Birzeit University - Faculty of Engineering & Technology
Electrical & Computer Engineering Department - ENCS313
Linux laboratory

Experiment #5
Compilation, Debugging, Project Management
Part II

0.1 Introduction

The experiment intends to present the `gcc` compiler, the `gdb` debugger and show students how to use `makefile`s to better manage projects. Students are assumed to be already familiar with the C-language. Students will be shown how to compile source code and create executables using the `gcc` compiler. Moreover, students will be shown how to create libraries (both static libraries and dynamic libraries) using the archive command `ar` and how to link to these libraries during the compilation phase. They will be shown how to use the `gdb` debugger in order to debug code and discover defects (or potential defects). Finally, students will be shown how to use a `makefile` that will help better manage a development project during compilation, setup and installation.

0.2 Objectives

The objectives of the experiment is to learn the following:

- Give a quick overview of the GNU project.
- Show students on how to use the `gcc` compiler and the `ar` archive command.
- Show students on how to use the `gdb` debugger.
- Show students on how to use a `makefile` to manage a development project.

0.3 The GNU debugger `gdb`

The GNU Debugger, usually called just `gdb` is the standard debugger for the GNU software system. It is a portable debugger that runs on many Unix-like systems and works for many programming languages, including Ada, C, C++ and Fortran.

`gdb` was first written by Richard Stallman in 1986 as part of his GNU system.

`gdb` offers extensive facilities for tracing and altering the execution of computer programs. The user can monitor and modify the values of programs' internal variables, and even call functions independently of the program's normal behavior.

`gdb` is still actively developed. As of version 7.0 new features include support for Python scripting, and "reversible debugging", allowing a debugging session to step backward, much like rewinding a crashed program to see what happened. The debugger does not contain its own graphical user interface, and defaults to a command-line interface. Several front-ends have been built for it, such as `xxgdb`[1] or `ddd` debuggers.

[1] http://en.wikipedia.org/wiki/GNU_Debugger.

0.3.1 Building the symbols table

In order to debug a program effectively, you need to generate debugging information when you compile it. This debugging information is stored in the object file; it describes the data type of each variable or function and the correspondence between source line numbers and addresses in the executable code. The debugging information will be stored in a *symbols table* that will be used by `gdb`. The symbols table contains names of variables, functions and types defined in your program. This information is inherent in the text of your program and does not change as your program executes.

To create the symbols table, you need to compile your C-files using the `gcc` compiler and use the option `g` as follows:

`gcc -g myFile.c -o myFile`.

Consider the following example:

```
#include <stdio.h>

/* print the sum of 2 variables */

main()
{
  int s1, s2, s3, s4, res1, res2;

  s1 = 1; s2 = 2;
  s3 = 3; s4 = 4;

  printf("res1 = %d\n", s1 + s2);
  printf("res2 = %d\n", s3 + s4);
}
```

Save the above code in a file called `gdb_1.c` and compile it as follows:

`gcc -g gdb_1.c -o gdb_1`

An executable named `gdb_1` will be created. Run the command:

`ls -l gdb_1`

and note the size of the executable.

Now compile the file `gdb_1.c` as follows:

`gcc gdb_1.c -o gdb_1`

and note again the new size of the executable `gdb_1`.

You will notice that the executable you get when using the option `-g` with `gcc` is bigger in size than the executable you get without using the `-g` option. The reason is that a symbols table that will be needed by the `gdb` debugger will be created in the first case.

Note:

If you invoke the `gdb` compiler on an executable that has not been generated using the `-g` option with the `gcc` compiler, a warning stating that *symbols table not found* will be displayed.

0.3.2 Invoking `gdb`

The most usual way to start `gdb` is with one argument, specifying an executable program as follows[2]:

[2] http://sourceware.org/gdb/current/onlinedocs/gdb.html

`gdb program`

To quit the debugger, type `quit`, `q` (for short) or `Ctrl-d`.

If you invoked the `gdb` debugger and need some help, type the command `help` to get assistance.

