Part III: WeatherForecast

The maximum score for part 3 is 190 points. 180 in Inspera!

3.1 Info

Good work! Now you come to part 3 of the exam, which are self-programming tasks. This part contains 18 tasks which together give a maximum score of around 200 points and count for approx. 60% of the exam. IMPORTANT:

The zip file you will download contains compilable (and executable) .cpp and .h files with pre-coded parts and a full description of the exercises in Part 3 as a PDF file. After downloading the zip file, you are free to use a development environment of your choice (such as VS Code) to work on the assignments.

In order to pass this exam, it is ABSOLUTELY ESSENTIAL THAT YOU UPLOAD THE ZIP FILE. After the end of the exam (13:00) you have 30 minutes to do this

The short answers in Inspera (parts 1 and 2) are automatically saved every 15 seconds, until the exam time is over. And the zip file (in part 3) can also be changed / uploaded multiple times. So if you want to make any changes after you upload the file, you can just change the code, rezip it, and upload the file again. When the trial period is over, the latest version of anything you've written or uploaded in Inspera will automatically be submitted as your current response, so make sure you upload at least once halfway through, and once before the time expires.

On the next page in Inspera you can download the .zip file, and later upload the new .zip file with your own code included. Remember that the PDF document with all the tasks is included in the zip file you download.

3.2 Intro

WeatherForecast is an application that renders a one week weather forecast for a city. The style of the rendering is in line with what is usually shown by weather forecasting apps. In the handed out zip file, you will find a runnable application skeleton lacking key pieces of functionality. Through a series of small tasks, we guide you through implementing the missing pieces to make the application fully functional. If all of the tasks in this assignment are completed successfully, the end result is shown in Figure 1, When running the handed out code before making any changes, you will be greeted by the window shown in Figure 2. In the following, we will introduce the application, its functionality and the file format used for storing the graphical representation of weather types, the files containing the location data and the files containing the weather forecast data.

3.3 Application overview

The application is implemented in several classes that relate to each other in a hierarchical fashion. The hierarchy is depicted in Figure 3.

The top-level class of the application is `Application`. This class is responsible for setting up the GUI and instantiating the `Forecast` class which reads a CSV file containing the weather forecast for a city and renders the resulting forecast. The CSV files containing the forecasts are located in the `forecasts` directory. A forecast shown by the application contains two things (Figure 1): the expected weather for 5 days and statistics for the expected weather such as the average and maximum temperatures.

Statistics generation and rendering takes place within the `Forecast` class and is implemented in assignments F3-F5. A day of the forecast is represented by the `Day` class. The `Forecast` class instantiates five `Day` classes, one for each day, with the weather for that day as read from the forecast CSV file. The forecast CSV files are found in the `forecasts` directory. We do not ask you to write code for parsing a CSV file or find the list of available forecasts as this is already implemented in the handed out code.

Finally, in order to show an icon representing the weather (e.g. a cloud or a sun for cloudy and sunny weather respectively) each `Day` class instantiates an `ImageRenderer` class for loading and showing the image file containing the appropriate icon. The weather icon image files are found in the `symbols` directory in the

