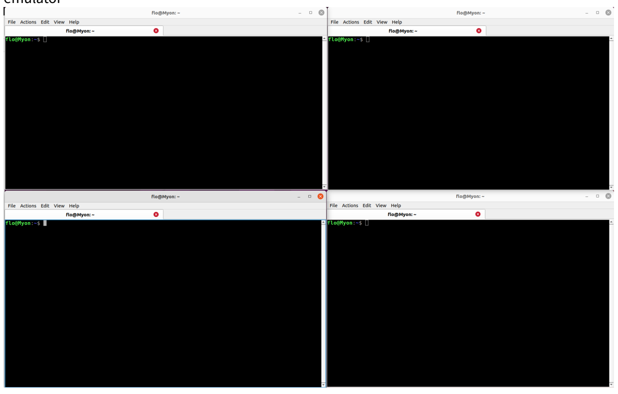
Compiler Construction
Prof. Dr. Florian Heinz
florian.heinz@oth-regensburg.de
Exercise Sheet 1

Exercise 1: System and Terminal

- Boot up Ubuntu or your own Unix-like System
- Open 4 shell windows (terminals) in parallel. Make sure that a monospaced font is used. Proportional fonts are strongly discouraged. On the CIP Pool hosts **QTerminal** is a suitable terminal emulator



```
flo@Myon:~$ 
```

ℹ Some Tips:

- Copy&Paste from/to the terminal window can be done by selecting text with the mouse and using `STRG+SHIFT+C` for copy and `STRG+SHIFT+V` for paste
- Try it out by writing "Hello World" in one terminal window and copying it into another one!
- The shell has a nifty TAB-completion feature. Try it out by typing "q" and use `TAB` twice to show possible completions. Type some more letters and try `TAB` to see, how the suggestions change. The completion is performed when only one possibility is left.
- Not only commands are completed, but also files in the filesystem. Try "ls /" and press `TAB` twice. What happens if you do that with "ls /sb"?
- Play around with `TAB`-completion a bit more.
- The shell itself has some useful line editing capabilities. Some examples:
  - `STRG+A`: Jump to the start of the line
  - `STRG+E`: Jump to the end of the line
  - `STRG+U`: Cut text to the beginning of the line
  - `STRG+K`: Cut text to the end of the line
  - `STRG+W`: Cut the previous word
  - `STRG+Y`: Paste text from the buffer
- Try out the shortcuts above. For example, from "Hello World" create "Hello WorldWorldWorldWorld"
- Search the web for "bash shortcuts" and try out some more. On the CIP pool hosts, the browsers **librewolf** and **falcon** are available.
- You can also set variables in a shell with `VAR=value` and retrieve it with `${VAR}`
  Try setting the variable `W` to the string "World" and output "Hello World" using this variable. Use the command "echo" for printing
- A variable can also iterate over a set of words. The syntax is:

```
for VAR in word1 word2 word3; do somecommand ${VAR}; done;
```

- Use this construct to produce the following output:

```
Color: red
Color: green
Color: blue
```

- Try out the command `seq` to produce all even numbers from 4 to 16. Consult the manpage of this command with `man seq` to find out how to do that

ℹ For most commands a manpage is available, as well as a short help when invoked with `--help`

- Try out the following: `echo Numbers: $(seq 1 10)`. What does the construct `$(cmd)` do?
- Combine this with a loop by iterating over the numbers from 1 to 100 and outputting:

```
Number: 1
Number: 2
...
Number: 100
```

Exercise 2: Filesystem

ℹ In this course we primarily work in the shell instead of using graphical file managers or IDEs.

- Get used to the filesystem commands:
  - ls -l - Directory listing
  - mkdir - Create a directory
  - cd - Change current working directory
  - touch - Create an empty file
  - mv source target - Move file source to target
  - cp source target - Copy file source to target
  - cp -a source target - Copy directory source to target
  - rm - Delete a file
  - rm -r - Delete a directory including all files and subdirectories in it (Caution!)
  - rmdir - Delete an empty directory
  - tar xvfz archive.tar.gz - Unpack a gzipped tar-archive
  - wget http(s)://someurl/ - Download a file from the web into the local directory
  - less filename - Display file contents (end with "q")

  ℹ Caution: Linux filenames are case-sensitive!

- Perform the following steps:
  - Create a directory "ex1.2"
  - Change into that directory
  - Download and unpack https://dcmc.othdb.de/public/make1.tar.gz with wget and tar
  - Delete the new directory `make1` with all files in it
  - Unpack it again
  - Switch into the directory `make1` and examine which files are in it
  - View the contents of the files
  - Type `make` and look again, which files have been created
  - An executable `hello` is now available in this directory. Start it.
  - Remove this executable
  - Invoke `make clean` to clear the build

Exercise 3: vi

ℹ VI is a standard editor available on almost all unix distributions.

