 **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** 3](#_Toc60761843)

[**Assertion macros** 4](#_Toc60761844)

[**Exceptions** 4](#_Toc60761860)

[**Test cases** 5](#_Toc60761864)

[**BDD-style test cases** 5](#_Toc60761865)

[**Test suites** 7](#_Toc60761866)

[**Decorators** 7](#_Toc60761867)

[**Value-parameterized test cases** 8](#_Toc60761868)

[**Logging macros:** 9](#_Toc60761869)

[**Messages which can optionally fail test cases:** 9](#_Toc60761870)

[**Command Line:** 9](#_Toc60761871)

[**Configuration:** 11](#_Toc60761872)

[**The main() entry point:** 12](#_Toc60761899)

[**Reporters:** 12](#_Toc60761902)

**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 allows tests to be written as "executable specifications”. This set of macros map on to TEST\_CASEs and SUBCASEs, 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 SUBCASEs
* **AND\_WHEN(** *something* **)**
* **AND\_THEN(** *something)*
  + Similar to WHEN and THEN are used to chain WHENs and THENs 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 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:**

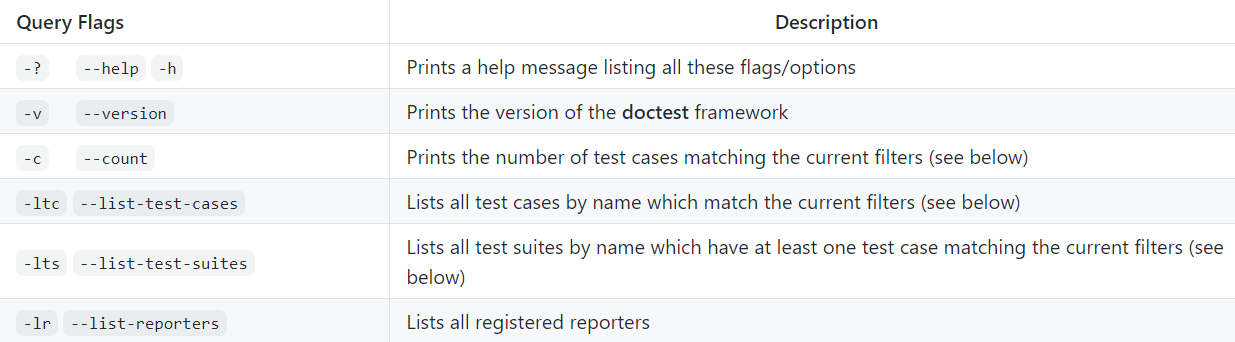
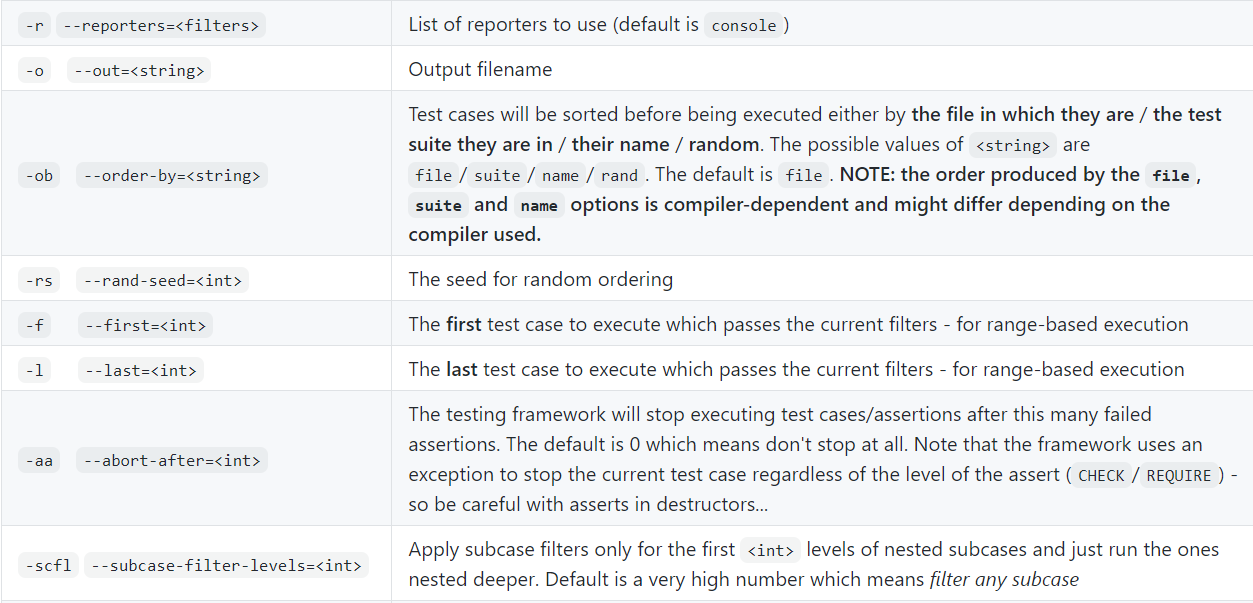
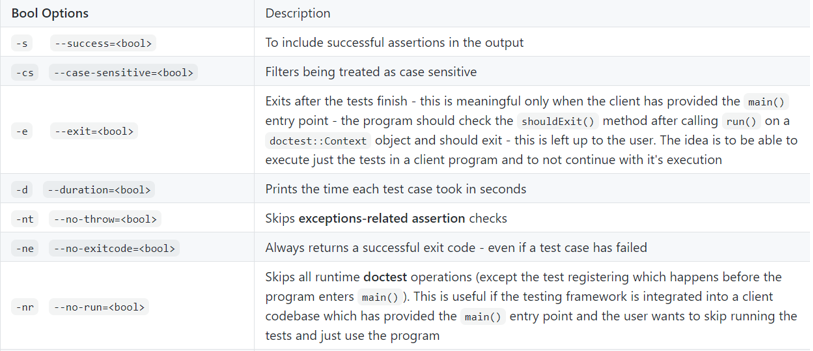
**There are a few other macros for logging information**

