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 your 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 CS 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 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

v1.1 — 3rd Mar 2022 Update test2.cpp:
- test2.cpp was not expecting an exception when one was asked for in the comments (Thanks Samuel Roach)

v1.0.1 — 28th Feb 2022 Minor fixes to framework and worksheet:
- Clarify student ID only needs to be in editable .cpp/.h files
- Update Catch2 framework
- Remove erroneous semicolon from 371pass.h
- Fix building with build.bat (Windows only)

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 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 (this happened last year!).

Program arguments

Your program must read in a series of program/command line arguments (not `stdin`!) and parse them. A library has been included that will handle and convert these values to `std::string` objects inside *371pass.cpp*. You will have to write additional code in this file to fully implement the functionality described below. See the next section for examples of the various outputs and error messages your program should give in different situations.

`--database value` Filename of the 371pass database (the default value *database.json* is assumed if one is not provided)

`--action value` For this coursework, your program needs to be able to do four things: '*create*', '*read*', '*update*', or '*delete*' categories, items, or entries. Accordingly, these are the four possible values for the `action` program argument. I recommend you focus on implementing code for when the action argument is `read` first, followed by `create`, `delete` and `update`.

`--category value` Apply the action to a category, where `value` is the name of the category. Identifiers cannot be empty. The functionality of the `category` argument depends upon the value of the `action` argument:

- If the `action` program argument is *create*, then a category should be created in the database with the identifier (often shortened to 'ident') equal to the value of the `category` argument. If a category exists already with that identifier, do nothing. In both cases, your program should not output anything and return an exit code of `0`.
- If the `action` program argument is *read*, and a category identifier matches the value of the `category` argument, output the data in that category as JSON to `stdout` and exit with a code of `0`. If such a category does not exist, output an error message to `stderr` and return an exit code of 1.
- If the `action` program argument is *update*, and you want to rename a category, then the category argument should be in the format `oldidentifier`:`newidentifier` such that your program will rename the category with identifier `oldidentifier` to `newidentifier`. If successful, your program will output nothing and return an exit code of 0. If such a category does not exist, output an error message to `stderr` and return an exit code of 1.
- If the `action` program argument is *delete* and there is a category in the database with the identifier equal to the value of the `category` argument, delete it. If successful, your program should not output anything and return an exit code of `0`. If not successful, (i.e., no category exists), output an error message to `stderr` and return an exit code of 1.

`--item value` Apply the action to an item, where `value` is the identifier for the item. Identifiers cannot be empty. You must also supply the `category` argument when supplying the `item` argument (i.e., all items must be in a category). The functionality of the `category` argument depends upon the value of the `action` and `category` arguments:

- If the `action` program argument is *create*, then an item should be created in the database with the identifier equal to the value of the `item` argument in a category specified by the `category` argument. If an item exists already with that identifier, do nothing. In both cases, your program should not output anything and exit with a code of `0`.
- If the `action` program argument is *read*, and a item identifier matches the value of the `category` and `category` arguments, output the data in that item as JSON to `stdout` and exit with a code of `0`. If such an item does not exist, output an error message to `stderr` and return an exit code of 1.
- If the `action` program argument is *update*, and you want to rename an item, then the item argument should be in the format `oldidentifier:newidentifier` such that your program will rename the item with identifier `oldidentifier` to `newidentifier` in the category specified by the `category` argument. If successful, your program will output nothing and return an exit code of 0. If such an item does not exist, output an error message to `stderr` and return an exit code of 1.
- If the `action` program argument is *delete* and there is an item in the database with the identifier equal to the value of the `item` argument in the category specified by the `category` argument, delete it. If successful, your program should not output anything and return an exit code of `0`. If not successful, (i.e., no item exists), output an error message to `stderr` and return an exit code of 1.

`--entry value` Apply the action to an item, where `value` contains either the key or value or both for an entry (depending on the `action` argument). Keys cannot

be empty. You must also supply the `category` argument when supplying the `item` argument. The functionality of the `category` argument depends upon the value of the `action`, `category`, and `item` arguments:

- If the `action` program argument is *create*, then the entry argument should be in the format `name,Martin` such that an entry with the key `name` and value `Martin` be added to the item specified by the `item` argument in a category specified by the `category` argument. If such an entry exists with this key, update the value. If the entry argument contains no comma (`,`) then assume the entire argument value is the key and an empty value. In all cases, your program should not output anything and exit with a code of `0`.
- If the `action` program argument is *read*, and there is an entry in the database with the key equal to the value of the `entry` argument in the item and category specified by the `item` and `category` arguments, output the value of the entry (not in JSON format) and exit with a code of `0`. If such an entry does not exist, output an error message to `stderr` and return an exit code of `1`.
- If the `action` program argument is *update*, and you want to rename an entry, then the entry argument should be in the format `name:Person` such that your program will rename the entry with key `name` to `Person` in the item and category specified by the `item` and `category` arguments. If you wish to update the value of an entry, the entry argument should be in the format `name,Albert` such that your program will update the entry with key `name` to have value `Albert`. If you wish to update the key and value of an entry, then the `entry` argument should be in the format `name:Person,Albert` such that your program will update the entry with key `name` to have key `Person` and value `Albert`. If successful, your program will output nothing and return an exit code of 0. If such an entry does not exist, output an error message to `stderr` and return an exit code of `1`.
- If the `action` program argument is *delete* and there is an entry in the database with the key equal to the value of the `entry` argument in the item and category specified by the `item` and `category` arguments, delete it. If successful, your program should not output anything and return an exit code of `0`. If not successful, (i.e., no item exists), output an error message to `stderr` and return an exit code of `1`.

