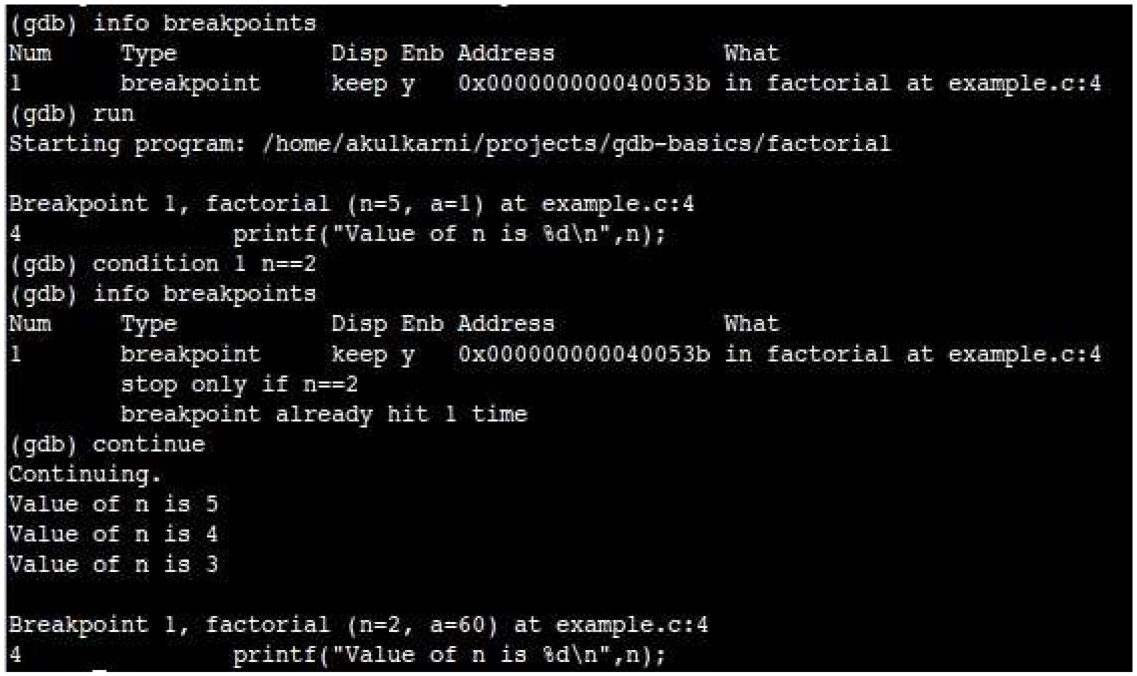
GDB — Basics

Amit Kulkarni · Follow
9 min read · Jul 2, 2017

Introduction to basic usage of gdb



```
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x000000000040053b in factorial at example.c:4
(gdb) run
Starting program: /home/akulkarni/projects/gdb-basics/factorial

Breakpoint 1, factorial (n=5, a=1) at example.c:4
4               printf("Value of n is %d\n",n);
(gdb) condition 1 n==2
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x000000000040053b in factorial at example.c:4
        stop only if n==2
        breakpoint already hit 1 time
(gdb) continue
Continuing.
Value of n is 5
Value of n is 4
Value of n is 3

Breakpoint 1, factorial (n=2, a=60) at example.c:4
4               printf("Value of n is %d\n",n);
```

A lot of people like reading or browsing code and given the convenience of sublime or any other IDEs/tools like cscope, the navigation is simple. The real complexity arises when you have to :

- Find a bug
- Understand the details of how program reacts to a certain kind of input
- Understand the details of your own code

This is where the IDEs come in and provide you some powerful tools and visual buttons to step through code and see the live action. But, if you are one of command line crowd that never uses a graphical IDE or visual debuggers with step-in/step-out buttons, welcome to the beautiful and simple world of GDB (GNU Debugger) 😊

Some uses of gdb:

- Step through a program line by line
- Set breakpoints that will stop your program
- Make your program stop on specified conditions
- Show the present values of variables
- Examine the contents of any frame on the call stack

So, here’s what we are going to do in this article:

- How to get gdb on Ubuntu
- How to compile a program to use with gdb
- Learn the basic information provided by gdb
- Step through a simple program to demonstrate various options of gdb

Specifications of the machine used for this article:

- OS : Ubuntu 14.04 LTS
- gdb version: 7.7.1
- gcc version: 4.8.4

If you need a refresher on gcc and it's usage, here is a great page to help you get started: GCC and Make

How to get gdb on Ubuntu

```
$ sudo apt-get install gdb
$ echo $?
0
$ gdb --version
```

Make sure gdb was installed correctly using return code check ($?)

