GDB (the GNU Debugger) is a debugging tool that provides many useful operations to help us debug programs, like pausing the program, inspecting its state, executing single lines at a time, and more.

Following along

All of the examples and GDB output snippets in this document are based on the C program located at `~uqjfenw1/public/debug/sample.c` unless otherwise specified. It is recommended that you build this program and follow along with the commands shown here in your own GDB session. A line in this guide starting with a `$` and written in `monospace font` is a command on a shell prompt, and a line beginning with `(gdb)` is a command on the GDB prompt. GDB and program output is also given in `monospace`, with extra line numbers and colour added for clarity, though note that some GDB output also includes line numbers; the GDB output is the text to the right of the first line numbers and the vertical bar.

Running GDB

A program must be compiled using the `-g` flag for GDB to be able to associate the program's execution with its source code.

```
$ gdb ./path/to/executable
```

will start GDB with your executable, which will not start running automatically.

```
 1 | GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-94.el7
 2 | Copyright (C) 2013 Free Software Foundation, Inc.
 3 | License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
 4 | This is free software: you are free to change and redistribute it.
 5 | There is NO WARRANTY, to the extent permitted by law. Type "show copying"
 6 | and "show warranty" for details.
 7 | This GDB was configured as "x86_64-redhat-linux-gnu".
 8 | For bug reporting instructions, please see:
 9 | <http://www.gnu.org/software/gdb/bugs/>...
10 | Reading symbols from /home/you/path/to/sample...done.
11 | (gdb)
```

Lines 1–9 are GDB's version and licence information. Line 10 tells us where GDB has read debug symbols from, and will be the full path to the program we ran GDB with. Line 11 is the GDB prompt, and is where we type our commands.

Starting our program

To start our program running under GDB, we use the `run [args]` command. The arguments are the command-line arguments to our program, written exactly as though we were running it in a shell.

```
11 | (gdb) run
12 | Starting program: /home/you/path/to/sample
13 | The arguments are the same backwards and forwards
14 | [Inferior 1 (process 21095) exited normally]
15 | (gdb)
```

Line 12 tells us that our program is starting, and repeats the path to our executable. Line 13 is our program's output, and will be printed until our program ends, crashes, or is paused by GDB. Line 14 is printed after our program has ended, and tells us its process ID and that it exited normally.

```
15 | (gdb) run a
16 | Starting program: /home/you/path/to/sample a
17 |
18 | Program received signal SIGSEGV, Segmentation fault.
```

```
19  0x0000000000400594 in compare_chars (c1=0x7fffffffe402 "a", c2=0x0) at sample.c:6
20  6           return *c1 == *c2;
21  (gdb)
```

Line 18 tells us that our program crashed due to a segmentation fault. Lines 19–20 give us the exact location of the segfault. We can see that this one happened on line 6 of the file `sample.c`, in the function `compare_chars`. This line is then printed for our convenience (including the line number).

Viewing code and state

If we want to more of the surrounding code, we can use the `list` command (or `l` for short). If the program is paused this will print the 10 lines around the program's current location, unless we explicitly provide a line number (`l 6`) or file and line (`l sample.c:6`). If we then repeat the `l` command, it will print out the next 10 lines of code.

```
21  (gdb) l 6
22  1       #include <stdio.h>
23  2       #include <string.h>
24  3       #include <stdbool.h>
25  4
26  5       bool compare_chars(char *c1, char *c2) {
27  6           return *c1 == *c2;
28  7       }
29  8
30  9       bool compare_words(char *w1, char *w2) {
31  10          for (int i = 0; i < strlen(w1); i++) {
32  (gdb) l
33  11              if (!compare_chars(w1 + i, w2 + i)) {
34  12                  return false;
35  13              }
36  14          }
37  15
38  16          return true;
39  17      }
40  18
41  19      bool check_words(char *words[], int numWords) {
42          20          bool mirrored = true;
43  (gdb)
```

Lines 22-31 & 33-42 show the source code of our program that we requested (include its line numbers).

