GDB TUTORIAL

STARTING GDB

The purpose of a debugger such as GDB is to allow you to see what is going on inside a program while it executes, or what a program was doing at the moment it crashed.

To prepare your program for debugging with GDB, you must compile it with the -g flag. For example:

```
% gcc   -g myprogram.c   -o myprogram
```

GDB can be started in three different modes:

```
% gdb myprogram                  // attaches gdb to myprogram and starts gdb
% gdb myprogram myprogram.core   // if myprogram has crashed
% gdb myprogram 208              // if myprogram is running with process id 208
```

GDB COMMANDS

Once started, GDB displays the prompt '(gdb)' and is ready to read commands from the terminal. You can now launch your program and step through it line by line, set breakpoints or track the values of variables. Some of the most frequently used GDB commands are:

Run

run [arglist] Start your program (with arglist, if specified).

Breakpoint *(makes your program stop at the beginning of the line where the breakpoint has been set)*

break [file:]fct Set a breakpoint at function fct (in file).
break [file:]lnum Set a breakpoint at line number lnum (in file).
break [+|-]offset Set a breakpoint some number of lines forward or back from the position at which execution stopped in the currently selected frame.

Watchpoint *(special breakpoint that stops your program when the value of an expression changes)*

watch expr Set a watchpoint for the expression expr.

Continue

c Continue running your program (after stopping, e.g. at a breakpoint).
next Execute next program line (after stopping); step *over* any function calls in the line.
step Execute next program line (after stopping); step *into* any function calls in the line.

Examining Data

display expr Add the expression expr to the list of expressions to display each time your program stops.
print expr Display the value of the expression expr (most often just a variable name!)
print file::variable Display the value of the static variable in the context file.
print function::variable Display the value of the static variable in the context function.

Examining the Stack Frames

bt Backtrace: display a stack frame for each active subroutine

Examining Source Lines

list + Print lines just after the lines last printed (by default 10 lines).
list - Print lines just before the lines last printed (by default 10 lines).
list [file:]fct Print lines centered around the beginning of function fct (in file).
list [file:]lnum Print lines around line number lnum (in file).

Deleting Breakpoints

clear Delete any breakpoints at the next instruction to be executed.
clear [file:]fct Delete any breakpoints set at entry to the function fct.
clear [file:]linenum Delete any breakpoints set at or within the code of the specified line.

Miscellaneous

info break Print a list of breakpoints and watchpoints.

help [name] Show information about GDB command name, or general information about using GDB.
quit Exit from GDB.

GCC OPTIONS

Some useful options of the gcc C compiler are listed here:

Debugging

-g Produce debugging information. GDB can work with this debugging information

Dump

-dH Produce a core dump whenever an error occurs.
-dm Print statistics on memory usage, at the end of the run, to standard error.

Optimization

-O1 The compiler tries to reduce code size and execution time.
-fast Optimize for maximum performance
(optimization available with Apple's standard gcc, but not with Ubuntu's standard gcc)
-Os Optimize for size.
-floop-optimize Perform loop optimizations: move constant expressions out of loops.
-foptimize-sibling-calls Optimize sibling and tail recursive calls.

Statistics

-ftime-report Makes the compiler print some statistics about the time consumed by each pass when it finishes.
-fmem-report Makes the compiler print some statistics about permanent memory allocation when it finishes.

EXAMPLE 1. FIRST GDB CONTACT

Assume that the file './swap.c' contains the program of figure 1.

```
(01)  void swap1(int i, int j)
(02)  {
(03)    int temp;
(04)    temp = i;
(05)    i = j;
(06)    j = temp;
(07)  }
(08)
(09)  void swap2(int *i, int *j)
(10)  {
(11)    int temp;
(12)    temp = *i;
(13)    *i = *j;
(14)    *j = temp;
(15)  }
(16)
(17)  main()
(18)  {
(19)    int i, j;
(20)    i=1; j=2;
(21)    swap1(i,j);
(22)    swap2(&i,&j);
(23)  }
```

