Tutorial 08 - 11.01./14.01.21

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

Dynamic Memory Allocation, Recursion, Practice Project "Parentheses Logic"

Today's Agenda

- Repetiton: "At runtime"
- Dynamic Memory Allocation
  - Malloc
  - Calloc
- Recursion
- Exercise 8.1: Practice Project - Parentheses Logic
- Free Practice

"At Runtime"

"At runtime" basically means "while the program is running".

If you change a value inside your code, recompile it and run it again, you *did not* change that value at runtime.

"At runtime" means that you don't have to change your code and therefore don't have to recompile it. The value will be determined during the progams execution lifecycle.

Static vs. Dynamic Memory Allocation

Until now, in order to initialize an array we needed to now its exact size before the execution:

```
// We need to now in advance that the array
// has to store (at most) 80 elements
int my_array[80] = {0};
```

But what if we don't now how much memory we will need in advance. And what about the case that this amount is determined at runtime?

```
int var = ... // Determined at runtime

int my_array[var] = {0} // Doesn't work!!
```

We call the type of memory allocation we've used until now **"static memory allocation"**.

In case the memory is not allocated in advance but at runtime we call this process **"dynamic memory allocation"**.

Dynamic Allocation

There are two ways we can dynamically allocate memory using `malloc` or `calloc` from the `stdlib` -library:

```
#include <stdlib.h>

int var = ... // Determined at runtime

int *my_array_1 = malloc(sizeof(int) * var);
int *my_array_2 = calloc(sizeof(int), var);
```

The difference between `malloc` and `calloc` is that `calloc` also initializes all the memory to 0, whereas `malloc` does not.

It is very important to **`free` up the memory** you dynamically allocated so it can be reused once you don't need it anymore.

```c
#include <stdlib.h>

int var = ... // Determined at runtime
int *my_array = calloc(sizeof(int), var);

...

free(my_array);
```

See `example_8_1_static_vs_dynamic_allocation.c`, `example_8_2_dynamic_allocation_malloc.c` and `example_8_3_dynamic_allocation_calloc.c`.

Dynamic Memory Allocation - Summary

In these code snippets `var` is a number (`var > 0`) which determines the size of the memory block, that you can now work with.

```
// Static memory allocation
int my_array_1[20] = {0};

// Dynamic memory allocation
int var = 20;
int *my_array_2 = calloc(sizeof(int), var);
```

You can treat `my_array_1` and `my_array_2` in the exact way (as an integer array). The only difference is, that you have to free the dynamic memory after you are done using it:

```
free(my_array_2);
```

What is Recursion?

The basic concept of a recursively defined function is that `f(n+1)` is calculated based on `f(n)`, `f(n-1)`, `f(n-1)`, ...

Semi-formal definition:

$$f(n) = g(f(n-1), f(n-2), ...)$$

You always need some termination point (where the recursion ends). For example:

$$f(0) = 0, \quad f(1) = 1$$

$$f(n) = f(n-1) + f(n-2) \quad \forall n > 1$$

Recursive Definitions - Examples

The **factorial operator** $x!$ for $x \in \mathbb{N}_0$:

$$x! = \begin{cases} 1 & \text{if } x = 0 \\ x \cdot ((x-1)!) & \text{otherwise} \end{cases}$$

The **modulo operator** $x \bmod m$ for $x \in \mathbb{N}_0, m \in \mathbb{N}$:

$$x \bmod m = \begin{cases} x & \text{if } x < m \\ (x-m) \bmod m & \text{otherwise} \end{cases}$$

Implementing the Factorial Operator

Task: Calculating the factorial of a number `n >= 0`

Traditional approach:
See `example_8_4_factorial_loop.c` on GitHub.

Recursive approach:
See `example_8_5_factorial_recursion.c` on GitHub.

Exercise 8.1: Practice Project - Parentheses Logic

Write a program that tests whether a given string including parentheses (`(` and `)`) is fulfilling the rules for setting parentheses:

1. For every opening parentheses there exists a closing parentheses and vice versa
2. Every closing paretheses appears after the respective opening parentheses.
3. Other characters do not play a role in this logic

Super Bonus: Expand your program to support different types of braces -> `(` / `)`, `{` / `}`, `[` / `]`. It is important that they don't interfere with each other, e.g. `([...)]` is invalid!

Hint: You can make use of the stack we built last week.

Free Practice

Choose the exercises you want to do from `additional_exercises`.

Smaller step-by-step problems with increasing difficulty:

Exercise 3: String Palindrome

Exercise 4: String Anagram

Exercise 5: Integer Palindrome

Exercise 6: Integer Anagram

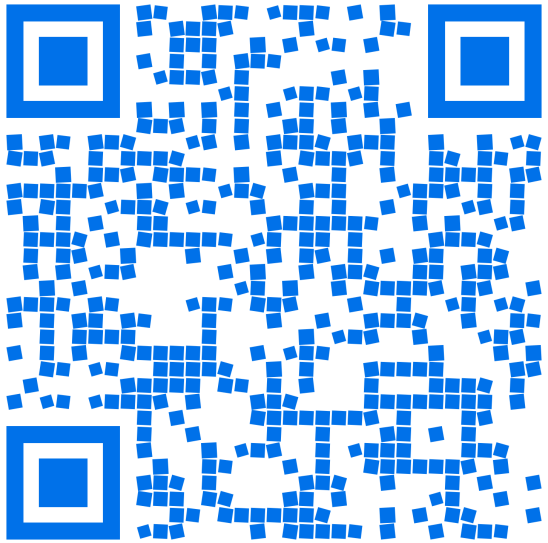
One big task (not that easy, but very rewarding). More of a take-home-project!

Exercise 13: Conway's Game of Life

See You Next Week!

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

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



Junior devs writing comments: