Using CMake Build System

Introduction

In the last two assignments we dealt with:

- writing tests using catch2
- code coverage using gcov, lcov

We saw that compiling, running the program and generating the coverage information takes a sequence of commands which can be tedious and error prone. In this assignment, we will use **CMake** to manage the build process of our program.

Problem 1: Minimal example for CMake [5]

- This problem is meant to be a quick start for CMake.
- Create a directory called **problem1** and inside it create the following directories:
  - **src** to store all source files
  - **tests** to keep all source files for the tests
  - **include** to keep all header files
  - **build** to store the executable and the coverage information
- We also need a starting point for our program. Create a file `main.cpp` and put it in the **problem1** directory. The following is the code for `main.cpp`

```
#include "hello_world.h"
```

```
int main()
{
cout << "Hello World from main.cpp" << endl;
cout << print_hello_world(true);
return 0;
}
```

- As you can see, we included our own header file called `hello_world.h`. Usually, header files should only contain declarations. Let's create it in the **include** directory.

```
#include <iostream>

using namespace std;

bool print_hello_world(bool);
```

- This header file contains a function declaration `bool print_hello_world(bool);`, we need to define it. Let's define it in the **src** folder.
- Create a new file called `hello_world.cpp` in the **src** folder and use the following code.

```
#include "hello_world.h"
bool print_hello_world(bool print)
{
if (print)
{
cout << "Hello World from hello_world.cpp" << endl;
}
else
{
cout << "No Hello World from hello_world.cpp" << endl;
}
cout << "Hi world" << endl;
return true;
}
```

- We can now use *CMake* to compile and run this minimal example. For this, we need to create a new configuration file for *CMake*. Create a file called *CMakeLists.txt* in the root directory i.e., **problem1** directory with the following content.

```
# Required for compatibility reasons
cmake_minimum_required(VERSION 3.10)

# Name of the project and the release version
project(HelloWorldProject VERSION 1.0)

# This specifies the C++ standard version we want our project to be
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED True)

# We are including the header files required
include_directories(include)

# The following specifies the name of the executable - 'HelloWorldProject'
add_executable(HelloWorldProject main.cpp src/hello_world.cpp)
```

- We can now compile and run the project. For this:
    - Go to the **build** directory and run the following commands
    - `cmake ..`
    - `make`
    - Run the executable `./HelloWorldProject` which should print out the following:

```
Hello World from main.cpp
Hello World from hello_world.cpp
Hi world
```

- We created a very simple minimal example for using *CMake*. You can now make changes to the code and simply use `make` to build the project and create the executable. This makes the overall workflow much simpler.

Adding Tests

- Now let's integrate tests to our workflow. In a previous assignment, we included `catch.hpp` to write tests. Copy the provided `catch.hpp` to the **include** folder.
- Now let's write tests. Create a new file in the **tests** directory called `test_hello_world.cpp` and write the following content.

```
#define CATCH_CONFIG_MAIN
#include "catch.hpp"

#include "hello_world.h"

TEST_CASE("Hello World from hello_world.cpp", "[hello_world]")
{
REQUIRE(print_hello_world(true) == true);
}
```

This tests the function `print_hello_world` written in the **src** folder. We need to integrate this to *CMake* configuration. For this we need to add the new lines to the *CMakeLists.txt* file. The updated configuration looks like the following:

```
cmake_minimum_required(VERSION 3.10)
# Name of the project and the release version
project(HelloWorldProject VERSION 1.0)

# This specifies the C++ standard version we want our project to be in
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED True)

# We are including the header files required
include_directories(include)

# The following specifies the name of the executable
add_executable(HelloWorldProject main.cpp src/hello_world.cpp)

# Added the following lines for integrating tests
add_executable(HelloWorldTest tests/test_hello_world.cpp src/hello_world.cpp)

enable_testing()

add_test(NAME HelloWorldTest COMMAND HelloWorldTest)
```

- Rest of the steps is similar as before:
  - Go to the **build** directory and run the following commands
  - `cmake ..`
  - `make`
  - Now run the tests using `./HelloWorldTest`
  - *Tip* You can also use the command `ctest` to run the tests instead of using the executable file `./HelloWorldTest`

Adding Code Coverage

The next step is to integrate code coverage to our minimal example. If you recall, for this we need to build our project using the flags `-fprofile-arcs -ftest-coverage`. Let's add it to *CMake*.