If our program crashes, GDB pauses its execution at the point of the crash. When our program is paused, we can use GDB to inspect the state of our program, including its call stack and variables, and even call functions from our program.

The call stack

The command to view the call stack is `backtrace` (`bt`) Our program has just crashed with a segfaut, so let's examine its call stack.

```
43  (gdb) bt
44  #0  0x0000000000400594 in compare_chars (c1=0x7fffffffe402 "a", c2=0x0) at sample.c:6
45  #1  0x00000000004005dd in compare_words (w1=0x7fffffffe402 "a", w2=0x0) at sample.c:11
46  #2  0x0000000000400687 in check_words (words=0x7fffffffe0e0, numWords=2) at sample.c:29
47  #3  0x00000000004006f4 in main (argc=2, argv=0x7fffffffe0d8) at sample.c:40
48  (gdb)
```

The call stack (lines 44–47) describes where our program is in the code currently, and what chain of functions was called to get there. The stack is made up of "frames", which each represent one function call. Each frame was called by the frame below it, all the way down to `main`. The top frame in the call stack (#0, line 44) is the current position (line 6 of `sample.c`, in `compare_chars`). `compare_chars` was called from line 11 in `compare_words` (stack frame #1, line 45), which was called on line 29 in `check_words` (frame #2, line 46), and so on. The backtrace also displays the values of the arguments to the called functions (in brackets, after the function names). In this example, `check_words`'s arguments were `0x7fffffffe0e0` and `2`.

Viewing variables

We use `p` (`print`) to see the value currently stored in a variable.

```
48  (gdb) p c1
49  $1 = 0x7fffffffe402 "a"
50  (gdb) p c2
51  $2 = 0x0
52  (gdb)
```

We can view global variables and the arguments and local variables of the function GDB is currently inspecting, which at the moment is `compare_chars` (frame #0), because that's where the program paused. `c1` and `c2` are both `char *`s, so GDB printed them as the pointer addresses. We can see on line 49 that GDB also printed the string pointed at by `c1`. It did not do this for `c2`, because `c2` is the null pointer.

Navigating the call stack

We can view the arguments and local variables of another function in the call stack by navigating GDB to the respective frame. This is done with the `up` and `down` commands.

```
52  (gdb) up
53  #1  0x00000000004005dd in compare_words (w1=0x7fffffffe402 "a", w2=0x0) at sample.c:11
54  11              if (!compare_chars(w1 + i, w2 + i)) {
55  (gdb) up
56  #2  0x0000000000400687 in check_words (words=0x7fffffffe0e0, numWords=2) at sample.c:29
57  29              bool wordsMatched = compare_words(word1, word2);
58  (gdb) l
59  24
60  25          for (int i = 0; i < numWords / 2; i++) {
61  26              char *word1 = words[lowerWord];
62  27              char *word2 = words[upperWord];
63  28
64  29              bool wordsMatched = compare_words(word1, word2);
65  30              mirrored = mirrored || wordsMatched;
66  31
67  32              lowerWord++;
68  33              upperWord--;
69  (gdb) p word1
70  $3 = 0x7fffffffe402 "a"
71  (gdb) down 2
72  #0  0x0000000000400594 in compare_chars (c1=0x7fffffffe402 "a", c2=0x0) at sample.c:6
73  6          return *c1 == *c2;
74  (gdb)
```

The `up` and `down` commands move GDB's focus up and down the call stack by the given number of frames, or by 1 frame if we don't provide a number. When we focus on a new frame, GDB prints out information about the frame (lines 53, 56, 72) in the same format as `backtrace`. The line of code that GDB is focused on is also printed (lines 54, 57, 73). This line is the last line of that frame that was executed, which will be the line that called the next frame (`l` shows the line's context). Line 70 shows that we can view local variables from the frame that we are currently focused on.

