Lecture 5: Debuggers

Youjip Won and Kyungsoo Park

KAIST EE

(Reference: The ART OF DEBUGGING with GDB, DDD, and ECLIPSE (TAD))

Introduction to GNU Debugger

- Today's topic
  - General strategy with debugging with GDB:
    - Execute the program to the point of interest
      - Use breakpoints and stepping to do that
    - Examine the values of variables at that point
  - Debugging example code
    - Insertion sorting

Typical Steps for Debugging with GDB

(a) Build with –g

```
(gdb) gcc -g insertsort.c –o insertsort
```

- Adds extra information to executable file that GDB uses
- Debugging symbols (e.g., line numbers, variable names, etc.)

(b) Run GDB in a different terminal

```
$ gdb insertsort
```

You can run GDB inside Emacs or VIM as well

(c) Set breakpoints, as desired

- the program would stop at each breakpoint when it's executed

```
(gdb) break main
```

- GDB sets a breakpoint at the first executable line of main()

```
(gdb) break process_data
```

- GDB sets a breakpoint at the first executable line of process_data()

Typical Steps for Debugging with GDB (cont.)

(d) Run (or continue) the program

(gdb) run

- GDB stops at the breakpoint in main()

(gdb) continue

- GDB stops at the breakpoint in process_data()

(e) Step through the program, as desired

(gdb) step (repeatedly)

- GDB executes the next line (repeatedly)

- Note: When next line is a call of one of your functions:
  - **step** command steps into the function
  - **next** command steps over the function, that is, executes the next line without stepping into the function

Typical Steps for Debugging with GDB (cont.)

(f) Examine variables, as desired

```
(gdb) print i
(gdb) print j
(gdb) print temp
```

- GDB prints the value of each variable

(g) Examine the function call stack, if desired

```
(gdb) where
```

- GDB prints the function call stack
- Useful for diagnosing crash in large program

(h) Exit gdb

```
(gdb) quit
```

Other Useful Tips

- How to run with command-line arguments?

```
(gdb) run arg1 arg2
```

- How to handle redirection of stdin, stdout, stderr?

```
(gdb) run < somefile > someotherfile
```

- Print values of expressions (later)
- Break conditionally (later)
- Materials so far are enough for basic usage of GDB

Debugging "insertsort.c" (from TAD)

```
// insertion sort, several errors
// usage: insert_sort num1 num2 num3 ...,
// where the numi are the numbers to be sorted

int x[10],      // input array
    y[10],      // workspace array
    num_inputs, // length of input array
    num_y = 0;  // current number of elements in y

void get_args(int ac, char **av)
{
   int i;
   num_inputs = ac - 1;
   for (i = 0; i < num_inputs; i++)
      x[i] = atoi(av[i+1]);
}

void scoot_over(int jj)
{
   int k;
   for (k = num_y-1; k > jj; k++)
      y[k] = y[k-1];
}
```

Debugging "insertsort.c" (from TAD)

```
void process_data()
{
 for (num_y = 0; num_y < num_inputs; num_y++)
    // insert new y in the proper place
    // among y[0],...,y[num_y-1]
    insert(x[num_y]);
}

void print_results()
{  int i;

   for (i = 0; i < num_inputs; i++)
      printf("%d\n",y[i]);
}

int main(int argc, char ** argv)
{
   get_args(argc, argv);
   process_data();
   print_results();
}
```

Debugging "insertsort.c" (from TAD)

```
void insert(int new_y)
{
   int j;
   if (num_y = 0) { // y empty so far
      y[0] = new_y;
      return;
   }

 // need to insert just before the first y
 // element that new_y is less than
   for (j = 0; j < num_y; j++)  {
      if (new_y < y[j])  {
         // shift y[j], y[j+1],... rightward
         // before inserting new_y
         scoot_over(j);
         y[j] = new_y;
         return;
      }
   }
}
```

Insertion Sort

- Supposed to do followings
  - Get the integer input from the command line
  - Sort them by insertion sort
  - Print out the result to stdout

```
$ ./insertsort 10 15 7 50 1
1
7
10
15
50
```

- Insertion sort?
  - You have a sorted array (initially it's empty)
  - For each new input value, add it to the right position in the sorted array
  - Repeat the second step until you process all input values
- Say input is 4, 1, 2, and the sorted array is initially empty ()
  - 4? add it to the right position in the sorted array (4)
  - 1? add it to the right position in the sorted array (1, 4) // 4 is pushed down
  - 2? add it to the right position in the sorted array (1, 2, 4) // 4 is pushed down

Build & Run the Program

```
$ gcc insertsort.c –o insertsort
$ ./insertsort 15 10
(running infinitely, user hits ctrl-c to stop the program)
```

Build & Run the Program

OK. Let's figure out what's wrong.

