
# System Programming
# C programming manual: lab 6
# 2018 - 2019

## *Bachelor Electronics/ICT*

*Course coördinator:* *Luc Vandeurzen*
*Lab coaches:* *Stef Desmet*
*Tim Stas*
*Jeroen Van Aken*
*Luc Vandeurzen*

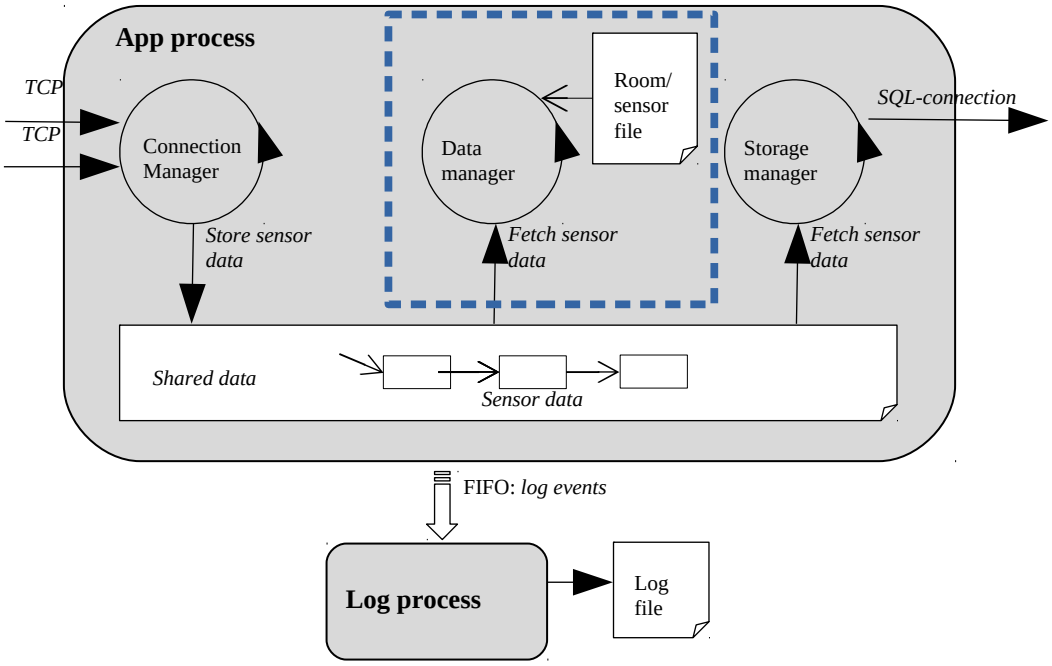
*Last update: October 23, 2018*

## *C programming*

*Lab targets: learn to create and use static and dynamic libraries, ldd, nm, and objdump; be able to implement code related to file reading and writing.*

*For your information*

*The picture below visually sketches the final assignment of this course. The relationship of this lab to the final assignment is indicated by the dashed blue line.*




**Exercise 1:** *create and use a static library*

Create a static library containing the list implementation. Copy the library to a local directory in your home folder, e.g. /home/lucvd/mylibs. Implement a main.c function that uses a list. Build main.c and the list library to an executable (assuming that the main.c file is not in the same directory of the static library). Run and test the program. Use ‘nm’ and 'objdump' to find out the addresses of the symbols in your code and check if the library code is really included in the executable.

**Exercise 2:** *create and use a dynamic library*

This is a similar exercise as the previous one, but this time we build a shared library containing the list implementation. Again, copy the library to the local directory in your home folder, e.g. /home/lucvd/mylibs. Use the main.c function from the previous exercise

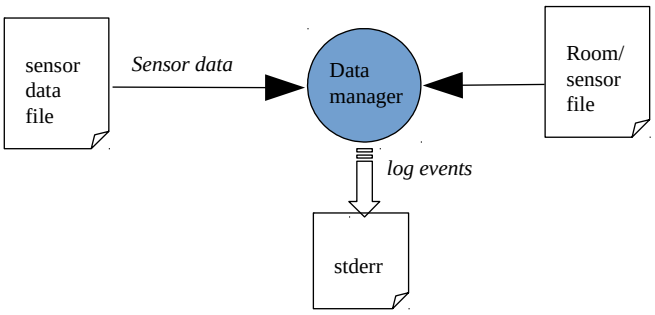
to build an executable using the list library (again assuming that the main.c file is not in the same directory of the dynamic library). Use 'ldd' to find out if the loader has a reference to the shared library. If that's ok, run and test the program. Again, Use ‘nm’ and 'objdump' to find out the addresses of the symbols in your code and check that the library code is NOT really included in the executable.