To run your executable `gdb_1` under `gdb`, execute the following command:

`gdb gdb_1`

followed by the command:

`run`

Your executable `gdb_1` will run until either completion or until an error is encountered. Note that your executable has run without arguments since it didn't need any. If your executable requires arguments, you can specify them after the command `run` as:

`run arg1 arg2 ...`

Consider the following C-code:

```
#include <stdio.h>

main(int argc, char *argv[])
{
  int i;

  i = argc;

  while ( i-- )
    printf("arg %d: %s\n", i, argv[i]);
}
```

Save the above code in a file called `gdb_2.c` and compile it using the `-g` option of the `gcc` compiler to create the executable `gdb_2`.

To run your executable `gdb_2` under `gdb`, execute the following command:

`gdb gdb_2`

followed by the command:

`run arg1 arg2 arg3`

Your executable `gdb_2` will run using the specified arguments until either completion or until an error is encountered.

0.3.3 Stopping and Continuing

The principal purposes of using a debugger are so that you can stop your program before it terminates; or so that, if your program runs into trouble, you can investigate and find out why.

Breakpoints

A *breakpoint* makes your program stop whenever a certain point in the program is reached. Under `gdb`, a breakpoint may be invoked using the command `break` or `b` for short as follows:

`break location`

or

`b location`

where `location` can specify a function name (e.g. `break coco`) or a line number (e.g. `break 120`).

A breakpoint may be removed (deleted) using the `delete` command (or `d` for short) as follows:

`delete [breakpoint number]`

or

`d [breakpoint number]`

where `[breakpoint number]` is an integer representing the breakpoint number.

Note 1:

If no argument is specified to the command `delete`, *all* breakpoints will be deleted.

Note 2

Rather than deleting a breakpoint, you might prefer to *disable* it. This makes the breakpoint inoperative as if it had been deleted, but remembers the information on the breakpoint so that you can *enable* it again later. Breakpoints can be disabled as follows:

`disable [breakpoints]`

or

`disable [range...]`

Breakpoints can be enabled as follows:

`enable [breakpoints]`

or

`enable [range ...]`

Continuing

Continuing means resuming program execution until your program completes normally, encounters a breakpoint or encounters an error. Under `gdb`, you can resume execution using the command `continue` or `cont` or even `c` for short.

Stepping

Stepping means executing just one more *step* of your program, where *step* may mean either one line of source code, or one machine instruction. Either when continuing or when stepping, your program may stop even sooner, due to a breakpoint or a signal.

Under `gdb`, you can employ stepping using the command `step`. If the command `step` encounters a function, it will step into it.

Note 1:

If you use the `step` command while control is within a function that was compiled without debugging information, execution proceeds until control reaches a function that does have debugging information. Likewise, it will not step into a function which is compiled without debugging information.

Note 2:

You can use the `gdb` command `next` or `n` for short in order to continue execution to the next source line in the current (innermost) stack frame. This is similar to the `step` command, but function calls that appear within the line of code are executed without stopping. Execution stops when control reaches a different line of code at the original stack level that was executing when you gave the `next` command.

Note 3:

The commands `continue`, `step` and `next` can take an optional arguemnt `[count]`. For example, invoking:

`cont [count]`

or

`next [count]`

or

`step [count]`

will continue running the application `count` times. If a breakpoint is reached, or a signal not related to stepping occurs before `count` steps, stepping stops right away.

0.3.4 Printing source lines

To print lines from a source file, use the `list` command (or `l` for short). By default, the command `list` prints 10 source lines. The command can be used as follows:

`list linenum`

The above command prints lines centered around line number `linenum` in the current source file. You can also use the command `list` as follows:

`list first,last`

The above command prints lines from `first` to `last`. Both arguments are linespecs.

You can also use the command `list` on functions as follows:

`list function`

which prints lines centered around the beginning of function `function`.

0.3.5 Specifying source directories

The debugger `gdb` needs to know where the source files are located while debugging an executable, in particular if you intend to use the command `list`. Keep in mind that in real-life projects, the source files might be scattered into multiple directories. If you need `gdb` to be able to list functions and lines that are located in scattered source files, use the command `directory` or (`dir` for short) as follows:

`directory dirname`.