Calling functions and evaluating expressions

We can use `p` to call a function or evaluate an expression as well as viewing variables.

```
74 (gdb) p compare_words("hello", "hello")
75 $4 = true
76 (gdb) p (char)('a' - 32)
77 $5 = 65 'A'
78 (gdb)
```

The syntax used on these lines is the same as in regular C code. Any code run or called in this fashion executes in the program's current state, so any variables or data accessed will have the values they held when the program was paused. Similarly, any data that is modified by the functions called will hold the new values after the program is resumed.

Breakpoints and stepping through code

Breakpoints are a tool used to automatically pause the program when it reaches a specific place in the code. The command to create a breakpoint is **`break`** (or `b`) and it takes a function name or a code position as an argument. When the program will pause when it reaches this position or the function is called.

```
78 (gdb) break compare_words
79 Breakpoint 1 at 0x4005af: file sample.c, line 10.
80 (gdb) break 10
81 Note: breakpoint 1 also set at pc 0x4005af.
82 Breakpoint 2 at 0x4005af: file sample.c, line 10.
83 (gdb) break sample.c:10
84 Note: breakpoints 1 and 2 also set at pc 0x4005af.
85 Breakpoint 3 at 0x4005af: file sample.c, line 10.
86 (gdb)
```

Lines 78, 80 & 83 are equivalent, as they all refer to the same line of code, and GDB warns us about this on lines 81 & 84. When we create a breakpoint, GDB tells us its number (in case we need to refer to it later), and where in the program it was created (see lines 79, 82, 85).

```
86 (gdb) info breakpoints
87 Num     Type           Disp Enb Address            What
88 1       breakpoint     keep y   0x00000000004005af in compare_words at sample.c:10
89 2       breakpoint     keep y   0x00000000004005af in compare_words at sample.c:10
90 3       breakpoint     keep y   0x00000000004005af in compare_words at sample.c:10
91 (gdb)
```

The **`info breakpoints`** command lists all breakpoints. The columns are: breakpoint number, breakpoint type, disposition, enable status, address, and location. Type, disposition, and enable status describe more advanced breakpoint features than we require, and the defaults will serve us well.

If you no longer need a breakpoint, you can supply its number to the **`delete`** command to remove it. GDB will not output anything when the breakpoint is deleted, but we can confirm that it was by viewing all breakpoints again.

```
91 (gdb) delete 2
92 (gdb) delete 3
93 (gdb) info breakpoints
94 Num     Type           Disp Enb Address            What
95 1       breakpoint     keep y   0x00000000004005af in compare_words at sample.c:10
96 (gdb)
```

If we restart the program, we can see the breakpoint in action. Because the program is currently running but paused, GDB will ask us to confirm that we would like to restart it.

```
 96  (gdb) run a b a
 97  The program being debugged has been started already.
 98  Start it from the beginning? (y or n) y
 99
100  Starting program: /home/you/path/to/sample a b a
101
102  Breakpoint 1, compare_words (w1=0x7fffffffe3fe "a", w2=0x0) at sample.c:10
103  10          for (int i = 0; i < strlen(w1); i++) {
104  (gdb)
```

The program executes normally until it reaches a line with a breakpoint set, and then it pauses before executing that line. If the breakpoint is a function, if will pause after the function was called but before the first line of the function body is executed. Line 102 describes the breakpoint that was hit (number and location), and the current stack frame (function and arguments). Line 103 shows the line that the program stopped before executing.

