Source of Life

In this assignment, we consider a simulation model of a system for managing and monitoring dams that provide water for the municipal water supply.

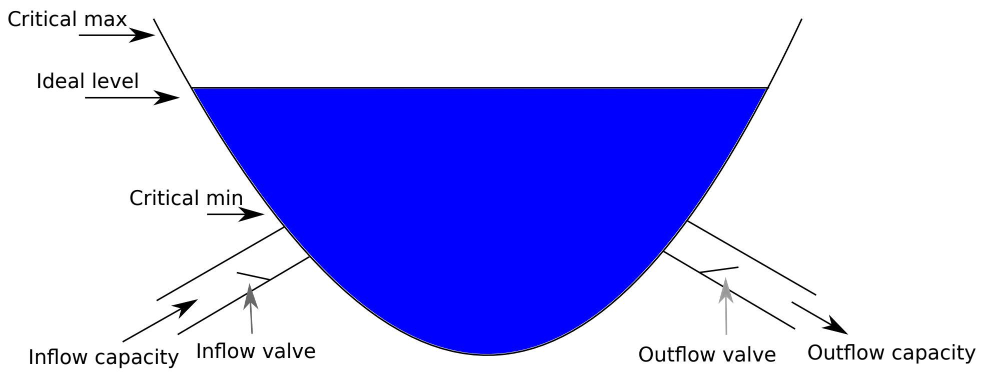

Figure 1: Schematic of a dam with the model parameters included in our simulation model

The purpose of the system is to monitor the water levels of the dams and adjust inflow and outflow valves as needed to maintain the water at an ideal level. As a simplifying assumption, inflow and outflow pipes have unlimited capacity and can therefore supply or consume as much water as needed. They do, however, have limited flow capacity. This means that only a limited amount of water can flow in or out from the dam during a timestep and it therefore takes time to restore the water level to normal following detection of an excess or lack of water. Figure 1 shows a schematic of a dam with the aforementioned model parameters.

Instead of measurements from actual sensors, the model of the system presented here examines a simulation scenario that progresses as a series of discrete timesteps. In each timestep, the simulation may receive sensor readings indicating that the water level of a dam has changed and, correspondingly, it may perform valve adjustments to bring the water level of the system closer to the ideal level[1]

Model parameters

This section describes the parameters briefly introduced in the above sections and introduce them in more precise contexts (cf. Figure 1).

Inflow capacity How much water that can flow into a dam during a timestep

Ideal level The ideal level of the dam. When the simulation starts, the dams are at this level.

Critical high The critical high level of the dam. An alarm is raised if the dam fills above this level.

Critical low The critical low level of the dam. An alarm is raised if the dam empties below this level.

The three different dams to simulate in this exam have parameters as shown in Table 1.

Name	Ideal level	Critical high	Critical low	Outflow capacity	Inflow capacity
dam01	1200	1800	400	40	50
dam02	600	800	200	30	20
dam03	800	1000	500	30	40

Table 1: Parameters for the simulated dams.

[1]This is analogous to a real-world deployment of the system which performs sensor read-outs at a specific frequency.