We can do so by adding the following lines to *CMakeLists.txt* ideally below other `set()` parts.

```
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fprofile-arcs -ftest-coverage")
set(CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} -fprofile-arcs -ftest-coverage")
```

We can build and run the program which will now generate `.gcno` and `.gcda` files.

Integrating lcov and genHTML

Next, we add the `lcov` and `genhtml` command sequence as a task to *CMakeLists.txt*.

```
# Adding coverage task

find_program(LCOV_PATH lcov)
find_program(GENHTML_PATH genhtml)

add_custom_target(coverage
    COMMAND ${LCOV_PATH} --capture --directory . --output-file coverage.info
    COMMAND ${LCOV_PATH} --remove coverage.info '/usr/*' '*/tests/*' '*/cmake_tests/include/*' --output-file coverage_filtered.info
    COMMAND ${GENHTML_PATH} coverage_filtered.info --output-directory coverage_report
    WORKING_DIRECTORY ${CMAKE_BINARY_DIR}
    COMMENT "Creating coverage"
)
```

We can run this task using the command `make coverage`

Bonus

Since finding and deleting the `.gcno` and `.gcda` files can be tedious, we can also create a task to delete them as follows:

```
add_custom_target(coverage_clean
    COMMAND find . -name "*.gcda" -delete
    COMMAND find . -name "*.gcno" -delete
    WORKING_DIRECTORY ${CMAKE_BINARY_DIR}
    COMMENT "Deleted the coverage files"
)
```

This can be run using `make coverage_clean`

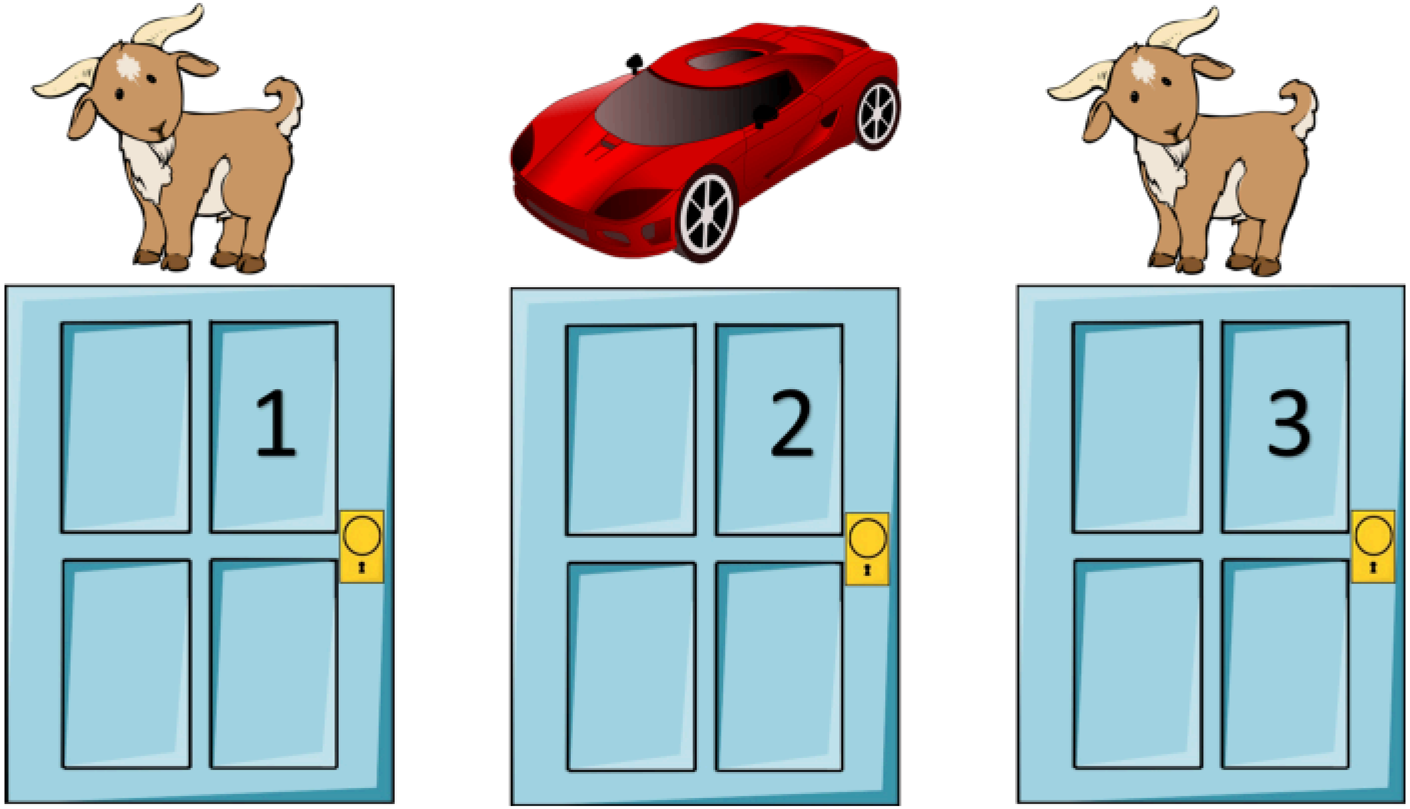
Problem Follow the tutorial and upload the folder as a `.zip` file. Full marks if the tutorial was faithfully reproduced.

