Intro Linux, C, PowerShell
Erik Hjelmås, August 4, 2025

Contents

1 Getting Started **3**

1.1 If student at NTNU: 3

1.2 If student at CISK: 3

2 Why Bother? **3**

2.1 What is "the Shell"? 4

2.2 Navigation 4

2.3 Looking Around 5

2.4 A Guided Tour 5

2.5 Manipulating Files 6

2.6 Working with Commands 7

2.7 I/O Redirection 7

2.8 Expansion 8

2.9 Permissions 10

3 Writing Shell Scripts **11**

3.1 Writing Our First Script and Getting It to Work 11

3.2 Editing the Scripts We Already Have 11

3.3 Here Scripts (aka "Here document") 11

3.4 Variables 13

3.5 Command Substitution and Constants 13

3.6 Flow Control - Part 1 14

3.7 Positional Parameters 15

3.8 Flow Control - Part 3 15

4 PowerShell **18**

5 C programming **20**

5.1 Compiling and Executing 20

5.2 The printf-function 23

5.3 Command line arguments 27

5.4 Exercises 28

1 Getting Started

1.1 If student at NTNU:

Create your own Linux virtual machine in SkyHiGh by following the instructions Basic Infrastructure Orchestration AND USE THE YAML FILE single_linux.yaml. Log in to the linux by following the instruction at the bottom of the page. If you need help, there is also a video (but note that in the video the name of the yaml file to use is not correct).

1.2 If student at CISK:

Create your own Linux virtual machine ("Linux-VM", "Linux-host", "Linux-machine" or just "Linux", we use these terms interchangeably) on your laptop by following the instructions: How to install Linux on Windows with WSL. Install the latest Ubuntu LTS-version (LTS is Long-Term Support) which by the time of this writing is Ubuntu 24.04 (24.04 because it was released in April (the fourth month) in 2024). You have two ways of accessing your Linux after you have installed it, and you need to master both ways:

Opening a terminal Start the "Ubuntu" program from the Windows menu.

Opening a ssh-connection To be able to do this you need to install a ssh server inside the Linux-VM first:

1. Access you Linux by opening a terminal as described above. Execute the following commands:

   ```
   sudo apt update
   sudo apt install openssh-server
   ```

2. Execute the command `ip address` to see all the IP-addresses that exist on different interfaces in your Linux. You should see an IP-address that starts with 172.16, write this down.
3. From your Windows-laptop you can now open PowerShell and write `ssh ubuntu@YOUR_IP_ADDRESS` where you replace `YOUR_IP_ADDRESS` with the 172.16-address you found (maybe you can also just write `ssh ubuntu@localhost`).

Whenever you see an exercise on the following pages that refers to ssh, access your Linux with ssh before doing that exercise.

2 Why Bother?

We will follow this excellent tutorial "Learning the shell" in detail. This tutorial together with this leaflet are your mandatory readings and include all the exercises you need to do.

2.1 What is "the Shell"?

Read the text and study/do the examples on What is "the Shell"?. Additional notes from me:

- Make sure you are using Linux as a regular user and not as the root user (doing some exercises as root and some as regular user will cause annoying access control problems for you later, trust me :)
- Notice how you use the arrow buttons for command line history, and how you can copy and paste text (you cannot use CTRL-C since that terminates a running program instead of copying text)

Exercises:

1. What is the difference between *a shell* and *a terminal*?
2. What is the name of the shell we are using?
3. What is the alternative way of interacting with computers instead of using a command line shell?

2.2 Navigation

Read the text and study/do the examples on Navigation. Additional notes from me:

- In the old days without smartphones/tablets we only had PCs and Laptops and the file hierarchy (tree structure) was always visible and easier to understand. The file hierarchy still exists on your smartphone, but it is hidden behind all the apps.

Exercises:

1. What is your *working directory*?
2. Are there any files present in this directory?
3. Change directory to `.ssh`.
4. Which file(s) is/are present in this directory?
5. Change directory to the *parent directory*.
6. What is your working directory now?
7. Change directory to `/home` using *absolute pathname*.

8. Change directory back to the `.ssh` directory.
9. Change directory to `/home` using *relative pathname*.
10. Change directory to the root of the file system.
11. Change directory to your home directory *in three different ways*.

