COMP26020 – Assignment 2
Processor Emulator in C++

A. Assignment

The goal of this assignment is to implement a set of C++ classes that emulate the operation of a very simple processor. The processor has a single 8-bit general purpose register, one 8-bit Program Counter, an address space with 256 bytes, and it can execute eight different instruction types. The implementation is composed of five files:

- **`instructions.cpp`** and **`instructions.h`** contain a class hierarchy of processor instructions, including how they are created, how they are represented as strings, and, most importantly, how executing them modifies the processor state.
- **`emulator.cpp`** and **`emulator.h`** describe the emulator itself: how the processor state is initialiased or loaded from a file, how instructions are executed, how breakpoints are managed.
- **`common.h`** contains header information that is needed by both the emulator and the instructions.

Additionally, we provide four more files that are used only for testing:

- **`catch.cpp`** and **`catch.hpp`** come from the Catch2 testing suite and allows us to do unit testing.
- **`structural-tests.cpp`** checks whether the public interface of the classes is correct.
- **`functional-tests.cpp`** checks whether the emulator and instruction classes work as intended

The provided implementation is 100% complete and correct.

Wait. What?

Well, the code is correct only in the sense that it produces the expected outputs and it passes all functional tests. It has memory errors, most of the code is very unsafe, it's very verbose, and it does not follow the conventions and guidelines around C++ code. It is the kind of code that a C programmer who just started learning C++ might write. ***This is not good C++ code***.

Your **actual goal** is to rewrite the five implementation files (**`emulator.{cpp,h}`**, **`instructions.{cpp,h}`** and **`common.h`**) using the concepts and approaches we covered in the lectures. This includes:

- RAII
- new/delete considered harmful
- C-arrays, C-strings considered redundant
- C++ standard library containers, utilities, and algorithms
- Improved type safety
- Using linters, static analyzers, and the C++ Core Guidelines to uncover errors and bad coding patterns

Your rewrite **should not materially change the externally visible behaviour of the code**: **any** possible external function calling **any** public member function should behave (mostly) the same way it did before the rewrite. In practice this means that:

a) You are allowed to edit only the five implementation files.
b) You are not allowed to remove existing public member functions (unless the comments say you can).
c) You are not allowed to change the visibility of existing public member functions.
d) You are significantly limited in how you can change the argument types or the return value types of public member functions (**but see the clarification below**).
e) Your public member functions need to return either the exact same thing they would have returned originally or something else that is functionally and semantically equivalent.

On the other hand, **you are free to**:

a) Add new member functions (private, protected, or public) (**with one exception, see below**).
b) Change the internal implementation of any member function.
c) Add/remove private member variables or change their types.
d) Remove any of the special functions that the compiler can auto-generate, assuming that the compiler versions of these functions will be correct (this will not change the public interface).

Changing argument/return types: Changes are allowed when the original and new types are a) implicitly convertible and/or b) syntactically interchangeable. For example, an integer with allowed values 0/1 can be replaced by a bool, because statements like `if(var)` or `if (var1 == var2)` are equally correct whether the variables contain 0/1 or false/true. Similarly, replacing an argument of `type1` with an argument of `type2` is okay if `type2` can be constructed from a `type1` object. The original implementation expects `type1`, so the testing code passes a `type1` object. If your code expects `type2` instead, the compiler will attempt to construct such an object from the argument we passed. If it's constructible, then there will be no error.

The testing code is designed to be as type agnostic as practically possible, so most meaningful type substitutions will be accepted. **If you are uncertain, use the structural tests**: they are designed to identify early whether type changes in function prototypes are valid. If the structural tests says no, then the type substitution is not allowed. If the structural tests says yes, then the type substitution is likely to be valid. If all the functional tests pass, the substitution is definitely valid.

Adding member functions: While adding member functions is allowed, **overloading member functions that were not originally overloaded is not allowed by the tests**. This is an artificial restriction. Normally this is a perfectly acceptable. But in the context of this assignment, it is complicated to apply structural tests on overloaded functions, so we only support overloading for functions that were originally overloaded.

