Struct

A `struct` forms a compound type out of several individual variables.

- It allows to store several pieces of data in one variable
- OSs generally use them for control data structures.
- individual components of a `struct` are called members

`fork()`

This function creates a new process. The return value is the zero in the child and the process-id number of the child in the parent, or -1 upon error.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char const *argv[])
{

    pid_t pid = fork();
    switch (pid) {
        case -1:
            printf("Error. fork failed\n");
            break;
        case 0:
            printf("I am the child.\n");
            break;
        default:
            printf("I am the parent.\n");
            wait(NULL);
            printf("Child terminated.\n");
    }

    return 0;
}
```

`execve()`

This can be used to replace the currently running program with another, for example to load a new binary into the **current** address space. When a shell

creates a new process to execute some program, it will first fork itself and then invoke `execve()` within the child process to replace the shell code with the code of the program that shall be executed.

Parameter Passing : Stack

Using a stack for parameter passing has two main advantages:

1. The context of each callee (its frame) is automatically pushed onto and popped from the stack, no additional actions are needed.
2. The parameters being pushed onto the stack are accessed via the stack frame pointer, thus you can implement procedures with a variable number of parameters very easily.

Function vs. Macro

The code of a function exists only once, independent of how often the function is being called.

For macros, each call results in its repective code being inserted at the call-site.

- In C, macros are expanded in the source code by the pre-processor.
- Code with many macros results in larger code than the same code using functions.
- Runtime behaviour: function calls incur a small overhead compared to macros for jumping to and back from the called function
- Macros are untyped: The compiler does not check whether the macro arguments have a specific type.
  - This offers flexibility but is also a well-known source of errors.