ICT283 Assignment 1

Objectives:

1. design and write good structured and object oriented C++ programs;
2. design and write well documented C++ programs that use programmer designed data structures;
3. design and execute test plans (unit tests and application tests);
4. draw Unified Modeling Language class diagrams that can be implemented;
5. discuss (and apply) the theory and application of data structures and the algorithms that use them and are used by them;
6. design and implement solutions that adhere to given specifications and requirements.

You do not work in groups for this assignment as this is an individual assignment.

Due: (Submission date/time in the LMS submission area will override the due day/time below.)
4 pm - Perth, WA time
Friday (7th teaching week since the start of semester/trimester)

How to submit (also see unit guide - section on Assignment/Project submission/return:
Internals: LMS
Externals: LMS

For submitting in LMS, zip up the entire folder. Make sure that you have included all needed files. Do not include temporary files or files not relevant to the assignment.

Name the zip file with the unit code, Assignment number, your name, student number.
ICT283_Asignment1_Samuel_A_Bent_00700707.zip (submission by Sam A. Bent with student number 00700707) or alternatively,
ICT283_Asign1_Samuel_A_Bent_00700707.zip

Textual submissions should be type-written. External documentation can only be in the following formats:
Text (.txt)
PDF (.pdf)
RTF (.rtf)
HTML (.html)
Image formats: PNG, GIF, JPG, TIFF, and BMP.

Assignment cover sheet requirements are listed in the unit guide found in the Admin area or “Getting Started” section in LMS.

LMS submissions do not require submission of a cover sheet but you should ensure that the requirements are met.

Mandatory Preparation:

Before attempting this assignment, you must complete all readings and lab work up to and including Topic 5. It is also advisable to test the Vector class in various ways as indicated in Lab 6.

Textbook: It is essential that you complete chapter 1, and all chapters till the chapter on "*Overloading and Templates*" from the unit textbook "*C++ Programming: Program Design Including Data Structures*" by D.S. Malik. You can use whichever edition is available at your institution's bookshop or provided to you.

Lecture notes and Lab work: All lecture material and laboratory work up to Topic 5. It is also advisable to complete lab 6, where your Vector class is tested in isolation. You should never incorporate a software component in your main build, until you have tested that software component first.

Assignment Question:

Please read the following very carefully to identify the requirements of the assignment. You should make separate notes.

Do not start the design or coding until you have understood the requirements. If unsure of something, check the Question and Answer (QandA) file for the assignment. If your query is not addressed in the QandA file for the assignment, please email your query to your lecturer.

You should also check the assignment's QandA file regularly as the file can be updated and it may have discussion of new issues that you might not have considered.

This assignment continues from laboratory 5 (Lab 5). You will be reusing the Vector, Date, and Time classes as well as data file reading code from Lab 5. You must ensure that these classes have been thoroughly unit tested before using in this assignment. See file "*LabExcTopic05.doc*" for Topic 5.

If you have not completed lab 5, you will need to complete question 5 from that lab before starting work on this assignment. Unit testing of the classes used in Lab 5 is absolutely essential.

Design an object-oriented solution and implement the solution in C++ to solve the problem described below.

The data files that you need for this assignment are made available to you in the ***data*** folder. This data comes from historical data recorded by sensors and was obtained from http://wwwmet.murdoch.edu.au/. Data is logged at intervals of 10 minutes. Sample data in comma-separated value text files is made available for this assignment. Each file contains a year's worth of data for multiple sensors. Data for each date-time recording are on separate rows. Within each row, a comma separates the value for each sensor. The sensor codes are found at http://wwwmet.murdoch.edu.au/sensors. Examine the data using a text editor like notepad++ **and** in a spreadsheet application. If you download your own data, you may want to remove the first few rows if the key list (sensor list) is there. The data that is supplied to you with this assignment does not contain the key list, and there is no need to download your data unless you specifically want to.

Note: Don't tick "Date/Time in UTC" when downloading the data yourself. When downloading your own data, you may find that the data column arrangement may not be the same if the backend code at the website has changed.