B. How to approach the assignment

As is, the code is extremely unsafe. Try to use (modern) C++ capabilities that attack that problem first. If you do it correctly, you will be able to remove large parts of the code and eliminate hundreds of warnings.

Do not start by focusing on using tools like clang-tidy or the Address Sanitizer. Tools often focus on symptoms instead of the underlying problem: use of unsafe code. As such, clang-tidy reports ~120 C++ Core Guidelines violations in the original code, but most of these violations are associated with only a small number of unsafe code patterns. Fixing individual violations will be tedious and error prone. Fixing unsafe code patterns, instead, will be faster and it will eliminate most guidelines violations and memory errors. After you've gone through the obvious fixes, using tools to identify some of the few remaining issues will be helpful.

A good first step is to eliminate C-strings. They're often associated with mallocs and manual memory management. Even when they're not, they are tedious and verbose to use. Replacing them with something more modern and safe will have a massive effect on the quality of your code.

Having done that, the compiler now complains about all the **various printing and scanning functions** that expect C-strings. Why not replace them too with something more modern and appropriate?

After that, attack the other two big problems we have discussed in the lectures: **new/delete and C-arrays**. Like with strings, eliminating them from your code will make it more safe, concise, and clear.

Look out for **bits of code that can be safely removed or replaced with simpler code**. If you can do that, do it. Remember the "Rule of Zero".

Prioritise correctness. Submitting changes that fail to pass the functional tests is really bad, both in the context of this assignment and in the real world. At best, it indicates that you don't know how to apply these changes correctly. At worst, it indicates that you did not apply these changes yourself. You will get a much better grade submitting a 100% correct solution that failed to use some modern C++ capability, than submitting code that uses that capability incorrectly.

Test your code after every single set of changes you make. Does it still compile? Does it still pass the tests? If you made a mistake, you want to know as soon as possible, not after another three rounds of edits.

C. Building and testing

The following subsections assume that you are using one of Linux workstations in the lab. If not, check Appendix A: Workflow options.

TD;LR:

Execute this once from the assignment's base directory:

```
cmake -B build -DCMAKE_CXX_COMPILER=g++-12
```

which should produce something like this:

```
-- The CXX compiler identification is GNU 12.3.0
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: <some-path>/build
```

To (re-) build and run the tests execute:

```
cmake --build build/ --target tests
```

which should produce something like:

```
[ 10%] Building CXX object CMakeFiles/catch.dir/catch.cpp.o
[ 20%] Linking CXX static library libcatch.a
[ 20%] Built target catch
[ 30%] Building CXX object CMakeFiles/emulator.dir/emulator.cpp.o
[ 40%] Building CXX object CMakeFiles/emulator.dir/instructions.cpp.o
[ 50%] Linking CXX static library libemulator.a
[ 50%] Built target emulator
[ 60%] Building CXX object CMakeFiles/functional-tests.dir/functional-tests.cpp.o
[ 70%] Linking CXX executable functional-tests
[ 70%] Built target functional-tests
[ 80%] Building CXX object CMakeFiles/structural-tests.dir/structural-tests.cpp.o
[ 90%] Linking CXX executable structural-tests
[ 90%] Built target structural-tests
[100%] Trying the tests
Randomness seeded to: 3007795231
===============================================================================
All tests passed (125 assertions in 16 test cases)

Randomness seeded to: 824987971
===============================================================================
All tests passed (41312 assertions in 63 test cases)

[100%] Built target tests
```

If you want to re-built and re-run the tests you don't have to issue the first cmake command, only:

```
cmake --build build/ --target tests
```

Test Suites

To run and test your implementation you are given two test suites, one structural and one functional. Each produces a C++ program (i.e. it contains a main function) that includes your implementation.

The **structural tests** use template meta-programming to statically check whether the argument types and the return types of your member functions agree with what the functional tests expect. If there is a problem, in almost all cases, the tests will fail to compile. If you cannot pass the structural tests, there is no point trying the functional tests: they will either fail to compile, or they will do something very different than what they should.

Assuming that your code is structurally valid, the **functional tests** instantiate emulator and instruction objects, call various member functions on them, and check whether they produce the right outputs for certain inputs.

