Cairo University
Faculty of Engineering

CMP461: Elective 2

Testing Project Report

Team :Thursday

DOCTEST C++

Name	Section	B.N	Email
Aly Ramzy Hassan	1	35	aly.aly981@eng-st.cu.edu.eg
Nour Ahmed	2	29	nour.aly98@eng-st.cu.edu.eg
Hager Ahmed	2	32	hager.abelkader97@eng-st.cu.edu.eg
Mohamed Ahmed Ibrahim	2	9	mohamed.elaziz98@eng-st.cu.edu.eg
Sara Yasser	1	24	sara.mdkor98@eng-st.cu.edu.eg

Table of Contents

Introduction

- Doctest is by far the fastest C++ testing framework in both compile time and runtime compared to other testing frameworks.

- Doctest is Thread-safe asserts (and logging) can be used from multiple threads spawned from a single test case.

- Ultra light on compile times both in terms of including the header and writing thousands of asserts.

- It is used mainly in unit testing.

- It brings the ability of compiled languages such as D / Rust / Nim to have tests written directly in the production code thanks to a fast, transparent and flexible test runner with a clean interface.

Assertion macros

This is The Explanations for Assertion Macros.

- **REQUIRE** (this level will immediately quit the test case if the assert fails and will mark the test case as failed.)
  `REQUIRE_EQ(actual,expected);`

- **CHECK (**this level will mark the test case as failed if the assert fails but will continue with the test case)

```
TEST_CASE("TEST CASE 0 :Main success scenario"){

    Node * root = newNode(1);

    root->left = newNode(2);

    root->right = newNode(3);

    root->left->left = newNode(4);

    root->left->right = newNode(5);

    root->right->left = newNode(6);

    root->right->right = newNode(7);

    CHECK (findLCA(root, 4, 5)==2);

    CHECK (findLCA(root, 4, 6)==1);

    CHECK (findLCA(root, 3, 4)==1);

    CHECK (findLCA(root, 2, 4)==2);

}
```

Exceptions

We Will discuss how to handle Exceptions Using DocTest

- For Example We Try To Construct Graph With –ve Number of Nodes , Which Will Cause Allocation Exception if the developer didn't handle the –ve Number of Nodes in The Constructor, so we will use DocTest to check that the Constructor shouldn't throw Exception using "CHECK_NOTHROW".

```
TEST_CASE("TEST CASE 2 :construct graph with -ve number of nodes"){

    CHECK_NOTHROW (Graph g(-1));}
```

Test cases

Test cases and subcases are very easy to use in practice:

- **TEST_CASE(** *test name* **)**
- **SUBCASE(** *subcase name* **)**

Names should also be string literals.

```
TEST_CASE("isFull")
    {
        Queue myq;

        SUBCASE("isFull is false when contain no element")
        {
            CHECK(!myq.isFull() == true);
        }

      }
```

Figure 1: Test case for checking the isFull() functionality of queue class, that contain subcases inside

Also note that asserts and other doctest functionality can be used in user code if checked for a testing context

```
void enQueue(int value) {
        if (isFull()) {
            if (doctest::is_running_in_test)
                CHECK(rear == MAX_SIZE - 1);
        }
        else {
            if (front == -1) front = 0;
            rear++;
            myqueue[rear] = value;
        }
    }
```

Figure 2: to check that when the queue is full, the rear of the Queue has the maximum number it could get

BDD-style test cases

Doctest supports an alternative syntax that allow tests to be written as "executable specifications. This set of macros map on to `TEST_CASE`s and `SUBCASE`s, with a little internal support to make them smoother to work with.

- **SCENARIO(** *scenario name* **)**
  - These macros map into `TEST_CASE`
- **GIVEN(** *something*)
- **WHEN(** *something* **)**
- **THEN(** *something* **)**
  - These macros map onto `SUBCASE`s
- **AND_WHEN(** *something* **)**

- **AND_THEN(** *something*
    - Similar to `WHEN` and `THEN` are used to chain `WHEN`s and `THEN`s together.

```
SCENARIO("isEmpty")
    {
        GIVEN("An empty queue") {
            Queue myq;
            CHECK(myq.isEmpty() == true);

            WHEN("the size is increased") {
                myq.enQueue(10);
                THEN("Queue is not empty!") {
                    CHECK(!myq.isEmpty() == true);
                }
            }

            AND_WHEN("the size is reduced") {
                myq.deQueue();

                THEN("Queue is empty again") {
                    CHECK(myq.isEmpty() == true;}
            }
        }
    }
```

Figure 3: Scenario that tests the isEmpty() of Queue class, checking its correctness through series of in sequence events using given, when, then and and-when macros.

