# Summative Coursework:
# Beth Yw? StatsWales Data Analysis Tool

Dr Martin Porcheron

You must submit your coursework by **26th March 2021** at **11am** to **both the Canvas assignment page for the Coursework and CS Autograder**. If there is submission to Canvas or CS Autograder after the deadline you will receive 0 marks for the coursework as per University policy. We will use the test outcomes for your last submission before the deadline to Autograder. Feedback for your final submission will be provided through Canvas.

CSC371 is piloting the use of the CS Autograder service to help you with your coursework as an *added benefit*. It has been used previously in Cryptography and IT-Security, but this is a much more resource-intensive pilot. We'll be monitoring the service throughout, but if we encounter issues, we may have to withdraw it. Because this is a pilot, you have been provided with test scripts to run your code locally as well. Despite being unlikely, the unexpected withdrawal of CS Autograder will not be grounds for extension—you can complete this coursework without relying on the service.

You can submit to CS Autograder a maximum of 10 times per day. Note that the results from CS Autograder may take a while to generate, especially if there are many simultaneous submissions, thus you are strongly discouraged from making last-minute changes and submissions just before the deadline. We will only take the last submission to CS Autograder, thus if you submit close to the deadline, your results may come after the deadline and may be lower if you broke something!

Please read this entire document through from start to finish before beginning to work on this assignment and before asking questions about the coursework. If anything is ambiguous or unclear, then you should seek clarification by posting in the Canvas group discussions for the coursework.

This coursework is worth 30% of your module mark.

## Unfair Practice Declaration

The university takes all cases of unfair practice seriously. The code you produce for this assignment **must be your own original work**. Do not submit any code copied from your peers, from the Internet, or from anyone else, as your own. Likewise, **do not share your code** with anyone, including sharing or posting **any portion** of it publicly or even in a private communication. Even if you do not intend for it to be copied by another student, this counts as *collusion* and is an academic misconduct offence.

**Reading** from documentation, tutorials, books, and online forums is acceptable, but copying code and passing this off as your own is an academic misconduct offence.

By submitting this coursework, you state that you fully understand and are complying with the University's policy on Academic Integrity and Academic Misconduct. The policy can be found at: https://myuni.swansea.ac.uk/academic-life/academic-misconduct.

Worksheet version: v1.1. Changelog:

- (v1.1) Task 8 has been clarified so that measures argument filtering is not respecified (Thanks Dominik Wojtasiewicz)

# About this coursework

As a software engineer you will often find yourself working within a team of developers on a common code base that everyone is contributing towards. The design and specification of what is being collectively built is designed via steering committee decisions or senior design architects, potentially in collaboration with clients. As a developer within one of potentially many sub-teams you only have remit to use the code as written by other sub-teams, and to produce code that strictly adheres to the design specification and is assigned for you to develop.

This coursework requires you to code according to a specification laid out in documentation and Behaviour Driven Development style (BDD) test suites. In this coursework, test suites are implemented using the Catch2 unit testing framework. When a test suite is compiled and executed it is a rich test application that provides a great deal of automated test information about how your code is executing and where it is not adhering to the specification.

## The Task: *Beth Yw? — StatsWales Data Analysis Tool*

You must develop a tool in C++ that can read CSV and JSON files, some of which have been downloaded from the StatsWales website and some of which have been compiled for you. StatsWales is website holding publicly-available data compiled by the Welsh Government. The datasets you have been provided all group series of *measures* over *years*, organised by *area*. For example, one measure is *population* for different *areas* in Wales over many *years*. Some of the files contain multiple measures, whereas other files only contain one measure. You will have to read in a series of program arguments, parse them, then import the appropriate dataset files, placing the data inside objects, before then printing this out to the standard output.

