GCC command in Linux with examples

GCC stands for GNU Compiler Collections which is used to compile mainly C and C++ language. It can also be used to compile Objective C and Objective C++. The most important option required while compiling a source code file is the name of the source program, rest every argument is optional like a warning, debugging, linking libraries, object file etc. The different options of gcc command allow the user to stop the compilation process at different stages.

Syntax:

```
gcc [-c|-S|-E] [-std=standard]
```

Example: This will compile the source.c file and give the output file as a.out file which is default name of output file given by gcc compiler, which can be executed using ./a.out

```
gcc source.c
```

Most Usefull Options with Examples: Here source.c is the C program code file.

-o opt: This will compile the source.c file but instead of giving default name hence executed using ./opt, it will give output file as opt. -o is for output file option.

```
gcc source.c -o opt
```

-Werror: This will compile the source and show the warning if any error is there in the program, -W is for giving warnings.

```
gcc source.c -Werror -o opt
```

-Wall: This will check not only for errors but also for all kinds warning like unused variables errors, it is good practice to use this flag while compiling the code.

```
gcc source.c -Wall -o opt
```

-ggdb3: This command give us permissions to debug the program using gdb which will be described later, -g option is for debugging.

```
gcc -ggdb3 source.c -Wall -o opt
```

-lm : This command link math.h library to our source file, -l option is used for linking particular library, for math.h we use -lm.

```
gcc -Wall source.c -o opt -lm
```

-std=c11 :This command will use the c11 version of standards for compiling the source.c program, which allows to define variable under loop initializations also using newer standards version is preferred.

```
gcc -Wall -std=c11 source.c -o opt
```

-c : This command compile the program and give the object file as output, which is used to make libraries.

-v : This option is used for the verbose purpose.

GDB command in Linux with examples

gdb is the acronym for GNU Debugger. This tool helps to debug the programs written in C, C++, Ada, Fortran, etc. The console can be opened using the gdb command on terminal.

Syntax:

```
gdb [-help] [-nx] [-q] [-batch] [-cd=dir] [-f] [-b bps] [-tty=dev] [-s symfile] [-e prog] [-se prog] [-c core] [-x cmds] [-d dir] [prog[core|procID]]
```

Example:

The program to be debugged should be compiled with -g option. The below given C++ file that is saved as gfg.cpp. We are going to use this file in this article.

```
#include <iostream>

#include <stdlib.h>

#include <string.h>

using namespace std;

int findSquare(int a)

{

   return a * a;
}
int main(int n, char** args)
{
   for (int i = 1; i < n; i++)
   {
      int a = atoi(args[i]);
      cout << findSquare(a) << endl;
   }
   return 0;
}
```

Compile the above C++ program using the command:

```
g++ -g -o gfg gfg.cpp
```

To start the debugger of the above gfg executable file, enter the command gdb gfg. It opens the gdb console of the current program, after printing the version information.

run [args] : This command runs the current executable file. In the below image, the program was executed twice, one with the command line argument 10 and another with the command line argument 1, and their corresponding outputs were printed.

quit or q : To quit the gdb console, either quit or q can be used.

help : It launches the manual of gdb along with all list of classes of individual commands.

break : The command break [function name] helps to pause the program during execution when it starts to execute the function. It helps to debug the program at that point. Multiple breakpoints can be inserted by executing the command wherever necessary. b findSquare command makes the gfg executable pause when the debugger starts to execute the findSquare function.

b

break [function name]

break [file name]:[line number]

break [line number]

break *[address]

break **any of the above arguments** if [condition]

b **any of the above arguments**

In the above example, the program that was being executed(run 10 100), paused when it encountered findSquare function call. The program pauses whenever the function is called. Once the command is successful, it prints the breakpoint number, information of the program counter, file name, and the line number. As it encounters any breakpoint during execution, it prints the breakpoint number, function name with the values of the arguments, file name, and line number. The breakpoint can be set either with the address of the instruction(in hexadecimal form preceded with *0x) or the line number and it can be combined with if condition(if the condition fails, the breakpoint will not be set) For example, break findSquare if a == 10.

continue : This command helps to resume the current executable after it is paused by the breakpoint. It executes the program until it encounters any breakpoint or runs time error or the end of the program. If there is an integer in the argument(repeat count), it will consider it as the continue repeat count and will execute continue command “repeat count” number of times.

continue [repeat count]

c [repeat count]

next or n : This command helps to execute the next instruction after it encounters the breakpoint.