Test fixtures

It's one of the ways in which **doctest** allows you to group tests together as subcases within a test case using a more traditional test fixture.

```
TEST_CASE_FIXTURE(Queue, "Check isFull() in Queue class")
{
   SUBCASE("isFull is false when contain no element")
    {
        CHECK(!isFull() == true);
    }
    enQueue(10);
    SUBCASE("isFull is false when contain only one element")
    {
        CHECK(!isFull() == true);
    }
    for (int i = 1; i < MAX_SIZE; i++)
        enQueue(10);
    SUBCASE("isFull is true when contain max elements")
    {
        CHECK(isFull() == true);
    }
    for (int i = 0; i < MAX_SIZE; i++)
        deQueue();
}
```

*Figure 4: test **fixture** to test isFull () functionality of class Queue, by creating uniquely-named derived classes of Queue and thus can access the protected method and member variables.*

Test suites

Test cases can be grouped into test suites. This is done with `TEST_SUITE()` or `TEST_SUITE_BEGIN()` / `TEST_SUITE_END()`

```
TEST_SUITE("Performance")
{
    TEST_CASE("Time Complexity: O(n) where n is the number of elements to push in
the queue." * doctest::timeout(1 / 1000))
    {

        Queue myq;
        for (int i = 0; i < 1e5; i++)
        {
            myq.enQueue(i);
        }
    }

}
```

Figure 5: Test suit that contains performance related test cases using TEST_SUITE()

```
TEST_SUITE_BEGIN("Performance");
TEST_CASE("Time Complexity: O(n) where n is the number of elements to push in the
queue." * doctest::timeout(1 / 1000))
{

    Queue myq;
    for (int i = 0; i < 1e5; i++)
    {
        myq.enQueue(i);
    }
}

TEST_SUITE_END();
```

Figure 6: Test suit that contains performance related test cases using TEST_SUITE_BEGIN () and TEST_SUITE_END ()

Decorators

Test cases can be *decorated* with additional attributes like this:

```
TEST_SUITE("Performance")

{
    TEST_CASE("Time Complexity: O(n) where n is the number of elements to push in
the queue." * doctest::timeout(1 / 1000))
    {        Queue myq;
        for (int i = 0; i < 1e5; i++)
        {
            myq.enQueue(i);
        }
    } }
```

Figure 7: timeout () fails the test case if its execution exceeds this limit (in seconds) - but doesn't terminate it.

Value-parameterized test cases

To perform data-driven testing in doctest, we do the assertions in a helper function and call it with a user-constructed array of data:

```
TEST_CASE("TEST CASE 0 :Main success scenario") {
        vector<int> Actual Output;
        Graph g(4);
        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 2);
        g.addEdge(2, 0);
        g.addEdge(2, 3);
        g.addEdge(3, 3);


        vector<int> Expected_Output ;
        Expected_Output.push_back(2);
        Expected_Output.push_back(0);
        Expected_Output.push_back(3);
        Expected_Output.push_back(1);
        Actual_Output = g.BFS(2);

        REQUIRE_EQ(Actual_Output.size(),Expected_Output.size());

        for(int i =0 ;i<Actual_Output.size();i++) {
            CAPTURE(Actual_Output[i]);
            CAPTURE(Expected_Output[i]);
            doChecks(Actual_Output[i],Expected_Output[i]);
         }
    }
```

Figure 8: the user has to manually log the data with calls to CAPTURE ().

We can also use subcases to initialize data differently, where any changes in the data are s seen inside the scope of the subcase:

```
TEST_CASE("TEST CASE 4 :Checking edges in the graph")
{
        Graph g(4);
        SUBCASE("Add edge between two nodes exists in the graph"){
            CHECK(g.addEdge(0,1)==true);
        }
        SUBCASE("Add edge between two nodes doesnt exist in the graph"){
            CHECK(g.addEdge(0,1)==true);

        }

    }
```

Figure 9: data initialized using subcases.

Logging macros:

Additional messages can be logged during a test case

- **INFO(** message , , ...**)**
  - The INFO() macro allows heterogeneous sequences of expressions to be captured by listing them with commas.
  - This message will be relevant to all asserts after it in the current scope or in scopes nested in the current one and will be printed later only if an assert fails.
- **CAPTURE**(variable_name)
  - The CAPTURE() macro is like Info() but takes one variable and print (variable_name:=value) when there is a failure**.**

Messages which can optionally fail test cases:

There are a few other macros for logging information

- **MESSAGE(**message**)**
  - The MESSAGE() macro just prints a message.
- **FAIL(**message**)**
  - The FAIL() macro fails the test case and exits it.
- **FAIL_CHECK(**message**)**
  - The FAIL_CHECK() macro fails the test case but continues with the execution.