2.3 Looking Around

Read the text and study/do the examples on Looking Around. Additional notes from me:

- Note the syntax and terminology
  `command -options arguments`
  (some options need arguments, some do not).
- Some frequently used options with ls are:
  ```
  ls -ltr  # sort files based on time, reverse order
  ls -ltrh # human readable output (file size in KB/MB/GB)
  ```

Exercises:

1. List all *hidden files* in your home directory.
2. What is the file name, modification time, size in bytes, group, owner and permissions of the file you found in the `.ssh` directory?
3. View the contents of the file `/etc/services` with `less`. Search for the characters `http`, repeat the search until it says "Pattern not found". Quit `less`.
4. (note: this exercise is only relevant if you have used ssh to access your Linux) What kind of file is the file in the `.ssh` directory? Do you understand how this file is related to a file you use when you connect to your Linux with ssh?

2.4 A Guided Tour

Read the text and study/do the examples on A Guided Tour. Additional notes from me:

- From now on, remember to always use *TAB completion* to fill out your commands and file names/paths.

Exercises:

1. Change directory to the `/boot` directory (remember to use TAB completion). What kind of file is `vmlinuz`? Ca how many megabytes (MB) is the file that actually is the Linux operating system kernel? (hint: include the option `h` to the `ls`-command, 'h' is short for "human-readable output").
2. Change directory to the root of the file system. Change to a subdirectory at least four levels below the root directory. Go back to the root directory and repeat two more times (with different directories each time) The goal here is to practice TAB completion. Hint: start with the `/usr/lib` directory.
3. In which directories to you find most of the programs you can execute?
4. In which directories to you find most of the configuration files? (these are files that contain settings that define behavior for the system and programs)

2.5 Manipulating Files

Read the text and study/do the examples on Manipulating Files. Additional notes from me:

- What is a file? It is data (the "contents") and metadata (filename, filesize, timestamps, permissions, etc.)

Exercises:

1. Copy the file `/etc/services` to your home directory. Rename it to `commonports`
2. Change directory to `/tmp`. Create a directory `mytmp` in `/tmp`. Move the file `commonports` from the previous exercise to the newly created `mytmp` directory. Verify that the file is there and no longer in your home directory.
3. Create a directory `backups`. Copy the directory `/etc/terminfo` including all files to the `backups`-directory by using the option `-a`. Using the option `-a` is an important way of copying called "archive" mode: this preserves all metadata (time stamps, permissions, owner, group) that the user performing the copy is allowed to preserve. Which metadata was not preserved in this copy operation?
4. Change directory to `/usr/bin`. List all files with file names that:
   (a) starts with a `s`
   (b) ends with either a `m` or a `n`
   (c) only have two characters

(d) contains a dot ('.')
(e) starts with a s followed by either c, e or f, and have one additional character of any kind (in other words the file name should have three characters in total).

2.6 Working with Commands

Read the text and study/do the examples on Working with Commands. Additional notes from me:

- We need to know this because sometimes a command exists as several variants under the same name, and we get confused if it behaves differently than expected when we use the wrong variant (this has happened to me with the time-command which I use quite frequently).

Exercises:

1. What are the four different kinds of commands?
2. What kind of command is pwd? python3? ll?
3. time is a command we can use to measure the execution time of a program. It exists as a *Shell keyword* which is different from a *Shell builtin*. A builtin is just a program compiled in as part of the shell (for better performance), while a keyword is a part of the shell's language (yes, a shell is also a programming language). time also exists as a separate program with more functionality than the keyword. Where in the file system is the time program located? What happens if you ask the shell what kind of command time is?
4. Browse the help page for the builtin echo. What is the meaning of the brackets? Try to output some text using echo with and without a newline.
5. View the man-page of the netcat-program. Search this man page for the word "examples" (remember that search in man-pages work just like search in the less-program where you can type 'n' for next match).

2.7 I/O Redirection

Read the text and study/do the examples on I/O Redirection. Additional notes from me:

- Important commands to master are cat, sort, uniq, head, tail, grep and wc.

Exercises:

1. Create a file containing just the number `1` with the command
   `echo 1 > test.txt`

   (a) Append the numbers `2` and `3` to the file so the output will be

   ```
   $ cat test.txt
   1
   2
   3
   ```

   (b) Create the file again with just the number `1`, but now append the numbers `2` and `3` *using two commands but without a line break, and then use a third command to add the line break* (to the end of the file) so the output will be

   ```
   $ cat test.txt
   1
   23
   ```

2. Download a file with names:
   `curl -O https://erikhje.folk.ntnu.no/navn.dat`

   (a) Use `wc` to count the number of lines and words in the file.