For all the following problems, you should use CMake to build it.

Problem 2 - Monty Hall Problem [40]

For this problem, there are no restrictions on the choice of header file.
Suppose you're on a game show, and you're given the choice of three doors:

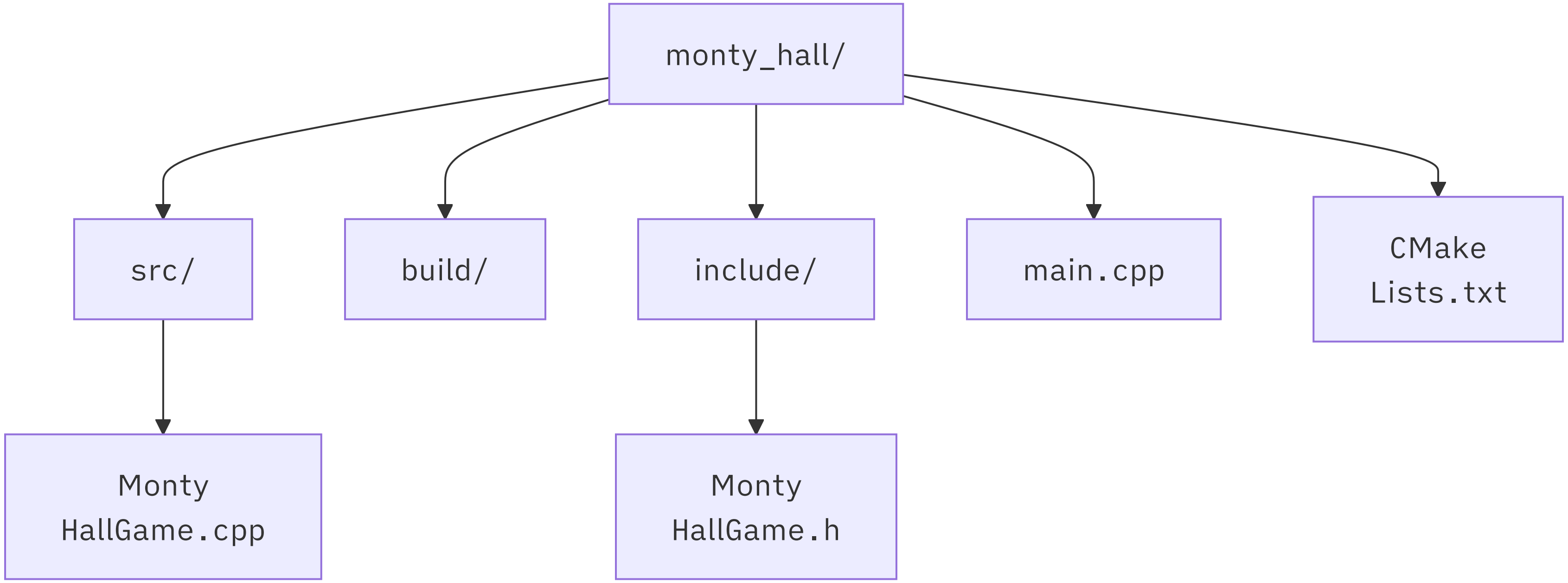
- behind one door is a car.
- behind the others, goats.
- You pick a door, say No. 3, and the host, who knows what's behind the doors, opens another door, say No. 1, which has a goat.
- He then says to you, "Do you want to switch to door No. 2 or stick with your choice?"
- **The problem: is it to your advantage to switch door?**



