Tutorial 08 - 10.12.2019

Group 06 - Moritz Makowski

Dynamic Memory Allocation, Recursion, Practice Project "Equation Solver"

Today's Agenda

- Repetiton: "At runtime"
- Dynamic Memory Allocation
  - Malloc
  - Calloc
- Recursion
- **Exercise 8.1: Practice Project - Equation Solver**
- **Exercise 8.2: Practice Project - Parentheses Logic**

"At Runtime"

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

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 itthe 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 = malloc(sizeof(int) * var);
```

```
int var = ... // Determined at runtime
int *my_array = 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 other programs can make use of it after your program's execution is over.

```c
#include <stdlib.h>

...

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

...

free(my_array);
```

See `example_8_1_static_allocation.c`, `example_8_2_dynamic_allocation_malloc.c` and `example_8_3_dynamic_allocation_calloc.c` on GitHub.

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)`, ...

If for example `f(n+1)` can be fully calculated by knowing `f(n)` and `f(n−1)` you only have to know two values of `f(...)` belonging to to consecutive values `a` and `a+1` in order to calculate `f(...)` for every value larger than these starting values.

Recursion Example:

Task: Calculating the factorial of a number `n` largen than 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 - Equation Solver

Write a program in which you defined a String inside the code which contains a mathematical expression and solve that expression.

Example:

```
char math_string[100] = "3*4+15+78-3*5";

// solve_equation is the function that you should write
// Its signature isn't required to look like this.
int result = solve_equation(math_string, 100);  // returns 90
```

Of course you can also have the user dynamically input an equation, if you want.

You can set your desired **level of difficulty** and work your way up:

```
char easy_string[100] = "1+40-55-30+16";

char hard_string[100] = "1+40*60-55-30/16";
```

See `additional-exercises` for even harder strings to solve.

Additional Idea: Whenever an equal sign `=` is found inside a string the thing is treated as an equation.

Example:

```
char equation_1[100] = "3*4+15=27";
char equation_2[100] = "3*4+15=23";
char equation_3[100] = "3*4+15=4*5+7=5*5+2";

int result = solve_equation(equation_1, 100);  // returns 1
int result = solve_equation(equation_2, 100);  // returns 0
int result = solve_equation(equation_3, 100);  // returns 1
```

Exercise 8.2: 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!

See You Next Week!

All **code examples** and **exercise solutions** (available right after my tutorial) on **GitHub**.
https://github.com/dostuffthatmatters/Engineering-Informatics-1-MSE-WS1920.


print statements
Full fledged
debugging libraries