   (b) Sort the contents of the file and write the output to a new file `snavn.dat`.

   (c) Show each name in the file only once.

   (d) Count the number of lines containing `Emil`

   (e) Show a list of the most frequent occurring names like this (hint: `man uniq`):

   ```
   5 Emil
   3 Frida
   2 Per Arne
   2 Emma
   2 Ella
   1 William
   1 Sofie
   etc
   ```

3. Write a command to show the first five lines of the file `~/.bashrc`.

4. Write a command pipeline to write lines 21-24 of `~/.bashrc` to a new file `strange.tmp`.

2.8 Expansion

Read the text and study/do the examples on Expansion. Additional notes from me:

- From the web page: "Each time we type a command line and press the enter key, *bash performs several processes upon the text before it carries out our command*".

Exercises:

1. Compute 32^3 using arithmetic expansion.

2. Create the following directory structure using `mkdir -p` and brace expansion.

```
.
|--A
|   |--0
|   |--1
|
|--B
|   |--0
|   |--1
|
|--C
    |--0
    |--1
```

   Verify that it has been created with the `ls` command by using the option for recursive listing (”recursive” in this setting means to show all subdirectories). Delete the entire directory structure you created with a single command.

3. See all the defined *environment variables* with `printenv | less`. Use `echo` to output the value of `SSH_CONNECTION`.

4. Execute the command `echo I am $USER`.

   (a) Repeat this command but replace `$USER` with *command substitution* and the command `whoami`.

   (b) Repeat the command but add quotes so the output will be
   `I am       ubuntu`

   (c) Repeat the command `echo I am $USER` but add quotes so the output will be
   `I am $USER`

5. Write a command pipeline using command substitution that will print the number of directories in `/usr`. It should output:
   `There are 12 dirs in /usr`

6. Download a file some text:
   `curl -O https://erikhje.folk.ntnu.no/MC.txt`
   Write three command lines to output the number of occurrences of the words `wish`, `pizza` and `rub` respectively in `MC.txt` to a file `wordstat.dat`. After you have executed the three command lines you should be able to do this

```
$ cat wordstat.dat
wish: 5
pizza: 2
rub: 3
```

7. Use `echo` with the *backslash escape characters* for newline and tab to produce the following output:

```
My test of linebreaks (newline)
and
	TAB
		and more TABS
```

2.9 Permissions

Read the text and study/do the examples on Permissions. Additional notes from me:

- Permissions on Linux are actually quite easy to understand as opposed to the permissions on Windows...

Exercises:

1. Before starting this exercise do `whoami`to make sure you are the `ubuntu` user, NOT `root`. Write commands for the following
   (a) Create a file `a.txt`
   (b) Change permissions on `a.txt` to `r--r-----`
   (c) Try to delete `a.txt` (redo steps above if you are able to delete it)
   (d) Change permissions on `a.txt` to `rw-------`
   (e) Add a user `mysil` (`adduser mysil --disabled-login` but you do not have access to do this, so what do you need to add to this command?).
   (f) Change the owner of `a.txt` to be `mysil`
   (g) Try to delete `a.txt` (redo steps above if you are able to delete it)
   (h) Move `a.txt` to `/tmp/`
   (i) Try to delete `/tmp/a.txt`

3 Writing Shell Scripts

3.1 Writing Our First Script and Getting It to Work

Read the text and study/do the examples on Writing Our First Script and Getting It to Work. Additional notes from me:

- Important for you to understand the `PATH` environment variable since it can be misused to trick (attack) users into running *trojan horse programs/scripts* (programs/scripts that appear to do something else than what they are actually doing).

Exercises:

1. A comment in a shell script starts with a hash (`#`), but a script[1] always starts with the two characters `#!`. This is called *shebang* (probably from ”shell-bang”) or *hash-bang*. What is special about this two-character combination that is always present in the beginning of a script? (in other words: what does it do?)
2. Follow the webpage Writing Our First Script and Getting It to Work:
   (a) Choose a text editor you would like to learn. If you don't have a special interest in this, I recommend `nano` where you can see a ”menu” at the bottom where e.g. `^X Exit` means to hold down the CTRL-key while you press the x-key, this will allow you to save and exit the file.
   (b) Do all the commands on the webpage to create your first `hello_world`-script and place it as an executable script in your `PATH`.

