# Linux kernel and C programming

## BLOCK 1: Introduction to the C language

Gergely Korcsák

Óbuda University
*korcsak.gergely@nik.uni-obuda.hu*

February 13, 2025

# Presentation Overview

# Presentation Overview

# Presentation Overview

# Introduction

Name: Gergely Korcsák

Email: korcsak.gergely@nik.uni-obuda.hu

# Requirements

The course will cover basic programming knowledge of the C language, and how to create linux kernel modules and user applications.

**Requirements:**

- Attendance
- 50p One midterm exam
- 50p One assignmet

The midterm exam and the assignment will worth 50 points each. The total of 100 points will result in the closing mark.

# Semester

1. Introduction to C (hello world, gcc, CMake)
2. Memory basics (cache, stack, heap, pahe(s))
3. CMake (variables, if/else)
4. Function pointers, unions and argc/args
5. Typedef, extern, static and compiler optimization
6. Introduction to Linux kernel modules (kmake, files)
7. Kernel proc files
8. Kernel dev files
9. C compile process (preprocess, compile, link)
10. CMake linker flags
11. Midterm exam practice
12. Midterm exam
13. Midterm assignment presentations
14. Midterm exam and assignment presentations retake

# Purpouse

The purpouse of this course it to get the technical knowledge of the C programming language.

This course is not for beginners, I expect you to have some prior programming experience. (like C)

# Introduction to C

# History

The C programming language is very simple. Altaugh using it can be quite difficult, due to memory management and other. But provides freedom.

C is a general-purpose programming language, and was created in the 1970s by *Dennis Ritchie*.

It was mainly developed to replace assembly, and create portable code/programs.

# Types

C is a strongly typed programming language. To declare a variable you need to write the `type` before it. The size of a variable will depend on the architecture, but lets see the sizes for a 64 bit processor.

| **Type** | **Size** | **Description** |
|---|---|---|
| `char` | `8 bit` | Signed Character (treated as int) |
| `short` | `16 bit` | Short Signed integer. |
| `int` | `32 bit` | Signed integer. |
| `long` | `64 bit` | Long Signed integer. |
| `float` | `32 bit` | Single precision floating point. |
| `double` | `64 bit` | Double precision floating point. |

You can read out the bytesize of a variable by using `sizeof()`.

# Variables

Variables are (?)not zeroed out at initialization. Therefore when we declare a variable, by default they will have the previous value of the memory they were assigned to.
To assign a variable we can use the **=**, **+=**, **-=**, ***=**, **/=** and **%=** operators

```
1 int a;
2 int b = 0;
3
4 int c = 2, d = 4;
```

We can do math between them, by using the **+**, **-**, ***** and **/** operators.

```
1 a = d - b;        // a: 4
2 int e = a * c     // e: 8
```

# Type modifiers

| Modifier | Description |
|---|---|
| `signed` | Ensure a variable is signed. |
| `unsigned` | Ensure a variable is unsigned. |
| `short` | Shortens the variable size by half. *(if possible)* |
| `long` | Doubles the variable size. *(if possible)* |

```
1 unsigned int a = 4;
2 long long b = 3;
3
4 b += a;
```

# Macros

Macros will only be used at preprocessing, and will be inserted into the code as is. They can contain function like variables, which will be used at preprocessing. Lets se a short example:

```
1  #define debug_string    "[debug]: "
2  #define mystring        "words in my head"
3  #define number1         75434
4  #define number2         1234
5  #define sum(a, b)       a + b
6  #define mul(a, b)       (a * b)
7
8  printf(debug_string "%s - %d\n", mystring, sum(number1, number2
       ));
9
0  // will be compiled as:
1  printf("[debug]: " "%s - %d\n", "words in my head", 75434 +
       1234));
2  // will print: [debug]: words in my head - 76668
```

# Memory

You can look at computer memory as an array of bytes. It can be indexed from 0 to the sky.

When running your program you will get a slice of memory to use from the kernel. This is where your application will be loaded.

# Pointers

Pointers are also variables, which point to an arbitrary memory location.

```
1 int a = 10;
2 int* p = &a;     // will set the value of 'p' to the address of
      'a'
3 *p = 5;          /* will set the value at the address inside 'p'
       to 5,
4                     therefore 'a' will be 5 */
5 p = 10;          /* will set the value of 'p' to 10,
6                     the value of 'a' will stay 5 */
```

# Pointer arithmetics

| Modifier | Description |
|---|---|
| `&p` | Get the memory address of variable `'p'`. |
| `*p` | Get the value pointed by `'p'`. |
| `p++` | Multiply the value of `'p'` by the bytesize of its type. |
| `p--` | Substract the value of `'p'` by the bytesize of its type. |
| `p + n` | Multiply the value of `'p'` by the bytesize of its type times `n`. |
| `p - n` | Substract the value of `'p'` by the bytesize of its type times `n`. |
| `p[n]` | The same as `*(p + n)`. |

# Arrays

In C, arrays are pointers, with a predefined length of memory. They can be declared like the following:

```
1 int array[10];
2 int array[5] = { 0, 1, 2, 3, 4 };
3 int array[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

We can access the element of an array, by its index like:

```
1 array[0] = 2;
2 array[2] = 4;
3 int a = array[3];
```

# Arrays

We can also user pointers as arrays:

```
1 int array[10];
2 int* p = array;
```

An array is just a pointer to an already allocated memory area. Therefore we can also get the n-th value like the following:

```
1 int array[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
2 int a = *(array + 3);    // a will be 3
```

# Conditions

C don’t have booleans, therefore a condifion is true if it is not 0, and false if it is.