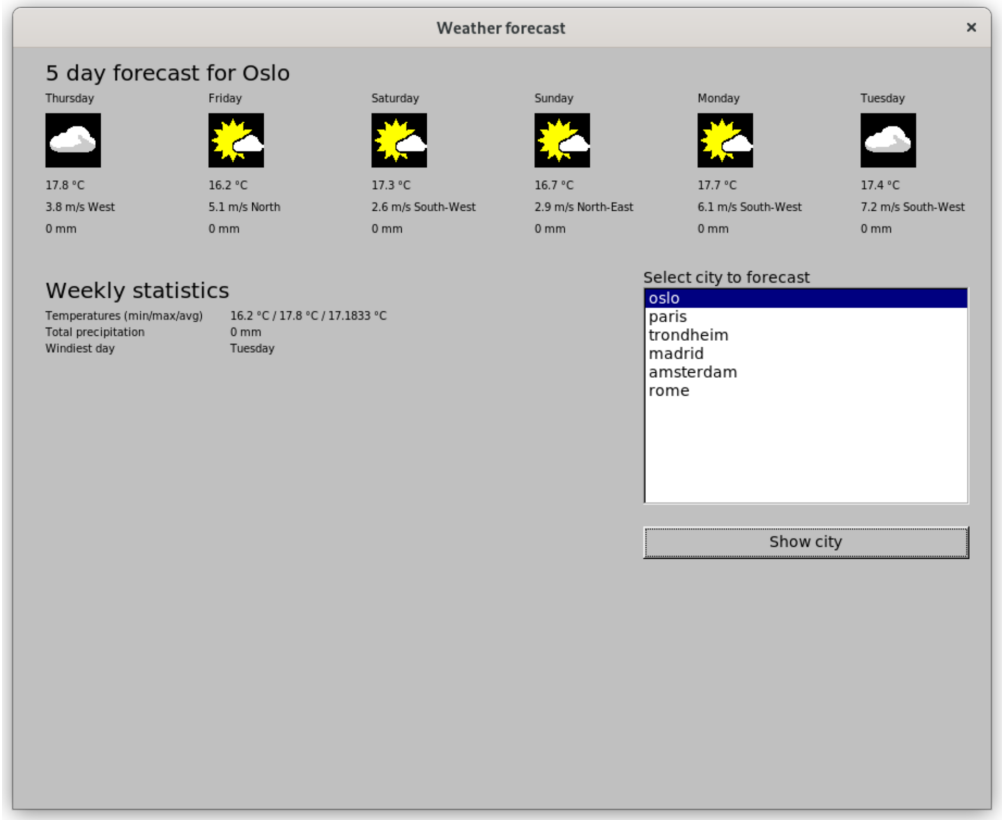

Figure 1: The complete application after selecting "Oslo" and clicking "Show city".

handout. We ask you to implement the functions for reading the image files containing weather symbols in tasks R1-R3 and we provide a detailed description of the image format in the introduction to these tasks.

How to solve part 3

```
int Forecast::get_day_placement(int day)
{
  // BEGIN: F1
  //
  // Write your answer to assignment F1 here, between the // BEGIN: F1
  // and // END: F1 comments. You should remove any code that is
  // already there and replace it with your own.

  return 0;

  // END: F1
}
```