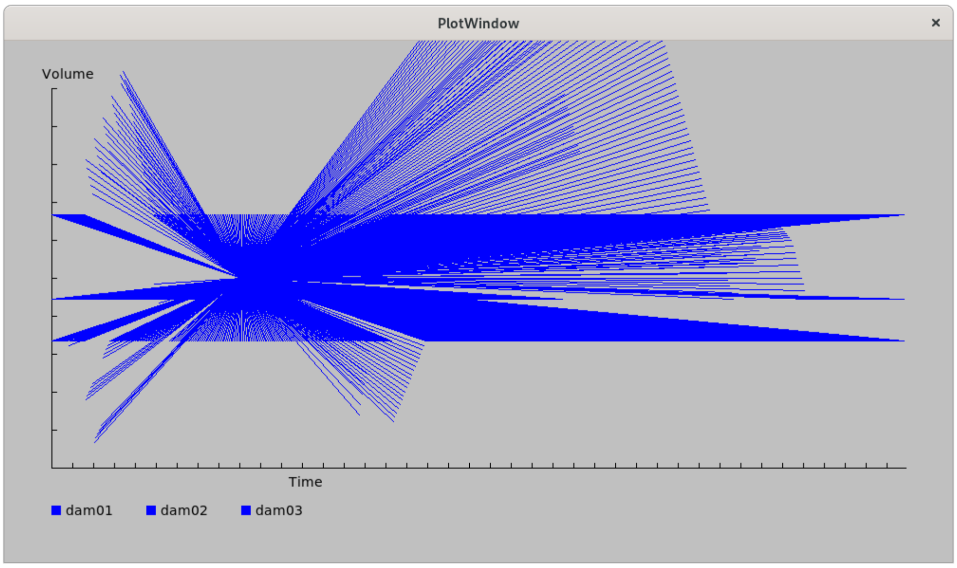

Figure 2: The initial plot produced by the handed-out code. This is not very useful.

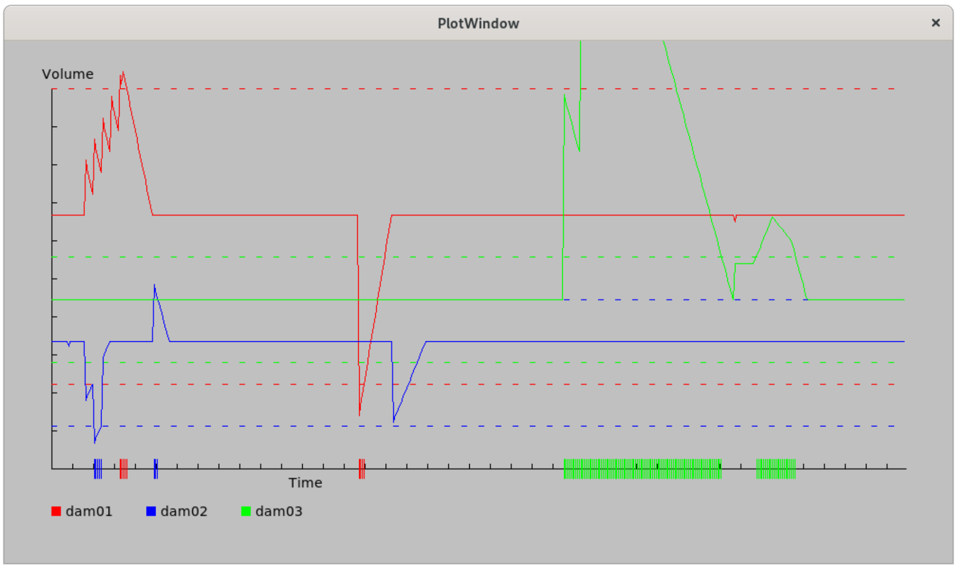

Figure 3: An example of a plot produced by the code after the plotting (P) functions have been implemented. The colored lines intersecting the x-axis indicates alarms for levels exceeding the critical maximums and minimum limits. The dashed horizontal lines indicate the limits.

Implementation overview

The implementation of the assignment is divided into several different files but you only need to change some of them in order to solve the assignments. A significant part of the code is merely scaffolding required to make the program run. Please keep this in mind, as it is not a good idea to spend time understanding the entire code base, since this is not required to be able to solve the assignment. To make the code easier to navigate, each assignment will tell you which files you need to change and which declarations you need to understand to solve that assignment.

The first time you compile and run the program you will be presented with the window shown in Figure 2. Obviously, this is not very useful and by solving the assignments, you will gradually implement the missing parts of the program and work towards the desired result shown in Figure 3. It is not a requirement that all the submitted code is actually running to pass this exam. However, we recommend that you do test your answers to verify their correctness.

The individual assignment pars that comprise this third assignment, indicate which source files that need to be changed to solve them. Throughout the source files, you will find the parts marked with a comment:

`// Assignment XX`

followed by the description of the task (in English). These descriptions are identical to the corresponding descriptions you will find in this document. You will also see comments stating

`// Used for assignment XX.`

These comments are used to draw your attention towards implementation details, e.g. declarations, which are needed to solve a specific assignment. Finally, associated with each assignment is a matching pair of comments

`// BEGIN: XX` and `// END: XX`.

It is crucially important that all of your answers are written between such comments to support the assessment mechanics. If any code is already written between the BEGIN and END comments in the handouts, you may, and often should, replace that code with your own implementation.

