ECS 98F - Using the GNU Debugger

Joël Porquet & Noah Rose Ledesma

GDB

GNU project

- Started by Richard Stallman in 1983
- Free software, mass collaboration project in response to proprietary UNIX
  - Copyleft license: GNU GPL
  - User programs: text editor (Emacs), compiler (GCC toolchain), debugger (GDB), and various utilities (ls, grep, awk, make, etc.)
  - Kernel: GNU Hurd

GDB

- GNU DeBugger
- Supports many languages
  - Including C and C++
- Inspection of program during execution
  - Execution flow
  - Data
- Helps finding errors like *segmentation fault*
- Read the fully-detailed manual: https://sourceware.org/gdb/current/onlinedocs/gdb/

GDB usage

Compilation flags

- Canonical compilation command line:

```
$ gcc [cflags] -o <output> <input>
```

- Optimize for speed (`-O2`)

```
$ gcc -Wall -Werror -O2 -o myprogram main.c
```

- Enable debugging support (`-g`)

```
$ gcc -Wall -Werror -g myprogram main.c
```

  - To balance performance with debugging experience use `-Og`
  - Not recommended to use debugging along with other optimizations
    - No optimization option is equivalent to `-O0`

- During development, very useful to be able to debug your program
- For production, probably better to disable the debug support and activate all possible optimization support
  - Reduce size of the executable (can easily be by 50%!)
  - Increase performance (can also be by 50%!)

GDB usage

Starting GDB

- Start GDB, specify the program to debug

```
$ gdb
...
(gdb) file myprogram
Reading symbols from myprogram...done.
(gdb)
```

- Or, start GDB with the program to debug as argument

```
$ gdb myprogram
...
Reading symbols from myprogram...done.
(gdb)
```

Running the program

- Without any argument:

```
(gdb) run
```

- With arguments:

```
(gdb) run argv1 argv2...
```

GDB usage

Interactive help

- GDB offers an interactive shell
  - History management
  - Auto-complete (with TAB)

In order to discover what you can do, just ask:

```
(gdb) help
List of classes of commands:
aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
...

(gdb) help breakpoints
Making program stop at certain points.
List of commands:
awatch -- Set a watchpoint for an expression
break -- Set breakpoint at specified location
...

(gdb) help break
Set breakpoint at specified location.
break [PROBE_MODIFIER] [LOCATION] [thread THREADNUM] [if CONDITION]
...
```

GDB usage

Possible scenarios

1. Program doesn't have bugs:
   - It will run fine until completion

```
$ ./myprogram
I worked, hurray!
```

2. *Best*-case scenario, regarding bugs:
   - *Segmentation fault*

```
$ ./myprogram
segmentation fault (core dumped) ./myprogram
```

3. *Worst*-case scenario:
   - Doesn't crash but wrong result

```
$ ./myprogram
I work��, ��rray!
```

   - Bugs that don't trigger any segmentation fault
   - In this case, you'll probably have to spend more time...

Segmentation faults

Example #1

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

size_t foo_len (const char *s)
{
    return strlen(s);
}

int main (int argc, char *argv[])
{
    char *a = NULL;

    printf ("size of a = %d\n", foo_len(a));

    return 0;
}
```

Execution

```
$ ./strlen-test
segmentation fault (core dumped)  ./strlen-test
```

Segmentation faults

Run with GDB

- *(After compiling the code with `-g`)*

```
$ gdb ./strlen-test
(gdb) run
Starting program: /home/noah/tmp/test/strlen-test

Program received signal SIGSEGV, Segmentation fault.
0x00007ffff7abc446 in strlen () from /usr/lib/libc.so.6
(gdb)
```

Backtrace

- First thing to do when getting a *segfault*:
  - Understand what is the sequence of calls that brought us there

```
(gdb) backtrace     # use just 'bt'
#0  0x00007ffff7abc446 in strlen () from /usr/lib/libc.so.6
#1  0x000000000040055e in foo_len (s=0x0) at strlen-test.c:7
#2  0x0000000000400583 in main (argc=1, argv=0x7fffffffd788) at strlen-test.c:14
```

Investigate

- `foo_len()` is supposed to receive a pointer
- Here it receives `0` (aka `NULL`)
- Looks like this `NULL` pointer probably gets dereferenced in `strlen()`...

Segmentation faults

Fix...

- Here, the problem is fairly obvious

```
size_t foo_len (const char *s)
{
    return strlen(s);
}

int main (int argc, char *argv[])
{
     char *a = "This is a valid string";

    printf ("size of a = %d\n", foo_len(a));

    return 0;
}
```

And, celebrate!