There are many solutions to the problem and one of them is simulation of the problem. Write a program in C++ that simulates the monty hall problem.

- Create a folder **monty_hall** and inside it three other folders **build**, **include** and **src**. For this problem, you **do not need** to write any tests using *catch2*.

- Create a *main.cpp* in the root directory. This should contain code that auto generates a scenario of the game where you are a contestant and the computer is the host. You should randomly pick a door where the computer reveals a door with the goat. Now you can either switch the door or stick with your choice. Based on your choice, the computer reveals if you win the car.
- After one round of play, simulate the game for 1000 times and provide a numerical response to which strategy is better. The output states the percentage each strategy wins. The *two* strategies being:
  - Switch the door
  - Stick with your original choice
- You should keep proper checks such that an user is not able select a door which is already opened
- The overall directory structure looks like below which should be followed:



- `zip` the **monty_hall** folder and upload it.
- I am presenting part of the content of `MontyHallGame.h`. You may **choose to not** follow it and implement your own way.

```cpp
#ifndef MONTYHALLGAME_H
#define MONTYHALLGAME_H

#include <vector>

class MontyHallGame
{
public:
    MontyHallGame();
    void generate(); // Use your own return type
    void switchDoor(); // Use your own return type
    void stickWithChoice(); // Use your own return type
    bool hasWon() const;
    // More code ..

private:
    // Your code ..
};

#endif // MONTYHALLGAME_H
```

- The following is part of *main.cpp*. You may **choose to not** follow it and implement your own way.

```cpp
#include <iostream>
#include "MontyHallGame.h"

int main()
{
    MontyHallGame game;
    // Code to play one round of the game.

    int simulations = 1000;
    for (int i = 0; i < simulations; ++i)
    {
     // your code
    }
    // your code to print out the best strategy (switching door/ stick to current choice?)
    return 0;
}
```

Hint

How to generate random numbers in C++?

```
#include <random>
#include <iostream>

using namespace std;

int main()
{
    unsigned int seed = 42; // Change this to get different random numbers
    mt19937 gen(seed);

    uniform_int_distribution<> dis(1, 100);
    uniform_real_distribution<> dis_real(0.0, 1.0);

    for (int i = 0; i < 10; ++i)
    {
        int random_number = dis(gen);
        double random_real = dis_real(gen);
        cout << "Random number: " << random_number << endl;
        cout << "Random number: " << random_real << endl;
    }
}
```

Produces the following output:

```
Random number: 52
Random number: 0.183435
Random number: 72
Random number: 0.59685
Random number: 83
Random number: 0.0580836
Random number: 75
Random number: 0.333709
Random number: 100
Random number: 0.708073
Random number: 3
Random number: 0.0564116
Random number: 2
Random number: 0.832443
Random number: 30
Random number: 0.000778765
Random number: 64
Random number: 0.183405
Random number: 33
Random number: 0.611653
```

Grade Distribution

- The implementation is correct and produces correct output [23]
    - Generates a random game and plays one round with input from user [5]

