# Tutorial 07 - 21.12./07.01.2021

Group 02/11 - Moritz Makowski - moritz.makowski@tum.de

## Higher-Order Functions, Pointers

# Today's Agenda

- Higher-Order Functions
- Pointers
- **Exercise 7.1: Pointers**
- Functions and Pointers
- Arrays and Pointers
- **Exercise 7.2: Arrays and Pointers**
- Structs and Pointers
- **Exercise 7.3: Practice Project - Stacks**

# Higher-Order Functions

Higher-order functions are functions which **receive a function as a parameter** and use that function inside their body.

By using this feature you can create logic with **less code** and **less functions**.

# Higher-Order Functions - Examples

Classic examples of higher order functions are:

- `map` : Apply a function to every element of an array
- `filter` : Given some array, remove all elements for which a given function returns false ( `0` )

See `example_7_1_ho_map_function.c` and `example_7_2_ho_filter_function.c` .

# Higher-Order Functions - Notation

```
void map(int (*func)(int), int *array, int length) {
    for (int i=0; i<length; i++) {
        array[i] = func(array[i]);
    }
}

int times_2(int x) {
    return 2 * x;
}

int main() {
    int my_array[10] = {1,2,3,4,5,6,7,8,9,10};
    map(times_2, my_array, 10);
}
```

*You **do not** have to fully remember this notation! Just keep it on your cheatsheet or know where to look it up.*

# What are Pointers?

Every time we create a variable, its value has to be stored somewhere in memory.

Each slot in memory has a specific address.

**A pointer is simply an adress of a slot in memory.**

*See* `example_7_3_pointer_intro.c`

# Getting the Address of a Variable

This is also called "Referencing".

Every pointer also has a type which is equal to type of its corresponding variable.

```
int *my_pointer;  // points to a memory slot which stores an integer
```

The star marks that `my_pointer` - which you can name however you want - is a pointer.

You can get the actual adress of a variable by using `&`:

```
int my_var = 55;
int *pointer_to_my_var = &my_var;
```

# Getting the Value Belonging to a Pointer (Address)

This is also called "**De**referencing".

You can get the corresponding value by using `*`:

```
int my_var_copy = *pointer_to_my_var;  // my_var_copy now also stores 55
```

```
printf("\nADDRESS of my_var = %p", pointer_to_my_var); // prints 0x7ff...
printf("\nVALUE of my_var = %d", my_var);               // prints 55
printf("\nVALUE of my_var = %d", *pointer_to_my_var);  // prints 55
```

# Exercise 7.1: Pointers

**(a)** Given `char c = 'K';`, what kind of type would I need to hold `&c`?

**(b)** Given `float *ff`, what kind of data does `ff` hold?

**(c)** Given `char *c`, what exactly is `&c`?

**(d)** What, if anything, is wrong with the following code?

```
int main(void) {
    char *some_value;
    char my_value = '!';

    some_value = my_value;

    return 0;
}
```

# Revision: Passing Data to Functions

What will be printed out?

```c
void add_10(int value) {
    value += 10;
}

int main() {
    int my_value = 5;
    add_10(my_value);
    printf("My value is %d\n", my_value);

    return 0;
}
```

See `example_7_4_passing_by_value.c`.

What happens inside `add_10`?

```
void add_10(int value) {
    // A local copy of "value" gets created
    value += 10;
    // The local copy now stores 20
}
```

To get this local variable "out of" the scope of `add_10` we'd have to include a `return` statement and a "re-assignment" inside `main`.

This way of passing data to functions is also called **"passing by value"**.

# Passing Data by Reference

You can also pass the pointer of a variable. Any copy of that pointer still points to the same slot in memory. And we don't modify the pointer itself but the memory space it points to.

```c
void add_10(int *value) {
    *value += 10;
}

int main() {
    int my_value = 5;
    add_10(&my_value);
    printf("My value is %d\n", my_value);  // prints 15

    return 0;
}
```