```
$ ./strlen-test
size of a = 22
```

Segmentation faults

Better fix

- Prevent the same bug from happening again

```
size_t foo_len (const char *s)
{
    assert(s && "String cannot be NULL here!");
    return strlen(s);
}

int main (int argc, char *argv[])
{
    char *a = NULL;

    printf ("size of a = %d\n", foo_len(a));

    return 0;
}
```

```
$ ./strlen-test
strlen-test: strlen-test.c:8: foo_len:
    Assertion `s && "String cannot be NULL here!"' failed.
```

Segmentation faults

Example #2

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

static const char STR[] = "Hello World!";

int main(int argc, char** argv) {
  char* reversed = malloc(strlen(STR));
  unsigned int reversed_index = strlen(STR) - 1;
  unsigned int str_index = 0;

  while(reversed_index >= 0) {
    reversed[reversed_index] = STR[str_index];
    reversed_index--;
    str_index++;
  }

  printf("%s\n", reversed);
  free(reversed);

  return 0;
}
```

strrev_segfault.c

Execution

```
$ ./strrev-test
segmentation fault (core dumped)  ./strrev-test
```

Segmentation faults

Run GDB

```
$ gdb ./strrev-test
(gdb) run
Starting program: /home/noah/tmp/test/strrev-test

Program received signal SIGSEGV, Segmentation fault.
0x000055555555517c in main (argc=1, argv=0x7fffffffe588) at strrev_segfault.c:16
16          reversed[reversed_index] = STR[str_index];
```

Backtrace

```
(gdb) bt
#0  0x000055555555517c in main (argc=1, argv=0x7fffffffe588) at strrev_segfault.c:16
```

- Except that here, it's not much of help...

Inspect variables

- Display indices so that we know which index in the array was being accessed:

```
(gdb) print reversed_index
$1 = 4294967295
(gdb) print str_index
$2 = 12
```

Segmentation faults

Fix...

- Problem is a case of overflow
  - An `unsigned int` type automatically wraps from 0 to 4294967295

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

static const char STR[] = "Hello World!";

int main(int argc, char** argv) {
  char* reversed = malloc(strlen(STR));
  int reversed_index = strlen(STR) - 1;
  unsigned int str_index = 0;

  while(reversed_index >= 0) {
    reversed[reversed_index] = STR[str_index];
    reversed_index--;
    str_index++;
  }

  printf("%s\n", reversed);
  free(reversed);

  return 0;
}
```

strrev_fixed.c

Tracking bugs

Behavior bugs

- Behavioral bugs more complicated to find because program doesn't crash
- It's just that the output is wrong

```c
#include <ctype.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    int i;
    char str[] = "Tracking bugs is my passion";

    printf("Before: %s\n", str);

    for (i = 0; i < strlen(str) - 1; i++)
        str[i] = toupper(str[i]);

    printf("After: %s\n", str);

    return 0;
}
```

Execution

```
Before: Tracking bugs is my passion
After: TRACKING BUGS IS MY PASSIOn
```

Tracking bugs

Setting breakpoints

- Stop the program during the execution at a designated point
- Set as many breakpoints as necessary
- GDB will always stop the execution when reaching them

Breaking at exact location in code

```
(gdb) break string-test.c:13
Breakpoint 1 at 0x4005ef: file string-test.c, line 13.

