# Exercise sheet 8 2022-12-22

**Due date: 2023-01-12 16:59**

The goal of this exercise sheet is to familiarize with ownership models, and specifically enforcing the implicit ownership models inherited by common C APIs.

At the end of the exercise you will have implemented a simple working TCP server and client, which can send messages to each other.

**NOTE:** Sadly, the C APIs used in this exercise aren't really portable between Linux and Windows. As all exercises are only tested on Linux, the Linux API counts. We highly recommend you using at least WSL for this exercise! They should work fine on MacOS.

Further, be careful when searching for code on the internet. Many examples use C code, and what is good and valid C, might not be good and valid C++. Make sure you use only what you understand, and not blindly copy and paste random examples. This is part of the challenge, to step by step abstract away the C code. And yes, you will need `reinterpret_cast` at some point (@_@).

## Exercise 1:

First, let's get familiar with netcat (please search for install instructions for your specific system). From here, I'll assume the GNU Netcat implementation. ncat should be a good option for all systems. They should all work simiarly, but one might need a couple of different flags, which you need to research yourself. This is a useful skill in itself!

Start two terminals, in one enter `nc -l -p 1337`', which tells netcat to listen on port 1337. In the other, enter '`nc localhost 1337`', which will connect to localhost on port 1337. Now you can type anything (e.g. *rolf*) in the second terminal, once you press enter it should be sent to the first terminal. Cool right? This also works with any remote machine you can access via an IP address. Exit with *Ctrl + C*.

Even better, you can send files with this method. Try it: run `cat somefile | nc -lp 1337` on the server and `nc host 1337 > file` on the client. Not enough? What about creating a compressed archive of a directory, sending that and unpacking? Easy: Just compressed the directory on the server using tar, and listen on the given port: `tar c directory/ | pv | nc -l -p 1337`. And on the client, just connect, and untar it: `nc host 1337 | pv | tar xv`. Here, `pv` is a pipeviwer, which will indicate progress. If you don't have it, leave it out.

**NOTE:** So much automated testing. So, please really do this task (https://i.imgur.com/mSHi8.jpg) ;-).

## Exercise 2:

You will be using many (potentially unknown to you) C functions in the next sub tasks. So, of course, you can search that on the web, but that is often unnecessary and slower than the Linux way.

Hop to the terminal again and type `man man`. `man` is an interface to the system reference manuals. Dennis Ritchie and Ken Thompson have written the first manpages, so they gotta be good! Read through the manual a bit and get familiar with it!

By default, you can move up and down with vim-like bindings, e.g. *j* and *k*, *Ctrl + D* and *Ctrl + U* for down and up respectievly, `/` to search. Now, exit with `q`.

Now, let us look at the man page of a function. Type `man strlen`. You will be presented with the `strlen` documentation of libc.

Sometimes, different functions, or search terms are present in multiple sections. E.g. compare the output of `man 2 fork` and `man 3 fork`. To refer to a specific one, you will see the notation *fork(2)* to refer to the first, and *fork(3)* to the second. You will see this notation used throughout the exercise to help you navigate the different functions needed. For most of the functions we are dealing in this exercise, sections 2 contains the libc documentation, and section 3, the POSIX specification (i.e. POSIX is the standard, and the libc documentation the implementation of the standard).

## Exercise 3:

Finally, let us get to implementing something. In Linux pretty much all in- or output like interfaces are represented using file descriptors (handles). For example, standard in, out and error are the filehandles 0, 1 and 2 respectively.

All other file descriptors need to be created by many functions e.g. `open`, `creat` or `socket` (check their main pages). They need to be closed using `close` (*close(2)* for the Linux implementation and *close(3)* for the POSIX manpage).

This is a good fit for a unique ownership model. I.e. the object should take hold a unique ownership of the file descriptors, once the file descriptor owns the descriptor it will be closed, when the object goes out of scope. No other object is allowed to close/destroy the underlying handle. Also, we do not want to share the ownership.

Implement this behaviour in `filedescriptor.cpp`. Specifically, implement the copy and move constructors and assignments. Think, how you can achieve the above ownership model with them.

## Exercise 4:

In this subtask, we implement a wrapper around the Linux socket. Sockets are endpoints for communication, represented by a file descriptor. In some sense they behave just like files, but Linux will create network packages for the data written to the socket, instead of writing to a file.

In this basic setting we are, sockets can either be client or server sockets. Server sockets bind to a port and start listening on it. Only once they listen on a port, they can accept connections from the port. Accepting connections is a blocking operations. In the sense, that no other function will be run, until a connection is made. Once a client connects to a server, a new socket is created. This is the socket you can read and write to communicate with the client from the server. On the client side, the socket simply connects to a destination address and port.

Once a connection is open, the client and server can send and receive messages. The order is important, in this simple example, they need to agree who writes and reads, it's not possible for both to send. Once the connection is closed, it is expected to close the socket it is connected with.

Implement the `Socket` and `Connection` in `socket.cpp` and `connection.cpp`.

## Exercise 5:

Finally, all that is needed is a nice wrapper for our client-server setup. The server takes care to listen on a port, and provides a facility to accept connections. The client can connect to a destination address and port.

Both the client and the sever need a single private member, which one?

Further, the server needs to

- Be construtible from a port.
- `accept` new connections.

The client needs to

- Be default construtible
- `connect` either to a given destination address (given as a `std::string`), and a port
- or `connect` to localhost and a given port.

The later forms an overload set, the real (vegan) meat, is in the first function.

**NOTE:** The type of ports should be `uint16_t` to avoid unnecessary warnings. Further, think what functions should be const functions. Lastly, the names should be clear from the highlighting from this document.

Finish both the interface and implementation of the `Server` and `client` classes, in `server.h`, `server.cpp`, `client.h` and `client.cpp`.

You can test your application using the `run.cpp` file. As with netcat, you can open two terminals in your build folder (assuming the executable is in the current folder), in one launch the executable with `./runhw08 server 1337`, on the other `./runhw08 client 1337`. Now write a message on the client side hit enter and magically (not really) the message will be send to the server.

Have fun!