0.3.6 Printing program variables

You may use the command `print` (or `p` for short) to print the value of varibales. Note that you can only print variables within the scope of the function you're currently in.

If you find that you want to `print` the value of an expression frequently (to see how it changes), you might want to add it to the *automatic display list* so that `gdb` prints its value each time your program stops. Each expression added to the list is given a number to identify it; to remove an expression from the list, you specify that number.

To display program variables, use the command `display` as follows:

`display expr`

The above command adds the expression `expr` to the list of expressions to display each time your program stops.

To remove the the display for a particular variable, use the command `undisplay` as follows:

`undisplay dnums`

The above command removes item numbers `dnums` from the list of expressions to display.

Instead of removing displays, you might choose to disable and re-enable them as needed. Displays

can be disabled as follows:

```
disable display dnums
```

The above command disables the display of item numbers `dnums`. A disabled display item is not printed automatically, but is not forgotten. It may be enabled again later.

Displays can be enables as follows:

```
enable display dnums
```

The above command enables the display of item numbers `dnums`. It becomes effective once again in auto display of its expression, until you specify otherwise.

0.3.7 Altering Execution

Once you think you have found an error in your program, you might want to find out for certain whether correcting the apparent error would lead to correct results in the rest of the run. You can find the answer by experiment, using the `gdb` features for altering execution of the program[3].

Assignment to variables

You can alter the value of a variable as follows:

```
print x=4
```

In the above example, you have stored the value 4 into the variable `x`, and then prints the value of the assignment expression (which is 4).

If you are not interested in seeing the value of the assignment, use the `set` command instead of the `print` command as follows:

```
set x=4
```

Returning from a function

You can cancel execution of a function call with the `return` command (without an argument) or with the `return expression` command (with an argument). You can think of this as making the discarded frame return prematurely.

The `return` command does not resume execution; it leaves the program stopped in the state that would exist if the function had just returned. In contrast, the `finish` command resumes execution until the selected stack frame returns naturally.

Calling program functions

You can use the command `call` if you want to execute a particular function from your program. For example, you can execute the function `drawCircle` with center at `x = 0, y = 0` and radius `r = 5` as follows:

```
call drawCircle(0, 0, 5)
```

That might be a good debugging technique if you want to experiment calling functions with specific arguments or calling certain functions before others.

0.3.8 User-defined commands

A *user-defined command* is a sequence of `gdb` commands to which you assign a new name as a command. The syntax is as follows:

```
define commandname
```

For example, you can define the following command:

[3] http://www.delorie.com/gnu/docs/gdb

```
define myPrint
set x=4
set y=4
set r=5
call drawCircle(x, y, r)
end
```

Whenever you need to call the function `drawCircle` with arguments `x = 4`, `y = 4` and `r = 5`, you can call the user-defined command `myPrint`.

User commands may accept as well up to 10 arguments separated by whitespace. Arguments are accessed within the user command via `$arg0...$arg9`. For example, you can define the following command:

```
define adder
print $arg0 + $arg1 + $arg2
```

To execute the command use:

```
adder 1 2 3
```

in order to add the 3 supplied arguments.

Note:

If the user-defined command contains one line only (as in the definition of the command `adder` above), there is no need to terminate the command by the keyword `end`.

0.3.9 Command files

A command file for `gdb` is a file of lines that are `gdb` commands. Comments (lines starting with #) may also be included. An empty line in a command file does nothing.

Using command files spare users from having to define their commands every time they call the `gdb` debugger and thus are very useful.

Command files for `gdb` are generally called `.gdbinit` and should be located in the same folder where you intend to call `gdb`. Once `gdb` is invoked, the file `.gdbinit` is read automatically.