How to compile a program to use with gdb

```
$ gcc -g -ggdb3 -o binaryfile source_code.c
```

```
-g : produces debugging information in the OSs native format (stabs,
COFF, XCOFF, or DWARF 2).

-ggdb : produces debugging information specifically intended for gdb.

-ggdb3 : produces extra debugging information, for example: including
macro definitions.

-ggdb by itself without specifying the level defaults to -ggdb2

-o binaryfile: Place the output in the specified binaryfile
```

Learn the basic information provided by gdb

Let's use a simple C program *example.c* to see the basic information provided by gdb

```
#include <stdio.h>

unsigned int factorial(unsigned int n, unsigned int a){
    if (n == 0)  return a;
    return factorial(n-1, n*a);
}

int main(int argc, char* argv[]){
    unsigned int number = 5;
    printf("Factorial of number %d is %d\n",
            number,
            factorial(number, 1));
    return 0;
}
```

The program finds the factorial of 5. Let's compile this program and see the information provided by gdb.

```
$ gcc -g -ggdb3 -o factorial example.c
$ gdb -q factorial

-q : Do not print introductory and copyright messages (Quiet mode)
```

I would first start with a command that tells me what information is available to me:

```
$ gdb -q factorial
Reading symbols from factorial...done.
(gdb) info
List of info subcommands:

info address -- Describe where symbol SYM is stored
info all-registers -- List of all registers and their contents
info args -- Argument variables of current stack frame
info auto-load -- Print current status of auto-loaded files
info auto-load-scripts -- Print the list of automatically loaded
Python scripts
info auxv -- Display the inferior's auxiliary vector
info bookmarks -- Status of user-settable bookmarks
info breakpoints -- Status of specified breakpoints (all user-
settable breakpoints if no argument)
...
info os -- Show OS data ARG
info types -- All type names
info variables -- All global and static variable names
---Type <return> to continue, or q <return> to quit---
```

There we go! A neatly sorted list of commands to get information.
Here is a list of some interesting commands in help, and a description of what a few of the info commands do:

```
(gdb) help status   # lists a bunch of info commands
(gdb) info frame    # list information about the current stack frame
(gdb) info locals   # list local variable values of current stack
frame
(gdb) info args     # list argument values of current stack frame
(gdb) info registers        # list register values
(gdb) info breakpoints      # list status of all breakpoints
```

Well, the information gets more interesting once we start debugging our binary file. Let's run our program. Oh wait! The whole purpose of coming to gdb was to pause, observe and proceed. There is no point in running a program without a breakpoint! Simplest way of putting a breakpoint is using the function name or a line number.

```
(gdb) list
1 #include <stdio.h>
2
3 unsigned int factorial(unsigned int n, unsigned int a){
4     if (n == 0)  return a;
5     return factorial(n-1, n*a);
6 }
7
8
9 int main(int argc, char* argv[]){
10  unsigned int number = 5;
(gdb)
11  printf("Factorial of number %d is %d\n",number,factorial(number,
1));
12  return 0;
13 }

(gdb) break 9
Breakpoint 1 at 0x40056d: file example.c, line 9.
```

Or get the list of functions in the program and break on a particular function.

```
(gdb) info functions
All defined functions:

File example.c:
unsigned int factorial(unsigned int, unsigned int);
int main(int, char **);
...
```

```
(gdb) break factorial
Breakpoint 2 at 0x40053b: file example.c, line 4.
(gdb) break main
Breakpoint 3 at 0x40056d: file example.c, line 10.
```

Forgot the list of breakpoints? Need to delete a breakpoint? Here is how you do it:

```
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
2       breakpoint     keep y   0x000000000040053b in factorial at
example.c:4
3       breakpoint     keep y   0x000000000040056d in main at
example.c:10
(gdb) delete breakpoints 2
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
3       breakpoint     keep y   0x000000000040056d in main at
example.c:10
```