## Directory structure

| | |
|---|---|
| *`LICENCE` | Licences for the libraries included in the *lib_*.[cpp\|hpp]* files |
| `README.md` | A Markdown file you can use to document your solution |
| `bin/` | Directory where compiled binaries are stored by default |
| `build.bat` | A Windows Batch file for compiling both Beth Yw? and the test suite |
| `build.sh` | A Bash script for compiling both Beth Yw? and the test suite |
| *`datasets/` | Offline copies of the datasets your coursework must parse |
| *`datasets.h` | A file simulating dynamically-constructed metadata for the files in the *datasets/* directory |
| *`lib_*.[cpp\|hpp]` | **The only permitted external libraries** |
| ^`area.[cpp\|h]` | The `Area` class, for storing data and `Measure` objects for an area |
| ^`areas.[cpp\|h]` | The `Areas` class, for importing data and storing `Area` objects |
| ^`bethyw.[cpp\|h]` | Bootstrap code that deals with the command line arguments passed into the program, starts the import, and prints the output |
| ^`input.[cpp\|h]` | The `InputSource` and `InputFile` classes, for opening streams to dataset files |
| ^`measure.[cpp\|h]` | The `Measure` class, for storing values over years for a specific measure and area |
| *`main.cpp` | A simple file that calls the bootstrap code in *bethyw.cpp* |
| `tests/` | Directory containing a number of Catch2 BDD unit tests |

You must not modify any file/directories with an asterisk (*). You only *need* to modify the files with a circumflex (^). Please do not create an additional files as our test scripts will miss these.

## Automated build script and test suite

*build.sh* and *build.bat* will compile your application from the command line on your local machine. The former works with POSIX OSs, i.e., Unix (including macOS) and GNU/Linux (including virtualised installations such as those through Windows Subsystem for Linux). The latter script works with Windows Command Prompt. Both scripts build to C++14 standards and place the compiled binary in the *bin/* directory as *bethyw(.exe)*. You can alternatively use your own toolchain for compilation if you wish, however we will be testing your coursework with GCC on a POSIX system.

### Testing your code locally

You have been provided with a suite of test programs *tests/test1.cpp*, *tests/test2.cpp*, ..., *tests/test12.cpp*. You can compile your program with one of these test suites by calling the build script with a single argument `testN` where *N* is the number of the test you wish to compile (i.e., `./build.sh test1` will build your program with the test suite in *tests/test1.cpp*). The compiled binary will be saved as *bethyw-test(.exe)* in the *bin/* directory. Execute this binary to run the test. Compile your application with all tests with the command `./build.sh testall`. The same options also work for *build.bat*. You can use your own IDE or build your compilation process (e.g., using make), but we cannot realistically provide support for this.

You have also been provided 10 sample outputs for different commands in the *tests/* directory. The filename of each output file is *outputN.* (where *N* is a unique number) followed by a command for your application, followed by a file extension, e.g., *output5. bethyw -d popden -j.json* is output test 5, which contains JSON data that should be the same as your program's output when executing the command `bethyw -d popden -j`.

Studying these files along with the documentation in the provided source code should give you an insight into how your functions should be declared and implemented. The tests should cover most of the functions you need to write, but **they do not offer complete coverage**. You are welcome to write additional tests to aid in your development. Grading your code will be done by running your coursework against the original versions of these test programs plus some additional tests.

Although you should generally get the same output with whichever compiler you use, your code will be tested using the GNU Compiler Collection 10.2.

### Testing your code on CS Autograder

The CS Autograder service, which forms part of the submission process for this module, uses GCC 10.2. You can submit your coursework up to 10 times a day up to the deadline. The service will compile and run your code against the provided output tests and the unit tests. You will be able to see the results of these tests, allowing you to refine your submission. Due to the number of students on this module, it may take a while for test results to be generated. I would strongly encourage you to do most of your testing locally, and periodically test using CS Autograder. I would strongly discourage you from submitting to CS Autograder close to the deadline as demand will be high, and your results may arrive after the deadline.

As stated above, CS Autograder is a pilot service being offered by the department to help and support students with coursework. It is not a substitute for submitting your work to Canvas.

## Program arguments

Your program must read in a series of command line arguments (not `stdin`!) and parse them. A library has been included that will handle and convert these values to `std::string` or `std::vector` objects inside *bethyw.cpp*. You will have to write additional code in this file to fully implement the functionality described below. Note that some arguments will be redefined in a task later.