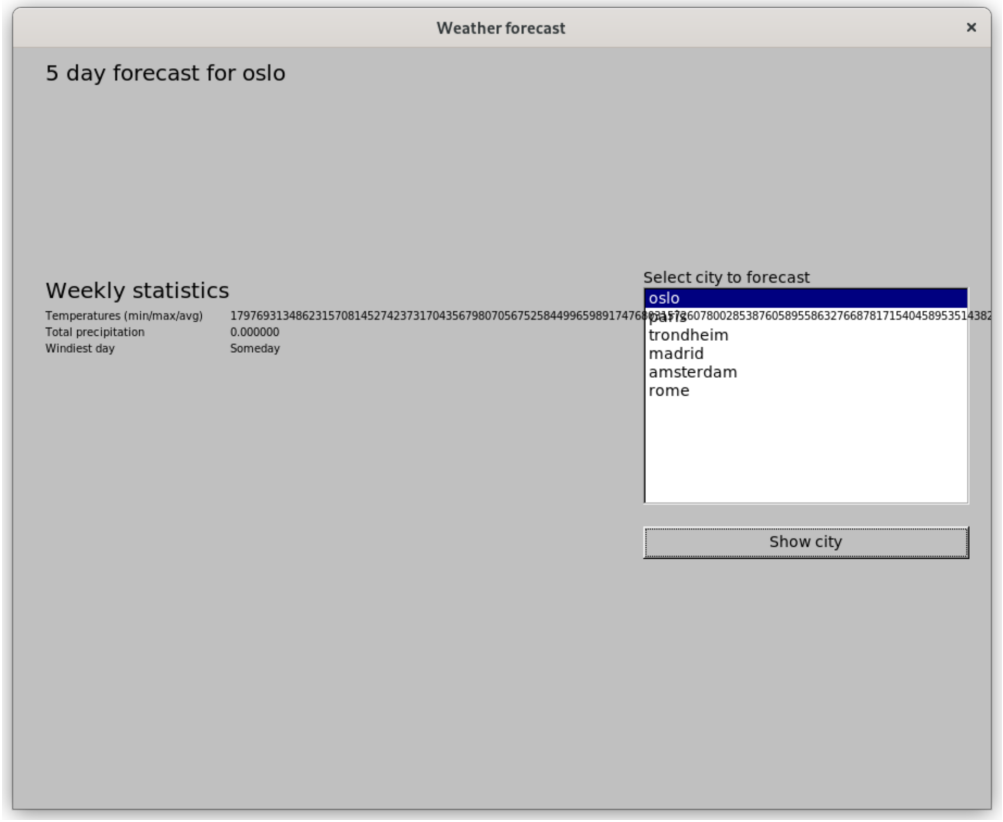

Figure 2: The handout application after selecting "Oslo" and clicking "Show city".

```
int Forecast::get_day_placement(int day)
{
  // BEGIN: F1
  //
  // Write your answer to assignment F1 here, between the // BEGIN: F1
  // and // END: F1 comments. You should remove any code that is
  // already there and replace it with your own.

  /* Din kode her */

  // END: F1
}
```

3.3.1 Where to find the tasks?

The assignments are named as a letter followed by a number. The letter indicates in which file the assignment is defined. Refer to Table 1 for the full list. For example, the assignment F1 is found in the file `forecast.cpp`

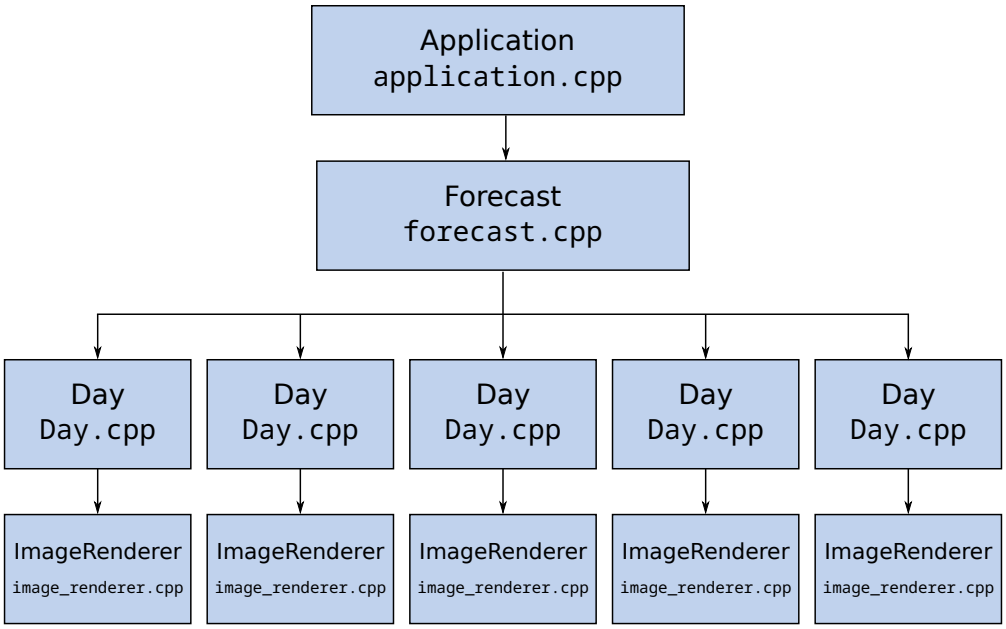

Figure 3: The class diagram of the WeatherForecast application.

Table 1: Overview of the task letter to file mappings

Letter	File
F	`forecast.cpp`
D	`day.cpp`
R	`image_renderer.cpp`
U	`util.cpp`

The tasks

Formatting values (40 points)

Any weather forecast must include nicely formatted numeric parameters describing the expected weather such as temperatures and wind speed. In this part of the assignment, we write the functions to format these values.

