Debugging

Variables

Lecturers: Mirko Jantschke, Pascal Scholz

4. Dezember 2018

Contents

Getting started

The slides are at fsr.github.io/c-lessons/materials.html

There will be tasks! You can find them at fsr.github.io/c-lessons

If you have questions, use the auditorium group:
https://auditorium.inf.tu-dresden.de/de/groups/110804109

In case of big trouble, write an e-mail to your tutor.

*** new only for a limited time ***
Hackerspace every foo from bar to foobar in room biz.

Bugs

It’s not a bug...

There are different kinds of errors.

- Compiletime errors
- Runtime errors (*bugs*)

Compiletime errors are easily handable since the compiler shows you where to fix them.

Bugs on the other hand are harder to find because you have no idea where to look for them.

… it’s a feature.

Bugs can appear due to different reasons

- Variable overflow
- Division by zero
- Infinite loops / recursions
- Range excess
- Segmentation fault
- Dereferencing *NULL pointers*
- …

The dungeon

We prepared a little ASCII dungeon.
You can find it in the repository (https://github.com/fsr/c-slides) in folder *materials/1_before/*

- Look at the code and try to understand what should happen.
- If you find mistakes, please leave them. We'll fix them later.
- Compile it (with *-std=c99*).
- And now run it.

The dungeon

We prepared a little ASCII dungeon.
You can find it in the repository (`https://github.com/fsr/c-slides`) in folder *materials/1_before/*

- Look at the code and try to understand what should happen.
- If you find mistakes, please leave them. We'll fix them later.

- Compile it (with *-std=c99*).

- And now run it.
- Try to fix all the mistakes using compiler flags.

GDB

The GNU DeBugger

There are tools helping with bugs, called debuggers. GDB is one of them.

To use it

- You have to install the package *gdb*
- You have to compile your program with the *-g* flag

```
$ gcc -g main.c
```

- After that you can start your program with gdb:

```
$ gdb a.out
```

Commands

- If you started gdb without a file you can load it with **file** *file_name*.
- Use **r[un]** to execute the program with gdb.
  You should begin with that. It will give you further information about the crash.
- You can set an arbitrary amount of breakpoints with **b[reak]** *line_number* or **b[reak]** *function_name*.
  Begin with a breakpoint at the point the program crashes.
- Print values with **p[rint]** *identifier*.
- Use **w[atch]** *identifier* to break and print a variable when it's changed.

Once you’re at a breakpoint

- Use n[ext] to execute the next program line only.
- s[tep] executes the next instruction.
- You can jump to the next breakpoint with c[ontinue].
- To see how you have come to this point in the program flow, type backtrace or bt.
  This shows you all functions you called to come there.

- By only hitting the *return key*, you repeat the last entered command.

Conditional breakpoints

After setting a breakpoint, GDB assigns an ID to it.
You can use this ID to extend the functionality of that breakpoint.

- con[dition] *breakpoint_ID expression* adds a condition to your Breakpoint:

```
(gdb) br 42
Breakpoint 1 at 0xbada55: file main.c, line 42.
(gdb) condition 1 i@=@@=@3
```

- For string comparison, set the string before comparing with **strcmp**:

```
(gdb) br main.c:42
Breakpoint 13 at 0xdeadbeef: file main.c, line 42.
(gdb) set $string_to_compare = "lolwut"
(gdb) cond 13 strcmp ( $stringtocompare, c ) @=@@=@ 0
```

Now it’s up to you

- Find and fix all bugs in the dungeon.

file	load program
r[un]	execute program
b[reak]	set breakpoint
p[rint]	print variable
w[atch]	break and print variable when it changes
n[ext]	execute next line and break
s[tep]	execute next instruction and break
c[ontinue]	execute until next breakpoint
backtrace / bt	How did I end up here?