An example of a `.gdbinit` file is shown below:

```
# In the below line, we specify the folders that contain source files
dir /home/user/application_1
dir /home/user/application_2

# Listing the function drawCircle and adding a breakpoint
list drawCircle
break 120

# Adding a breakpoint at function drawRectangle
break drawRectangle

# Defining the command myPrint
define myPrint
set x=4
set y=4
set r=5
call drawCircle(x, y, r)
set u=0
```

```
set v=0
call drawRectangle(u, v, x, y)
end

# Defining the command adder
define adder
print $arg0 + $arg1 + $arg2
```

If you want to bypass the usage of the initialization file `.gdbinit`, you can invoke the `gdb` debugger with the `-nx` option as follows:

```
gdb -nx
```

Note:

If you have multiple initialization files for `gdb` and you want to use specific files at different invocations of `gdb`, you can do the following:

```
gdb -nx     # Do not use the .gdbinit file
source "initialization_file_of_your_choice"
```

0.3.10 Debugging an already-running process

If you want to debug an already running process, one that was started outside `gdb`, you can attach it to `gdb` as follows:

`gdb` *name-of-executable* `-pid` *process-id*

Alternatively, you can attach a running process inside the `gdb` debugger as follows:

`attach` *process-id*

The command takes as argument a process ID (that you can get using the shell command `ps -ef`.

When you use the `attach` command, the debugger finds the running process and stops it. You can examine and modify an attached process with all the `gdb` commands that are ordinarily available when you start processes with command `run`. You can insert breakpoints; you can step and continue; you can modify storage. If you would rather the process continue running, you may use the `continue` command after attaching `gdb` to the process.

When you have finished debugging the attached process, you can use the `detach` command to release it from `gdb` control. Detaching the process continues its execution. After the `detach` command, that process and `gdb` become completely independent once more, and you are ready to `attach` another process or start one with command `run`.

Note:

If you exit `gdb` or use the `run` command while you have an attached process, you kill that process. By default, `gdb` asks for confirmation if you try to do either of these things; you can control whether or not you need to confirm by using the `set confirm` command as follows:

```
set confirm on
```

The above command enables confirmation requests (the default). Or you can disable confirmation requests as follows:

```
set confirm off
```

You can use as well the following command to display the state of confirmation requests:

```
show confirm
```

0.3.11 Using `ddd` interface to `gdb`

The `ddd` utility (Data Display Debugger) shown in Figure 1 below is a graphical front-end for the `gdb` debugger and other command-line debuggers. It provides basically the same functionality as what you get under the `gdb` debugger. However, it allows the usage of the mouse and provides the main `gdb` commands as push buttons where users can click instead of typing them down. Since most of the students come from microsoft Windows environment background, they tend to feel more at ease dealing with graphical interfaces instead of command-line tools. However, keep in mind that the `gdb` debugger is a standard tool while the `ddd` is not.

You need first to make sure that the `ddd` interface is installed on your machine. Execute:

```
which ddd
```

and check on the output that you get. If `ddd` is installed, the output of the above command should probably be:

```
/usr/bin/ddd
```

If `ddd` is not installed, you can install it as follows (you need to switch to `root` user to do that):

```
yum install ddd
```

The above command installs `ddd` utility on Fedora systems. Those who have Ubuntu installed on their machines can replace the command `yum` by the command `apt-get`