You *could (but don't if you have cmake)* compile the programs manually:

```
g++-12 -std=c++20 catch.cpp structural-tests.cpp instructions.cpp emulator.cpp -o
build/structural-tests
g++-12 -std=c++20 catch.cpp functional-tests.cpp instructions.cpp emulator.cpp -o
build/functional-tests
```

and execute the test suites with:

```
./build/structural-tests
./build/functional-tests
```

The structural tests should produce this output:

```
Randomness seeded to: <some random number>

===============================================================================

All tests passed (125 assertions in 16 test cases)
```

While the functional tests should produce this:

```
Randomness seeded to: <some random number>

===============================================================================

All tests passed (41312 assertions in 63 test cases)
```

Unless you have done something really really questionable, **passing the tests means that your code is functionally correct.** It is possible to pass the tests with an invalid solution, for example if your code is handcrafted to only satisfy the tests. We will test your code using the same testing code, but different inputs to guard against such invalid solutions. Solutions that pass the released inputs but not the marking inputs will be checked manually.

More information about how the tests, including how Catch2 works, can be found in Appendix B: Catch2.

CMake

Instead of building and running the tests manually, we provide a **CMake** project that automates and accelerates these two tasks. For more details, check Appendix C: CMake. Below is the basic information you need to know.

CMake prefers building "out-of-tree". That means it builds in some folder separate from the code. In the following examples, we're using **`build/`** but you can use whichever folder name you want.

To tell CMake to prepare the `build/` folder for building, you execute this **from the assignment's base directory**:

```
cmake -B build/ -DCMAKE_CXX_COMPILER=g++-12
```

You only have to execute this once (unless you move your build/ folder between different systems – see "Using Multiple Systems" in 7. Using multiple systems)

To build and run the tests you then execute:

```
cmake --build build/ --target tests
```

To run clang-tidy on your code, you can execute:

```
cmake --build build/ --target tidy
```

We suggest you use the Address Sanitizer[1] (ASan), a compiler plug-in from Google that discovers memory errors. ASan is built-in in gcc and clang and enabled with the compiler flag "`-fanalyze=sanitizer`". The following pseudo-target, automatically applies ASan on the functional tests and runs them:

```
cmake --build build/ --target sanitized
```

After the end of the functional tests, ASan will print a list of all code locations with memory errors.

If you want to see exactly what these cmake commands are doing, append **`-v`** to the command line.

All these commands indirectly invoke the underlying build system. On the lab workstations, this means Make. Instead of going through CMake, you could choose to invoke Make directly. **From inside the build directory**, you just execute:

```
make <target-name> # e.g. make tests, make tidy, or make sanitized
```

D. Submission

Deliverables, Submission & Deadline

There are five deliverables: the completed `emulator.{cpp,h}`, `instructions.{cpp,h}`, and `common.h` files. The submission is made through the CS Department's Gitlab. You should have a fork of the repository named "*26020-lab2-S-CPlusPlus_<your username>*". **Make sure you push to that precise repository and not another one**, failure to do so may result in the loss of some/all points. Submit your deliverables by pushing the corresponding files on the master branch and creating a tag named **`lab2-submission`** to indicate that the submission is ready to be marked.

The deadline for this assignment is 29/11/2024 6pm UTC.

Submission Checklist

1. Does your submission compile successfully **on the lab machines**?
2. Does your code **pass all tests**?
3. Have you **committed all local changes**? Check using git status.
4. Have you pushed your changes to the **right repository**? Go to your repo's web interface and check the list of commits.
5. Have you tagged your **latest** commit with "**lab2-submission**"?

Marking Scheme

The exercise will be marked out of 10. Marks are awarded for implementing the changes described in the first section of this document. Given that the starting point of this assignment is a codebase that already compiles and passes all tests, we treat that as the marking baseline: marks are awarded only for improving the code beyond that baseline, not for just passing the structural and functional tests.

Failing the tests however means that your code is wrong and you could not fix it. On top of that, this makes it harder/impossible to apply some of the marking tools on your submission. As such, test errors are penalised.

