:

1. If a core dump happened, then what statement or expression did the program crash on?

2. If an error occurs while executing a function, what line of the program contains the call to that function, and what are the parameters?

3. What are the values of program variables at a particular point during execution of the program?

4. 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

5. Even though GDB can help you in finding out memory leakage related bugs, but it is not a tool to detect memory leakages.

6. GDB cannot be used for programs that compile with errors and it does not help in fixing those errors.

2. INSTALLING GDB

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.

1. Make sure you have the prerequisites for installing *gdb*:
    - An ANSI-compliant C compiler (gcc is recommended - note that gdb can debug codes generated by other compilers)
    - 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.

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.

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
```

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

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

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

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

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

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

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.

3. DEBUGGING SYMBOL TABLE

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
```

4. 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

5. 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 (not fun).
- 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 (r arg1 arg2), 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 (h breakpoints). 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 (l myfunction).
- 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 (also called step out). 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 (when x == 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 (breakpoints are numbered when created).

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.

6. 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.

Let us write a program to generate a core dump.

```
#include <iostream>
using namespace std;

int divint(int, int);

int main() {
  int x = 5, y = 2;
  cout << divint(x, y);
  x =3; y = 0;
  cout << divint(x, y);
  return 0;
}

int divint(int a, int b)
{
  return a / b;
}
```

To enable debugging, the program must be compiled with the -g option.

```
$g++ -g crash.cc -o crash
```

NOTE: We are using g++ compiler because we have used C++ source code.

Now, when you run this program on your linux machine, it will produce the following result:

```
  Floating point exception (core dumped)
```

You will find a *core* file in your current directory.

Now to debug the problem, start gdb debugger at the command prompt:

```
$gdb crash
    # Gdb prints summary information and then the (gdb) prompt

    (gdb) r
    Program received signal SIGFPE, Arithmetic exception.
    0x08048681 in divint(int, int) (a=3, b=0) at crash.cc:21
    21        return a / b;
    # 'r' runs the program inside the debugger
    # In this case the program crashed and gdb prints out some
    # relevant information.  In particular, it crashed trying
    # to execute line 21 of crash.cc.  The function parameters
    # 'a' and 'b' had values 3 and 0 respectively.

    (gdb) l
    # l is short for 'list'.  Useful for seeing the context of
    # the crash, lists code lines near around 21 of crash.cc

    (gdb) where
    #0  0x08048681 in divint(int, int) (a=3, b=0) at crash.cc:21
    #1  0x08048654 in main () at crash.cc:13
    # Equivalent to 'bt' or backtrace.  Produces what is known
    # as a 'stack trace'.  Read this as follows:  The crash occurred
    # in the function divint at line 21 of crash.cc.  This, in turn,
    # was called from the function main at line 13 of crash.cc

    (gdb) up
    # Move from the default level '0' of the stack trace up one level
    # to level 1.

    (gdb) list
    # list now lists the code lines near line 13 of crash.cc


    (gdb) p x
    # print the value of the local (to main) variable x
```

In this example, it is fairly obvious that the crash occurs because of the attempt to divide an integer by 0.

To debug a program 'crash' that has crashed and produced a core file named 'core', type the following at the command line:

```
gdb crash core
```

As this is mostly equivalent to starting gdb and typing the 'r' command, all of the commands above could now be used to debug the file.

Debugging Example 2

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

Let us write another program that will cause a core dump due to non-initialized memory.

```
#include <iostream>
using namespace std;
void setint(int*, int);
int main() {
  int a;
  setint(&a, 10);
  cout << a << endl;
  int* b;
  setint(b, 10);
  cout << *b << endl;
  return 0;
}
void setint(int* ip, int i) {
  *ip = i;
}
```

To enable debugging, the program must be compiled with the -g option.

```
  $g++ -g crash.cc -o crash
```

NOTE: We are using g++ compiler because we have used C++ source code.

When you run this program on your linux machine, it will produce the following result:

```
segmentation fault (core dumped)
```

Now let us debug it using gdb:

```
 $ gdb crash
    (gdb) r
    Starting program: /home/tmp/crash
    10
    10
    Program received signal SIGSEGV, Segmentation fault.
    0x4000b4d9 in _dl_fini () from /lib/ld-linux.so.2
    (gdb) where
    #0  0x4000b4d9 in _dl_fini () from /lib/ld-linux.so.2
    #1  0x40132a12 in exit () from /lib/libc.so.6
    #2  0x4011cdc6 in __libc_start_main () from /lib/libc.so.6
    #3  0x080485f1 in _start ()
    (gdb)
```

Unfortunately, the program will not crash in either of the user-defined functions, **main** or **setint**, so there is no useful trace or local variable information. In this case, it may be more useful to single-step through the program.

```
    (gdb) b main
    # Set a breakpoint at the beginning of the function main
    (gdb) r
    # Run the program, but break immediately due to the breakpoint.
    (gdb) n
    # n = next, runs one line of the program
    (gdb) n
    (gdb) s
    setint(int*, int) (ip=0x400143e0, i=10) at crash2.C:20
    # s = step, is like next, but it will step into functions.
    # In this case the function stepped into is setint.
    (gdb) p ip
    $3 = (int *) 0x400143e0
    (gdb) p *ip
    1073827128
```

The value of *ip is the value of the integer pointed to by ip. In this case, it is an unusual value and is strong evidence that there is a problem. The problem in this case is that the pointer was never properly initialized, so it is pointing to some random area in memory (the address 0x40014e0). By pure luck, the process of assigning a value to *ip does not crash the program, but it creates some problem that crashes the program when it finishes.

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.

7. WHAT IS NEXT?

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.