3.2 Editing the Scripts We Already Have

This chapter is not mandatory to read, it is not part of our curricula, but read it if you are interested (in other words: this webpage not be part of the exam).

3.3 Here Scripts (aka ”Here document”)

Read the text and study/do the examples on Here Scripts. Additional notes from me:

- We are going to follow the text and create webpages using a shell script.

[1] This applies to scripts in other languages like Python, Perl, etc. as well.

Exercises:

1. Choose one of the following to ways[2] of creating webpages (the solutions you will get are based on option 1b with a webserver on your Linux-VM):

   (a) Use your existing home on the web which NTNU provides to you:

      i. ssh to `login.stud.ntnu.no` with your NTNU username and password.

      ii. Put your HTML-files in the `public_html` directory.

      iii. See the webpage at `https://folk.ntnu.no/USERNAME/FILENAME` (replace `USERNAME` and `FILENAME`)

   (b) Install a webserver on your Linux-VM and put your webpages there:

```
# Are there any programs running listening on ports?
ss -tlpn # will probably show port 22 for ssh and 53 for local DNS

# Install a web server that will start listening on port 80:
sudo apt install nginx # choose OK if asked any questions
                       # (use TAB and ENTER keys)

# Is there now a program listening on port 80?
ss -tlpn # the "Local Address:Port" column should list port 80

# From which directory does nginx serve web pages?
grep -r root /etc/nginx/ | grep www
# This is where you can put your webpages to see them
# at your 10.212. ip-address
```

2. What is the advantage of using a *here script* (aka *here document*) instead of just the `echo`-command with double quotes (or single quotes)?

3. Create the web page using the last example script you see on the bottom of Here Scripts. You should:

   (a) Copy and paste the script into your own file.

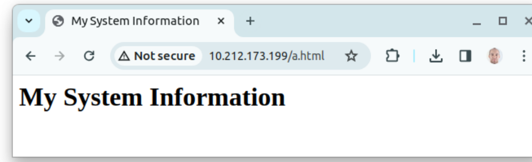
   (b) Change permissions on the file, so you can execute the file as a script.

   (c) Execute the script and redirect the output to an HTML-file.

   (d) Copy the HTML-file to directory where the web pages are (note: if you are on the Linux-VM you need to prefix the command with `sudo` since an ordinary user is not allowed to write to nginx root directory).

   (e) View the file in you web browser:

[2] CISK-students must choose the second option: installing a webserver on your own Linux-VM.



4. What is the special feature in this script (the one on the bottom of Here Scripts) that was not a part of the previous script that also used a here script?

3.4 Variables

Read the text and study/do the examples on Variables. Additional notes from me:

- Remember that when you assign values to variables, you cannot have spaces around `=` since the shell will interpret this as a command with options instead of an assignment.

Exercises:

1. Modify your script to include a variable and an environment variable[3] (in the same way as in Variables). Perform the same operations as the previous exercise to update the webpage so it will look something like this:

3.5 Command Substitution and Constants

Read the text and study/do the examples on Command Substitution and Constants. Additional notes from me:

- Command substitution is so powerful! It lets you use the output from other command lines or scripts into your current command line (or script).

[3]Actually `HOSTNAME` is not an environment variable in the strict sense, it is created automatically by Bash, but you will not see it when you execute `printenv`.

Exercises:

1. Modify your script to include command substitution according to Command Substitution and Constants. Perform the same operations as the previous exercise to update the webpage so it will look something like this:

3.6 Flow Control - Part 1

Read the text and study/do the examples on Flow Control - Part 1. Additional notes from me:

- "The if command is fairly simple on the surface; it makes a decision based on the *exit status* of a command."

Exercises:

1. Do `help -m test | less` and search this text for `Arithmetic` so you understand how you can compare numbers in if-tests.
2. Write an if-test that checks if -1 is less than 0. If true it should output "Yes, it's true!".
3. What is the difference between the two comparison operators `=` and `-eq`?
4. Store the file name `mysil` in a variable (the file does not have to exist). Write an if-else test on the command line which checks if the file exists. If the file exists, the command should output
   `mysil er her :)`
   If the file does not exist, the command should output
   `mysil er ikke her :(`
5. Write a script that has its first command line:
   `read -p "Enter a string: " mysil`
   (this will read a string from the user into the variable `mysil`)
   The script should then check if the user entered an empty string or not.