**Exercise 3:** *working with text and binary files, using a pointer list to organize data*

*PLEASE UPLOAD YOUR SOLUTION OF THIS EXERCISE AS A **ZIP FILE** ON https://labtools.groept.be .*

YOU ARE STRONGLY ENCOURAGED TO TEST AND DEBUG YOUR SOLUTION UNTIL THE CRITERIA FOR THIS EXERCISE AS DESCRIBED ON https://labtools.groept.be/ ARE SATISFIED!

Assume that a sensor network is used to monitor the temperature of all rooms in an office building. Write a program, called the ‘data manager’, that collects sensor data and implements the sensor system intelligence. For example, the data manager could apply decision logic to control a HVAC installation. Obviously, controlling the HVAC installation of a real building is not an option (*and we are happy for that!*).

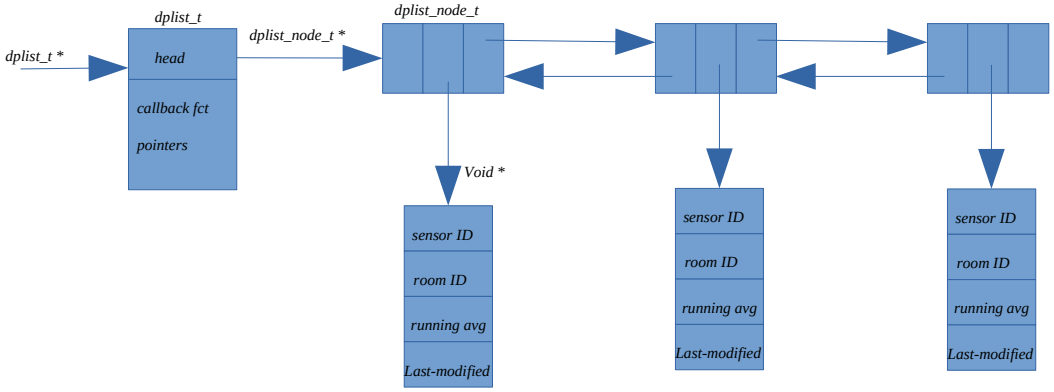



Before handling sensor data, the data manager should first read a file called 'room_sensor.map' containing all room - sensor node mappings. The file is a text file (i.e. you can open and modify the file in a standard editor) with every line having the format:

*<room ID><space><sensor ID><\n>*

A room ID and sensor ID are both positive 16-bit integers (uint16_t).
The data manager organizes all sensor nodes in a **pointer list** data structure. Use the library implementation of a pointer list implemented in the previous exercises to do this (try the static and dynamic version). An element in this list maintains **at least** information on (i) sensor node ID, (ii) room ID, (iii) data to compute a running average, and (iv) a last-modified timestamp that contains the timestamp of the last received sensor data used

to update the running average of this sensor. The picture below visualizes this data structure.




The data manager starts collecting sensor data and computes for every sensor node a running average. We define a running average in this context as the average of the last RUN_AVG_LENGTH sensor values. If this running average exceeds a minimum or maximum temperature value, a log-event (message send to stderr) should be generated indicating in which room it's too cold or too hot. RUN_AVG_LENGTH should be set at compile-time with the preprocessor directives RUN_AVG_LENGTH=<some_value>. If this isn't done, the default value of 5 should be used. It should also be possible to define the minimum and maximum temperature values at compile-time with the preprocessor directives SET_MIN_TEMP=<some_value> and SET_MAX_TEMP=<some_value> where <some_value> is expressed in degrees Celsius. Compilation should fail and an appropriate error message should be printed when these preprocessor directives are not set at compile-time.
Sensor data is defined as a struct with the following fields (see also the given 'config.h' file):

- sensor_id: a positive 16-bit integer;
- temperature: a double;
- timestamp: a time_t value;

The data manager reads sensor data from a binary file called 'sensor_data' with the format:

*<sensor ID><temperature><timestamp><sensor ID><temperature><timestamp> ...*

Notice this is a binary file, not readable in a text editor, and spaces and newlines (\n) have no meaning.
A program 'file_creator' is given to generate the 'room_sensor.map' and 'sensor_data' files to test your program. If you compile this program with the '-DDEBUG' flag on, the

sensor data is also printed to a text file. You may assume that the sensor data on file is sorted on the timestamps (earliest sensor data first). Sensor data of a sensor ID that did not occur in the room_sensor.map file is ignored, but an appropriate message is logged.
Finally, organize your code wisely in a main.c, a datamgr.c and a datamgr.h file. This will help you to easily re-use the data manager code for the final assignment!