Fig. 1. A simple program with two swaps

Try out the following commands (remember: % is the prompt of the shell, and (gdb) the prompt of GDB):

```
% gcc  -g swap.c  -o swap
% gdb swap
```

```
(gdb) break main      // set some breakpoints, i.e. at the beginning of line 20
(gdb) break swap1     // set breakpoint at line 4
(gdb) break swap2     // set breakpoint at line 12
(gdb) break 23        // set breakpoint at line 23, i.e. just before quitting the pragram swap

(gdb) run             // run the program swap with the above breakpoints;
                      // stop at the first breakpoint, i.e. breakpoint main at line 20
(gdb) print i         // print the value of i at the beginning of line 20
(gdb) step            // execute the next program line (i.e. line 20)
(gdb) print i         // print the value of i at the beginning of line 21

(gdb) c               // continue to the next breakpoint (i.e. breakpoint swap1, line 4)
(gdb) print i         // print the value of i at the beginning of line 4
(gdb) break +2        // set new breakpoint at line 4+2

(gdb) c               // continue to the next breakpoint (i.e. breakpoint at line 6)
(gdb) print i         // print the value of i at the beginning of line 6
(gdb) ...
(gdb) quit
%
```

EXAMPLE 2. DEBUGGING OPTIMIZED BINARY CODE

Assume that the file './loop.c' contains the program of figure 2.

```
(01)  #include "stdio.h"
(02)
(03)  void loop1(void)
(04)  {
(05)    int i,j,k;
(06)    for (i=1; i<=3; ++i) {
(07)       j=i;
(08)       k=3;
(09)       printf("i=%i, j=%i, k=%i\n",i,j,k);
(10)    }
(11)  }
(12)
(13)  void loop2(void)
(14)  {
(15)    int i,j,k;
(16)    for (i=1; i<=3; ++i) {
(17)       j=i;
(18)       k=3;
(19)    }
(20)  }
(21)
(22)  main()
(23)  {
(24)    loop1();
(25)    loop2();
(26)  }
```

Fig. 2. A simple program with two loops

Try out the following commands (remember: % is the prompt of the shell, and (gdb) the prompt of GDB):

```
% gcc   -g loop.c   -o loop
% gcc   -g -O1 loop.c   -o loop_opt1
% gcc   -g -fast loop.c   -o loop_fast    // flag -fast is available with Apple's standard gcc,
                                          // but not with Ubuntu's standard gcc
% ./loop ; ./loop_opt1 ; ./loop_fast  // just for verifying that all three programs are correct

% gdb loop         // attach loop to the gdb debugger and run gdb
(gdb) b main       // set a breakpoint at the beginning of the main function
```

```
(gdb) r          // run the program loop and stop at the breakpoint main
(gdb) c          // continue to the end of the program loop

(gdb) r          // run the program loop again
(bdb) s          // observe now, step by step, what is happening
                 // by issuing a series of s commands, interleaved with some 'print i' commands
                 // (try also some 'print j' and 'print k' commands)
...
```

Now repeat the same scenario with the two optimized programs 'loop_opt1' and 'loop_fast', and watch carefully what is happening!

EXAMPLE 3. POINTER TRACKING

Assume that the file 'inc.c' contains the program of figure 3.

```
(01)  #include "stdio.h"
(02)
(03)  void inc(int *i)
(04)  {
(05)     ++*i;
(06)  }
(07)
(08)  main()
(09)  {
(10)    int i=2;
(11)    inc(i);  // instead of inc(&i);
(12)  }
```

Fig. 3. A common pointer bug

Try out the following commands (remember: % is the prompt of the shell, and (gdb) the prompt of GDB):

```
% gcc   -g inc.c   -o inc
(gdb) b main      // set a breakpoint at the beginning of the main function
(gdb) r           // run the program loop and stop at the breakpoint main
(bdb) s           // observe now, step by step, what is happening
```

Assume that the file 'seg_fault.c' contains the program of figure 4.

```
(01)  main()
(02)  {
(03)    int *p; // p is uninitialized and points
(04)            // to a random location in memory !
(05)    *p = 12;
(06)  }
```

Fig. 4. The most common pointer bug

Try out the following commands (remember: % is the prompt of the shell, and (gdb) the prompt of GDB):

```
% gcc   -g seg_fault.c   -o seg_fault
(gdb) b main      // set a breakpoint at the beginning of the main function
(gdb) r           // run the program loop and stop at the breakpoint main
(bdb) s           // observe now, step by step, what is happening
```

References

[ST 02] Richard M. Stallman and Roland H. Pesch : "Debugging with GDB: The GNU Source-Level Debugger", GNU Press, 2002.
Free available at https://www.sourceware.org/gdb/documentation/

See also: ktrace, http://en.wikipedia.org/wiki/Ktrace.