| | |
|---|---|
| `--dir arg` | The directory `arg` contains the datasets that will be used (default: *datasets/*). This argument is optional (i.e., if the program is run without the argument supplied, the value *datasets* is used). |

| | |
|---|---|
| `-d arg` or `--datasets arg` | The dataset(s) to import and analyse as a comma-separated list of codes. This argument is optional. If it not set, or it is set and contains the value `all`, then all datasets should be imported. Possible values can be found in *datasets.h* as the member variable `CODE` in the `InputFileSource` struct inside the `InputFiles` namespace. |
| `-a arg` or `--areas arg` | The areas(s) to import and analyse as a comma-separated list. Initially, `arg` should be implemented such that filtering is done using local authority codes, as found within the dataset files. This argument is optional. If it not set, or it is set and contains the value `all`, then all areas should be imported. |
| `-m arg` or `--measures arg` | The measure(s) from the dataset(s) to import and analyse as a comma-separated list. Initially, `arg` should be implemented such that filtering is done using the codenames for measures, as found within the dataset files. This argument is optional. If it not set, or it is set and contains the value `all`, then all measures from all imported datasets should be imported. |
| `-y arg` or `--years arg` | The years(s) from the dataset(s) to import. `arg` can either be of the form YYYY (e.g. `2012` to import data for the year 2012) or YYYY–ZZZZ (e.g. `2012–2015` to import the 4 years from `2012` to `2015`). This argument is optional. If it not set, or it is set and contains the value `0` or `0–0`, then all years should be imported. |
| `-j` or `--json` | Output the data as JSON instead of tables (see below). |

## Included datasets

A number of data files are stored in the *datasets/* directory and are declared in *datasets.h*. Do not replace or modify any of these files—your coursework will be marked with a fresh copy of them.

Read the code in *datasets.h* that provides details about these files, comparing this code to the explanation below. This file contains code that is not a particularly efficient way of storing this information (i.e., with maps storing data on the heap). The intention here was to create a file that resembles what might have been generated by some form of runtime computation of available datasets (like a future version of the system might have). For ease in this coursework, I have provided it as statically-defined data so you can inspect all the relevant information in code.

In *datasets.h*, each `InputFileSource` instance in the `InputFiles` namespace includes: the program argument value for filtering the dataset (using `-d`/`--datasets`), a human-readable name for the dataset, a filename, a `SourceDataType` enum value (used by functions in *areas.cpp*), and a map whose type is aliased as `SourceColumnMapping`, linking the enum `SourceColumn` (which includes a list of all possible relevant columns) to the string value of the column. Read the explanation of these enums in this file before proceeding.

### *areas.csv*

A list of areas in Wales, mapping the local authority code to a name in English and a name in Welsh. It should be handled by specific code for parsing this authority code CSV, and has three columns, one for the authority code (`AUTH_CODE`), one for the name in English (`AUTH_NAME_ENG`) and one for the name in Welsh (`AUTH_NAME_CYM`).

This is a special dataset that should always be imported irrespective of the `-d`/`--datasets` argument (but the areas imported can still be filtered by the `-a`/`--areas` argument).

### *popu1009.json*

The StatsWales JSON file for population density by local authority. This dataset file has the identifier `popden` (i.e. it should be included if the `-d`/`--datasets` argument is not supplied, contains

`all`, or contains `popden`). It is given the human-readable name of "Population density". It is a JSON file that should be handled by our function for parsing StatsWales JSON files. There are three measures in this file: *Population* (codename: `pop`), *Population Density* (codename: `dens`), and *Land Area* (codename: `area`).

StatsWales JSON files have three high-level key:value pairs, with the data we need stored in the value accessed by the key `value`. `value` itself is an array/list of rows in the dataset, organised as key:value pairs. There is much data here, but we only need the following pairs:

- The value of `Localauthority_Code` contains the local authority code, which is mapped to the `AUTH_CODE` `enum` column heading in *datasets.h*, and used in the `-a`/`--areas` program argument filter
- The value of `Localauthority_ItemName_ENG` contains the the local authority name in English, which is mapped to the `AUTH_NAME_ENG` `enum` column heading in *datasets.h*
- The value of `Measure_Code` contains an identifier for the measure, which is mapped to the `MEASURE_CODE` `enum` column heading in *datasets.h*, and used in the `-m`/`--measure` program argument filter
- The value of `Measure_ItemName_ENG` contains the English name for the measure, which is mapped to the `MEASURE_NAME` `enum` column heading in *datasets.h*
- The value of `Year_Code` contains a year for a particular value, which is mapped to the `YEAR` `enum` column heading in *datasets.h*, and used in the `-y`/`--years` program argument filter
- The value of `VALUE` contains the value for the given local authority, measure, and year, and is mapped to the `VALUE` `enum` column heading in *datasets.h*

Note that a bug on the StatsWales website truncates this JSON file and it misses many areas.