1. (10 points) **U1: Weekday numbers to names**

   Write a function that maps an integer between 0 and 6 to a weekday name. The first day of the week (value 0) is Monday. If the value of the `weekday_num` parameter is outside of that range, the function should throw an exception.

2. (10 points) **U2: Friendly wind directions**

   Write a function that bins the compass angles of wind directions (0 to 360) degrees to 8 different directions (e.g. North, Norh-West, etc.). If the value of the `heading` argument is outside of the valid range, the function should throw an exception.

   The mappings are as follows:

   - 337.5 - 22.5: North

- 292.5 - 337.5: North-West
- 247.5 - 292.5: West
- 202.5 - 247.5: South-West
- 157.5 - 202.5: South
- 112.5 - 157.5 South-East
- 67.5 - 112.5: East
- 22.5 - 67.5: North-East

3. (10 points) **U3: Get unit as string**

   Consider the enum `Unit` defined in `util.h` as follows

   ```
   enum Unit {
       UNIT_MS, // Meters per second, m/s
       UNIT_MM, // Millimeteres, mm
       UNIT_DC // Degrees celcius, °C
   };
   ```

   Write a function that given a `Unit` enum value returns the corresponding string representation. For example, `get_unit(UNIT_MM) == "mm"`.

4. (10 points) **U4: Format units**

   Write a function that formats numeric weather parameters for presentation by rounding floating point numbers to a specified number of decimals and appending a unit. The function takes three parameters: The `value` to be formatted, the number of `decimals` that the value should be rounded to and the `unit` that should be appended to the formatted string. Use the function `get_unit` implemented in U3 to convert the `unit` value to a string.

   For example, the function should return values that makes the following comparisons true (assuming that the `get_unit` function is correctly implemented): `format_value(3.94, 1, UNIT_MM) == "3.9 mm"` and `format_value(3.587, 2, UNIT_MS) == "3.59 m/s"`

Showing a day in the forecast (10 points)

5. (10 points) **D1: Rendering a day**

   Now, we use the helper functions defined previously to render a day of the weather forecast. To add text to the forecast, the function `draw_text(Point pos, string value)` is provided. Where applicable, round floating point numbers to one decimal and make sure that the appropriate unit is appended. For positioning the text, use the variables `xpos` and `ypos` which holds the coordinates of the upper right corner of a day. The list below tells you which information should be present in the forecast, the text's relative y-offset and the variable of the `Day` class where the information is found. Use the variables `xpos` and `ypos` together with the relative y-offsets to construct the `Point` parameter `pos` of the `draw_text` function.

   - The day of the week. Y-position: 20, variable: `day`
   - The temperature. Y-position: 100, variable `temp`
   - The wind speed and direction: Y-position: 120, variables: `wind_speed`, `wind_dir`
   - The amount of precipitation: Y-position: 140, variable: `precip_amount`

Showing the forecast (50 points)

6. (10 points) **F1: Forecast layout**

Write a function that returns the horizontal pixel offset for each day in the forecast. Each day is a 150 pixels wide. If the value of the `day` parameter is invalid, i.e. it is outside the range $0 \leq \text{day} \leq 6$ the function should throw an exception.

7. (10 points) **F2: Capitalize a string**

   Write a function that capitalizes the first character of a string such that `capitalize("trondheim") == "Trondheim"`. This function only needs to change the first character of the string. Remember that the function `toupper` can be used to change a `char` value to uppercase.

8. (10 points) **F3: Precipitation statistics**

   Write a function that calculates the total amount of precipitation expected across all the days covered in the forecast and save the result in the `total_precip` class variable. When this function is called, `total_precip` is 0.

   To do this, iterate over the `days` vector in the current `Forecast` class instance. Remember that the `days` vector is defined as `vector<unique_ptr<Days>> days` in `forecast.h`. To get the amount of precipitation expected for a day, use the public `precip_amount` variable of the `Days` class instance.