If your submission fails to compile or run successfully, we will fix minor issues that could have been caused by you not testing your code on the lab workstations, for example missing included headers. We might fix other minor bugs, but it's up to the individual marker to decide what "minor bug" means. Bugs that require changing multiple lines of code are definitely not minor. In all cases where we had to fix your code, an appropriate penalty will be applied. **So, check that your code compiles and runs on the lab machines before you submit.**

Below is the provisional marking scheme (which might change slightly before marking begins):

1 https://github.com/google/sanitizers/wiki/AddressSanitizer

Level of correctness → How much the code quality marks are scaled

- Tests do not build (code is syntactically wrong, internally inconsistent, or implements the wrong public interface) → 0.0x
- Tests build but do not run successfully (never terminate or terminate with a segmentation fault) → 0.25x
- The tests run but there are three or more failing test cases → 0.5x
- The tests run but there are at most 2 failing test cases → 0.75x
- The program passes all tests, but with some compilation warnings → 0.9x
- The program passes all tests with no warnings → 1x

Quality criteria → Awarded marks

- Asan does not report any memory errors (leaks, use-after-delete, out-of-bounds accesses, etc) → 2
- The code follows RAII principles → 2
- The code uses Standard Library containers and data types whenever appropriate → 2
- The code takes advantage of C++ capabilities to make the code clearer and more concise → 1
- The library code uses Standard library algorithms whenever there is a clear benefit from doing so → 1
- The library code follows the C++ Core Guidelines (the ones that we discussed in the lectures, as well as the ones tested by clang-tidy) → 1
- The code is clear, well commented, and follows good practices → 1 (This will be interpreted generously, unless we are forced to carefully read your code)

Using AI

You are not allowed to use code generated by AI. This is considered plagiarism. Also, research has consistently shown that using AI stops students from actually learning, which translates to worse outcomes in the exams or in their future careers.

Similarly to plagiarism, this applies not only to code that was copy-pasted but to any kind of code that you "just followed closely" to write your solution.

Students often claim to be using AI for "ideas" or for explaining something about the code. Current AI is a statistical tool which by definition will be often wrong, especially when dealing with tasks outside the ones in its training set. In the context of this assignment, AI hasn't seen similar exercises, previous solutions, or even the material we teach. AI might tell you something correct, but you have absolutely no guarantee that this is actually correct, which defeats the purpose of asking AI for help. You are paying good money to have access to actual experts that can give you personalised advice and help, so why use AI instead?

Plagiarism/Collusion

This is an individual coursework exercise. You should not share your code with others, you should not try to see or copy the source of anyone else, and you should not work together with other people. Exchanging ideas can also be collusion[2] and it will definitely treated as such if the submissions are unreasonably similar.

All submissions will be checked using a standard code plagiarism detection program, as well as a checker tailored to this assignment. We will manually examine any submissions that are flagged. If their similarities are because they contain the code we gave you or because they applied the same obvious changes, then there will be no further action. If this is not the case and their similarities are extremely unlikely to be due to chance, they will either be penalised or forwarded to a disciplinary panel.

CS Guidance UoM-wide Guidance Academic Malpractice Procedure

2 Academic Malpractice Procedure – 3.1.2.3:"The methods of collusion may include, but are not limited to, sharing of work, ideas or plans by social media or other electronic communication means, and/or physical sharing of work, ideas or plans."