Command Line:

Some options that provide more control over the run time of the script.
They have multiple types.

- **Query flags:** options that provide information about the tool, or about the run time of a test script.

Query Flags	Description
`-?` `--help` `-h`	Prints a help message listing all these flags/options
`-v` `--version`	Prints the version of the **doctest** framework
`-c` `--count`	Prints the number of test cases matching the current filters (see below)
`-ltc` `--list-test-cases`	Lists all test cases by name which match the current filters (see below)
`-lts` `--list-test-suites`	Lists all test suites by name which have at least one test case matching the current filters (see below)
`-lr` `--list-reporters`	Lists all registered reporters

- **Int/String/bool options:** options that provide control over the running of the test script.

Int/String Options	Description
`-tc` `--test-case=<filters>`	Filters test cases based on their name. By default all test cases match but if a value is given to this filter like `--test-case=*math*,*sound*` then only test cases who match at least one of the patterns in the comma-separated list with wildcards will get executed/counted/listed
`-tce` `--test-case-exclude=<filters>`	Same as the `-test-case=<filters>` option but if any of the patterns in the comma-separated list of values matches - then the test case is skipped
`-sf` `--source-file=<filters>`	Same as `--test-case=<filters>` but filters based on the file in which test cases are written
`-sfe` `--source-file-exclude=<filters>`	Same as `--test-case-exclude=<filters>` but filters based on the file in which test cases are written
`-ts` `--test-suite=<filters>`	Same as `--test-case=<filters>` but filters based on the test suite in which test cases are in
`-tse` `--test-suite-exclude=<filters>`	Same as `--test-case-exclude=<filters>` but filters based on the test suite in which test cases are in
`-sc` `--subcase=<filters>`	Same as `--test-case=<filters>` but filters subcases based on their names
`-sce` `--subcase-exclude=<filters>`	Same as `--test-case-exclude=<filters>` but filters based on subcase names
`-r` `--reporters=<filters>`	List of reporters to use (default is `console`)
`-o` `--out=<string>`	Output filename
`-ob` `--order-by=<string>`	Test cases will be sorted before being executed either by **the file in which they are / the test suite they are in / their name / random**. The possible values of `<string>` are `file` / `suite` / `name` / `rand`. The default is `file`. **NOTE: the order produced by the `file`, `suite` and `name` options is compiler-dependent and might differ depending on the compiler used.**
`-rs` `--rand-seed=<int>`	The seed for random ordering
`-f` `--first=<int>`	The **first** test case to execute which passes the current filters - for range-based execution
`-l` `--last=<int>`	The **last** test case to execute which passes the current filters - for range-based execution
`-aa` `--abort-after=<int>`	The testing framework will stop executing test cases/assertions after this many failed assertions. The default is 0 which means don't stop at all. Note that the framework uses an exception to stop the current test case regardless of the level of the assert (`CHECK` / `REQUIRE`) - so be careful with asserts in destructors...
`-scfl` `--subcase-filter-levels=<int>`	Apply subcase filters only for the first `<int>` levels of nested subcases and just run the ones nested deeper. Default is a very high number which means *filter any subcase*
Bool Options	**Description**
`-s` `--success=<bool>`	To include successful assertions in the output
`-cs` `--case-sensitive=<bool>`	Filters being treated as case sensitive
`-e` `--exit=<bool>`	Exits after the tests finish - this is meaningful only when the client has provided the `main()` entry point - the program should check the `shouldExit()` method after calling `run()` on a `doctest::Context` object and should exit - this is left up to the user. The idea is to be able to execute just the tests in a client program and to not continue with it's execution
`-d` `--duration=<bool>`	Prints the time each test case took in seconds
`-nt` `--no-throw=<bool>`	Skips **exceptions-related assertion** checks
`-ne` `--no-exitcode=<bool>`	Always returns a successful exit code - even if a test case has failed
`-nr` `--no-run=<bool>`	Skips all runtime **doctest** operations (except the test registering which happens before the program enters `main()`). This is useful if the testing framework is integrated into a client codebase which has provided the `main()` entry point and the user wants to skip running the tests and just use the program

