Lesson 1

Christian Schwarz, Jakob Krebs

28.10.2019

Contents

Introduction

Getting started

- we will use mostly linux
- all slides, examples and tasks will be available at `https://jkrbs.github.io/c_lessons/`
- the source resides on github at `https://github.com/jkrbs/c_lessons`
- all tasks can be send to us via e-mail and we will prvide feedback `c-lessons@deutschland.gmbh`
- weekly lessons **Mondays, 13:00-14:30**

development envirement

- you can use any editor of your choice
- you also can use an ide like vscode, atom, ...
- we will use a commandline, vim and gcc

gcc for Unix-based operating systems

Ubuntu / Debian:

```
$ sudo apt-get install gcc
```

Arch Linux:

```
$ sudo pacman -S gcc
```

Mac OS X:

```
$ brew install gcc
```

... and you're done ;-)

gcc for Windows 10 (using WSL)

For convenience you should use the new Ubuntu-based Linux subsystem.

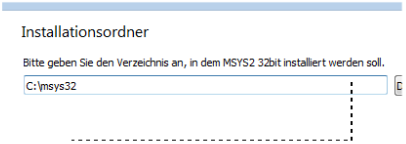
- In *Settings*, got to *Update & Security* > *For Developers* and switch to *Developer Mode*
- In the *Control panel*, go to *Programs* > *Turn Windows Features On or Off* and enable the *Windows Subsystem for Linux (Beta)*
- Reboot as you're prompted
- Search for "bash" and run the *bash* command
- Follow the installation instructions

You may now continue as if you were using Ubuntu ;-)

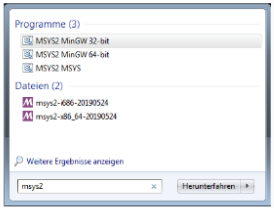
gcc for older versions of Windows (using msys2)

1. www.msys2.org → msys2-x86_64-20190524.exe →

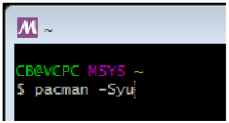
2.



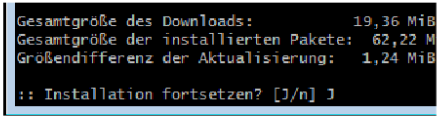
→



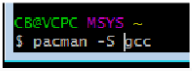
3.



→



4.



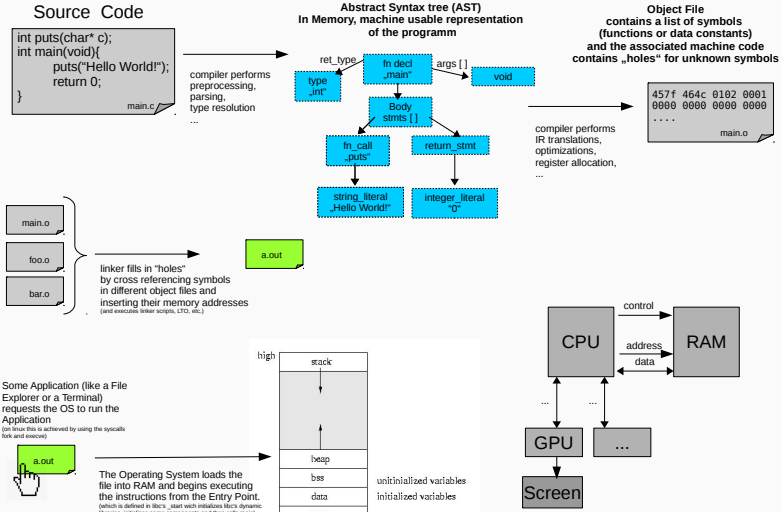
Hello World!

The first program

- Create a new file named "**main.c**".
- Open it in your text editor of choice.
- Fill it as follows:

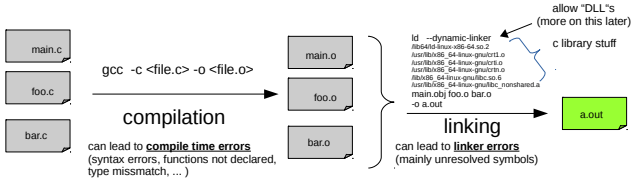
```
int puts(char* c);
int main(void) {
    puts("Hello World!");
    return 0;
}
```

From source to binary

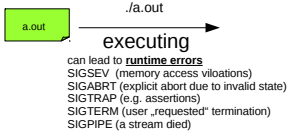


Compiletime vs Runtime

„compile time“



runtime



Program structure

A basic program

```
#include <stdio.h>

int main(void) {
        puts("Hello World!");
        return 0;
}
```

} Preprocessor statements

} Main function

Preprocessor statements

- Processed before compilation
- Have their own language; start with a #

```
#include <stdio.h>
```

- Includes the *input/output header* from the **C standard library**
- Needed to use *puts()*

Preprocessor statements have way more use cases,
but they form their own language which is very different from actual C.

In this course, we will use them for inclusions only.

The main function

- Core function of every program
- Exists **exactly once** in every program
- Called on program start

```
int main(void) {
```

- As a function, *main()* can take parameters and return a value
- Get used to *void* and *int*. They will be explained later
- '{' marks the start of the main function scope

The main function scope

- Contains program statements
- They are processed from top to bottom

```
        return 0;
}
```

- Last statement; ends main function (and thus the whole program)
- *0* tells the OS that everything went right
- '}' marks the end of the main function scope

Statements

- Instructions for the computer
- End with a ; (semicolon)

```
puts("Hello World!");
```

- Here is the empty statement:

```
;
```

- All statements are located in function blocks

Comments

- Information for you and others who use your code
- Cut out before compilation

Single-line comments:

```
// Prints "Hello World!" on the command line
```

Block comments (multi-line):

```
/* Prints "Hello World!"
   on the command line */
```

Better style of block comments:

```
/*
 * Prints "Hello World!"
 * on the command line
 */
```

Style

A few words on style

- There can be multiple statements on one line
- Indentation is not necessary at all

A few words on style

- There can be multiple statements on one line
- Indentation is not necessary at all
- **But**...

```
#include <stdio.h>
int
main     (          void ){puts("Hello World!");
                   // Prints
/*"Hello World!"                              */
                   return 0;}
```

Write enjoyable code

- Put each statement onto its own line
- Indent every statement in the main function by one *tab* or a fixed number of *spaces*
- Decide on a commenting style and stick to it (/* .. */ recommended)
- Leave blank lines between different parts of the program
- Use *spaces* and *newlines* consistently
- Later on just install **clang-format** and stop worrying