Now, let’s run our program with a sample argument just to demonstrate how to run a program in gdb with arguments

```
(gdb) run a_random_argument another_random_argument
Starting program: <path>/factorial a_random_argument
another_random_argument
```

```
Breakpoint 3, main (argc=3, argv=0x7fffffffdf78) at example.c:10
10  unsigned int number = 5;
(gdb) info args
argc = 3
argv = 0x7fffffffdf78
```

We are at the breakpoint 3 (in main() function) and we are able to see that our program has received 3 arguments. How to see what were the arguments? Use the ‘print’ command.

```
(gdb) print argv[0]
$1 = 0x7fffffffe2e4 "<path>/factorial"
(gdb) print argv[1]
$2 = 0x7fffffffe312 "a_random_argument"
(gdb) print argv[2]
$3 = 0x7fffffffe324 "another_random_argument"
```

You could also print the values in hex/binary/octal/other formats.

```
# Printing in hex
(gdb) p/x argc
$4 = 0x3
# Printing in Binary (t stands for two(2))
(gdb) p/t argc
$5 = 11
```

How to expand macros? *'macro expand'* to our rescue!

```
(gdb) b main
Breakpoint 1 at 0x400815: file <path>/main.c, line 5.
(gdb) r
Starting program: <path>/testmacros

Breakpoint 1, main () at <path>/main.c:5
5  int err = 1;
(gdb) n
6  if(err == FILE_READ_ERROR){
(gdb) macro expand FILE_READ_ERROR
expands to: 3
```

NOTE: The program has to be run before expanding macros.

At any point in the program, you can use the *'list|l'* command to view the next lines gdb is going to execute.

```
(gdb) l
5     return factorial(n-1, n*a);
6 }
7
8
9 int main(int argc, char* argv[]){
10  unsigned int number = 5;
11  printf("Factorial of number %d is %d\n",number,factorial(number,
1));
```

```
12  return 0;
13 }
```

And if you need any help in knowing the options of a command:

```
(gdb) help all
...
...

(gdb) help breakpoints
Making program stop at certain points.

List of commands:

awatch -- Set a watchpoint for an expression
break -- Set breakpoint at specified line or function
...
delete breakpoints -- Delete some breakpoints or auto-display
expressions
delete display -- Cancel some expressions to be displayed when
program stops
disable breakpoints -- Disable some breakpoints
disable display -- Disable some expressions to be displayed when
program stops
enable -- Enable some breakpoints
enable breakpoints -- Enable some breakpoints
enable breakpoints count -- Enable breakpoints for COUNT hits
enable breakpoints delete -- Enable breakpoints and delete when hit
enable breakpoints once -- Enable breakpoints for one hit
...
```

Let’s achieve our basic goal of debugging: pause-observe-proceed. The 4 gdb commands that empower you to do that are : *‘next’, ‘step’ , ‘continue’ and ‘finish’.*

next|n: Proceed to the next line of execution (Doesn’t step into a function call in the current line)

```
Breakpoint 1, main (argc=1, argv=0x7fffffffdfa8) at example.c:10
10  unsigned int number = 5;
(gdb) next
11  printf("Factorial of number %d is %d\n",number,factorial(number,
1));
(gdb) next
Factorial of number 5 is 120
12  return 0;
(gdb) next
13 }
```

step|s: Step into the function call in the line being executed (even if there is no breakpoint for that function)

```
Breakpoint 1, main (argc=1, argv=0x7fffffffdfa8) at example.c:10
10  unsigned int number = 5;
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x000000000040056d in main at
```

```
example.c:10
 breakpoint already hit 1 time
(gdb) n
11  printf("Factorial of number %d is %d\n",number,factorial(number,
1));
(gdb) step
factorial (n=5, a=1) at example.c:4
4     if (n == 0)  return a;
```

continue|c: Continue execution till the next break point or end of program
finish: Finish execution of the current function

Of course this is just the basic usage of these commands. Feel free to explore more options using :

```
(gdb) help continue
Continue program being debugged, after signal or breakpoint.
Usage: continue [N]
If proceeding from breakpoint, a number N may be used as an argument,
which means to set the ignore count of that breakpoint to N - 1 (so
that the breakpoint won't break until the Nth time it is reached).
(gdb) help step
Step program until it reaches a different source line.
Usage: step [N]
Argument N means step N times (or till program stops for another
reason).
```