For example, for assignment P1 you see the following code in the handed out `plotter.cpp`

```cpp
Color::Color_type GraphColor::next_color()
{
  // BEGIN: P1
  (void)cur_color;
  return Color::black;
  // END: P1
}
```

After implementing you solution, you should end up with this instead

```cpp
Color::Color_type GraphColor::next_color()
{
  // BEGIN: P1

  // Your code here

  // END: P1
}
```

Note that the BEGIN and END comments **should NOT be removed**.

Finally, if you find any of the assignments unclear, you should state your assumptions as comments in the submitted code.

Theme 1: Plotting

All of the assignments in this section relate to the implementation of the plotting functionality implemented in the files `plotter.h` and `plotter.cpp`

3.1 P1: next_color

Implement the function `next_color` such that it each call to the function returns the sequentially next color from the vector `colors` defined in the `Color` class (`plotter.h`). When there are no more colors in the list, the function should throw an exception with a descriptive message. Use the private class member variable `cur_color` to keep track of the current location in the `colors` vector.

Example: The first call to `next_color` returns `Color::red` as it is first in the `colors` vector. The second call returns `Color::blue` as it is second in the list, and so on.

3.2 P2: Random color

The class `GraphColorRandom`, defined in `plotter.h`, extends the `GraphColor` class and modifies its functionality such that its `next_color` member function now returns the colors in a random order. Note that each color should still only be returned once. Implement the constructor of the `GraphColorRandom` class so that it shuffles the `colors` vector when the class is instantiated.

To test your implementation, change the instantiation of `GraphColor colors` in the class definition of `Plotter` in `plotter.h` as indicated in that file.

Implement the `GraphColorRandom` class.

Hint: A common algorithm for shuffling a list is Fisher-Yates shuffle[2] given below

```
for i from 0 to n-2 do
    j ← random integer such that i <= j < n
    exchange a[i] and a[j]
```

3.3 P3: get_color

Implement the function `get_color` which takes the name of a dam as a parameter and returns the plot color assigned to the dam. The function should look up the color in the class member map `dam_colors` defined in `plotter.h`. If a dam does not exist in the map, the function should create an entry for the dam and assign a color using the `colors.next_color()` function. `colors` is the `GraphColor` instance that is defined in the `Plotter` class in `plotter.h`

3.4 P4: Track last point

The plots of the water levels of the dams are composed of a series of line segments. For each timestep, we only get the current value of the given dam. Therefore, we need to keep track of where the line segment for the previous timestep ended in order to know where to start the line segment for the current timestep. This is the purpose of the `get_last_point` function.

Implement the function `get_last_point` which returns the point corresponding to the end of the previous line plotted for the dam. The function takes two parameters; the name of the dam and the next point to be plotted. The last points should be stored in the class member map `last_points` defined in `plotter.h` which maps the name of a dam to the previous point plotted for that dam. If no entry in the map exists for the dam, the function should create a new entry for it and return the point `(0,0)`. If an entry is found, the function should return the previous point and update the map with the next point.

3.5 P5: Dam alarms

If a dam encounters a critical condition, i.e. its water level goes beyond the critical maximum and minimum limits, we want to visualize this in the graph.

[2] `https://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle`

Implement the function `plot_alarm` which plots a vertical line intersecting the x-axis of the graph (as shown in Figure 3) indicating that a water level alarm occurred. To color the lines, use the `get_color` function from assignment P3.

The `plot_alarm` function takes three parameters: The name of the dam the alarm was triggered for, the timestep during which the alarm occurred and the type of alarm. The `timestep` value is used to calculate the x-coordinate placement of the line. Note that the function `calc_x` maps a timestep to a value along the x-axis of the coordinate system. The alarm is given as a parameter of the type `Dam::WaterAlarm`, an `enum` defined in `dam.h`. The private member variable `y_len` contains the height of the y-axis. You can use this to determine the height of the line.

The `plot_alarm` function is called on every `timestep`, and the value of the `alarm` parameter is the return value of the function `Dam::get_alarm` (from assignment D2) in `dam.cpp`. You should only draw the "alarm line" if the value of the alarm parameter indicates that an alarm occurred at that `timestep`. Note that our default implementation of the `get_alarm` function simply produces some random alarms. Therefore, even if you have not yet solved assignment D2, you can still use `get_alarm` for testing now.

Hint: To get lines intersecting the x-axis (as shown in Figure 3) you can draw lines with the start- and end points

`(calc_x(timestep), y_len + 40)` and `(calc_x(timestep), y_len + 60)`

respectively.

