Homework 3

Due Wednesday, Apr 9, 2025 at 8pm ET

The learning objectives of this homework are to:

- Review function templates and the iterator pattern in C++
- Gain experience writing parallel code using asynchronous tasks
- Understand how to implement the singleton pattern in C++
- Practice metaprogramming using C++ macros

Use the following commands to download and unpack the distribution code:

```
$ wget https://eecs390.github.io/homework/hw3/starter-files.tar.gz
$ tar xzf starter-files.tar.gz
```

You may work alone or with a partner. Please see the syllabus for partnership rules. As a reminder, you may not share any part of your solution outside of your partnership. This includes code, test cases, and written solutions.

Exercises

1. *Function templates, iterators, and asynchronous tasks.* In this question, we implement a parallel version of `std::find_if()`, which finds the first element in a sequence that passes a given predicate.

   a) Start by implementing the sequential version of `eecs390::find_if()` in `find_if.hpp`. See the documentation of the function template in the provided file, and write your code there as well.

      You may **not** `#include` or use any libraries in your implementation.

      When you are done, you should pass the first test case that uses `find_if_test.cpp`:

      ```
      $ make find_if_test1
      ```

   b) Proceed to complete the parallel version, `eecs390::async_find_if()`, in `async_find_if.hpp`. Use `std::async()` to launch individual tasks that each invoke the sequential implementation from the previous part. Make sure that your code is parallel -- do not wait on the result of a task until all tasks have been launched.

      - The `num_tasks` parameter specifies how many asynchronous tasks you should launch. Use the `results` vector to store the result of launching the tasks.
      - Since `Iterator` must be a LegacyRandomAccessIterator, you can do "pointer" arithmetic on `begin` and `end` to obtain the subsequences to pass to each task.
      - For each task, you will need to compare against the end iterator passed to that task to determine whether or not it found a value for which the predicate is true. The end iterators will be different for each task.

      When you are done, you can run the second test case that uses `find_if_test.cpp`:

      ```
      $ make find_if_test2
      ```

      You can also manually run the test to see what kind of speedup you get with 8 tasks:

      ```
      $ ./find_if_test.exe 100000000 280 285 8
      ```

2. *Unit-test framework.* In this question, we implement a basic version of the unit-test framework used in EECS 280. When you are done, the test case should pass:

   ```
   $ make utf_test
   ```

a) Start by implementing `UTFTestSuite::get()` to use the singleton pattern. Use the private member variable `UTFTestSuite::test_suite` to keep track of the singleton instance.

b) Now write the `TEST()` macro, which generates the code needed to define and register a test case.

The user syntax for specifying a test case is as follows, which is demonstrated in `utf_test.cpp`:

```
TEST(test_name) {
  ... // test body
}
```

The code that `TEST()` generates needs to incorporate the user-provided body, which syntactically requires the generated code to end with a function definition. To make this work and also register the test case with the singleton test-suite instance, the generated code must consist of the following:

- A new derived class of `UTFTestCase`. The constructor for the latter is already implemented to register the test case.
  You will need to declare, but not define, the override for the `run()` method within the definition of derived class. We will repurpose the user-provided body that follows the `TEST()` invocation as the implementation of the `run()` method.
- A global object that is an instance of the new derived class. This will register the test case upon its initialization.
- The header for the definition of the `run()` method of the new derived class.

You may find it useful to invoke the preprocessor to examine the generated output:

```
$ g++ -std=c++20 -E utf_test.cpp
```

We have provided a `Makefile` target to facilitate running the preprocessor after stripping out the `#include` directives:

```
$ make utf_test.preprocess
```

This saves the preprocessed result in `utf_test.E.cpp`. We recommend autoformatting the file (e.g. using your IDE) before examining it.

When you are done with this part, you should be able to compile and run `utf_test.cpp`:

```
$ make utf_test.exe
$ ./utf_test.exe
```

However, the output will not match the expected result until you complete the next part.

c) Finally, implement the `UTF_CHECK()` macro, which checks whether or not the given value is a true value. If not, the generated code should throw a `UTFFailure` exception with a message that can be constructed as follows:

```
diagnostic " at " __FILE__ ":" + std::to_string(__LINE__)
```

The `__FILE__` and `__LINE__` macros are built in, and they expand to the filename and line number respectively.

Submission

Place your solutions to question 1 in the provided `find_if.hpp` and `async_find_if.hpp` files, and to question 2 in the `utf.hpp` file. Submit `async_find_if.hpp`, `find_if.hpp`, and `utf.hpp` to the autograder before the deadline. Be sure to register your partnership on the autograder if you are working with a partner.