### *econ0080.json*

The StatsWales JSON file for active businesses by area and year, which has the identifier `biz`. There are 8 different measures in this file: *the number of active enterprises* (codename: `a`), *the number of newly opened enterprises* (codename: `b`), *the number of enterprises closing* (codename `d`), *the number of active enterprises per 10,000 of the population aged 16 to 64* (codename: `pa`), *the number of births per 10,000 of the population aged 16 to 64* (codename: `pb`), *the number of deaths per 10,000 of the population aged 16 to 64* (codename: `pd`), *the birth rate as a percentage of active enterprises* (codename: `rb`), and *the death rate as a percentage of active enterprises* (codename: `rd`). The column headings are defined in the same way as *popu1009.json* (see *datasets.h*).

### *envi0201.json*

The StatsWales JSON file for air quality indicators, by local authority, which has the identifier `aqi`. There are three measures in this file: *$NO_2$* readings (codename: `no2`), *PM10* readings (codename: `pm10`), and *PM2.5* readings (codename: `pm2-5`). The column headings are defined in the same way as *popu1009.json* (see *datasets.h*).

### *trans0152.json*

The StatsWales JSON file for rail passenger journeys by local authority and year, which has the identifier `trains`. This is a different type of dataset as there is only one measure in it. As a result, there are no `MEASURE_CODE` and `MEASURE_NAME` columns for this dataset. Instead, we use the following two hardcoded values in `SINGLE_MEASURE_CODE` and `SINGLE_MEASURE_NAME`. These two values do NOT map to columns in the JSON file, but rather are the *values* you should use as the measure code and name when saving the data to your `Measure` object.

Note that this file had incomplete local authority codes but I have fixed these for you. Also, the data was not collated by calendar year (e.g., years were listed as 2002-03, 2003-04, etc.). For the sake of simplicity in this coursework, I have changed them for you.

***complete-popu1009-[area|pop|popden].csv***

There are three CSV files corresponding to the three measures in *popu1009.json*, but with the missing data also added. They have the dataset identifiers `complete-popden`, `complete-pop`, and `complete-area`. These are only single-measure files, thus they have hardcoded measure code and measure name values in `SINGLE_MEASURE_CODE` and `SINGLE_MEASURE_NAME` respectively, as well as an `AUTH_CODE` column. There is no need for a `VALUE` in CSV.

These are manually modified versions of the CSV files generated by the StatsWales website (which produces invalid CSV files by default and does not include local authority codes). These CSV files may need a different parser to the areas CSV file above, thus have a different `SourceDataType` enum value (`AuthorityByYearCSV`).

## Expected output

The expectation is that a user can combine the above program arguments to query information and retrieve this either as tables in the terminal output or as JSON. Below are some examples. Block comments throughout the provided code detail the formatting of output too.

### Textual output

To enhance readability, try to align columns and line spacing as per the examples below. Note in the examples below that when you output as tables to the standard output, you need to calculate the mean for the presented data, the difference between the first and last year, and the percentage difference between the first and last year.

If we request data from the popden dataset, for the areas `W06000011` and `W06000010`, between `1990` and `1993`, the command is:

```
$ ./bin/bethyw -d popden -a W06000011,W06000010 -y 1990-1993
```

...and you should get the output:

```
Carmarthenshire / Sir Gaerfyrddin (W06000010)
Land area (area)
       1991        1992        1993     Average    Diff.  % Diff.
2370.276200 2370.276200 2370.276200 2370.276200 0.000000 0.000000

Population density (dens)
     1991      1992      1993   Average     Diff.   % Diff.
71.605579 71.411509 71.407290 71.474793 -0.198289 -0.276918

Population (pop)
         1991          1992          1993       Average       Diff.   % Diff.
169725.000000 169265.000000 169255.000000 169415.000000 -470.000000 -0.276919

Swansea / Abertawe (W06000011)
Land area (area)
      1991       1992       1993    Average    Diff.  % Diff.
377.596400 377.596400 377.596400 377.596400 0.000000 0.000000

Population density (dens)
      1991       1992       1993    Average     Diff.   % Diff.
608.435356 608.083128 607.391914 607.970133 -1.043442 -0.171496

Population (pop)
         1991          1992          1993       Average       Diff.   % Diff.
229743.000000 229610.000000 229349.000000 229567.333333 -394.000000 -0.171496
```