This assignment should be implemented in the file `plotter.cpp`.

3.6 P6: Dam limits

To make it easier to tell when the water level of a dam is crossing its maximum or minimum boundaries, we want to visualize these limits in the graph as well.

Implement the function `plot_limit` which draws a dashed horizontal line on the graph (some examples were shown in Figure 3). The function takes the name of the dam and a limit as parameters and it is called when the dams are initialized. Recall that you can map a value to a y-coordinate using the `calc_y` function and that the `calc_x` function called with values between 0 and `timesteps` will generate the entire possible range of x-coordinates. Draw the lines in the color that is returned by the `get_color` function. Finally, remember to add the `Shapes` you draw to the `shapes` vector, so that they are deallocated when the `Plotter` class is destroyed.

Theme 2: Dam models

The assignments in the Dam class are concerned with the dam model implemented in the file `dam.cpp`. Also refer to the definition of the `Dam` class in `dam.h`

3.7 D1: Water level delta

Implement the function `get_water_level_delta` which returns the difference between the current and ideal water level. The function should return the absolute value of the delta, so you can use the function `abs` for that. Recall that the current water level and the ideal water level are stored in the member variables `water_level` and `ideal_level`, respectively, in the `Dam` class. The `Dam` class is defined in `dam.h`.

3.8 D2: Water level alarms

Implement the function `get_alarm` which indicates to the caller if the water level of the dam is above the critical high level or below the critical low level. The function should return one of the values defined in the `WaterAlarm` `enum` in `dam.h`. Also if the water level is within the critical bounds, return the appropriate value from the `enum`.

Recall that the current water level, critical high and critical low levels are stored in the private member variables `water_level`, `water_level_max` and `water_level_min` respectively. The variables are all defined in `dam.h`.

3.9 D3: Water level restoration

If the water level goes outside the critical bounds of the dam the system must take action to restore the water level to ideal.

As per the physical limitations of a dam, described in the introduction, the amount of water that can flow in or out of the dam per timestep is given by the `outflow_capacity` and `inflow_capacity` class member variables. This means that it takes time for the water level to return to normal. Note that if the difference between actual and ideal water level are less than the inflow or outflow capacity you can simply set the water level to ideal directly. You can use the `water_level_delta` function defined in assignment D1 to check this.

Implement the function `restore_water_level` which restores the water level in the dam if it is over or under the ideal level.

Theme 3: File input/output

The assignments in this file are concerned with reading and writing from and to text files in Comma Separated Values (CSV) format. To solve the assignments, you need to change the files `damsim.cpp` and `sensors.cpp`.

3.10 F1: CSV file output

The solution for this assignment should be implemented in the file `damsim.cpp`.

Write a function which writes simulation data in CSV-format to the file stream `output_file`. You should write the lines to the file stream in the following format:

```
<dam name>,<current timestep>,<current water level>
```

You can get the name of the dam and the current water level from the `get_name` and `get_water_level` functions of the Dam object passed to the function as a reference. Examine the declaration of the `Dam` class in `dam.h` for more details. The current `timestep` is passed directly through the `cur_timestep` parameter.

```
timestep,dam,inflow,outflow
10,dam02,0,40
10,dam01,40,0
20,dam01,300,0
20,dam02,0,300
25,dam01,300,0
25,dam02,0,300
...
```

Figure 4: Excerpt from the sensor data CSV file `sensor_data.csv`.

3.11 F2: CSV file reading

Currently, the sensor readings used for the simulation are hard-coded in the constructor of the `SensorsFile` class in `sensors.cpp`.

In this assignment, we make the dam simulator more flexible by extending it with the ability to read sensor data from simple CSV files[3]. Several of these files are provided with this assignment in the `data/` directory. The sensor data files are named `sensor_dataX.csv` where X is an integer.

Figure 4 shows the first few lines of the file `sensor_data1.csv`. As you can see, the format is as follows. The first line of the file contains the names of the fields. In this case the first field is `timestep`, the second is `dam`-id, etc. The subsequent lines contains the actual readings. In this example, the first reading occurs on `timestep` 10 for dam02 registering an inflow of 0 and an outflow of 40.