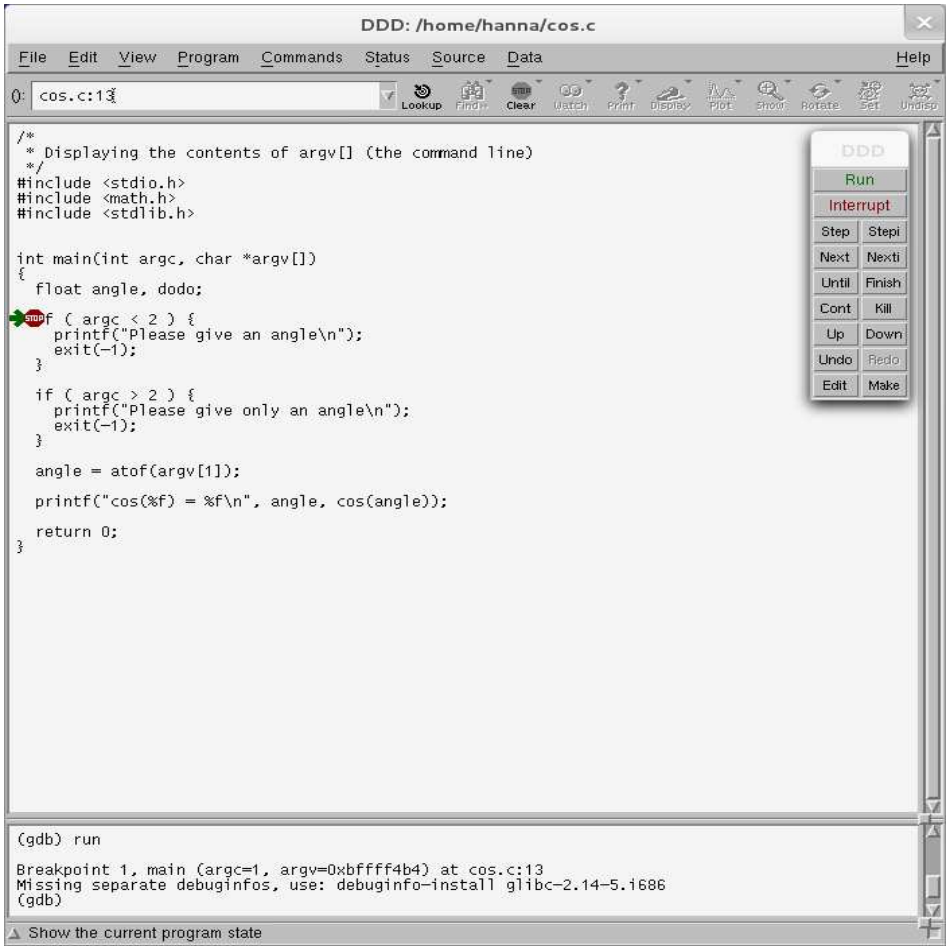

Figure 1: The `ddd` debugger interface.

To invoke the `ddd` utility, you execute:

```
ddd executableName
```

where `executableName` is the name of the executable you want to debug. You can place break-

points using the mouse by placing the cursor on the line you intend to place the breakpoint at and then clicking on the `Break` push button (placed in the upper menu with a stop sign). You can place as many breakpoints as you need.

To run a program, you can click on the `Run` push button (placed on top of the menu that is on the right side of the `ddd` application). If running your program requires arguments, then you can type the command `run` followed by your arguments in the lower small window of the `ddd` utility. You will notice that the program execution will stop at the first encountered breakpoint and a green arrow gets displayed to the left of the breakpoint `Stop` sign as shown in Figure 1.

You can as well continue the execution of the program using the `Cont` push button, step in a particular function using the `Step` push button, print the value of variables by first selecting them with the mouse and then clicking the `Print` push button and so on.

Using the `ddd` utility should be intuitive enough so students should be able to find out their way on their own.

0.3.12 Debugging programs with multiple processes

On most systems, `gdb` has no special support for debugging programs which create additional processes using the `fork` function. When a program forks, `gdb` will continue to debug the parent process and the child process will run unimpeded. If you have set a breakpoint in any code which the child then executes, the child will get a `SIGTRAP` signal which (unless it catches the signal) will cause it to terminate.

However, if you want to debug the child process there is a workaround. Put a call to `sleep` in the code which the child process executes after the `fork`. While the child is sleeping, use the `ps` command to get its process ID. Then tell `gdb` (a new invocation of `gdb` if you are also debugging the parent process) to attach to the child process. From that point on you can debug the child process just like any other process which you attached to.

By default, when a program forks, `gdb` will continue to debug the parent process and the child process will run unimpeded. If you want to follow the child process instead of the parent process, use the command `set follow-fork-mode` as follows:

```
set follow-fork-mode child
```

The above command sets the debugger response to follow the new process after `fork`. The parent process runs unimpeded.