To understand the nature of the data, you must complete the lab for topic 5. You need to understand the nature of the data files and how to read the data and load the data into the required data structures. You will need the code that you developed in lab 5 to continue work on this assignment.

Design and then write an Object Oriented program in C++ that meets the specifications shown below. You should provide a suitable menu with an exit option in your main program. When designing the output, imagine yourself as the user of the program. You want the user interaction to be concise but user friendly on the command line. Do not use GUI interaction.

Sample output formats shown below use *made up* data for the year 1905.

Menu options are:

1. The average *wind speed* and average *ambient air temperature* for a specified month and year. (print on screen only)

   Example output format if there is data:
   January 1905: 5.5 km/h, 25.5 degrees C

   Example output format if there is no data:
   March 1905: No Data

2. Average *wind speed* and average *ambient air temperature* for each month of a specified year. (print on screen only)

   Example output format is:
   1905
   January: 5.5 km/h, 25.5 degrees C
   February: 4.5 km/h, 27.5 degrees C
   March: No Data

   …

3. Total *solar radiation* in kWh/m^2 for each month of a specified year. (print on screen only)

Example output format is:
1905
January: 196.4 kWh/m^2
February: 200.3 kWh/m^2
March: No Data
…

4. Average wind speed (km/h), average ambient air temperature and total solar radiation in kWh/m^2 for each month of a specified year. (write to a file called "*WindTempSolar.csv*")

   Output Format:
   Year
   Month,Average Wind Speed, Average Ambient Temperature, Solar Radiation

   Example output format is:
   1905
   January,5.5,25.5,196.4
   February,4.5,27.5,200.3
   …

   Year is printed on the first line and the subsequent lines list the month and the average *wind speed*, average *ambient air temperature* and the total *solar radiation* for each month. The values are comma separated.

   For menu item 4: If data is not available for any month, do not output the month. In the example, March 1905 has no data. Nothing is output for March. If the entire year's data is not available, output just the year on the first line and the message "No Data" on the second line.

5. Exit the program.

The user specifies the year and/or month. Your program asks for these on the command line and the user types in the required values and presses the "Enter" key. Date and month entries on the command line must be numeric. For example, the user types in the value 1 and not the string January or Jan to represent the first month of the year.

Although there a number of data columns in the data file, you will only use columns with labels ***WAST*** (date and time), ***S*** (Wind Speed), ***SR*** (Solar Radiation) and ***T*** (Ambient air temperature).

Convert units carefully as the output units are not all the same as the units in the data files. For example input column S is in m/s but the output needed is km/h. The units for solar radiation in the input data file and the output are also not the same.

Processing:

Your program loads the data first from the *data* folder.

The program for assignment 1 will read only one input data file from the ***data*** folder. Do not read data files from anywhere else other than this folder.

After loading the data into the required data structures (see below), your program displays the menu to the user. The required data structures (see below) must be used for menu items 1 to 4.

Make sure the design is modular to cater for future iterations of the assignment requirements. For example, future iterations might require handling of more data fields, use of different data structures. New output requirements may be needed.

If you do not attempt to "future-proof" your design (modularise, increase cohesion, reduce coupling), you will find that you will be re-doing all (or most of) the work to cater for new or modified Assignment 2 requirements within a very much-restricted time frame (can be less than 2 weeks).

Heed the advice and lessons learned in Labs 1 to 5. Complete all readings **and** lab work from topics 1 to 5 before starting to work on this assignment. You can, of course, write small programs to test out ideas needed for this assignment: like how to read and extract data from the given data files (lab for topic 5); test out algorithms for doing the required processing and unit test basic classes like the Date, Time and Vector classes from lab 5 for use in this assignment.

Completion of Lab 5 is important for this assignment as a number of needed classes are created and unit tested in Lab 5 and in previous labs.