E. FAQ (Check blackboard for updates)

0. I have the X problem on my Y laptop	If you are not confident that you can fix your problems easily, just connect to the lab workstations and work there. This is the point of lab workstations to begin with: a single hardware/software configuration where the code is guaranteed to run and you can be confident that we will get the same results you get when we test your code.
1. command cmake not found	You don't have cmake installed on your system or it's not in your PATH. Either install it or add it in your PATH.
2. I get something like "error: 'concept' does not name a type" when compiling the tests	Your compiler does not fully support C++20. Install a more recent compiler or point your cmake to another installed compiler using the configuration flag "-DCMAKE_CXX_COMPILER=<compiler path>"
3. I get "TIDY-NOTFOUND: not found" when running the tidy target	CMake cannot find a clang-tidy. Either install one or don't use this target. If you install one, delete the build directory and set it up again.
4. Do I need to use clang-tidy and ASan? Can't figure out how to use them on my system.	No, you can identify all code issues without using the two tools. But why make your life difficult? Just use the lab workstations.
5. Do I need to eliminate all clang-tidy warnings?	Ideally yes, except for the various suppressed warnings. These are in library code, so they are beyond the scope of this assignment.
6. If I eliminate all guideline warnings, will I get the full mark?	Not necessarily. clang-tidy does not enforce all guidelines, so there will be issues with your code that it will not complain about.
7. How will I know that I have fixed the code? What does the perfect solution look like?	There is no way of telling you that without giving you an exhaustive list of what you need to do. But if you attack the big problems that we kept talking about in the lectures (i.e., "new considered harmful", RAII, "C-arrays considered redundant", avoiding explicit bounds checking), you should already be close to a perfect solution.
8. Do I need to fix every single issue to get a 10?	No. You need to fix all major issues and most of the minor issues. Missing a few minor things will probably have no effect on your grade. Missing a few more minor issues will only have a small effect.
9. Can I modify the argument types of any member functions?	Partially. Substitutions are usually okay if the new type is constructible from objects of the old type or if both types are syntactically interchangeable. The testing code is doing its best to handle substitutions gracefully, but there are limits. The structural tests are designed to fail early and clearly if you make an illegal substitution. If you want to change how you use a function internally but you cannot change its arguments, just create a new function with your preferred argument types and use that instead.
10. Can I mix raw and smart pointers?	Yes. Smart pointers are meant to be used only for owning data. Replacing non-owning raw pointers with smart pointers is usually wrong.
11. Can we add a destructor for class X, even if it was not originally declared? Can we remove the destructor for class Y, even if was originally declared? Can we remove a copy/move constructor/assignment operator?	Yes, yes, and yes. The class normally has the set of five special methods whether you declare them explicitly or not (unless you set them to deleted of course). Assuming you don't do anything that affects visibility, declaring or removing an explicit constructor/destructor has no effect on the public interface. The constructors and destructors will be there regardless of what you do. The only change is whether the special methods are explicit or not. If there is a problem, the structural tests will tell you so early.
12. Can we remove/modify/add private variables?	Yes. The testing code cannot see private variables or functions, so any changes you make cannot affect the tests. But it might affect the quality of your solution.
13. The test suite reports something like: functional-tests.cpp:201: FAILED: REQUIRE(some_fnct()) with expansion: false	REQUIRE checks whether whatever is in the following () evaluates to true or 1. The message indicates that some_fnct() returns the wrong value. Try to figure out why.

14. Can I change the arguments/return values/methods to const?	Definitely not the return values. Const return values place a constraint *on the calling code*: my code is not allowed to modify the values. If this code depends on them being modifiable, even if they are not actually modified, there will be a compiler error. Modifying arguments and methods should be okay. Const arguments place a constraint *on your code*, so there is not effect on the testing code.
15. Can I change an argument from pointer to reference?	NO! Pointers and references have similar properties but they are not syntactically interchangeable. If the testing code expects a pointer argument and your code expects a reference argument, the compiler will just refuse to compile it.
16. Can I merge all code in a single file?	NO! This will break CMake and modular compilation in general, i.e. your code will not compile when tested. If this wasn't a correctness problem, the marking scheme would explicitly penalise that kind of awful coding practice.
17. Can we use standard library header X/Y/Z?	You are allowed to use any standard library header you want, as long as that header exists in C++ 20. It's not a great idea to use some of these headers, but it will not break the testing code.
18. "No space left on device"! What happened?	You've ran out of space. If you are using a lab machine, you don't have unlimited space in your home folder. Even if you *think* you've barely used that folder, believe me, it's full. Use the following command, from your home folder, to figure out what's using all your space: `find . -mindepth 1 -maxdepth 1 -type d -print0 | xargs -0 -n1 du -hs` This will list all subfolders of your home including the space they occupy. If you see a folder that's using a lot of space but it shouldn't or you don't need it any more, delete it and try again. Alternatively go into that folder and execute the above command again to find the specific subsubfolder that's causing the problem.
19. My tagged commit did not modify any of the five files of my submission, but an earlier commit did. Will this cause a problem?	No, it will be fine. Our marking code will clone your whole repo at the point in time you tagged it, not just the files you committed when you tagged it.
20. What is uint8_t?	It's an integer type that is guaranteed to be 8 bits wide and unsigned. It's somewhat similar to unsigned char (but read below).
21. Can I replace uint8_t with unsigned char?	While the two types are in most aspects the same (8-bit unsigned integers), using the word "char" might send the wrong signal. Whenever I am using uint8_t, the variable is not meant to hold characters. And an array of uint8_t is not meant to hold a string. uint8_t makes this clear to other developers (unfortunately not the compiler though). So don't replace it.