Implement the constructor of the `SensorsFile` class in the `sensors.cpp` file to read the file name passed as the `fname` parameter taken by the constructor. For every line you read, you should call the `insert_reading` function to register the readings. We do not expect you to gracefully handle exceptional conditions, e.g., file format errors. You may, therefore, assume that your function will only be used to parse files conforming to the format described above.

As the hard-coded default sensor readings correspond to the readings found in the `sensor_data1.csv` file, you should still see the plot shown in Figure 3 if you implemented the function correctly. If you wish, feel free to try plotting some of the other sensor reading files by changing the `measurement_file` variable in the `main` function in the file `main.cpp`.

[3]Note that the CSV files you encounter in the wild may be more complicated than this.

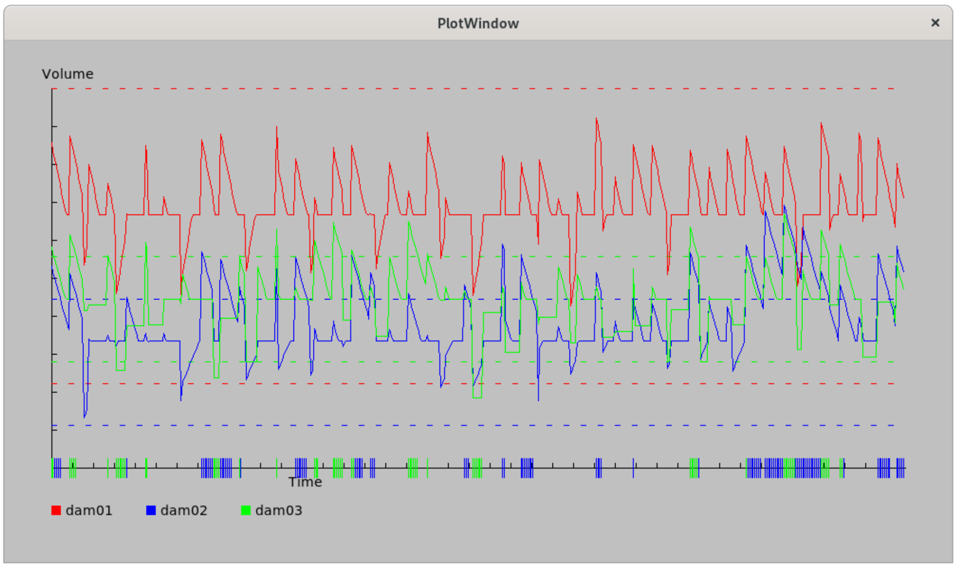

Figure 5: An example of what the simulation plot with randomized sensor data might look like.

Theme 4: Class inheritance

The assignments in this part are solved by modifying the files `sensors.h` and `sensors.cpp`. Additionally, In order to test your implementation, you need to change the template parameter of the instantiation of `DamSim` in `main.cpp`. The relevant location in `main.cpp` is marked. Also, you must uncomment the line at the bottom of `damsim.h` as directed.

By default, the implementation as handed out uses an instance of the class `SensorFile` to get sensor readings. This file implements the abstract class `Sensors` and reads sensor readings from a file. In this part of the exam, we ask you to implement a sensor reading generator which generates random sensor readings rather than reading them from a file.

A `Sensor` has a single function: `get_reading`. The `get_reading` function returns a `Reading struct` (defined in `sensors.h`) containing the measurements for the given `Dam/timestep`. If there are no sensor values to read for a given `timestep and dam` combination, the function should just returns a `Reading struct` with both values set to 0.

When you have implemented all of the sub-questions in this part you may end up with a plot similar to Figure 5. Your result will, of course, differ from this depending on which algorithm you choose for generating random readings in your `get_reading` implementation.

3.12 S1: Virtual class declaration

In `sensors.h`, write a class declaration for `SensorsRandom` which declares the functions of the abstract base class `Sensors`. Implement a delegating constructor which calls the constructor of the virtual class `Sensors`.

3.13 S2: Generating reading

Implement the function `get_reading` defined in the `SensorsRandom` class. The function gets the current timestep and the name of the dam as parameters and returns the struct `Readings` (defined in `sensors.h`) containing the sensor values.

You are free to decide the logic for this function, however, it must return sensible values for inflow and outflow generated using the `rand()` function. For example, a simple approach would be to return random values between 0 and 100 for inflow and outflow every 10th timestep.

End of this exam.