```
1 if (0) {
2     // Will always be false
3 }
4 if (1) {
5     // Will always be true
6 }
7 if (-5) {
8     // Will always be true
9 }
```

# Conditions

We can use the following conditional operators:

| Operator | Description |
|---|---|
| ! | Not |
| > | Greater than |
| < | Lesser than |
| == | Equals |
| != | Not equals |
| && | And |
| \|\| | Or |

# Loops

We can talk about `while`, `do while` and `for` loops.
While and do while will run until the provided condition is not 0.

```c
int i = 5;

// Will print the numbers from 5 to 1
while (i) { // can also be (i > 0)
    printf("%d\n", i);
    i--;
}

// Will print the numbers from 0 to 4
while (i < 5) {
    printf("%d\n", i);
    i++;
}
```

Listing 1: While loop

# Loops

```c
int i = 0;

// Will print the numbers from 0 to 4
for (i = 0; i < 5; i++) {
    printf("%d\n", i);
}
```

# Bitwise operations

Bitwise operations execute the following on eather one or two different variables:

| **Operator** | **Description** |
|---|---|
| & | And |
| \| | Or |
| << | Shift Left (x2) |
| >> | Shift Right (/2) |
| ~ | Inverse |
| ^ | XOR |

# Structs

C structs are complex variables with named fields. The variables will be declared at the same memory location, after each other. They are often used in headers, like in TCP.

```
1  struct my_struct {
2      int a;
3      int b
4      float c;
5      bool d;
6  }
7
8  struct my_struct my_struct_var1 = { 0 };     // fill with 0
9  struct my_struct my_struct_var2 = {
0      .a = 5;
1      .c = 0.6f;
2  };
3
4  my_struct_var1.b = 12;
```

# Functions

```c
/*
 * <return type> function_name(<void/int a, ...>) {
 *      // do stuff
 * }
 */

int main() {
    printf("hello world!");
    return 0;
}
```

# Headers

Header files are like interfaces in other programming languages. Depending on the usage they contain defines, types definitions and function definitions.
We can include them with: `#include "header.h"`.

They shouldn't contain function declarations!

You will also see `#include <stdio.h>`. The difference between refering to an include with "" and <> is that brackets are used for the standard library and system headers, while double quotes refers to user defined "local" headers.

# Headers example

```c
// my_header.h
struct my_struct {
    int a;
}

int my_func(void);
```

```c
// my_program.c
struct my_struct struct_var = { 0 };

int my_func() {
    return 13;
}
```

# Headers example

```c
// main.c
#include <stdio.h>
#include "my_header.h"

int main() {
    struct_var.a = my_func();
    printf("My a: %d", struct_var.a);
    return 0;
}
```

# Strings

Strings are an array of characters. They can eather be stored in an ”array”, or a pointer.
They are always terminated by a trailing `0` character, which can be written with .

```
1 char* my_string = "Hello World!";
2
3 printf("My message: %s\n", my_string);
```

# Printf

`printf()` is part of the standard library. It requires a format string, and extra arguments if they are defined in the format string. The syntax is:

```
1 #include <stdio.h>
2 printf("format_string", args...);
```

The format string can contain format specifiers:

| Specifier | Description |
|---|---|
| %d | Signed integer |
| %u | Unsigned integer |
| %c | Character |
| %f | Floating-point number |
| %x | Hex number format |
| %s | String closed by a zero character \0 |
| %% | Print the % character |

# Printf

The format specifier can also be extended for prefilling by writing the charecter and the length of characters printed. For example:

```
1 printf("My Hex: 0x%08x\n", 0xAB13);
2 // will print: My Hex: 0x0000ab13
3
4 printf("My Hex: 0x%08X\n", 0xAB13);
5 // will print: My Hex: 0x0000AB13
```

The precision of floating-point numbers can also be specified as:

```
1 printf("My float: %.2f\n",      2.3456);
2 // will print: My float: 2.34
3 printf("My float: %02.1f\n",    2.3456);
4 // will print: My float: 02.3
```

# Compileing

# GCC

# CMake

# Hello World

Lets write our first application. Create a `hello_world.c` file, and write the following:

```c
#include <stdio.h>

int main() {
    printf("Hello World!\n");
    return 0;
}
```

Lets compile our program with:

```
gcc hello_world.c -o hello_world
```

# Excercises

# The End

Questions? Comments?