Let's say you've a pointer or an iterator in a function and you obviously do not want to spend time on using the 'n' command until the iterator reaches a value or the loop reaches a certain state. Conditional breakpoints to the rescue!

```
$ gdb -q conditional
Reading symbols from conditional...done.
(gdb) l
1 #include<stdio.h>
2 int main(){
3  int iterator;
4  for(iterator = 0; iterator < 5; iterator++){
5   printf("The current value of iterator is %d\n", iterator);
6  }
7  return 0;
8 }
(gdb) break conditional.c:5 if iterator == 4
Breakpoint 1 at 0x40053e: file conditional.c, line 5.
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x000000000040053e in main at
conditional.c:5
 stop only if iterator == 4
(gdb) run
Starting program: /home/akulkarni/projects/gdb-basics/conditional
The current value of iterator is 0
The current value of iterator is 1
The current value of iterator is 2
The current value of iterator is 3

Breakpoint 1, main () at conditional.c:5
5   printf("The current value of iterator is %d\n", iterator);
```

```
(gdb) print iterator
$1 = 4
```

You may also choose to add a condition to a breakpoint after you've set the breakpoint.

```
$ gdb -q factorial
Reading symbols from factorial...done.
(gdb) b factorial
Breakpoint 1 at 0x40053b: file example.c, line 4.
(gdb) i b
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x000000000040053b in factorial at
example.c:4
(gdb) help condition
Specify breakpoint number N to break only if COND is true.
Usage is `condition N COND', where N is an integer and COND is an
expression to be evaluated whenever breakpoint N is reached.

(gdb) condition 1 (n == 1)
(gdb) i b
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x000000000040053b in factorial at
example.c:4
 stop only if (n == 1)
```

Usually, the header files containing the struct definitions and function signatures are written in separate files. What if we would want to know the type of a variable? What if it is a struct and has more data types? What if we want to know the signature of a function? '*ptype*' to our rescue.

ptype prints the definition of the type specified. Let's say we have the following struct:

```
struct evp_cipher_st {
    int nid;
    int iv_len;
    unsigned long flags;
    int (*ctrl) (int type, int arg);
    /* Application data */
    void *app_data;
};

typedef struct evp_cipher_st EVP_CIPHER;
```

We have some source code that creates a pointer to an instance of type EVP_CIPHER. Give the pointer to the *ptype* command and prepare to be amazed!

```
(gdb) r
Starting program: <path>/ptype-demo

Breakpoint 1, main () at ptype-demo.c:24
24  EVP_CIPHER *evp_cipher = malloc(sizeof(EVP_CIPHER *));
(gdb) n
25  evp_cipher->nid = 120;
(gdb) ptype(evp_cipher)
type = struct evp_cipher_st {
    int nid;
    int iv_len;
    unsigned long flags;
    int (*ctrl)(int, int);
    void *app_data;
} *
(gdb) n
27  evp_cipher->ctrl = controller;
(gdb) ptype(evp_cipher->ctrl)
type = int (*)(int, int)
(gdb) ptype(evp_cipher->app_data)
type = void *
```

Just to complete the basics 😄, here is how we quit gdb:

```
(gdb) quit
```

or

```
(gdb) q
```

Tips:

- To pretty print a data structure, use the following setting of gdb:

  ```
  (gdb) set print pretty on
  ```

- Press return key to execute the previous command automatically. For example: Your previous command was ‘***list***’ and you wish to see the next 10 lines after that. All you have to do is hit the return key.
- Try to use the shorthand notation of commands for productivity. For example, use *’n’* for ‘*next*’, *‘s’* for ‘*step*’, *‘l’* for ‘*list*’, *‘q’* for ‘*quit*’, *‘c’* for ‘*continue’* and so on.

Well, I hope the title was justified and now you are armed with the commands and tools necessary to dissect a C program using the basic commands of gdb.

Hope that helped!

Resources

- A guide to gdb
- GDB Documentation
- GDB Quick Reference Card
- Debugging with gdb — Richard Stallman, Roland Pesch, Stan Shebs, et al.