`-nv` `--no-version=<bool>`	Omits the framework version in the output
`-nc` `--no-colors=<bool>`	Disables colors in the output
`-fc` `--force-colors=<bool>`	Forces the use of colors even when a tty cannot be detected
`-nb` `--no-breaks=<bool>`	Disables breakpoints in debuggers when an assertion fails
`-ns` `--no-skip=<bool>`	Don't skip test cases marked as skip with a decorator
`-gfl` `--gnu-file-line=<bool>`	`:n:` vs `(n):` for line numbers in output (gnu mode is usually for linux tools/IDEs and is with the `:` separator)
`-npf` `--no-path-filenames=<bool>`	Paths are removed from the output when a filename is printed - useful if you want the same output from the testing framework on different environments
`-nln` `--no-line-numbers=<bool>`	Line numbers are replaced with `0` in the output when a source location is printed - useful if you want the same output from the testing framework even when test positions change within a source file
`-ndo` `--no-debug-output=<bool>`	Disables output in the debug console when a debugger is attached

All flags/options also come with a prefixed version (--dt- before the flag/option) to make sure that these flags/options don't conflict with other tools/systems.

Configuration:

Doctest provides a set of identifiers to allow configuring how the tool is built, these identifiers should be defined before the inclusion of the framework header, the available identifiers are:

- DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN
- DOCTEST_CONFIG_IMPLEMENT
- DOCTEST_CONFIG_DISABLE
- DOCTEST_CONFIG_IMPLEMENTATION_IN_DLL
- DOCTEST_CONFIG_NO_SHORT_MACRO_NAMES
- DOCTEST_CONFIG_TREAT_CHAR_STAR_AS_STRING
- DOCTEST_CONFIG_SUPER_FAST_ASSERTS
- DOCTEST_CONFIG_USE_STD_HEADERS
- DOCTEST_CONFIG_VOID_CAST_EXPRESSIONS
- DOCTEST_CONFIG_NO_COMPARISON_WARNING_SUPPRESSION
- DOCTEST_CONFIG_OPTIONS_PREFIX
- DOCTEST_CONFIG_NO_UNPREFIXED_OPTIONS
- DOCTEST_CONFIG_NO_TRY_CATCH_IN_ASSERTS
- DOCTEST_CONFIG_NO_EXCEPTIONS
- DOCTEST_CONFIG_NO_EXCEPTIONS_BUT_WITH_ALL_ASSERTS
- DOCTEST_CONFIG_ASSERTION_PARAMETERS_BY_VALUE
- DOCTEST_CONFIG_COLORS_NONE
- DOCTEST_CONFIG_COLORS_WINDOWS
- DOCTEST_CONFIG_COLORS_ANSI
- DOCTEST_CONFIG_WINDOWS_SEH
- DOCTEST_CONFIG_NO_WINDOWS_SEH
- DOCTEST_CONFIG_POSIX_SIGNALS

- DOCTEST_CONFIG_NO_POSIX_SIGNALS
- DOCTEST_CONFIG_INCLUDE_TYPE_TRAITS

After defining the configurations, we include the framework header.

The main() entry point:

Doctest by default provides a main() for you if you want to run some test scripts existing in a single file and terminating the program after the test is done, but in production environment you would need to continue executing your program.

Doctest provides the ability to build your custom main() entry point, passing to it parameters from the command line to control it, and the ability to continue the execution of the main program after the test is done, this example shown how you can do it.

```
#define DOCTEST_CONFIG_IMPLEMENT

#include "doctest.h"

int main(int argc, char ** argv)

{

    doctest::Context context(argc, argv);

    int test_result = context.run(); // run queries, or run tests unless --no-run

    if(context.shouldExit()) // honor query flags and --exit

        return test_result;

    //normal program operation continues

}
```

Reporters:

Doctest has a modular reporter/listener system with which users can write their own reporters and register them. The reporter interface can also be used for "listening" to events, the default available reporters to use are:

- **Console**: writes normal lines of text with coloring if a capable terminal is detected.
- **Xml:** writes in xml format tailored to **doctest**.

The output by default is written to stdout but it can be configured from the command line with the `--out=<filename>` **option.**

It is possible to provide a custom reporter by defining a class inherited from the IReporter **class.**

Work Distribution

Name	1	2	3	4
Aly Ramzy	LCA	Euler Tour	BFS	MinMax
Nour Ahmed	Dijkstra	Merge Sort	Stack	Binary Search
Hager Ahmed	Queue	Priority Queue	Insertion Sort	BST Traversal
Mohamed Ahmed	KMP	Kruskal	Segment Tree	LPS
Sara Yasser	Convex Hull	LCS	LIS	Quick Sort

- For Each Algorithm We have covered Correctness Test and Performance Test.

Project Structure

1. Code Files “CodeFiles” : The Folder That contains the code to be tested.
2. Test Scripts “TestScripts” : The Folder That contains the test scripts for each code in the code files.
3. Tool “Tool” : The Header File of Doctest.
4. Main.cpp : The Main file to run the whole project.
5. Bash Scripts : the scripts used to compile and run the project.