Data Structures:

Reuse the **Date, Time and template Vector** classes from the laboratory exercises. A template vector class, called **Vector** must be used and you must write your own minimal and complete **template** Vector class to store data in a linear structure (from Lab 5). For the purposes of the assignment, a Vector class is a *dynamic array encapsulated in a class called Vector*. Access to the private array is through the Vector's public methods. This Vector is **not** the same as the STL vector. The Vector for the assignment provides the same functionality as an array with controlled access. The Vector also contains only a few methods that are absolutely essential for the Vector. Nice to have functionality should not be implemented as methods. Such nice to have functionality can be provided by helper routines that are not methods (and not friends). As an example, in lab 4 you had the I/O operators that were routines that provided I/O convenience to the classes. Helper routines can also be functions and procedures that operate on the Vector class.

The Vector should allow for resizing. If more space is needed in the Vector than what is available, the Vector would increase its size. As required in lab 5, the client of Vector should not need to make this request to increase size.

To better understand the template requirement, you should complete the textbook chapter on "*Overloading and Templates*" and complete Lab 5.

Make sure you know the difference between a template class and STL class.

Make sure that the implementation of a method is separate from the method's prototype declaration (interface) in a class. This ensures that the implementation and the interface are separate. For template classes both interface and implementation will be in the same *.h (header)* file but in separate parts of the file. For non-template classes, the interface will be in *.h* files and the implementation will be in *.cpp* files.

In all cases, method interfaces (declarations) should have separate implementations outside the class declaration. The declarations will not have method code body. The code body will be outside the class declaration. If you need to revise this idea, go through the readings for topic 1.

Header files *(.h)* must be documented using doxygen style comments as shown in the file *ModelFile.h* in *doxyexample* folder from earlier topics. See *DoxyExample* from Topic 1. For convenience, a copy of ModelFile.h is provided with this assignment. Comments should be indented to the right so that method prototypes stand out from the comments. Follow the style in ModelFile.h.

As indicated earlier, you should design your classes so that they can be used in the future with different specifications of this assignment. See the Lab 5 exercise where you are asked to "future-proof" your design.

You should be careful that you do not have data structure classes that have I/O methods or friends. Completion of laboratory session 4 is also essential. If you have data structure classes that do I/O, aside from losing marks, you will have to do a lot more re-coding (i.e. a lot more work) when the I/O requirements change. You may want to have dedicated I/O classes instead or let the main program deal with I/O. Be mindful of the principles of cohesion and coupling as these concepts underlie some of the SOLID principles (https://en.wikipedia.org/wiki/SOLID) and GRASP https://en.wikipedia.org/wiki/GRASP_(object-oriented_design) or more detailed in this PDF file link (if interested).

STL data structures/algorithms cannot be used in this assignment.

You may use std::string and string stream classes in your program instead of using C like strings. You may use iostream and file handling classes and objects in C++. See laboratory exercises.

Any advice and further clarifications to these requirements would be found in the QandA file in the assignment 1 area. Questions and Answers (if any) would also be found in this file.

Notes:

A. The five program menu items listed above are part of assignment 1 requirements. There can be other requirements that you do not need to implement in this assignment but your design and implementation should be such that additional requirements could be added without major re-design and re-write of the code. As background information (not relevant to this assignment), research papers (see https://www.sciencedirect.com/science/article/pii/S1876610213000829, https://www.sciencedirect.com/science/article/pii/S0960148108001353) suggest that solar panel (PV) performance can be affected by temperature.

B. Solar radiation data is recorded in the text data file as W/m^2 (Watts per square meter). This is the amount of solar energy being measured per second over an area of one square meter. Or, in other words, the amount of power that is being detected over an area of one square meter. The actual meaning of the value is found here http://wwwmet.murdoch.edu.au/details. As the value recorded is the average W/m^2 over a 10-minute period, you need to convert this to kWh/m^2. This is done by converting the power in Watts (W) over a 10-minute period to Watts-hours. 10 minutes is 1/6 hour. So if the power is 120W for 10 minutes, this would equate to 120W x 1/6 hour = 20Wh. To convert Wh to kWh, divide this value by 1000. Thus you have 0.02 kWh. So $120W/m^2$ for 10 minutes is $0.02\ kWh/m^2$. You may want to view http://en.wikipedia.org/wiki/Kilowatt_hour for a further discussion if you are interested. Manually check calculations done by your code. If the output is wrong, your program would be considered as not working.

