Logging gdb Into a Text File

When dealing with a long stack trace or multi-thread stack trace, viewing and analyzing gdb output from a Terminal window can be inconvenient. However, we can log either an entire session or specific output into a text file first, then browse it offline later using other text editor tools. To do this, we need to use the following command:

```
(gdb) set logging on
```

When we execute this command, `gdb` will save all the Terminal window outputs into a text file named `gdb.txt` in the currently running `gdb` folder. If we want to stop logging, we can just type the following:

```
(gdb) set logging off
```

One great thing about GDB is that we can turn set logging commands on and off as many times as we want, without worrying about the dumped file names. This is because all the outputs are concatenated into the `gdb.txt` file.

Here is an example of returning `ch14_gdb_2.out` with the `gdb` output being dumped:

```
~/wus1/Chapter-13$ gdb ch14_gdb_2.out                    //cmd 1
...
Reading symbols from ch14_gdb_2.out...done.
(gdb) set logging on                                     //cmd 2
Copying output to gdb.txt.
(gdb) b ch14_gdb_2.cpp:24 if i==1                        //cmd 3
Breakpoint 1 at 0xa84: file ch14_gdb_2.cpp, line 24.
(gdb) r                                                  //cmd 4
...
Breakpoint 1, dotproduct (x=0x7fffffed68, y=0x7fffffed68, n=5) at
ch14_gdb_2.cpp:24
24 s += (*p) * (*q);
(gdb) p i                                                //cmd 5
$1 = 1
(gdb) p s                                                //cmd 6
$2 = 1
(gdb) finish                                             //cmd 7
Run till exit from #0 dotproduct (x=0x7fffffed68, y=0x7fffffed68,
n=5) at
ch14_gdb_2.cpp:24
0x00000055555559e0 in main () at ch14_gdb_2.cpp:11
11 sxx = dotproduct( x, x, 5);
Value returned is $3 = 55
(gdb) delete breakpoints 1                               //cmd 8
(gdb) set logging off                                    //cmd 9
Done logging to gdb.txt.
(gdb) c                                                  //cmd 10
Continuing.
dot(x,x) = 55.000000
dot(x,y) = 55.000000
[Inferior 1 (process 386) exited normally]
(gdb) q                                                  //cmd 11
~/wus1/Chapter-13$ cat gdb.txt                           //cmd 12
```

The commands that were used in the preceding code are as follows:

- `cmd 1`: `gdb` is launched.
- `cmd 2`: We set the logging flag to on. At this point, `gdb` says the output will be copied into the `gdb.txt` file.
- `cmd 3`: A conditional `break point` is set.

- `cmd 4`: We run the program, and it stops when it reaches the conditional `breakpoint` in file `ch14_gdb_2.cpp`, *line 24*.
- `cmd 5` and `cmd 6`: We print the values of `i` and `s`, receptively.
- `cmd 7`: By executing the step out of the function command, it shows that `sxx` is `55` (after calling `sxx=dotproduct( x, x, 5)`) and that the program stops at line `sxy=dotproduct( x, y, 5)`.
- `cmd 8`: We delete `breakpoint` 1.
- `cmd 9`: We set the logging flag to off.
- `cmd 10`: Once a continue instruction is given, it runs out of the main function and gdb waits for a new command.
- `cmd 11`: We input q to quit `gdb`.
- `cmd 12`: When it goes back to the Terminal window, we print the content of the logged `gdb.txt` file by running the cat command in the OS.

So far, we have learned enough GDB commands to debug a program. As you may have noticed, it's time-consuming and thus very costly. Sometimes, it becomes even worse because of debugging in the wrong places. To debug efficiently, we need to follow the right strategies. We will cover this in the following subsection.