If you intend to follow the parent process after `fork` is invoked (which is the default), use the following command:

```
set follow-fork-mode parent
```

If you need to display the current debugger response to a `fork` or `vfork` call, use the following command:

```
show follow-fork-mode
```

0.3.13 Debugging programs with multiple threads

The `gdb` debugger provides the facilities for debugging multi-thread programs. For example, users get automatic notification of new threads once it enters the system.

You can inquire about existing threads by using the command:

```
info threads
```

In addition, you can switch between threads by calling the command:

```
thread threadno
```

where `threadno` is the number of thread you intend to switch to. You can get the number for each thread by calling the command `info threads`. The `gdb` debugger will display the message `Switching to ...` to indicate that it switched focus to a new thread.

Note 1:

The `gdb` thread debugging facility allows you to observe all threads while your program runs. However, whenever `gdb` takes control, one thread in particular is always the focus of debugging. This thread is called the *current thread*. Debugging commands show program information from the perspective of the current thread.

Note 2:

An asterisk `'*'` to the left of the `gdb` thread number indicates the current thread.

Note 3:

Whenever `gdb` stops your program, due to a breakpoint or a signal, it automatically selects the thread where that breakpoint or signal happened. `gdb` alerts you to the context switch with a message of the form `'[Switching to ...]'` to identify the thread.

Note 4:

Whenever your program stops under `gdb` for any reason, *all* threads of execution stop, not just the current thread. This allows you to examine the overall state of the program, including switching between threads, without worrying that things may change underfoot. Conversely, whenever you restart the program, *all* threads start executing.

As an example on debugging a multi-threading application, consider the following code:

```
#include <stdio.h>       /* standard I/O routines                 */
#include <pthread.h>     /* pthread functions and data structures */

/* function to be executed by the new thread */
void * do_loop(void* data)
{
    int i; /* counter, to print numbers */
    int j; /* counter, for delay        */
    int me = *((int*)data);     /* thread identifying number */

    for (i = 0; i < 10; i++) {
      for (j = 0; j < 500000; j++); /* delay loop */
        printf("'%d' - Got '%d'\n", me, i);
    }

    /* exit the thread */
    pthread_exit(NULL);
}

/* like any C program, program's execution begins in main */
int main(int argc, char* argv[])
{
    int        thr_id;         /* thread ID for the newly created thread */
    pthread_t  p_thread;       /* thread's structure                     */
    int        a         = 1;  /* thread 1 identifying number            */
    int        b         = 2;  /* thread 2 identifying number            */

    /* create a new thread that will execute 'do_loop()' */
    thr_id = pthread_create(&p_thread, NULL, do_loop, (void*) &a);
```

```
    /* run 'do_loop()' in the main thread as well */
    do_loop((void*) &b);

    return 0;
}
```

Save the above code in a file called `pthread_create.c` and compile it as follows:

```
gcc -g pthread_create.c -o pthread_create -lpthread
```

The linking with the POSIX threads library (`lpthread`) is necessary to get thread-spcific functions.

Run the `gdb` debugger on the executable `pthread_create` and place 2 breakpoints, the first at line:

```
thr_id = pthread_create(&p_thread, NULL, do_loop, (void*) &a);
```

and the second one at function `do_loop`.

Afterwards, execute the command `run` followed by the command `info threads` once execution stops at the first breakpoint. Note the number of threads currently available and which thread has currently the focus.

Execute the command `next` and note the messages that get displayed by the debugger. Note the new number of threads by running the command `info threads`.

You can now move focus between the main thread and the newly created thread using the command:

```
thread threadno
```

where `threadno` can be either `2` (thread `2`) or `3` (thread `3`). You'll notice that the local variable `me` in function `dp_loop` changes between the values `1` and `2` depending on the thread of focus.

0.4 Profiling with `gprof`

The command `gprof` is a profiling program which collects and arranges statistics on your programs. Basically, it looks into each of your functions and inserts code at the head and tail of each one to collect timing information. Then, when you run your program normally, it creates a file called `gmon.out` which contains raw data that the `gprof` program turns into profiling statistics[4].

