# Summative Coursework: 371pass

Dr Martin Porcheron

You must submit your coursework by the due date listed on Canvas and CS Autograder at **11:59pm** to **the CS Autograder submission page** (not Canvas!). Feedback for your final submission will be provided by email to you university email address.

You can submit to Autograder a maximum of 10 times per day. Note that the results from 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 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 test that Autograder works for you now** by uploading the coursework framework. Some users occasionally encounter system errors with Autograder that seem to be caused by cookies. If this happens to you, try logging into Autograder in a private browsing/incognito window.

This coursework is worth 100 marks, and equates to 20% of your module grade.

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.

## 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.

## Changelog

**v1.0 — 28th Feb 2022** Coursework released

# About this coursework

As a software engineer you will often find yourself working within a team of developers, following a specification of what is being collectively built is designed via steering committee decisions or senior design architects, potentially in collaboration with clients.

## Learning objectives

This coursework is designed to simulate the sorts of things you may face in an industry job. Namely, you may have to complete code, using someone else's code as a starter. Unfortunately, decisions about how your program should work may well be made for you already. Someone else may have also selected which external libraries you can and cannot use (often corporations have sets of preferred tools). These are all realistic challenges that should not be overlooked. It can be challenging to code using someone else's framework, and using third party libraries will require you to read documentation and even experiment with different ideas to figure out how to use them correctly. Through completing this coursework, you should develop an appreciation of how coding in industry is likely to unfold.

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. As such, you will be expected to learn how to write code in C++ according to unit tests.

## The Task: 371pass

You will be implementing a simple application to store and manage personal data such as passwords, logins to websites, bank accounts, and credit cards. For this coursework, you will implement some basic functionality in C++ for your program to work over the command line. Your data will be saved to a single JSON file, which your program should also be able to read from.

## Directory structure

| | |
|---|---|
| `*LICENCE` | Licences for the framework and libraries included |
| `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 371pass and the test suite |
| `build.sh` | A Bash script for compiling both 371pass and the test suite |
| `^database.json` | A sample database you can use during development |
| `^src/371pass.[cpp\|h]` | Files that contain your main program logic |
| `^src/category.[cpp\|h]` | The `Category` class, for storing `Item` objects |
| `^src/item.[cpp\|h]` | The `Item` class, for storing entries as key/pair values |
| `*src/lib_∗.[cpp\|hpp]` | The only permitted external libraries |
| `^src/wallet.[cpp\|h]` | The `Wallet` class, for storing `Category` objects |
| `*src/main.cpp` | A simple file that calls the bootstrap code in *371pass.cpp* |
| `tests/` | Directory containing a number of Catch2 BDD unit tests and JSON files used in the 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/test9.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 *371pass-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.

Studying these files, along with the documentation in the provided source code and this PDF, 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 Autograder

The CS Autograder service, which is the main 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 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 discourage you from submitting to Autograder close to the deadline as demand will be high, and your results may arrive after the deadline.

## 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.

## Completing this coursework

Below is an abbreviated list of tasks you need to complete, designed to help you structure your progress. The source files provided include extensive comments that explain each function. The test suite is written such that it provides an explanation for all inputs and outputs for each test.

Note, even if you do not complete all the entire coursework, it is still possible to score very 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 *371pass.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. Examine the documentation fully

Don't try to simply jump into programming this coursework. Good software development requires developers to think through what they are going to program first. Read the guidance on how the program arguments should work and the actions that should be taken with each one. Read the comments and provided code in the various *.h* and *.cpp* files in the *src/* directory and think of how you are going to modify these files to accomplish this task.

## 3. Write just enough code to pass the first test

Next, look through the test scripts, understand what is happening. The test scripts are designed so that you can follow them through logically. Try compiling and running your code with the first test script (*tests/test1.cpp*) and observe the output. This test script compiles, but when you run it your test fails.

The first test relates to the action program argument and determines whether your implementation of the `parseActionArgument()` function in the App namespace (found in *371pass.h/cpp*) can convert the string representation passed in from the command line argument into a valid of the `Action` enum (*371pass.h* includes a description of this enum) or throws an exception.

Look at this function's existing implementation and modify it to pass the test. Once your test passes, think: are there other edge cases not covered by this test? Could this be done more efficiently? Is this the most elegant solution? If so, improve the correctness and efficiency of your code.

## 4. Passing the other tests

Once you have satisfied yourself that your implementation of `parseActionArgument()` is correct and done well, move on to the next test script. Remember, marks are awarded in this coursework for both correctness and style.

For the additional tests, the test may not even compile—this is because the test calls functions that are not yet implemented. You will need to read the test file to see which ones fail. Reading the test script is very useful in understanding how the function should work. In the various *.cpp* files in the *src/* directory, there are `TODO` comments that describe each function you must implement.

## 5. Write code not covered by the tests

Some functions do not have unit tests that cover them, and some functionality of your program may not be tested by the provided tests. Once you have passed the tests, continue to look through your code in the *src/* directory and ensure you have implemented every function and feature that has been requested.

## 6. Edit, edit, edit!

Every book you've read, every set of lecture slides you've glanced over, and every program's source code you've used have all been *edited*. Very few programs are perfect first time—look over your code, think through if you're doing things the right way. Read the documentation again and decide whether your code meets what is expected. Write additional tests, or try out different program arguments. For many of you, this will be one of your last pieces of software you write in your degree, and at this point, you should have an appreciation for programming as a craft.

## Submission instructions

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 marking.

You must submit your code to the CS Autograder submission page before the deadline. Autograder only accepts specific files and will ignore any unneeded files. Autograder does not accept directories or ZIP files—you must upload your .cpp and .h files and your README.md file.

**Please test that Autograder works for you now** by uploading the coursework framework. Some users occasionally encounter system errors with Autograder that seem to be caused by cookies. If this happens to you, try logging into Autograder in a private browsing/incognito window. Autograder is the only submission mechanism and you have ample time to test this before the deadline.

## Grading criteria

The assignment is graded out of 100 marks:

**30 marks** are awarded for successfully passing automated unit tests provided in the *tests/* directory. You can test your code against these tests on the 'My Submissions' on CS Autograder. The output of the Catch2 test is visible if you expand each test.

**30 marks** are awarded for successfully passing additional automated tests, which have not been provided. These tests include greater coverage of your program, and test for things described in the framework comments, but not necessarily tested in the provided test suite.

**40 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).
- 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
- 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 160 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.

You may gain marks for adding additional functionality to your programs beyond the specification, but be careful not to break any of the existing tests.