If we request data from the popden and `trains` datasets, for the area `W06000011`, with the `rail` and pop meausre, between 2015 and 2018, the command is:

```
$ ./bin/bethyw -d popden,trains -a W06000011 -m rail,pop -y 2015-2018
```

...and you should get the output:

```
Swansea / Abertawe (W06000011)
Population (pop)
```

```
         2015          2016          2017          2018       Average       Diff.  % Diff.
242316.000000 244462.000000 245480.000000 246466.000000 244681.000000 4150.000000 1.712640

Rail passenger journeys (rail)
         2015          2016          2017          2018       Average        Diff.  % Diff.
910878.000000 914448.000000 921736.000000 927841.000000 918725.750000 16963.000000 1.862269
```

#### JSON output

You must format your JSON output like so. The top level is a key:value mapping where keys are local authority codes. The values should be an object that contains two key:value pairs. One must have the key `names`, and contain an object of the various names (mapping three-letter language codes to area names). The other must have the key `measures` and contain the data.

For example:

```
$ ./bin/bethyw -d popden,trains -a W06000011 -m rail,pop -y 2015-2018 -j
```

...and you should get the output (ignore the whitespace/indentation):

```
{
   "W06000011":{
      "measures":{
         "pop":{
            "2015":242316.0,
            "2016":244462.0,
            "2017":245480.0,
            "2018":246466.0
         },
         "rail":{
            "2015":910878.0,
            "2016":914448.0,
            "2017":921736.0,
            "2018":927841.0
         }
      },
      "names":{
         "cym":"Abertawe",
         "eng":"Swansea"
      }
   }
}
```

### External libraries included

Three external libraries have been included in the coursework in files with a name starting with *lib_*. **These are the only external libraries you may use**. They are the Catch2 Unit testing framework, used for the test scripts in *tests/*; CXXOpts, used to assist parsing the command line arguments; and JSON for Modern C++, used to help you parse JSON files quickly and easily.

## Your tasks

Below is an abbreviated list of tasks you need to complete, designed to help you structure your progress. The source files provided to you include extensive comments in Javadoc style that explain each function, and the test suite includes tests for expected outputs. There are a number of expected exceptions that should be thrown. The test suite includes the exception class and message that you should throw.

Note, even if you do not complete all these tasks, it is still possible to score well on the coursework. I strongly encourage you to focus on producing good, clean, safe code for some of the tasks than sloppy code for all of them (the latter will potentially lead to a lower mark!).

## 1. Add your student number

Before you do anything else, you need to add your student number to the various files. In *bethyw.h*, edit the `const STUDENT_NUMBER` and in all the files above with the circumflex, edit the initial block comment, replacing `<STUDENT NUMBER>` with your student ID number.

## 2. Implement program argument parsing functions

You should implement the various functions in *bethyw.cpp* (and add the declarations to *bethyw.h*) to parse the program arguments correctly. Read through this file first, implement the functions, and then uncomment the following lines in *bethyw.cpp*:

```
auto datasetsToImport = BethYw::parseDatasetsArg(args);
auto areasFilter      = BethYw::parseAreasArg(args);
auto measuresFilter   = BethYw::parseMeasuresArg(args);
auto yearsFilter      = BethYw::parseYearsArg(args);
```

## 3. Implement `populateFromAuthorityCodeCSV` parsing

First, read through the comments in *input.h* and *input.cpp* and then declare and implement the `InputSource` and `InputFile` classes. Make sure you do this first and test that you correctly open files and retrieve a reference to a `std::istream`. *areas.csv* should always be imported, irrespective of the `datasets` program argument.

Next you will need to partially implement the `Areas` and `Area` classes in *area.cpp* and *areas.cpp*. Read through these files first and then start with the `Area` class, before the `Areas` class.

In *areas.h*, you will have to select an appropriate container for these `Area` objects to be stored within (modify the declaration of `AreasContainer` and the `Areas` class accordingly). The `Areas::populate()` and `Areas::populateFromAuthorityCodeCSV` functions are partially implemented for you. The former calls the latter when the `SourceDataType` passed to it is a `BethYw::AuthorityCodeCSV`. You will have to modify the implementation of these two functions to parse `areas.csv` and generate `Area` objects, which you then store inside your chosen nested container. In contrast to the lab assignments, I *strongly* recommend you use the C++ stream operators for parsing the file inputs (e.g., std::getline).

