Lesson 6

Christian Schwarz, Jakob Krebs

2.12.2019

Contents

File IO and Debugging

Sources and Solutions

- we publish all code written in this course at `https://github.com/jkrbs/c_lessons`
- we will publish example solutions of the tasks on same site
- send us questions or your solutions to c-lessons@deutschland.gmbh

fopen and fclose

`stdio.h` provides the following functions to open and close a file:

```
1 FILE* fopen (char* filename, char* mode);
2 int fclose (FILE* stream);
3
4 //example
5 FILE* test = fopen("test.txt", "w");
6 fclose(test);
```

Filenames can either be absolute (`"/home/foo/bar.txt"`) or relative (`"test.txt"`). Relative paths are relative to the "current working directory". That is the current directory of your shell when you execute the program. Shells can usually change this directory using `cd` (change directory), and display it using `pwd` (print working directory).

This is not necessarily the directory that the program executable lies in.

file modes

The `"w"` mode in `fopen(filename, mode)` specifies that we only want to **w**rite to the file. There are multiple different modes available:

mode	access	if file exists	if file doesn't exist
r	read-only	read from start	return NULL
w	write-only	overwrite contents	create new
a	write-only	append	create new
r+	read+write	read from start, overwrite	return NULL
w+	read+write	read from start, overwrite	create new
a+	read+write	read from start, but append at the end	create new

”File”

- `FILE*` can be thought of as a pointer to a FILE structure managed by the C standard library that remembers all the necessary information to interact with the file.
- ”File” should not be taken too literally here. Stream might have been the better term. For example, `stdin` and `stdout` are also `FILE*`s.
- Streams or ”`FILE`”s really just represent an object that bytes can be written to and / or read from.

fread and fwrite

```
size_t fread(void* buffer, size_t size, size_t count, FILE* stream);
size_t fwrite(void* buffer, size_t size, size_t count, FILE* stream);
```

- `fread` reads bytes from the `stream` and writes them into `buffer`.
- `fwrite` reads bytes from `buffer` and writes them out to the `stream`.

The functions read/write `size` bytes for up to `count` times, or until the stream has no more contents.

They return the number of elements (of size `size`) successfully read/written.

Sometimes this is useful, e.g. if we want to read up to 20 `int`s:

```
size_t ints_read = fread(buffer, sizeof(int), 20, file);
```

But mostly we use them like this:

```
size_t bytes_read = fread(buffer, 1, sizeof(buffer), file);
```

file io example

```
1  FILE* logfile = fopen("log.txt", "a+");
2  // very unlikely to fail since "a+" creates nonexistant files
3  assert(logfile != NULL);
4
5  char buffer[1024];
6  do{
7      size_t size = fread(&buffer, 1, sizeof(buffer), logfile);
8      display_log(&buffer, size); // use the data
9  } while(size > 0);
10
11 char* msg = "we accessed the log file\n";
12 size_t size = fwrite(msg, strlen(msg), 1, log);
13 assert(size == 1); // was our data written successfully ?
14
15 fclose(config);
```

Debugging

Debugging

There's multiple possibilities why a program doesn't work as intended. As we discussed, the broad classification is between.

- Compiletime (+ link time) errors
- Runtime errors (also called *bugs*)

Compiletime errors are easily handable since the compiler shows you where and what they are

Bugs are oftentimes much harder to find because they could be anywhere in your program and nobody warns you.

Different kinds of Bugs

Bugs can appear due to different reasons

- Variable overflow
- Division by zero
- Infinite loops / recursions
- Range excess
- Segmentation fault
- Dereferencing `NULL` (or other invalid) pointers
- ...

The GNU Debugger (gdb)

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
```

Using gdb

```
$ gdb -g intermediate_06_asciidungeon.c
$ gdb a.out
(gdb) start
Temporary breakpoint 1, main (argc=1, argv=0x7fffffffe028) at ...
41          player = init_entity(5, 8, 100, 'J');
(gdb) next
42          monster1 = init_entity(2, 3, 100, '*');
(gdb) step
init_entity (x_pos=2, y_pos=3, health=100, symbol=42 '*') at ...
104         struct entity *new_ent = malloc(sizeof new_ent);
(gdb) backtrace
#0  init_entity (x_pos=2, y_pos=3, health=100, symbol=42 '*') at ...
#1  0x0000555555554938 in main (argc=1, argv=0x7fffffffe028) at ...
(gdb)
```

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.
  If you have a segfault, it's a good idea to begin with that. It will give you further information about the crash location.
- If you want to debug from the beginning use **sta[rt]** to run and immediately break
- 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 right before 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* sets a condition for 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
```

- use **con[dition]** *breakpoint_ID* to remove the condition:

gdb –tui Mode

If you want a nicer interface where you can see multiple lines of your program, use

```
gdb --tui a.out
```

useful gdb commands

file	load program
r[un]	execute program
b[reak]	set breakpoint
sta[rt]	execute program and break immediately
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?

Task: Bughunting in the Ascii Dungeon

We (or more precisely, the fsr) prepared a little ascii dungeon littered with bugs. You can find it at.

`https://jkrbs.github.io/c_lessons/tasks/intermediate_06_asciidungeon.c`

Or just click on `Lesson 6 Intermediate Task: Bughunting in the Ascii Dungeon` on our website (`https://jkrbs.github.io/c_lessons`).

Task: Fix all the bugs in the program using `gdb`, until you can run around on the Screen using wasd (+ Enter)!