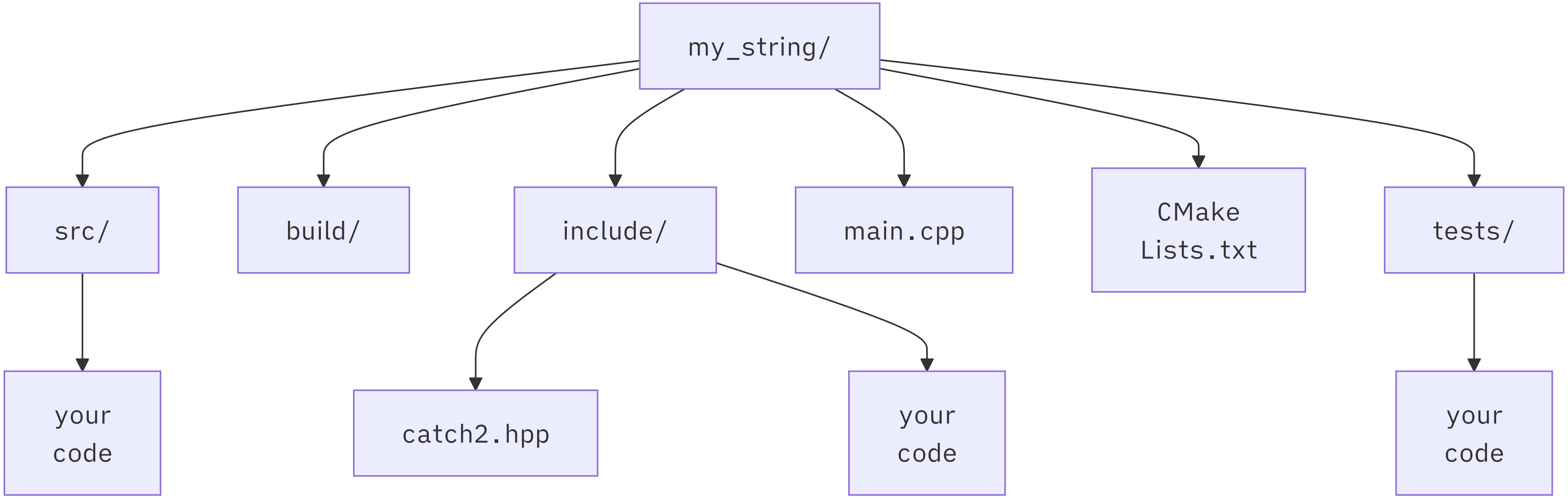
  - Simulates the game 1000 times without any response from user [2]
  - Correctly implement the program and produces correct response. [16]
- Above and followed the proper directory structure [2] else [0]. The program is wrong but followed directory structure [0].
- Above and the program can be built using cmake [10] else [0]. The program is wrong but can be built using cmake [5].

Problem 3 - Custom String Class [55]

The allowed external header files for the following problem are:

- `#include <iostream>`
- `#include <cstring>`
- Create a class called `MyString` that has the following. A newly created `MyString` is by default initialized to `""` Assume that we will only use objects of `MyString` type to perform the operations (i.e., we will not mix with built-in types):
  - A default constructor [2]
  - A parameterized constructor that takes `const char* s` as a parameter [2]
  - A copy constructor [2]
  - A destructor [2]
  - A copy assignment operator (overload = operator) for performing deep copy [4]
  - Overload `+` operator to concatenate two strings [2]
  - Overload == operator to compare two strings which is not case sensitive [3]
  - Friend function to reverse the string in-place (the function should have a return type of `void`). [4]
  - Overload `<<` operator to print out the string [2]
  - Write `catch2` tests [5]
  - Achieve 100% line coverage. [15]
  - Make sure that it is possible to build the entire project using cmake [10]

Now, `zip` the **my_string** directory and upload. You should follow the following directory structure:


my_string/
src/
build/
include/
main.cpp
CMake Lists.txt
tests/
your code
catch2.hpp
your code
your code