Update *bethyw.h* and *bethyw.cpp* by adding the `loadAreas(areas, dir, areasFilter)` function, and uncomment the code which calls this in `run()`.

## 4. Complete the `Measure` and `Area` classes

Next, implement the `Measure` class as well as the code to store instances of `Measure` objects inside `Area` objects. By this point, you should have implemented all the functions in the `Area` and `Measure` classes.

## 5. Implement output

Implement the << overload operator for the `Meausre`, `Area`, `Areas` classes and the `Areas::toJSON()` function such that your imported data can be exported as a terminal output and as a JSON string. Uncomment the code in `bethyw.cpp` to print these outputs.

## 6. Implement `WelshStatsJSON` parsing

Now you will need to declare and implement the `Areas::populateFromWelshStatsJSON` function. I have selected and included a very simple JSON library you should use. The block comment for this function explains and demonstrates how to use this with streams, but you can also simply check the website for the library (figuring out how to use libraries is a key part of programming). Work through the datasets one at a time, making sure your code works for each one. If when parsing a dataset, you encounter additional areas, you should import these.

When multiple datasets are imported, the imported data should be combined such that subsequent imported values replace any existing values stored inside objects (i.e. you will never have more than one `Areas` object, more than one `Area` object per area, or more than one `Measure` object per measure per area.

You will need to make sure the filters from the command line arguments work correctly with your functions such that areas can be filtered by local authority code, measures by codename, and years either with exact years or with a range value.

## 7. Implement `AuthorityByYearCSV` parsing

Declare and implement a function for parsing the final three CSV files. These include some of the data missing from *popu1009.json*. You must make sure you correctly add the data such that there are not duplicate *Measure* objects from different datasets.

## 8. Implement extended argument filtering

Adjust your filtering code so that rather than having to filter areas by the exact code value, we can also use partial values for both the codes and the names in Welsh or English. You should implement this in a case insensitive way. For example, `bethyw --areas swan` should print out all the data for all the areas that contain 'swan' in any part of their code or area name.

## Submission instructions

You must submit your code to both the Canvas assignment page for the Coursework **and** CS Autograder before the deadline. CS Autograder only accepts specific files and will ignore any unneeded files. For submission to Canvas, your should put your assignment in a directory named as `<student number>` where `<student number>` is your student number. You should then convert this directory to a `.zip` file for submission. The entirety of your coursework solution **must** be in this directory.

You **should** modify the README file in this directory to concisely explain any of the known caveats or issues with your implementation that you would like to be known during grading. You should include in this file the compiler you used.

## Grading criteria

The assignment is graded out of 100 marks:

**16 marks** are awarded for passing a series of tests based on the runtime outputs of your program against the specification given in this document and in the comments. You can test your code against 10 of the marks to the output files in the *tests/* directory.

**42 marks** are awarded for successfully passing automated unit tests. You have been provided with *some* of these tests in the directory *tests/*, equating to 20 marks.

**42 marks** are for good coding practice, and awarded in independently of the completeness marks above. You must produce clean, concise, and readable code. A non-exhaustive list of things that will you will lose marks for:

- Code that compiles with any warnings in GCC (e.g., using the `–pedantic` and `–Wall` flags). If you are not using GCC on your machine, you will be able see generated warnings on CS Autograder under *Output tests* → *Compilation completed without warnings*.
- Poor or inconsistent choice of variable names, or poorly formatted, non-indented, or incorrectly structured code
- Code that is inefficient/requires excessive computation
- Incorrect or missing usage of `const` keyword for parameter types, return types, and member function declarations
- Not throwing or catching exceptions as appropriate
  - You must not use the wildcard catch (i.e. `catch(...)`)
- Incorrect or unsafe use of pointers and references, or unsafe memory management
- Unneeded duplication of computation or use of redundant code or variables
- Producing code that is at-risk of throwing uncaught or unexpected exceptions
- Not including all functions as prototypes in header files
- Removal, reordering, or not using the block comments present in the provided files
  - These comments are as much to help you understand the specification as they are to help grade your work efficiently
  - It is infeasible to grade 180 assignments unless they are consistently formatted and each part of your implementation can be found predictably
- Excessive or non-existent use of commenting. Use comments to highlight code you have written, or code which may not have an obvious meaning. This can help the marker identify places to award marks. You should assume the marker knows C++, so do not need to comment obvious code.