3.7 Positional Parameters

From Positional Parameters you only need to read this part about *positional parameters* (providing command line arguments to your script):

All of these features involve using command line options and arguments. To handle options on the command line, we use a facility in the shell called *positional parameters*. Positional parameters are a series of special variables (`$0` through `$9`) that contain the contents of the command line.

Let's imagine the following command line:

```
[me@linuxbox me]$ some_program word1 word2 word3
```

If `some_program` were a bash shell script, we could read each item on the command line because the positional parameters contain the following:

- $0 would contain "some_program"
- $1 would contain "word1"
- $2 would contain "word2"
- $3 would contain "word3"

Here is a script we can use to try this out:

```
#!/bin/bash

echo "Positional Parameters"
echo '$0 = ' $0
echo '$1 = ' $1
echo '$2 = ' $2
echo '$3 = ' $3
```

Detecting Command Line Arguments

Often, we will want to check to see if we have comand line arguments on which to act. There are a couple of ways to do this. First, we could simply check to see if `$1` contains anything like so:

```
#!/bin/bash

if [ "$1" != "" ]; then
    echo "Positional parameter 1 contains something"
else
    echo "Positional parameter 1 is empty"
fi
```

Second, the shell maintains a variable called `$#` that contains the number of items on the command line in addition to the name of the command (`$0`).

```
#!/bin/bash

if [ $# -gt 0 ]; then
    echo "Your command line contains $# arguments"
else
    echo "Your command line contains no arguments"
fi
```

3.8 Flow Control - Part 3

Read the text and study/do the examples on Flow Control - Part 3. Additional notes from me:

- Always remember that you can write a for-loop directly on the command line, you do not have to put it inside a shell script.

Exercises:

1. Do `echo {01..05}` to see that it generates a list of numbers. Write a for-loop to output these numbers one on each line.
2. Do `echo {a..c}` to see that it generates a list of characters. Write a nested for-loop to output the following:

```
01.a
01.b
01.c
02.a
02.b
02.c
03.a
03.b
03.c
```

3. Write a shell scripts that checks if there are exactly three command line arguments (it should exit with error if not three), and processes these three in a for loop to echo them like this:

```
$ ./argcheck.sh en to
$ ./argcheck.sh en to tre
en is an argument
to is an argument
tre is an argument
```

4. Write a shell script that iterates over all command line arguments and checks if they are files:

```
$ ./filecheck.sh a mysil fil.txt
a is a file
fil.txt is a file
```

5. Write a shell script that takes a list of directories as command line arguments, counts the number of files and directories in each of these directories and outputs them in sorted order (the highest number on top):

```
$ ./numfiles.sh /tmp /usr/share/ /usr/ /var
109 files and dirs in /usr/share/
14 files and dirs in /var
12 files and dirs in /usr/
7 files and dirs in /tmp
```

6. We do not include Shell functions as mandatory readings, but notice the use of the commands `du`, `find` and `uname` in the functions at bottom of Flow Control - Part 3.

Table 1: Relevant aliases.

Linux	**PowerShell**
`pwd`	`Get-Location`
`ls`	`Get-ChildItem`
`cd`	`Set-Location`
`cp`	`Copy-Item`
`mv`	`Move-Item`
`rm`	`Remove-Item`
`cat`	`Get-Content`
`sort`	`Sort-Object`
`tee`	`Tee-Object`
`echo`	`Write-Output`
`ps`	`Get-Process`
`kill`	`Stop-Process`
`(wc)`	`Measure-Object`
`(head/tail)`	`Select-Object`

4 PowerShell

See separate document on PowerShell. Read quickly through the first sections, READ CAREFULLY FROM Objects and Get-Member up to if, comparison, tests, ”branching”.

Based on what we have covered in the first three chapters about Linux, table 1 lists the relevant aliases for some of the commands you probably already know.

Exercises:

1. Find out which PowerShell cmdlet are mapped to the aliases `cd`, `echo`, `cat`, `cp`, `rm` and `sort`. Try all these six cmdlets.
2. Store the output of
   `(Resolve-DnsName ftp.uninett.no).IP4Address`
   in a variable. Pipe the variable to `Get-Member` to see which properties and methods the variable (which is an object like ”everything” is in PowerShell) has.
   (a) Print the Length of the variable.
   (b) Use the `split()`-method to split the four octets of the IP address so they are printed each on a line by themselves.
3. Write a command line which recursively outputs all directories in your home directory. It should not list files, only directories (hint: `Get-Help -Online Get-ChildItem`).