- Go into the directory `make1` and open the file `hello.c` with the vi editor

  ℹ VI has two modes: command and insert. When starting vi, the command mode is active

- Move around in the source file using the commands **h j k l**. This is a bit like "wasd" in computer games.
- Move to the line with the "printf" command and press **y**. The line is now copied to the paste buffer
- Insert the paste buffer with **p**
- Delete the additional line again by issuing **d d**. Hint: The deleted line is additionally copied to the paste buffer.
- You can undo the last edits by pressing **u** (repeatedly, if necessary)
- It is also possible to redo edits with `STRG+r`
- Try that out by undoing the deletion performed previously and redoing it afterwards
- To write text we have to change into the **insert mode**. Move the cursor after the word "Hello" and press **i**
- Now you can write text. Change the line to read "Hello World"
- Leave the insert mode and enter command mode again by pressing `ESC`
- You can save your precious work by pressing **:w** in command mode
- Find out how to do search/replace with vi by looking for vi cheat sheets in the web. Replace "World" with "World!"
- Read the cheat sheets a bit further to find additional useful commands, there are a lot!
- Leave the editor with the command **:wq**. This also saves the file additionally.

Exercise 4: Makefiles

Makefiles (GNU make) are used to create a build of a project. The structure of a Makefile is dependency based.

- Open the file make1/Makefile and have a look at its structure:
  The special variables CC and CFLAGS are used when compiling a C-sourcefile. CC specifies the name of the c compiler binary, CFLAGS are added as compiler flags when translating a source file.

  ```
  CC=gcc
  CFLAGS=-O2 -Wall
  ```

  General purpose variables, like LDFLAGS and OBJ in this example, can be set and used with $(OBJ)

  ```
  LDFLAGS=-lpthread
  OBJ=main.o hello.o
  ```

  A Makefile rule consists of a target name, a dependency list and actions:

  ```
  target: dep1 dep2 dep3
       action1
       action2
  ```

  The dependencies are expected files. If there is no file present with that name (dep1 dep2 dep3), then a target with that name is looked for. This target then should <u>create</u> a file with that name.
  Example:

  ```
  print: hello.txt
      cat hello.txt

  hello.txt:
      echo Hello World > hello.txt
  ```

  When invoking make a target that should be built can be passed, in this case:

  ```
  make print
  ```

  If no target name is passed, then the first defined target is built.

- Now create a new directory make2 on the same level of make1
- Change into that directory
- Use the editor **vi** to create a Makefile
- Type in the two targets **print** and **hello.txt**. Note that the actions have to be indented using TAB, not spaces
- Save and exit the editor
- Type make print to build the target **print**. What happens?
- Run make again. What happens now? Why is it different?
- What would happen if we had an additional action touch print for the target "print"? Try to anticipate the behavior before actually testing it.

  Make is often used for building C projects. For that reason, targets for C object files are implicitly defined. View the file "make1/Makefile" again.

  The target "hello" has two dependencies (carried by the variable OBJ) *hello.o* and *world.o*. If you look closely, you notice that no targets for these are defined, yet the project builds when running make. The reason is, that the following implicit targets for these exist:

  ```
  hello.o: hello.c
      $(CC) $(CFLAGS) hello.c -c -o hello.o

  world.o: world.c
      $(CC) $(CFLAGS) world.c -c -o world.o
  ```

  This is true for all targets with suffix *.o*.

  So, if you start make in the directory make1 the following happens:

- The build system starts to build the first defined target "hello"
- Now, the make system expands the variable OBJ to the two dependencies
- Make looks for the first dependency "hello.o", which is not present, so the target "hello.o" is invoked
- Target "hello.o" is implicitly defined and looks for the dependency hello.c, which is present.
- The implicit rule invokes gcc to compile the object file hello.o from hello.c
- Same procedure for the second dependency "world.o"
- Now, that all dependencies for the target "hello" are fulfilled, the action for that target is executed:

  ```
  $(CC) -o $@ $^
  ```

  which expands to

  ```
  gcc -o hello hello.o world.o
  ```

  so the special variable **$@** expands to the current target, while **$^** expands to the dependencies for the current target. "gcc" is not invoked as a compiler here, but used to link the two object files to a single binary "hello".
- If you run make multiple times in that directory, what happens after the first time?
- Change the printf output in the source file "world.c" with vi and re-run make. What happens and how does that work?

  Make checks, if the modification timestamp of the dependency files is more recent than the timestamp of the target. If it is, it assumes that the target has to be rebuilt, because a dependency changed.

Exercise 5: Other tools

Make sure that the following tools are available at your system:

- gcc
- make
- objdump
- vi
- bash
- GNU flex
- GNU bison
- A terminal emulator (e.g.: xterm, qterminal, gnome-terminal)

Exercise 6: Programming exercise

- Create a folder prog1
- Create a source file "check.c", that contains a function "int check(char *nr)" that takes a numeric string and returns 1, if the represented number is divisible by 3. Use the vi for this
- Create a source file "main.c", that contains the main function. This function should open a file "numbers.txt", read it line by line, invoke the "check" function on it and print out the number if the check was successful
- Now write a Makefile to build the project. The main target should be called "check" and produce a binary from the two source files above. Additionally, a dependency and target "numbers.txt" should exist that generates a list of numbers from 1 to 1000 with step 7 (1 8 15 22 ...) and writes it into a file "numbers.txt". Make sure the source files compile with full warning (CFLAGS -Wall) and optimization level 2 (-O2).
- Try out your program if it works as expected.

ℹ Use the manpages. They also contain manuals for the C standard functions (e.g. try `man fgets`). Try to use graphical tools or the browser as few as possible.