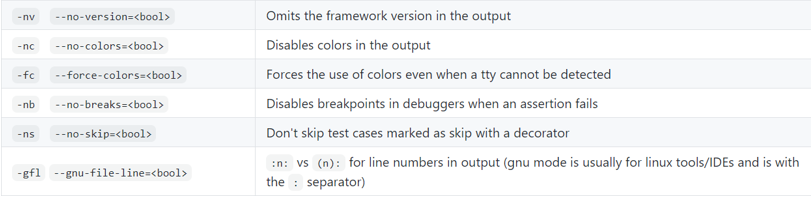
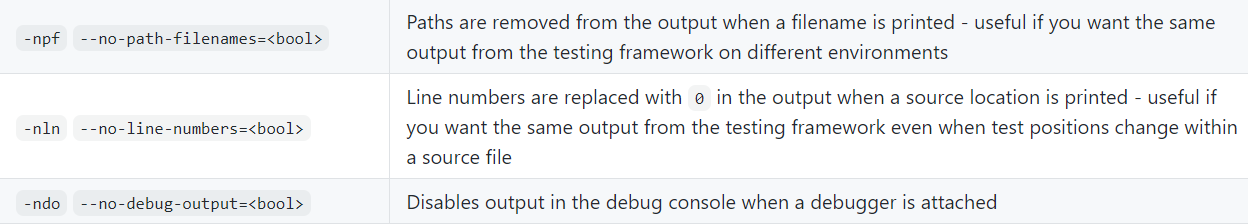
* **MESSAGE(**message**)**
  + The MESSAGE() macro just prints a message.
* **FAIL(**message**)**
  + The FAIL() macro fails the test case and exits it.
* **FAIL\_CHECK(**message**)**
  + The FAIL\_CHECK() macro fails the test case but continues with the execution.

**Command Line:**

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



**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.