*See* `example_7_5_passing_by_reference.c` *on GitHub.*

# Arrays and Pointers

An array variable is actually a pointer to the first element of the array.

```c
int main() {
    int my_array[10] = {0,1,4,9};

    printf("ADDRESS of the 1st element = %p\n", &(my_array[0]));
    printf("ADDRESS of the 1st element = %p\n\n", my_array);

    printf("ADDRESS of the 2nd element = %p\n", &(my_array[1]));
    printf("ADDRESS of the 2nd element = %p\n\n", my_array + 1);

    printf("VALUE of the 3rd element = %d\n", my_array[2]);
    printf("VALUE of the 3rd element = %d\n", *(my_array + 2));

    return 0;
}
```

*See* `example_7_6_array.c` *on GitHub.*

*See `example_7_7_array_index_access.c` on GitHub.*

*See `example_7_8_array_pointer_access.c` on GitHub.*

# Exercise 7.2: Arrays and Pointers

```
char string_buffer[1000];
char *my_char_ptr = string_buffer;
```

**(a)** What kind of data would `my_char_ptr` be?

**(b)** What about `*my_char_ptr`?

**(c)** What about `*string_buffer`?

**(d)** Let's say you want to store `&(string_buffer[2])` somewhere. What type of variable do you need to store this?

**(e)** What is wrong with the following code?

```
char old_buffer[1000];
char *my_char_ptr = &old_buffer;
```

**(f)** What is wrong with the following code?

```
char old_buffer[1000];
char new_buffer[1000];
new_buffer = old_buffer;
```

**(g)** If you pass an array to a function and you change elements of the array in the function, what do you think happens when you leave the function?

# Structs and Pointers

There is a shortcut in accessing element inside a struct, that is present as a pointer.

```
struct Point {
    float x,
    float y,
    float z
};

struct Point *pointer_to_struct;
```

Access by dereferencing the struct first

```
(*pointer_to_struct).x = 1.2;
```

Clean version (does exactly the same):

```
pointer_to_struct->x = 1.2;
```

# Exercise 7.3: Practice Project - Stacks

Some background information about stacks: A stack is similar to a list but a stack only allows certain types of operations on its contents.

There are two main methods you can perform on a stack:

- `push` : Add an element to the stack (in case its storage capacity is not reached)
- `pop` : Remove and return the most recently added element from the stack (in case the stack is not empty)

*In german it is officially called "Keller", however "Stapel" is probably a better name.*

You can have further read about Stacks on Wikipedia:
https://en.wikipedia.org/wiki/Stack_(abstract_data_type)

## Your tasks

Download this folder from GitLab: `tutorial-07/exercises/stack_boilerplate`

This folder includes three files: `stack.c`, `stack.h` and `main.c`. Compile them with:

```
gcc -Wall -Werror -std=c99 stack.c main.c
```

... or inside the CMake file:

```
add_executable(
    stack_boilerplate
    exercises/stack_boilerplate/stack.c
    exercises/stack_boilerplate/main.c
)
```

**Tasks:** Implement the functions marked with `\\YOUR CODE HERE` so that the Stack behaves like it is supposed to.

*You can modify `main.c` as you like to test all of the functionality.*

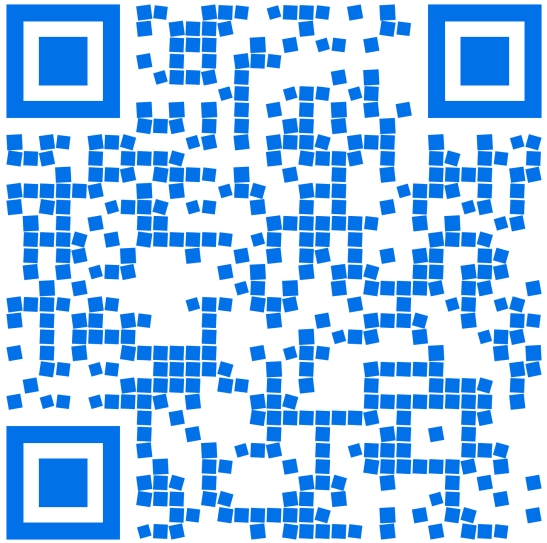
*You can also test your implementation with a proper test procedure:*

```
gcc -Wall -Werror -std=c99 stack_boilerplate/stack.c stack_solution/main_correction.c
```

# See You Next Week!

All **code examples** and **exercise solutions** on **GitLab** (solutions right after my tutorial):

https://gitlab.lrz.de/dostuffthatmatters/IN8011-WS20



When you copy a snippet from StackOverflow and it doesn't works