4. Use `Get-Member` to list all properties and methods in the the objects you get from `Get-ChildItem`. Pipe to `more` to page-by-page (by hitting space) or line-by-line (by hitting enter). Repeat with `Get-Process` instead of `Get-ChildItem`.

5. Piping to `Select-Object -Property *` is many times useful to see all the properties an object has (remember: ordinary output on screen only shows a subset of all the properties).

   (a) List all the user accounts on your computer with `Get-LocalUser`, choose a user and see all the properties for that account: when was the user last logged on?

   (b) List all the files in your home directory with `Get-ChildItem $env:HOMEPATH`, choose a file and see all the properties for that file: when was the file last accessed?

6. Write a PowerShell command line that will list all files in the directory `C:\Windows` that are larger than 10KB. Then add to the pipeline:

   (a) Make the output sorted based on file size (Length)

   (b) Make the output show only the three largest files

   (c) Make the output show only file name, file size, last access time and last write time

7. Write a PowerShell command line that will show how much space is used by all the files in all the directories (including all subdirectories) in `$env:HOMEPATH`. The output should be in MB (MegaBytes).

8. Write a command line which outputs all processes which have the property `StartTime` within the last hour.

9. Write a command line which prints the name of all directories (from your current directory) which contain at least 10 files/subdirectories.

10. Create a variable `$golf` with the value `Viktor Hovland`. Use `Write-Output` to print "My golf hero Viktor Hovland" using this variable. Advanced: Now try to use the `Split()` method of the variable to print only "My golf hero Hovland" (hint: you can use the `Split()`-method just like this without any arguments).

11. (OPTIONAL EXERCISE) Using splatting, write a command line which recursively outputs all files (not directories, only files) in `C:\Windows\System32\LogFiles` and also have the parameter ErrorAction set to SilentlyContinue.

5 C programming

A nice video to watch accompanying this text is Writing a simple Program in C (but note that this uses the vi/vim editor which I recommend everyone really interested in Linux to learn, but those of you who want an easier way forward should use the nano editor instead).

5.1 Compiling and Executing

A very simple C-program looks like this:

```
#include<stdio.h>

int main(void) {
    printf("Yo Mysil\n");
    return 0;
}
```

Let's explain it line by line:

`#include<stdio.h>`
This tells the compiler that the following code will use functions (in our case: the printf-function) that are defined in the library file `stdio.h`

`int main(void) {`
The program starts executing at the start of the function called `main`. `int` means that the program should provide a *return/exit code/value/status* (sometimes we say "exit status" sometimes we say "return value", etc.) and that it should be an integer. `void` is a way of saying that this function does not take any arguments (or "parameters" if you will). Finally, the opening brace indicated that "now comes the actual code".

`printf("Yo Mysil\n");`
`printf` is an extremely useful and widely used function across many programming languages (it also exists on the command line in Linux). The argument to printf is in this case the text `Yo Mysil` which will be printed on the screen with a following newline indicated by `\n`. The program statement is ended with a semicolon `;`.

`return 0;`
This is the exit status mentioned above. Programs should return an exit status of `0` if they complete successfully. If a program does not exit successfully, e.g. there is missing input, it will typically have an if-statement checking for a condition and returning `1` instead of `0` (or some other values dependent on the types of error).

```
}
```

The closing brace indicated the end of the main function, and by so the end of the program.

If we create this program called `mysil.c` with a text editor like `nano` or by using a here document, we can verify its content with `cat`:

```
$ cat mysil.c
#include<stdio.h>

int main(void) {
    printf("Yo Mysil\n");
    return 0;
}
```

To compile and execute the program we do:

```
$ gcc -Wall mysil.c -o mysil
$ ./mysil
Yo Mysil
```

Let's explain it line by line:

```
$ gcc -Wall mysil.c -o mysil
```

`gcc` is our compiler (GNU project C and C++ compiler). When we have such a nice compiler we ask it to help us write good code by using the command line option `-Wall` which is short for "enable all warnings" or "Warnings all". `mysil.c` is the source code we want to compile. Finally, we ask the compiler to create the executable file `mysil` from the compilation process. If we do not specify this `-o mysil`, the compiler will create an executable file called `a.out` instead. Since we like to identify what our program is by its name, we typically specify it with the `-o` option since the name `a.out` does not tell us anything. Notice the logic in the command line options:

- `gcc -Wall -o mysil mysil.c` is also correct.
- `gcc -o mysil mysil.c -Wall` is also correct.
- `gcc -o -Wall mysil mysil.c` is NOT correct.
- `gcc -Wall mysil -o mysil.c` is NOT correct.

```
$ ./mysil
```

We execute the program by prefixing it with `./` (unless the program is in a directory specified by the `PATH` environment variable). We do not have to change permissions on the program like we had to do with shell scripts. The compiler have taken care of this for us.