   We use our own test data files to test your program. You will not have access to our test data files and so it would not be possible for you to fake results. Our test data files will have the same format as that provided to you but the data will be different.

C. You will notice that the data also has solar radiation recorded at night. So to simplify the problem, *only solar radiation **values >= 100 W/m2** are to be used in your program*.

D. Examine the requirements carefully. Do you need to keep the data for each of the 10-minute readings or can you aggregate for the day or the month? For example, a cloud floating by would cause the solar reading to drop temporarily. Are these short-term changes relevant to working out solar radiation received from one month or year to the next? What if the requirements change? Think of what other information can be extracted from this data in the future. You will need to justify the approach you took.

E. You need to keep to the requested specifications. So calculating and presenting anything more (or less) than what is asked for violates the specifications and your work may get penalised. The specifications also require certain data structures to be used in the solution.

F. If you are interested in solar power, an easy starting point is http://en.wikipedia.org/wiki/Solar_power. Similarly, for wind power, you may want to start at http://en.wikipedia.org/wiki/Wind_power. Neither of these sites is needed for this assignment. It only serves to provide a background to the work you are doing.

G. Any advice and further clarifications to these requirements would be found in the QandA file in the assignment 1 area.

Submission requirements (for all students):
You must provide **all** of the following in LMS;

- UML design and Data Dictionary (diagrams should show high level **and** the detailed version)
  - Data Dictionary to accompany the UML diagram. Present this in the form of a table as shown in the lecture notes.
  - Written rationale for the design – answer "why" you did something in a particular way or why something is needed. "What it does", is written in the code comments and not in the rationale. Provide rationale for **each** method and attribute in your Vector class and any other class that you write. We would like to know why you designed something in a particular way – i.e. what is your thinking behind the design. *You do **not** have to provide a rationale for simple setters/getters. Add an extra column to the **Data Dictionary** shown in topic 4, Lecture 11. Label the column "Rationale".*
- Algorithm – so that a non C++ programmer can implement your approach. The algorithm should be understandable by a programmer who does not know C++ but may know Java, or some other programming language. If you like, you can use the algorithm writing style used in our reference book *Introduction to Algorithms by Cormen, Leiserson, Rivest and Stein*. One example is on page 18 of the third edition of the reference book. Alternatively, you may also decide to use the algorithm writing style from our ICT159 text *Simple Program Design by Robertson*. You should use meaningful names relevant to this assignment problem.
- Source code with doxygen style comments. All *.h* files should have doxygen comments as shown in *ModelFile.h.* Implementation files *(.cpp*) have normal code comments.
- Doxygen output (only html output) in a sub directory called "html".
- Program that builds in Code::Blocks and runs. All we would need to do to build your program is to load your Code::Blocks project file (.cbp) and select "build". Although we will use our own data file(s), you must still provide the data file you are using so that the program builds and works as submitted.
- Test plan
- Output of test run(s)
- A declaration indicating what works and what does not work in your program. This declaration should be provided as a separate document called "***evaluation.txt***". The declaration is a summary of your test plan and output of test runs. Test plan and output of test runs have a lot of detail and are separate documents. *The file evaluation.txt is only a summary – like an executive summary* – done as dot points.

*Printed versions of documents apply **only** when there is a notification in LMS asking for hard copies.* Do not print code. Code will only exist as soft copy.

Marking

All marking/feedback via LMS. There may be situations where feedback is done personally during class time.

UML diagram (High level and Low level)	10
Written rationale for the design with Data Dictionary	10
Non-programming language specific algorithm. *Marks not allocated if algorithm is word processed code*	10
Program that builds and works (includes coding, coding style including readability, doxygen comments, C++ classes). *Marks not allocated if program does not build or doxygen output is not provided.*	30
Non-STL Vector class implementation and usage. *Marks not allocated if STL data structures/algorithms used.*	30
Evaluation, Test plan and testing. *Marks only if evaluation.txt is also provided*.	10
Total	100