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[5].

In this section we'll show you how to use the program `gprof`.

0.4.1 How to use gprof

To use `gprof`, you need to do the following:

- 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

[4] http://www.cs.duke.edu/ ola/courses/programming/gprof.html
[5] http://www.thegeekstuff.com/2012/08/gprof-tutorial/

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.

The examples seen below have been extracted from the site shown in the legend [6]. You are thus invited to visit that site for more information.

Consider the following example:

```
//test_gprof.c
#include<stdio.h>

void new_func1(void);

void func1(void)
{
    printf("\n Inside func1 \n");
    int i = 0;

    for(;i<0xffffffff;i++);
    new_func1();

    return;
}

static void func2(void)
{
    printf("\n Inside func2 \n");
    int i = 0;

    for(;i<0xfffffffaa;i++);
    return;
}

int main(void)
{
    printf("\n Inside main()\n");
    int i = 0;

    for(;i<0xffffff;i++);
    func1();
    func2();

    return 0;
}
```

Save the above code in a file called `test_gprof.c`.

```
//test_gprof_new.c
#include<stdio.h>
```

[6] http://www.thegeekstuff.com/2012/08/gprof-tutorial/

```
void new_func1(void)
{
    printf("\n Inside new_func1()\n");
    int i = 0;

    for(;i<0xffffffee;i++);

    return;
}
```

Save the above code in a file called `test_gprof_new.c`.

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.

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

The option `-pg` instructs the compiler to 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.

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.

```
./test_gprof
```

You'll get the following output

```
 Inside main()

 Inside func1

 Inside new_func1()

 Inside func2
```

You should notice at this point that a file called `gmon.out` has been created.

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
```

Comprehending the profiling information

The file `analysis.txt` that was generated in the previous step contains roughly 2 parts:

- Flat profile
- Call graph

Execute the command `cat analysis.txt` and check that you get an output similar to below:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 35.87     21.21    21.21        1    21.21    21.21  new_func1
 35.73     42.34    21.13        2    10.56    21.17  func1
 28.21     59.02    16.68                              func2
  0.19     59.13     0.11                              main

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
   else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
   function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
   the function in the gprof listing. If the index is
   in parenthesis it shows where it would appear in
   the gprof listing if it were to be printed.

     Call graph (explanation follows)


granularity: each sample hit covers 4 byte(s) for 0.02% of 59.13 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]     71.8    0.11   42.34                 main [1]
               21.13   21.21       2/2           func1 [2]
-----------------------------------------------
               21.13   21.21       2/2           main [1]
[2]     71.6   21.13   21.21       2         func1 [2]
               21.21    0.00       1/1           new_func1 [3]
-----------------------------------------------
               21.21    0.00       1/1           func1 [2]
[3]     35.9   21.21    0.00       1         new_func1 [3]
```

```
-----------------------------------------------
                                                 <spontaneous>
[4]     28.2   16.68    0.00                  func2 [4]
-----------------------------------------------

 This line lists:
     index A unique number given to each element of the table.
Index numbers are sorted numerically.
The index number is printed next to every function name so
it is easier to look up where the function in the table.

     % time This is the percentage of the 'total' time that was spent
in this function and its children.  Note that due to
different viewpoints, functions excluded by options, etc,
these numbers will NOT add up to 100%.

     self This is the total amount of time spent in this function.

     children This is the total amount of time propagated into this
function by its children.

     called This is the number of times the function was called.
If the function called itself recursively, the number
only includes non-recursive calls, and is followed by
a '+' and the number of recursive calls.

     name The name of the current function.  The index number is
printed after it.  If the function is a member of a
cycle, the cycle number is printed between the
function's name and the index number.

...
...

Index by function name

   [2] func1                   [1] main
   [4] func2                   [3] new_func1
```

As you can notice, the file contains description about the different columns that you get in the file `analysis.txt`. In case all the verbose information contained in the `analysis.txt` is not required, you can suppress it by using the `-b` option of the command `gprof` as follows:

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

Execute the command `man gprof` to get to know the different options that it offers.