(gdb) r
Starting program: /home/noah/tmp/test/string-test
Before: Tracking bugs is my passion

Breakpoint 1, main () at string-test.c:13
13              str[i] = toupper(str[i]);
```

Tracking bugs

Breaking at a particular function

```
(gdb) b main
Breakpoint 1 at 0x40059f: file string-test.c, line 8.

(gdb) r
Starting program: /home/noah/tmp/test/string-test

Breakpoint 1, main () at string-test.c:8
8           char str[] = "Tracking bugs is my passion";
```

Breaking only if condition is satisfied

```
(gdb) b string-test.c:13 if i == 5
Breakpoint 1 at 0x4005ef: file string-test.c, line 13.

(gdb) r
Starting program: /home/noah/tmp/test/string-test
Before: Tracking bugs is my passion

Breakpoint 1, main () at string-test.c:13
13              str[i] = toupper(str[i]);

(gdb) print i
$1 = 5
```

Tracking bugs

Dealing with breakpoints

- Set at least one breakpoint before running the program
  - Otherwise the program will run until completion
- Once the program stops and the gdb shell is available, a few options:
  1. Continue the execution until hitting the same or another breakpoint

     ```
     (gdb) continue      # or just 'c'
     ```

  2. Execute only the next line of code and break again

     ```
     (gdb) step          # or just 's'
     ```

     Careful, `step` enters function calls
  3. Jump over function calls

     ```
     (gdb) next          # or just 'n'
     ```

Tip: typing `<enter>` in the interactive GDB shell repeats the last command

Tracking bugs

Printing variables

```
int a = 2;
char b = 'x';
int *c = &a;
char *s = "A string";
                        // <= breaking here
...
```

- Inspect the value of all your variables with command `print`

Default

- By default, prints variables according to their type

```
(gdb) print a
$1 = 2
(gdb) p b
$2 = 120 'x'
(gdb) p c
$3 = (int *) 0x7fffffffd65c
(gdb) p s
$4 = 0x40070b "A string"
```

Tweak

- Can tweak both the way `print` prints and what it prints

```
(gdb) print /x a
$1 = 0x2
(gdb) p /c b+2
$2 = 122 'z'
(gdb) p *c
$3 = 2
(gdb) p s[0]
$4 = 65 'A'
```

Tracking bugs

Printing data structures

```
struct entry {
    int     key;
    char    *name;
} obj = {
    .key    = 2,
    .name   = "toto",
};

struct entry *e = &obj;
```

- With `print`, you can access the pointer and the object it's pointing to:

```
(gdb) print e
$1 = (struct entry *) 0x7fffffffd640
(gdb) print &obj
$2 = (struct entry *) 0x7fffffffd640
(gdb) p *e
$3 = {key = 2, name = 0x400734 "toto"}
(gdb) p e->key
$4 = 2
(gdb) p obj.name
$5 = 0x400734 "toto"
```

Misc

Setting watchpoint

- Breakpoints are for interrupting the execution flow at a specific location
- Watchpoints are for interrupting the program when a variable is modified

```
(gdb) watch i
Hardware watchpoint 2: i

(gdb) c
Continuing.
Before: Tracking bugs is my passion

Hardware watchpoint 2: i

Old value = 0
New value = 1
0x0000000000400612 in main () at string-test.c:12
12          for (i = 0; i < strlen(str) - 1; i++)
```

Misc

Other useful commands

- `finish`
  - Runs until the current function is finished
- `until`
  - When executed in a loop, continues the execution until the loop ends
- `info breakpoints`
  - Shows informations about all declared breakpoints

```
(gdb) info b
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x000000000040059f in main at string-test.c:8
breakpoint already hit 1 time
2       hw watchpoint  keep y                      i
breakpoint already hit 2 times
```

- `delete`
  - Deletes a breakpoint

Misc

Conclusion

- GDB is a versatile tool for "looking under the hood"
- Not a magical tool for fixing bugs in program
- Debuggers serve a specific purpose and should only be used in the right circumstances

- Doesn't work for programs with realtime behavior
- Debugging support disables optimizations and adds overhead
- Not ideal for programs that run for a very long time