(1) Compile with –g option

```
$ gcc insertsort.c -g -o insertsort
```

(2) gdb with insertsort, run & ctrl+C

```
$ gdb insertsort
…
(gdb) run 15 10
Starting program: /home/kyoungsoo/debug/insertsort 15 10
^C
Program received signal SIGINT, Interrupt.
0x0000000008000806 in process_data () at insertsort.c:52
52              insert(x[num_y]);
(gdb) print num_y
$1 = 1
```

```
47 void process_data()
48 {
49   for (num_y = 0; num_y < num_inputs;
num_y++)
50     // insert new y in the proper place
51     // among y[0],...,y[num_y-1]
52     insert(x[num_y]);
53 }
```

Stopped at process_data().

See what happens at insert() when `num_y == 1`.

(3) Stop at `insert()` when `num_y == 1`

```
(gdb) break insert if num_y == 1
Breakpoint 1 at 0x8000761: insert. (3 locations)
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/kyoungsoo/debug/insertsort 15 10

Breakpoint 1, insert (new_y=10) at insertsort.c:30
30          if (num_y = 0)  { // y empty
```

(4) Move on with `next` command

```
(gdb) next
36      for (j = 0; j < num_y; j++)  {
```

We are at line 36 and `num_y == 1`, so we expect to enter the loop.

(5) Move on with `next` command

```
(gdb) next
45        }
```

What? Why skip the loop?

```
27 void insert(int new_y)
28 {
29     int j;
30     if (num_y = 0) { // y empty so far
31       y[0] = new_y;
32       return;
33     }
…
36     for (j = 0; j < num_y; j++)  {
37         if (new_y < y[j])  {
…
40           scoot_over(j);
41           y[j] = new_y;
42           return;
43         }
44       }
45     }
```

Debugging Insertsort

6) Check out the value of num_y!

```
(gdb) print num_y
$2 = 0
```

Surprising! num_y is 0. A bug between line 30 and 45.

OK, found a bug!

`if (num_y = 0) -> if (num_y == 0)`

Fix the bug, and run it again!

```
$ ./insertsort 15 10
15
0
```

No infinite loop, but wrong output! What happened to 10 (second number)?

```
27 void insert(int new_y)
28 {
29     int j;
30     if (num_y = 0) { // y empty so far
31       y[0] = new_y;
32       return;
33     }
…
36     for (j = 0; j < num_y; j++)  {
37         if (new_y < y[j])  {
…
40           scoot_over(j);
41           y[j] = new_y;
42           return;
43         }
44       }
45     }
```

Debugging Insertsort

(7) Stop at `insert()` when `new_y == 10`

```
(gdb) break insert if new_y == 10
(gdb) run 15 10
Breakpoint 1, insert (new_y=10) at insertsort.c:30
30    if (num_y == 0)  { // y empty so far
```

(8) A bit more investigation with `next/print`

```
(gdb) next
36    for (j = 0; j < num_y; j++)  {
(gdb) next
37    if (new_y < y[j])  {
(gdb) print y[0]
$3 = 15
(gdb) next
40 scoot_over(j);
(gdb) print y
$4 = {15, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

OK. A bug in `scoot_over()`! After `scoot_over()`, `15` should have been moved like

```
y[] = {0, 15, 0, 0, 0, 0, 0, 0, 0, 0}
```

```
27 void insert(int new_y)
28 {
29    int j;
30    if (num_y == 0) { // y empty so
far
31      y[0] = new_y;
32      return;
33    }
…
36    for (j = 0; j < num_y; j++)  {
37        if (new_y < y[j])  {
…
40          scoot_over(j);
41          y[j] = new_y;
42          return;
43        }
44      }
45    }
```

Debugging Insertsort

```
27 void insert(int new_y)
28 {
29     int j;
30     if (num_y == 0) { // y empty so
far
31       y[0] = new_y;
32       return;
33     }
…
36     for (j = 0; j < num_y; j++)  {
37         if (new_y < y[j])  {
…
40           scoot_over(j);
41           y[j] = new_y;
42           return;
43         }
44       }
45     }
```

```
void scoot_over(int jj)
{
    int k;
    for (k = num_y-1; k > jj; k++)
       y[k] = y[k-1];
}
```

At line 40, `scoot_over(0)` is called where `jj == 0` and `num_y == 1`

So, `k=num_y-1` becomes `0`, and `k > jj` fails!

Fix: `k = num_y;`

Debugging Insertsort

OK, rerun the program

```
$ ./insertsort 15 10
Segmentation fault (core dumped)
```

Killed as it accesses memory that is not allowed.

(9) Run gdb with insertsort