Stepping through code

Now that our program has hit breakpoint and is paused, we may wish to procede through the code slowly, seeing how its state changes as each line executes. The `step` (`s`) and `next` (`n`) commands are used to execute the program one line or less at a time. `next` runs the next line of code in the current function, including any functions it calls, and then pauses before executing the next. `step` moves to the line of code that needs to be executed next, no matter where the line is. The difference between these two is clearest when the next line in the current function calls a function itself.

```
104  (gdb) l 44
105  39      int main(int argc, char *argv[]) {
106  40          if (check_words(&argv[1], argc)) {
107  41              printf("The arguments are the same backwards and forwards\n");
108  42          } else {
109  43              printf("The arguments are not the same forwards and backwards\n");
110  44          }
111  45      }
112  (gdb) b main
113  Breakpoint 1 at 0x4006df: file sample.c, line 40.
114  (gdb) run
115  Starting program: /home/you/path/to/sample
116
117  Breakpoint 1, main (argc=1, argv=0x7fffffffe0e8) at sample.c:40
118  40          if (check_words(&argv[1], argc)) {
119  (gdb) next
120  41              printf("The arguments are the same backwards and forwards\n");
121  (gdb)
```

When using `next` in this situation, the entire `check_words` function is executed, and the program is paused before executing the next line in `main`. Like when a breakpoint is hit, the next line is printed after one is run with `next`.

```
121  (gdb) run
122  Starting program: /home/you/path/to/sample
123
124  Breakpoint 1, main (argc=1, argv=0x7fffffffe0e8) at sample.c:40
125  40          if (check_words(&argv[1], argc)) {
126  (gdb) step
127  check_words (words=0x7fffffffe0f0, numWords=1) at sample.c:20
128  20          bool mirrored = true;
129  (gdb) l
130  15
```

```
131  16          return true;
132  17      }
133  18
134  19      bool check_words(char *words[], int numWords) {
135  20          bool mirrored = true;
136  21
137  22          int lowerWord = 0;
138  23          int upperWord = numWords - 1;
139  24
140  (gdb)
```

When using `step`, the function will be entered and pause before its first line is executed. Because a new stack frame was entered, the frame information is printed as well as the next line to execute (lines 127 & 128).

Pausing and resuming execution

If your program is caught in a loop, waiting for IO, or otherwise taking a long time to execute, you can pause it by pressing `CTRL-C` in GDB. The following example is from a program that contains only an infinite while loop.

```
140  (gdb) l main
141  1       int main(int argc, char *argv) {
142  2           while (1) {
143  3           }
144  4       }
145  (gdb) run
146  Starting program: /home/me/path/to/loop
147  ^C
148  Program received signal SIGINT, Interrupt.
149  main (argc=1, argv=0x7fffffffe0e8 "\321\343\377\377\377\177") at loop.c:3
150  3           }
151  (gdb)
```

The program runs until `CTRL-C` is pressed (displayed is `^C` in GDB, line 147). Line 148 says that the program received `SIGINT`, which is what a shell sends when the user types `CTRL-C`. GDB treats this much like a breakpoint; the program is paused at its current point of execution, and the location and stack frame information are printed (lines 149 & 150).

```
151  (gdb) c
152  Continuing.
```

If you don't want to step through line-by-line after the program is paused, you can resume regular execution using `continue` or `c`.

Conditional breakpoints

A useful ability of breakpoints is to only pause the program when a certain condition holds. This can be invaluable when you know a problem only arises in a certain situation and don't want to pause the program unless that situation is encountered. The command to create a conditional breakpoint is `break location if condition`.

```
153  (gdb) break main if argc > 1
154  Breakpoint 1 at 0x4006df: file sample.c, line 40.
155  (gdb) run
156  Starting program: /home/you/path/to/sample
157  The arguments are the same backwards and forwards
158  [Inferior 1 (process 61401) exited normally]
159  (gdb) run test
```

```
Starting program: /home/you/path/to/sample

Breakpoint 1, main (argc=2, argv=0x7fffffffe0c8) at sample.c:40
40          if (check_words(&argv[1], argc)) {
(gdb)
```

This example shows a conditional breakpoint being made that pauses the program immediately if it is given any command line arguments. When run without arguments, it executes and exits, but when it is run with an argument the breakpoint is triggered and the program is paused.