9. (10 points) **F4: Temperature statistics**

   Write a function that calculates the maximum, minimum and average temperatures expected for the days included in the forecast and save the resulting values in the `max_temp`, `min_temp` and `avg_temp` class member variables respectively. When this function is called `max_temp` and `min_temp` are set to the minimum and maximum values supported by the C++ `double` type respectively.

   As in the previous tasks, iterate over the `days` vector and access the `temp` member variable of the `Day` class instance to retrieve the temperature expected for a day.

10. (10 points) **F5: Wind statistics**

    Write a function that finds the windiest day of the week and save the result to the `windiest_day` class member variable. Keep in mind that we are interested in identifying the *day number* (i.e., an integer between 0-6) and not the actual wind speed.

    The wind speed for a day can be found in the public `wind_speed` member variable of the `Day` class instances.

Rendering images (90 Points)

In this final part of the assignment we implement the functions for reading and rendering the image format used for storing the weather symbols in the forecast. Before digging into the tasks, we describe the image format that the subsequent tasks asks you to implement[1].

The image file format is divided in three parts: The image header, color definitions and Image data. The parts are defined as follows:

Image header One line with three numbers separated by spaces. The first two numbers gives the width and the height of the image and the final number gives the number of colors in the image.

Color definitions This section maps a *color reference* to a *color name*. The color reference is a single character representable by the C++ type `char` and the color name is a `string`. The character can be anything except a space and each line contains a single definition. The number of lines, and thus the number of definitions, is given by the last number of the image header. Thus, if the last number of the image header is 5, as in Figure 4, you should expect the 5 lines following the header to be color definitions.

[1] For historical reference, the file format used in this assignment is a slight simplification of the XPM2 file format that was used for storing images in early versions of the X11 window system.

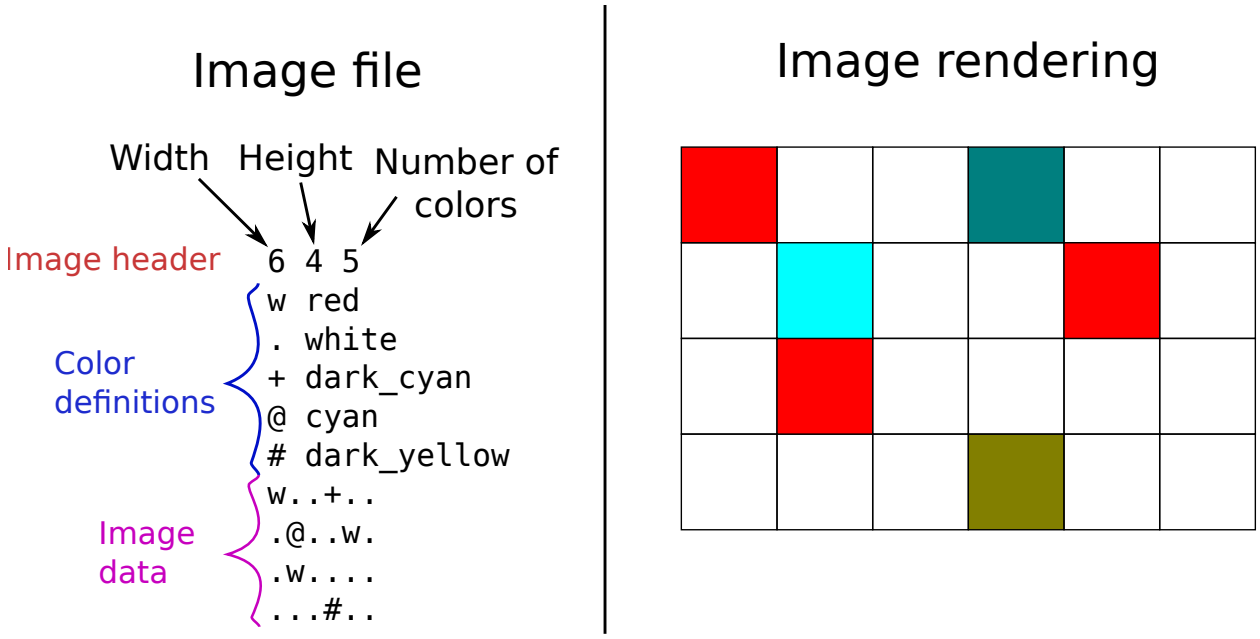

Figure 4: An example of a simple image file (left) and its rendering (right).