Whenever it encounters the above command, it executes the next instruction of the executable by printing the line in execution.

delete : This command helps to deletes the breakpoints and checkpoints. If the delete command is executed without any arguments, it deletes all the breakpoints without modifying any of the checkpoints. Similarly, if the checkpoint of the parent process is deleted, all the child checkpoints are automatically deleted.

d

delete

delete [breakpoint number 1] [breakpoint number 2] ...

delete checkpoint [checkpoint number 1] [checkpoint number 2] ...

In the above example, two breakpoints were defined, one at the main and the other at the findSquare. Using the above command findSquare breakpoint was deleted. If there is no argument after the command, the command deletes all the breakpoints.

clear : This command deletes the breakpoint which is at a particular function with the name FUNCTION_NAME. If the argument is a number, then it deletes the breakpoint that lies in that particular line.

clear [line number]

clear [FUNCTION_NAME]

In the above example, once the clear command is executed, the breakpoint is deleted after printing the breakpoint number.

disable [breakpoint number 1] [breakpoint number 2] …. : Instead of deleting or clearing the breakpoints, they can be disabled and can be enabled whenever they are necessary.

enable [breakpoint number 1] [breakpoint number 2] …. : To enable the disabled breakpoints, this command is used.

info : When the info breakpoints in invoked, the breakpoint number, type, display, status, address, the location will be displayed. If the breakpoint number is specified, only the information about that particular breakpoint will be displayed. Similarly, when the info checkpoints are invoked, the checkpoint number, the process id, program counter, file name, and line number are displayed.

info breakpoints [breakpoint number 1] [breakpoint number 2] ...

info checkpoints [checkpoint number 1] [checkpoint number 2] ...

checkpoint command and restart command : These command creates a new process and keep that process in the suspended mode and prints the created process's process id.

For example, in the above execution, the breakpoint is kept at function findSquare and the program was executed with the arguments "1 10 100". When the function is called initially with a = 1, the breakpoint happens. Now we create a checkpoint and hence gdb returns a process id(4272), keeps it in the suspended mode and resumes the original thread once the continue command is invoked. Now the breakpoint happens with a = 10 and another checkpoint(pid = 4278) is created. From the info checkpoint information, the asterisk mentions the process that will run if the gdb encounters a continue. To resume a specific process, restart command is used with the argument that specifies the serial number of the process. If all the process are finished executing, the info checkpoint command returns nothing.

set args [arg1] [arg2] … : This command creates the argument list and it passes the specified arguments as the command line arguments whenever the run command without any argument is invoked. If the run command is executed with arguments after set args, the arguments are updated. Whenever the run command is ran without the arguments, the arguments are set by default.

show args : The show args prints the default arguments that will passed if the run command is executed. If either set args or run command is executed with the arguments, the default arguments will get updated, and can be viewed using the above show args command.

display [/format specifier] [expression] and undisplay [display id1] [display id2] … : These command enables automatic displaying of expressions each time whenever the execution encounters a breakpoint or the n command. The undisplay command is used to remove display expressions. Valid format specifiers are as follows:

o - octal

x - hexadecimal

d - decimal

u - unsigned decimal

t - binary

f - floating point

a - address

c - char

s - string

i - instruction

In the above example, the breakpoint is set at line 12 and ran with the arguments 1 10 100. Once the breakpoint is encountered, display command is executed to print the value of i in hexadecimal form and value of args[i] in the string form. After then, whenever the command n or a breakpoint is encountered, the values are displayed again until they are disabled using undisplay command.

print : This command prints the value of a given expression. The display command prints all the previously displayed values whenever it encounters a breakpoint or the next command, whereas the print command saves all the previously displayed values and prints whenever it is called.

print [Expression]

print $[Previous value number]

print {[Type]}[Address]

print [First element]@[Element count]

print /[Format] [Expression]

file : gdb console can be opened using the command gdb command. To debug the executables from the console, file [executable filename] command is used.

GPROF Tutorial – How to use Linux GNU GCC Profiling Tool

Profiling is an important aspect of software programming. Through profiling one can determine the parts in program code that are time consuming and need to be re-written. This helps make your program execution faster which is always desired.

