Network and Multiplayer Gaming
Practical 02 – Threads and shared data structures

Julien Cordry

February 2, 2023

1 Before writing any code

- Those exercises are not intended to work with your project, but they let you become familiarised with some concepts seen in the lectures.
- Revise how to run threads before doing this.
- Look up the syntax of the threads launched on an object and threads running using a **lambda**.
- Revise the **Counter** example with its solutions.
- You can use Code::Blocks to code the exercises, making sure you configure your project to have a C++11 or C++14 compatibility that allows you to use threads.

2 Threads

2.1 Exercises

1. Create a `Print` class with a `print` function that prints a simple friendly message. Use your `main` function to execute a thread executing this `print` function over an object. Try to detach your thread from `main` first and run the program a few times, then change it to join. What is the difference?
2. Change your code to run the thread using a **lambda** instead.
3. Create another class `Loop` that has a `run` function. The `run` function loops around indefinitely waiting for a `bool` member variable named `hasToStop` to be set to `true`. Use a thread to execute the `run` function while another thread using a `lambda` waits a random amount of time (using `chrono`) before setting `hasToStop` to `true`. Make sure your `main` function awaits your two threads.

2.2 Fight!

- We want to simulate a fight between **Rolfor** the barbarian and **Guard Dan** the gnome druid. They should be put in their own classes with a `run` function that we intend to execute in a thread.

- `run` should loop around with a random wait and exit when the HP of the combatant becomes 0 or less (meaning the character has died).
- After a random wait in `run`, a character should try to attack their opponent. The success of this attack has to be random as well and should take into account the AC or defence value of the defender.
- The attackers want to know when the attack is successful by reading the number of HP left in their opponent. If the opponent has no more HP, we can print out a winning message and exit the `run` function. Since the combatants will want to know about each other, one idea could be to give them pointers to each other to be set in the `main` function.
- You typically want to have a trace of the combat by printing messages on the standard output.
- Don't forget to join the threads in your main.
- Note that the above program is **not** thread-safe and will sometimes run for ever even though one character has died.
- If everything sort of works, you can add some complexity with additional additional threads/characters/classes. You will have to make sure that you are also adding pointers to every other combatants in the classes, so that they can attack each other.

3 Mutex

- We want to implement a simple producer consumer pattern over one variable. Both producer and consumer threads should loop a few times (try 50 at least).
- One producer thread will produce a random integer, add it to a container and then sleep for a random amount of time. It could print the number as it is being created. The producer ends up looping back to the beginning.
- One consumer thread will "consume" a number from the data structure and then sleep a random amount of time. It should also print the number that is being consumed. The consumer ends up looping back to the beginning.
- We could use a simple `struct` as the Counter seen in the lecture to store the shared `int` however we will also need a wrapper around that `struct` so that we allow accessing the stored value in a thread-safe way. Think of using a mutex.
- It is possible that no new `int` has been produced when the consumer tries to consume it. We could use a `bool` to be aware of that.
- Similarly the producer should have a way of knowing if the consumer has consumed the preceding random value. We could use a `bool` to know if that has happened.