5.2 The printf-function

Let's have a look at another C-program called `printfdemo.c`:

```
$ cat printfdemo.c
#include<stdio.h>
int main(void) {
    int x = 42;
    char str[] = "Mysil";
    printf("Tall er %d og tekst er %s\n",x,str);
    return 0;
}
```

Let's explain the new lines line by line:

`int x = 42;`
This creates a variable named `x` with a value `42`, and the variable is of data type integer.

`char str[] = "Mysil";`
C does not have its own data type for a string, so we create a string as an array of the data type character. In this case the variable is named `str` with a "value" `Mysil` (we say "value" since it is actually an array of characters and not a single value).

`printf("Tall er %d og tekst er %s\n",x,str);`
The printf-family of functions are used to format and output text. We will only use the `printf` function which outputs to the screen, but you should note that there is a `fprintf` which we can use to output to file, and a `snprintf` which outputs to a string variable (a char array), and several other similar variants of printf.
`"Tall er %d og tekst er %s\n"`
is a *format string*. A format string contains *conversion specifiers*, in our case the conversion specifiers are `%d` which says "replace me with an integer (a digit)", and `%s` which says "replace me with a string". The conversion specifiers are replaced with values from arguments following the format string, and by default they will replace them in the order they are listed:

- `%d` will be replaced with the value of `x` (42)
- `%s` will be replaced with the value of `str` (Mysil)

Let's compile and execute the program to verify:

```
$ gcc -Wall printfdemo.c -o printfdemo
$ ./printfdemo
Tall er 42 og tekst er Mysil
```

Let's take a look at some interesting details of format strings using a simplified version of `printfdemo.c` that only uses integers:

Reading from memory What if we make a mistake and have more conversion specifiers than we have arguments:

```
#include<stdio.h>
int main(void) {
    int x = 3, y = 5;
    printf("Tall er %d og %d og %d\n",x,y);
    return 0;
}
```

We compile and execute:

```
$ gcc -Wall printfdemo.c -o printfdemo
<we get some warnings, but it does compile>
$ ./printfdemo
Tall er 3 og 5 og 288173504
```

WHAT? Where did 288173504 come from? Printf keeps reading from our computers memory if we have not specified enough arguments! In other words: we have a *memory leak* and memory leaks can be quite bad, see Heartbleed for one of the most famous memory leak vulnerabilities.

Writing to memory Printf has a special conversion specifier `%n`:

```
#include<stdio.h>
int main(void) {
    int x = 3, y = 5, z;
    printf("Tall er %d og %d.%n\n",x,y,&z);
    printf("z er %d\n",z);
    return 0;
}
```

We compile and execute:

```
$ gcc -Wall printfdemo.c -o printfdemo
$ ./printfdemo
Tall er 3 og 5.
z er 15
```

WHAT? Where did 15 come from? The conversion specifier `%n` *writes* the number of characters written so far into the corresponding variable[4] (`z`) in the argument list. 15 comes from the fact that the text string "Tall er 3 og 5." has a total of 15 characters. *We can write to a computer's memory using printf!*

Controlled writing to memory The conversion specifiers in printf also has the option of specifying which argument in the argument list they process by adding `m$` after the percentage-character where `m` is the argument number, e.g.
`printf("Tall er %2$d og %1$d\n",x,y);`
will switch the order of `x` and `y` in the output because `2$` in the first conversion specifier tells it to insert the second argument `y` instead of the first argument `x` (which is the default behaviour). This means that if we use this option (`m$`) together with the conversion specifier `%n` we can control which variable we write to:

```
#include<stdio.h>
int main(void) {
    int x = 3, y = 5;
    printf("Tall er %d og %d.%2$n\n",x,&y);
    printf("y er %d\n",y);
    return 0;
}
```

We compile and execute:

```
$ gcc -Wall printfdemo.c -o printfdemo
<we get some warnings, but it does compile>
$ ./printfdemo
Tall er 3 og 1359008880.
y er 24
```

WHAT? Yes, we wrote to the variable `y` the value 24 (the number 1359008880 was output for `y` in the first printf-statement because of the `&`-operator which gives the address of the variable). So a bit simplified we can say that `%n` allows us to write to memory and `m$` allows to control where in memory we write.

