Lesson 9

Christian Schwarz, Jakob Krebs

15.12.2019

Contents

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

unions

Why do we need Unions?

Suppose you have a struct representing one of different kinds of events in an game:

```
1  struct game_event{
2      enum event_kind {
3          PLAYER_INTERACTION, PROJECTILE_HIT //...
4      } kind;
5      // only used by PLAYER_INTERACTION events
6      enum button_kind button;
7      // only used by PROJECTILE_HIT events
8      float projectile_hit_speed;
9  };
```

- This struct layout wouldn't be very efficient, since `PLAYER_INTERACTION` events never need `projectile_hit_speed`, but waste memory for it anyway.
- This can become a problem if we have many events or start to transmit events over the network in multiplayer games.

How can we save memory using unions?

We can tell C that we only need one of a list of fields at a time using `unions`.

```
struct game_event{
    enum event_kind {PLAYER_INTERACTION, PROJECTILE_HIT} kind;
    union {
        enum button_kind button;
        float projectile_hit_speed;
    } payload;
};
```

- Syntactically, a union behaves a lot like a struct.
- But, C will use the same memory block for all members of the `union`
- Therefore, the size of the `union` will be equal to it's biggest member.
- Writing to one union member and then reading from another causes undefined behavior (Though it's quite commonly done anyways)

Anonymous Unions

Sometimes we just want to share memory without giving a name to the created substructure. For this, `C11` (2011) introduced / standardized anonymous unions and structs:

```
1  struct game_event{
2      enum event_kind {PLAYER_INTERACTION, PROJECTILE_HIT} kind;
3      union {
4          enum button_kind button;
5          struct {
6              int projectile_damage;
7              float projectile_hit_speed;
8          }; //struct members don't share memory inside a union
9      };
10 };
11 game_event g;
12 g.projectile_hit_speed = 3; //we can access the members directly
```

procedural macros

Procedural Macros

So far we only talked about the simplest types of macros:

```
1 #define MY_CONSTANT 5
```

But Macros can also be used like functions to and take parameters:

```
1 #define MULT(a, b) a * b
```

- Like with all macros, this is just a textual replacement.
- Therefore `MULT(1 + 1, 2)` will yield 3, not 4, as it expands to `1 + 1 * 2`
- we can write `#define MULT(a, b) ((a) * (b))` to avoid such problems
- Why not just use a function?
  - C doesn't evaluate function calls at compile time, which prevents using the result e.g. for array bounds.
  - Macros can be used to generate boilerplate code like slightly differing structs, functions or constants
  - Macro arguments can be (partial) C code!

Function like Macros

When a macro contains multiple statements but is meant to be called like a function with a trailing semicolon and behave as expected e.g. when we do `if(some_cond) MY_MACRO(3);` then we need a little hack:

```
1 #define MY_MACRO(x) do{                \
2     foo(x);                           \
3     bar();                            \
4 } while(false) //no semicolon!
```

- The backslashes are necessary to continue the macro line, therefore no backslash is necessary on the last line
- if we had just put the statements without the do block, an `if` would only conditionalize the first statement.
- by leaving out the semicolon after the `while(false)` we create semantics equal to a (void) function call
- It's not possible to 'return' a value using this trick

Variadic Macros

Macros can also take a variable amount of arguments, the so called parameter pack is referenced using the special macro `__VA_ARGS__`

```
1 #define printfln(fmt, ...) printf(fmt "\n", __VA_ARGS__)
```

- parameter packs must always contain at least one argument, otherwise we might get an error
- many compilers allow it anyways or offer tricks to deal with empty parameter packs (e.g. gccs `##__VA_ARGS__`)

unix privileges

basics

In unix the file permissions can be set for the user, the group and for all users on the system.

A file can be executable, readable and writable for everyone/group/user.

Each user has a user id `uid` and each group has a group id `gid`.

permission numbers

octal	permission
0	`---`
1	`--x`
2	`-w-`
3	`-wx`
4	`r--`
5	`r-x`
6	`rw-`
7	`rwx`

easy to calculate `4(r) + 1 (x) = 5 (r-x)`

```
1 > ls -lah
2 -rw-r--r--  1 jakob jakob 2.7K Dec  7 11:49 default.php
3 drwxr-xr-x  2 jakob jakob 4.0K Dec  5 13:25 dl.txrx23.de/
4 -rw-------  1 root  root     5 Jan  3 08:42 extnetif.txt
```

setuid and setgid

With the setuid and setgid bit set the file will always be executed with the permissions of the owning group or the owning user.

This is indicated by a `s` in the user or group executable position

```
1 ~/u/c/c/src> ls -lah a.out
2 -rwsr-xr-x 1 root root 17K Jan  5 18:33 a.out*
```

in c

```c
#include <unistd.h>

uid_t getuid(void);
uid_t getgid(void);

uid_t geteuid(void);
uid_t getegid(void);

int setuid(uid_t uid);
int setgid(uid_t uid);
int seteuid(uid_t uid);
int setegid(uid_t uid);
```

The `getuid()` function shall return the real user ID of the calling process.

The `geteuid()` function shall return the effective user ID.

dropping privileges

we want to run with as little privileges as possible. So after needing privileges for starting our application, we drop as many as possible.

For this we want to set the effective user id of our program to a uid with less privileges

```
1 if (setegid(target_groupid) != 0)
2     //err
3 if (seteuid(target_userid) != 0)
4     //err
```

project

project

We are finished with the content, which is appropriate for a c course. So we thought it would be time for a more challenging task.

- Do you want to work on the project as a group?
- Do you have ideas on a small game we could write?
- Shall it be a game or something different?
- Should we have small blocks of content at the beginning of the next lessons?

Our proposal is a multiplayer rouge-like game. (basically a small ascii dungeon in which two people have to fight monsters)