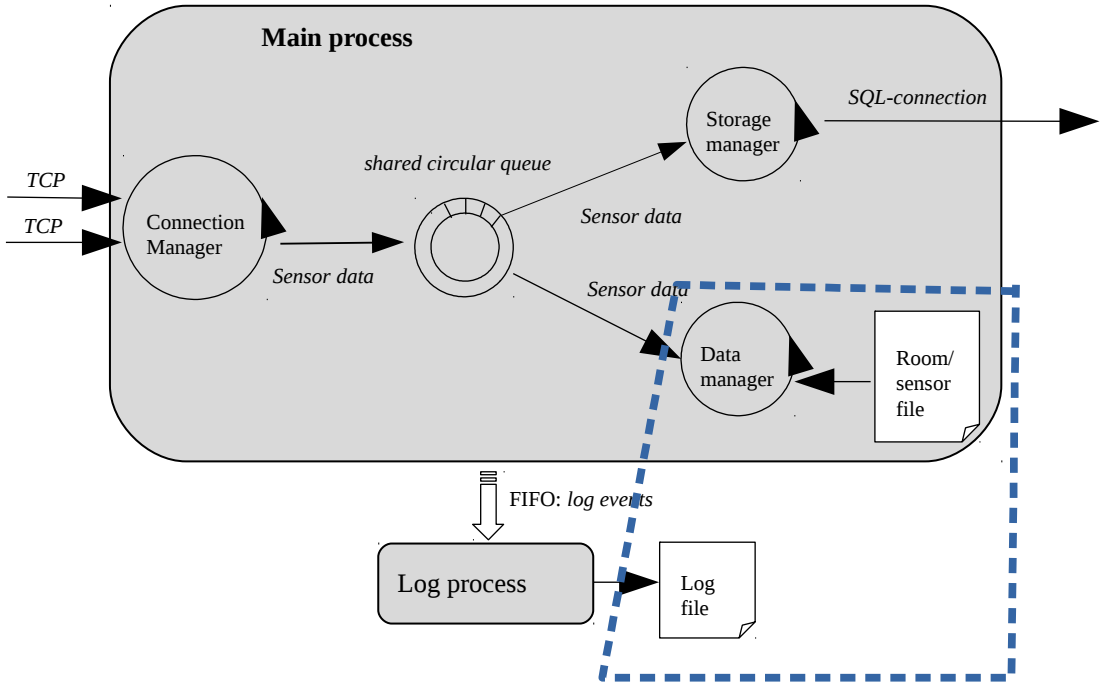
Lab 6 – Libraries, pointer lists and files

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.



THE SOLUTION OF THIS EXERCISE NEEDS TO BE UPLOADED ON TOLEDO BEFORE THE NEXT LAB.

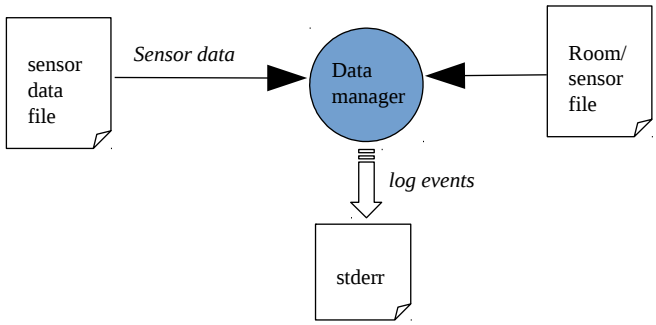
Your solution is only accepted if the following criteria are satisfied:

1. *Compilation with '-Wall -Werror' option generates no warnings or errors*
2. *Running 'cppcheck –enable=all --suppress=missingIncludeSystem' on the source code results in no warnings or errors*
3. *Run your application with 'valgrind --tool=memcheck –leak-check=yes' to check no memory leaks are detected by this tool.*
4. *The application should run on the given 'sensor_data' file without run-time errors.*
5. *The implementation should consists of at least a main.c, a datamgr.c, a datamgr.h, a liblist.so library and the list.c and .h files used to create the liblist.so library. Upload these files on Toledo!*

DO NOT upload code that doesn't satisfy these criteria.

Basic version

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!*).

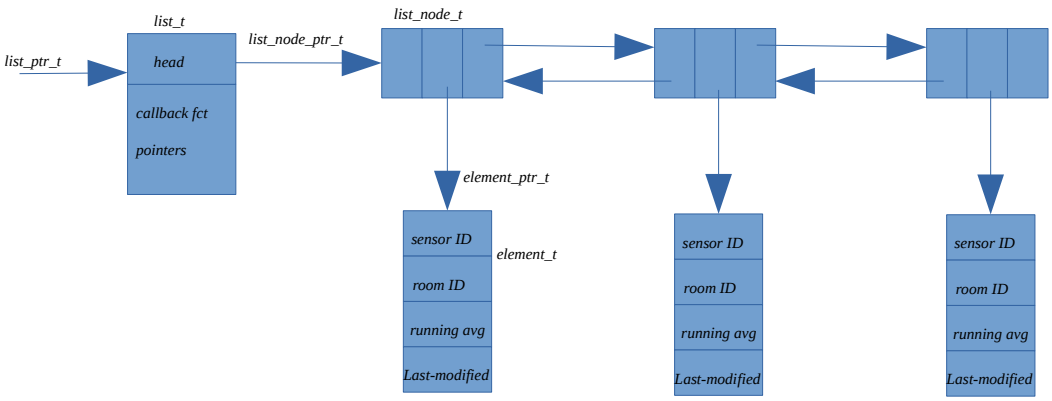


Before handling sensor data, the data manager should first read a file called 'room_sensor.map' containing all room - sensor node mappings. The file is a text file (i.e. you can open and modify the file in a standard editor) with every line having the format:

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

A room ID and sensor ID are both positive 16-bit integers.

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 lab to do this. 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.



Next, 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 5 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. It should 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:

- 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 sample file is given to test your program. 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!

Extended version

Several extensions and improvements can be made to the basic version. We just mention a few interesting ones:

1. Keep the pointer list sorted on sensor IDs such that searching for a sensor ID can be optimized.
2. Delete sensor elements from the list when there is no activity for a

certain TIMEOUT time. Use the 'last-modified' element field to make this decision.

3. Implement a small program that generates sensor data files. The sensor node simulator from lab 1 can be a good starting point.