–h or ––help — Output the program help information. This command has been implemented for you.

Note that as a result of the above, category and item identifiers cannot contain a colon (`:`) and entries cannot contain a colon (`:`) or a comma (`,`) in their key or value. This is simply a design limitiation and you should not worry about it. Your code will not be tested in this situations.

Expected outputs

Below are a number of sample commands and outputs expected using the sample `database.json` file provided in the coursework framework. Ignore line breaks in the terminal output. Your program will be tested with similar commands to the below using a semi-automated process. Although these outputs are not tested by the unit tests in the *tests/* directory, you are expected to ensure your program has outputs as below.

`$ ./bin/371pass`

stdout: *empty*

stderr:
```
Error: missing action argument(s).
```

exit: 1

$ ./bin/371pass --action 123

stdout: *empty*

stderr:
```
Error: invalid action argument(s).
```

exit: 1

$ ./bin/371pass --action read

stdout:
```
{"Bank Accounts":{"Starling":{"Account Number":"12345678","Name":"Mr John Doe",
"Sort Code":"12-34-56"}},"Websites":{"Facebook":{"password":"pass1234fb",
"url":"https://www.facebook.com/","username":"example@outlook.com"},"Google":
{"password":"pass1234","url":"https://www.google.com/","username":
"example@outlook.com"},"Twitter":{"password":"r43rfsffdsfdsf","url":
"https://www.twitter.com/","username":"example@outlook.com"}}}
```

stderr: *empty*

exit: *0*

$ ./bin/371pass --action read --category "Bank Accounts"

stdout:
```
{"Starling":{"Account Number":"12345678","Name":"Mr John Doe","Sort Code":"12-34-56"}}
```

stderr: *empty*

exit: *0*

$ ./bin/371pass --action read --category "Bank Accounts" --item "Starling"

stdout:
```
{"Account Number":"12345678","Name":"Mr John Doe","Sort Code":"12-34-56"}
```

stderr: *empty*

exit: *0*

**$ ./bin/371pass --action read --category "Bank Accounts" --item "Starling"
--entry "Account Number"**

stdout:
```
12345678
```

stderr: *empty*

exit: *0*

**$ ./bin/371pass --action read --category "Bank Accounts" --item "Starling"
--entry "Account"**

stdout: *empty*

stderr:
```
Error: invalid entry argument(s).
```

exit: 1

**$ ./bin/371pass --action read --category "Banks" --item "Sterling"
--entry "Account"**

stdout: *empty*

stderr:
```
Error: invalid category argument(s).
```

exit: 1

$ ./bin/371pass --action read --entry "Account"

stdout: *empty*

stderr:
```
Error: missing category argument(s).
```

exit: 1

$ ./bin/371pass --action read --item "Starling" --entry "Account"

stdout: *empty*

stderr:
```
Error: missing category argument(s).
```

exit: 1

`$ ./bin/371pass --action read --category "Bank Accounts" --entry "Account"`

stdout: *empty*

stderr: `Error: missing item argument(s).`

exit: 1

`$ ./bin/371pass --action create`

stdout: *empty*

stderr: `Error: missing category, item or entry argument(s).`

exit: 1

`$ ./bin/371pass --action create --category "Banks" --item "Sterling"`
`--entry "Account,9876543"`

stdout: *empty*

stderr: *empty*

exit: *0*

`$ ./bin/371pass --action update --category "Banks" --item "Sterling"`
`--entry "Account,123456"`

stdout: *empty*

stderr: *empty*

exit: *0*

`$ ./bin/371pass --action delete --category "Banks"`

stdout: *empty*

stderr: *empty*

exit: *0*

`$ ./bin/371pass --action test`

stdout: *empty*

stderr: `Error: invalid action argument(s).`

exit: 1

`$ ./bin/371pass --action update --category "Bank Accounts" --item "Starling"`
`--entry "Account Number:"`

stdout: *empty*

stderr: `Error: invalid entry argument(s).`

exit: 1

These outputs should given you an idea of the expected outputs across the program, but if you have questions, please ask on the Canvas discussion board for the coursework.

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 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 .cpp/.h 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 the 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.

All submissions to Autograder count as coursework submissions, thus for each submission, you must comply with the University's policy on Academic Integrity and Academic Misconduct (even if you later replace this submission with a new version).

Grading criteria

The assignment is graded out of 100 marks:

20 marks are awarded for successfully passing automated unit tests provided in the *tests/* directory. You can see the result of 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.

15 marks are awarded for successfully passing automated output tests provided through CS Autograder. You can see the result of your code against these tests on the 'My Submissions' on CS Autograder. The output of your program and the expected output is visible if you expand each test. Due to a limitation in the design of autograder, marks for automated output tests are not shown (each output test is 1 mark).

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.

35 marks are for good coding practice, and awarded in independently of the completeness marks above. You must produce clean, concise, and readable code. These marks are awarded in a negative marking approach (i.e., you lose marks for bad coding practice) as by now, you should always be working towards good coding standards. 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 (e.g., shadowing variables), or poorly formatted, non-indented, or incorrectly structured code
- 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.

- Code that is inefficient/requires excessive computation including using '`using namespace std`'
- Incorrect or missing usage of `const` keyword for parameter types, return types, and member function declarations
  - Or using a cast to remove a qualifier such as `const`
- 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

You may gain marks for designing your implementation well, making good use of the Standard Library, and adding additional functionality to your programs beyond the specification, but be careful not to break any of the existing tests.