In very large projects, profiling can save your day by not only determining the parts in your program which are slower in execution than expected but also can help you find many other statistics through which many potential bugs can be spotted and sorted out.

In this article, we will explore the GNU profiling tool ‘gprof’.

How to use gprof

Using the gprof tool is not at all complex. You just need to do the following on a high-level:

Have profiling enabled while compiling the code

Execute the program code to produce the profiling data

Run the gprof tool on the profiling data file (generated in the step above).

The last step above produces an analysis file which is in human readable form. This file contains a couple of tables (flat profile and call graph) in addition to some other information. While flat profile gives an overview of the timing information of the functions like time consumption for the execution of a particular function, how many times it was called etc. On the other hand, call graph focuses on each function like the functions through which a particular function was called, what all functions were called from within this particular function etc So this way one can get idea of the execution time spent in the sub-routines too.

Step-1 : Profiling enabled while compilation

In this first step, we need to make sure that the profiling is enabled when the compilation of the code is done. This is made possible by adding the ‘-pg’ option in the compilation step.

From the man page of gcc :

-pg : Generate extra code to write profile information suitable for the analysis program gprof. You must use this option when compiling the source files you want data about, and you must also use it when linking.

So, lets compile our code with ‘-pg’ option :

```
$ gcc -Wall -pg test_gprof.c test_gprof_new.c -o test_gprof

$
```

Please note : The option ‘-pg’ can be used with the gcc command that compiles (-c option), gcc command that links(-o option on object files) and with gcc command that does the both(as in example above).

Step-2 : Execute the code

In the second step, the binary file produced as a result of step-1 (above) is executed so that profiling information can be generated.

```
$ ls

test_gprof  test_gprof.c  test_gprof_new.c

$ ./test_gprof  Inside main()

 Inside func1

 Inside new_func1()

 Inside func2

$ ls

gmon.out  test_gprof  test_gprof.c  test_gprof_new.c

$
```

So we see that when the binary was executed, a new file ‘gmon.out’ is generated in the current working directory.

Note that while execution if the program changes the current working directory (using chdir) then gmon.out will be produced in the new current working directory. Also, your program needs to have sufficient permissions for gmon.out to be created in current working directory.

Step-3 : Run the gprof tool

In this step, the gprof tool is run with the executable name and the above generated ‘gmon.out’ as argument. This produces an analysis file which contains all the desired profiling information.

```
$  gprof test_gprof gmon.out > analysis.txt
```

Note that one can explicitly specify the output file (like in example above) or the information is produced on stdout.

```
$ ls
```

```
analysis.txt  gmon.out  test_gprof  test_gprof.c  test_gprof_new.c
```

So we see that a file named 'analysis.txt' was generated.

So (as already discussed) we see that this file is broadly divided into two parts :

1. Flat profile

2. Call graph

The individual columns for the (flat profile as well as call graph) are very well explained in the output itself.

Customize gprof output using flags

There are various flags available to customize the output of the gprof tool. Some of them are discussed below:

1. Suppress the printing of statically(private) declared functions using -a

If there are some static functions whose profiling information you do not require then this can be achieved using -a option :

```
$ gprof -a test_gprof gmon.out > analysis.txt
```

2. Suppress verbose blurbs using -b

As you would have already seen that gprof produces output with lot of verbose information so in case this information is not required then this can be achieved using the -b flag.

```
$ gprof -b test_gprof gmon.out > analysis.txt
```

3. Print only flat profile using -p

In case only flat profile is required then :

```
$ gprof -p -b test_gprof gmon.out > analysis.txt
```

4. Print information related to specific function in flat profile

This can be achieved by providing the function name along with the -p option:

```
$ gprof -pfunc1 -b test_gprof gmon.out > analysis.txt
```

5. Suppress flat profile in output using -P

If flat profile is not required then it can be suppressed using the -P option :

```
$ gprof -P -b test_gprof gmon.out > analysis.txt
```

6. Print only call graph information using -q

```
gprof -q -b test_gprof gmon.out > analysis.txt
```

7. Print only specific function information in call graph.

This is possible by passing the function name along with the -q option.

```
$ gprof -qfunc1 -b test_gprof gmon.out > analysis.txt
```

8. Suppress call graph using -Q

If the call graph information is not required in the analysis output then -Q option can be used

```
$ gprof -Q -b test_gprof gmon.out > analysis.txt
```