```
$ gdb insertsort
…
(gdb) run 15 10
…
Program received signal SIGSEGV, Segmentation fault.
0x0000000008000741 in scoot_over (jj=0) at insertsort.c:24
24              y[k] = y[k-1];
(gdb) print k
$1 = 984
(gdb) print num_y
$2 = 1
```

```
20 void scoot_over(int jj)
21 {
22   int k;
23   for (k = num_y; k > jj; k++)
24      y[k] = y[k-1];
25 }
```

OK. Found another bug!
`k` keeps on increasing (`k++`).
But it has to *decrease* each time!
Fix: `for (k = num_y; k > jj; k--)`

Debugging Insertsort

OK, rerun the program

```
$ ./insertsort 15 10
10
15
```

Great! Is it done?

```
$ ./insertsort 15 10 16 8
8
10
15
0
```

Oh No! `16` is missing!

(10) Stop at `insert()` if `new_y == 16`

```
(gdb) b insert if new_y == 16
(gdb) run 15 10 16 8
Breakpoint 1, insert (new_y=16) at insertsort.c:30
30          if (num_y == 0)  { // y empty so far, easy case
```

b=break

```
27 void insert(int new_y)
28 {
29     int j;
30     if (num_y = 0) { // y empty so far
31       y[0] = new_y;
32       return;
33     }
…
36     for (j = 0; j < num_y; j++)  {
37         if (new_y < y[j])  {
…
40           scoot_over(j);
41           y[j] = new_y;
42           return;
43         }
44       }
45     }
```

(11) Look at the loop

```
(gdb) n
36    for (j = 0; j < num_y; j++)  {
(gdb) n                              n=next
37    if (new_y < y[j])  {
(gdb) p y                            p=print
$1 = {10, 15, 0, 0, 0, 0, 0, 0, 0, 0}
(gdb) n
36    for (j = 0; j < num_y; j++)  {
(gdb) n
36    for (j = 0; j < num_y; j++)  {
(gdb) n
45     }
```

Found a problem! Not storing `new_y` when `new_y >=` all elements!

Fix: `y[num_y]=new_y` after line 44. Now all bugs are fixed!

More on Breakpoints

Various breakpoints

```
(gdb) break 30              // stop at line 30 of the current file
(gdb) break fileA.c:40      // stop at line 40 on fileA.c
(gdb) break func if expr    // stop at func() if expr is true
```

`info`: show all breakpoints

```
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
2       breakpoint     keep y   0x000000000000089b in main at insertsort.c:64
3       breakpoint     keep y   0x000000000000075e in insert at insertsort.c:30
        stop only if new_y == 16
```

`delete`: delete breakpoints

```
(gdb) delete 2 3    // remove breakpoints 2 and 3
```

`watch expr`: stops when the value of "`expr`" changes

```
(gdb) watch i    // stops when the value of i changes
(gdb) watch (i|j>12) && I > 24 && strlen(name) > 6 // expr can be fairly flexible
```

list/where/up/down

`list`: show the source code

```
(gdb) list          // show the source code of the c
(gdb) list func     // show the source code of func()
```

`where`: show the call stack of where I am

```
(gdb) where
#0  insert (new_y=1) at insertsort.c:30
#1  0x000000000800081d in process_data () at insertsort.c:53
#2  0x00000000080008b6 in main (argc=2, argv=0x7fffffee3b8) at insertsort.c:65
```

`up`: go up to the caller function in the call stack

```
(gdb) up
#1  0x000000000800081d in process_data () at insertsort.c:53
53   insert(x[num_y]);
```

`down`: go down to the callee function in the call stack

```
(gdb) down
#0  insert (new_y=1) at insertsort.c:30
30   if (num_y == 0)  { // y empty so far, easy case
```

Resuming Execution

`continue`: continue execution

```
(gdb) continue  // resume execution from where we stop
(gdb) cont      // cont = continue
```

`until`: execute the code until the current loop ends

```
(gdb) until  // execute machine instructions until
             // it reaches a memory address (location)
             // larger than the current address
(gdb) until 17 // execute until it reaches line 17
```

`finish`: execute the code until the current function returns

```
(gdb) finish
(gdb) fin     // fin = finish
```

Assignment for Lecture 5

- Deadline: before the start of the lecture in the next week
- Follow the instructions in this lecture with insertsort.c
- Fix all bugs in insertsort.c and submit the bug-free file to KLMS
  - Use gcc instead of gcc209 for this assignment (Since gcc209 is safer, you can't compile the code with gcc209 due to a compiling error)
  - `$ gcc -g -o insert insert.c`