F. Appendices

Appendix A: Workflow options

0. Summary

	Similarity to the marking environment	Does not requires configuration	Support	Convenience
Lab workstations (locally)	✓✓✓	✓✓✓	✓✓✓	✗✗
Lab workstations (remotely)	✓✓✓	✓✓✓	✓✓✓	✗
Your own Ubuntu	✓✓	✓	✓	✓ or ✓✓✓ (depending on whether you already have it)
Your favourite development environment	✓ to ✗✗✗	✗ to ✗✗✗	✗✗✗	✓✓✓

1. Editing and building

We suggest you avoid proper IDEs and use some kind of source editor instead. Anything between vim and VSCode should be okay, assuming you don't go overboard with complicated plugins. We have two main reasons for that:

1. Configuring an IDE correctly and consistently with our testing configuration is a time-consuming task
2. The complexity of the things you need to do is not high enough for an IDE to be helpful

Basically, we believe that using an IDE will waste your time.

Regardless of the editor you choose, we suggest you build and run the tests using the command line. This requires zero configuration (apart from perhaps installing CMake that you need regardless of your workflow) and produces consistent results across multiple hardware and software configurations.

2. Using lab workstations locally

The instructions above and the various support tools (e.g. CMake) are guaranteed to work on any of our lab's Linux workstations. Using a lab machine means that you don't have to worry about technical issues and system configuration. Everything works directly out of the box. Also, this is where we will test your code while marking. **If your code compiles and runs correctly on a lab machine, it will run correctly when we test it**.

3. Using lab workstations remotely

Same advantages as above, but you can connect to them from anywhere in the university's network. You will need an ssh client, such as openssh (Linux) or PuTTY (Windows). To connect set the remote hostname to one in the range **`e-10cki18001.it.manchester.ac.uk`** to **`e-10cki18060.it.manchester.ac.uk`**. For example in Linux you can do:

```
ssh -Y <username>@e-10cki18012.it.manchester.ac.uk
```

If the command says that it cannot find a route to the host, try a different hostname in the valid range.

4. Using your own Ubuntu Installation (VM or not)

For Ubuntu 24.04 (preferred) install a few packages using apt:

```
$ sudo apt install make g++-12 cmake clang-tidy-17
```

For Ubuntu 22.04:

```
$ sudo apt install make g++-12 cmake clang-tidy-14
```

Follow the same steps as above and you should be getting the same results as what you would get in the lab. But keep in mind that we will not be able to help you if you run into any problems caused by differences in the setup. **To be on the safe side, test your code on a lab machine before submitting**.

5. Other systems (command line + plus some editor)

Now we are getting into "you're on your own" territory.

You need to install a recent C++ compiler using your system's preferred approach. On linux systems use your package manager to install the latest gcc/g++ and the latest clang-tidy. On MacOS, the default Clang packaged with Xcode might be enough (we don't know for sure because the only way to test MacOS systems is to have an Apple machine), otherwise you will need to install gcc through Homebrew. On Windows, the compiler packaged with Visual Studio (**not VSCode**!) works okay.

Your system might already have CMake. If not, get it from the CMake website: https://cmake.org/download/

If you run into problems, you might have to tell CMake explicitly which compiler to use by adding an argument in the cmake command setting up the build directory:

```
cmake -B build/ -DCMAKE_CXX_COMPILER=<full path of your compiler>
```

The commands for building and testing the code should remain the same. That's the best part of using CMake.