Finally: The real format string vulnerability! We have now seen that printf has some interesting features, and yes we call them features. But features turn into *software vulnerabilities* when users get a way to exploit them. As explained in the original

[4] In this case we prefix the variable `z` with an ampersand (&) because we need to give the memory address of the variable instead of just its name. C has a lot of details like this, but we don't worry about those now.

posting by Tim Newsham about Format String Attacks on the Bugtraq mailing list in September 2000, the most obvious error to make is if the programmer is going to output a string `str` that the user controls and decides to do
`printf(str);` UNSAFE!
instead of
`printf("%s", str);` SAFE!
If the programmer allows the user to control the arguments to printf, they can abuse this. E.g. in the following code it should not be possible to have the program output ”Welcome!”:

```
#include <stdio.h>
int login_OK = 0;
int main(void) {
    char password[256];

    printf("Enter password: ");
    fgets(password, sizeof(password), stdin);
    printf(password); /* VULNERABILITY */

    if ( login_OK == 1 ) {
        printf("Welcome!");
    }
    return 0;
}
```

Here a ”hacker” can try different input and possible construct an input with the features we have described above to overwrite the `login_OK` variable and thereby abuse the program.

Note that this is not some amazing discovery, we just need to read the man-pages. `man 3 printf` tells us:

> Code such as printf(foo); often indicates a bug, since foo may contain a % character. If foo comes from untrusted user input, it may contain %n, causing the printf() call to write to memory and creating a security hole.

If you would like to know all the quite advanced details involved in format string attacks, I recommend watching the excellent videos from Fabian Faessler (aka LiveOverflow) (in addition to reading the Bugtraq-post mentioned above and the printf-manpage of course).

5.3 Command line arguments

To provide command line arguments to a C-program, we change
`int main(void)` to
`int main(int argc, char *argv[]))`

- `argc` contains the number of arguments provided, but it includes the program name, so it has a value of at least one. If you provide one command line argument, `argc` will have a value of two. If you provide two command line arguments, `argc` will have a value of three, and so on.
- `argv` is an array containing the actual arguments provided. Since C does not have its own string data type, `argv` is an array of "addresses" to char arrays containing the arguments, but don't worry about if you don't understand what that means, what we need to know is that how we can access them:
  - Output the first argument:
    `printf("%s\n",argv[1]);`
  - Copy the first argument to another string:
    `char *first = strdup(argv[1]);`

A small demo program that outputs two command line arguments:

```
#include<stdio.h>
#include<string.h>
int main(int argc, char *argv[]) {
    char *first  = strdup(argv[1]);
    char *second = strdup(argv[2]);
    printf("%d args, %s og %s.\n",argc-1,first,second);
    return 0;
}
```

We compile and execute:

```
$ gcc -Wall argtest.c -o argtest
$ ./argtest solan reodor
2 args, solan og reodor.
```

Notice that we don't worry about robustness in our little introduction to C, so if we don't provide exactly two command line arguments to this program it will access variables that does not exist and this typical leads to a memory access violation called "segmentation fault":

```
$ ./argtest solan
Segmentation fault (core dumped)
```

5.4 Exercises

1. How is the main-function in C different from other functions?
2. Write the command line for compiling a C-program called `solan.c` into an executable file `solan`. Include the option for enabling all warnings in the compilation process.
3. Assume `int x=2, y=3;`, what will be the output of the following printf statements?
   (a) `printf("%d",x);`
   (b) `printf("%d\n%d\n",x,y);`
   (c) `printf("%2$d og %1$d\n",x,y);`
   (d) `printf("%2$d og %2$d\n",x,y);`
4. Explain as detailed as you can the problem with this C-program:

```
#include<stdio.h>
int main(int argc, char *argv[]) {
    printf(argv[1]);
    return 0;
}
```

5. Download the file `mysil` and make it executable:
   `curl -O https://erikhje.folk.ntnu.no/mysil; chmod 755 mysil`
   Find out if it has a format string vulnerability.
6. The syntax for an if-statement in C is
   `if ( <condition> ) { <code> } else { <code> }`
   Write a C program `solan.c` that returns 1 if the program has less than two command line arguments, and returns 0 otherwise.
7. Write a Bash command line `if` test that says "Yes!" if the exit code of solan (the executable version of the program you wrote above) is zero.