Image data The last section of the image contains the actual image data. Each line of the image corresponds to a line in the image file and therefore, if an image has 6x4 pixels (as in Figure 4) the image format section contains 4 lines with 6 characters each. Each character corresponds to one of the defined colors. For example, the two first horizontal pixels of the image in Figure 4 are given by the characters "`w.`". From the color definition section we see that `w` maps to the color `red` and `.` maps to the color `white`. Therefore, the two first pixels of the image are red and white respectively. For convenience, we provide the map `color_map` defined in the `ImageRenderer` class that you can use to convert the color names used in the image file to the color values used in the drawing functions.

Internal representation. When an image is loaded from an image file it is transformed to the internal representation used in the `ImageRenderer` class. The internal representation is comprised of two datatypes defined in `image_renderer.h` as follows:

```
1 map<char, Graph_lib::Color::Color_type> colors;
2 vector<unique_ptr<vector<Graph_lib::Color::Color_type>>> image;
```

The map `colors` contains the mappings between *color references* and the color codes used in `Graph_lib`. We implement a helper function for adding entries to this map in task R2 and the function for reading color mappings from the image file in task R6. The vector `image` contains the actual image data. Note that it is defined as a vector of vectors of colors. The outer vector contains one vector per row of the image and the inner vectors contain the colors of the pixels comprising each row. So, after the image in Figure 4 has been read, the `image` vector contains 4 vectors (corresponding to the height of the image) each of which contains 6 colors (corresponding to the width of the image).

The following three tasks asks you to implement parsers for the three parts of the image format. Then, we implement the function for rendering the internal representation of an image on-screen.

11. (10 points) **R1: Color lookup**

Write a function that returns the `Graph_lib` color type corresponding to the color name given in the parameter `color`. To do this, you may use the map `color_map` defined in `image_renderer.h`. If the parameter `color` refers to an invalid color the function should throw an exception.

12. (10 points) **R2: Add color references**

    Write a function that given a color reference (parameter `ref`) and a color name (parameter `color_name`) adds an entry to the `colors`-map. Use the function `get_color_value` to look up the color names.

13. (10 points) **R3: Add image rows**

    Write a function that adds a new vector to the `image` vector. The intention is that this function is called for every additional row of pixels found when reading the image file. Remember that the new vector must be instantiated as a `unique_pointer`.

14. (10 points) **R4: Add image pixels**

    Write a function that adds a pixel to the image. Since the image data is read line-by-line, always add the pixel to the most recently added vector in the `image` vector.

15. (10 points) **R5: Reading the header**

    Complete the function for reading the image header. Most of the hard work is already done for you but the part for reading the actual values into the variables `width`, `height`, and `ncolors` is missing. Your task is to add this.

16. (20 points) **R6: Reading the colors**

    Implement the function for reading the color definitions of an image. An example of color definitions is given in Figure 4

    The image file stream is passed in the `image_file_stream` parameter and the number of colors to be read are passed in the `ncolors` parameter. When the function is called, `image_file_stream` points to the beginning of the colors section of the file (assuming you implemented the previous task correctly). For every color definition read, call the function `add_color` to add the color to the internal representation. The `add_color` function takes two parameters: a `char` containing the color reference and a `string` containing the color name.

17. (20 points) **R7: Reading the pixels**

    Implement the function for reading the pixels part of the image. From the `width` and `height` variables initialized in the `read_image` function, you know the dimensions of the image and therefore the number of lines of image data to read and the number of characters each line has. For every line you read, make a call to the `add_line` function without arguments. For every pixel you read, call the `add_pixel` function with the name of the color of the pixel. The mappings between color references and values are stored in the map `colors`. Thus, you can use the color reference `char` you read from the image file to perform a lookup in the `colors` map to get the needed argument to the `add_pixel` function.