While all major compilers that support C++20 are supposed to behave the same, there are slight differences in how they process code and how their standard libraries are structured. In one system, including header X might also bring in header Y (that you also need). In another header X does not bring in header Y, so you need to manually include header Y too. This means that code that compiles perfectly well on your machine, might fail to compile on our machines. **To be on the safe side, test your code on a lab machine before submitting**.

Again, we will not provide any technical support on these systems, so don't use them unless you are confident that you can solve any technical issue that might arise.

6. Other systems (full-fledged IDEs)

You can use IDEs but, again, doing so is a mistake.

Most IDEs can parse CMake files, so if you import the assignment folder, they will be (probably) able to create a project that is somewhat capable of building the test executables. IDEs will use their own CMake presets and will ignore certain parts of the CMake configuration, so not everything will work correctly though. Fixing these issues is doable, but it requires extra time and understanding of the internals of your IDE.

If you are determined to make the mistake of using an IDE, **please make sure you test your code on a lab machine before submitting**.

7. Using multiple systems

If you are using multiple systems, for example doing most development locally but then connecting to a lab machine to test your code, **you need to handle the `build/` directory carefully**. The build directory is setup for the specific system where the initial cmake command was invoked. It uses the full path of your compiler and the full paths of your source and target files. Using something like `cmake --build build/` on a `build/` folder copied from somewhere else will not work. If you are lucky, this command will fail immediately with a clear error message. If you are less lucky, the command will partially succeed and then fail with a long list of hard to decipher errors.

To avoid this problem ideally **avoid copying your whole assignment directory when moving from one system to another.** If you are using git to move your files, **do not `git add` the build/ folder**. The repository we created for you has a .gitignore file that explicitly forbids adding the build folder, so you will be okay (unless you explicitly override gitignore). If you are creating zip files (or other archives), avoid adding the build/ folder. Otherwise, delete it from the zip file before copying the file to another machine.

An alternative is to delete the contents of the `build/` folder after copying it and running `cmake -B build/` again on the new machine to reinitialise the build directory.

A final alternative is to have multiple build folders with different names, each one used from a different machine. E.g. build_lab for building from a lab machine and build_laptop from your laptop. While this works, it can be error prone.

Appendix B: Catch2

Both sets of tests use the *Catch*[3] test framework, which is composed of two files: `catch.cpp` and `catch.hpp`. You definitely do not need to understand these files. Just know that they exist.

The tests in `structural-tests.cpp` and `functional-tests.cpp` are structured around individual test cases. A test case is basically a function enclosed in a `TEST_CASE() { ... }`. The code in each test case will be executed as a unit. Each test case, sets up some state, creates new objects of the tested classes, and calls their member functions to either change or get their state. We check whether these functions executed correctly by specifying assertions that say what the expected outcome was. We have three main types of assertions:

- `CHECK( <statement> )` → assert that the statement is true, but keep executing the test case even if the statement fails.
- `REQUIRE( <statement> )` → assert that the statement is true and stop the test case immediately if it fails. Mostly used in places where a wrong result means that the code is seriously broken and further testing is pointless
- `REQUIRE_THAT( <value1>, Catch::Matchers::Equals(value2) )` and `CHECK_THAT( <value1>, Catch::Matchers::Equals(value2) )` → assert that the return value of the statement is semantically equal to value2

You don't need to understand the testing code fully, but if you get failed tests, it might be useful to check the code and the comments that are associated with it in order to understand what went wrong.

Appendix C: CMake

CMake is a build file generator: its input is a file (`CMakeLists.txt`) describing the relationships of the source files with the various target executables. Its output is the actual build files (e.g. Makefile, Ninja, Nmake, etc) that when executed will build the target executables. CMake supports multiple different build systems, not only Make, so it's more portable: after you've installed CMake correctly, the same set of CMake commands will run correctly regardless of the OS, the build system, or the compiler.

Appendix D: UML Diagram

An automatically generated (and potentially inaccurate) UML diagram is contained in `uml.pdf`

Appendix E: Doxygen-based Documentation

Detailed documentation, including classes, public member functions and variables, class hierarchies, function descriptions, etc can be found in the folder `html/`.

A good entry point is `html/inherits.html`

3 https://github.com/catchorg/Catch2