CTest

# Overview

CTest is a C test harness. It provides a test runner and a set of assert macros that can be used to write tests.

CTest is designed to:

* Only use C for writing tests in order to minimize mixing C and C++ (sometimes the mix is just not desired)
* Maximize portability, thus trying to avoid at any costs compiler implementation specific features.

# Using CTest

The following steps are required in order to use CTest:

* Include CTest.h in a .c file
* Write the tests in the .c file as part of a test suite
* Link the test runner in your executable
* As part of main (or any other function), execute the test suite

# The first test

The below example shows a simple test written by using CTest:

#include "CTest.h"

#include "SomeUnitUnderTest.h"

CTEST\_BEGIN\_TEST\_SUITE(SimpleTestSuiteOneTest)

CTEST\_FUNCTION(Test1)

{

// arrange

// act

int x = SomeFunction();

// assert

CTEST\_ASSERT\_ARE\_EQUAL(int, 42, x);

}

CTEST\_END\_TEST\_SUITE(SimpleTestSuiteOneTest)

In order to run the suite, the main function would contain:

#include "CTest.h"

int main(int argc, char\* argv[])

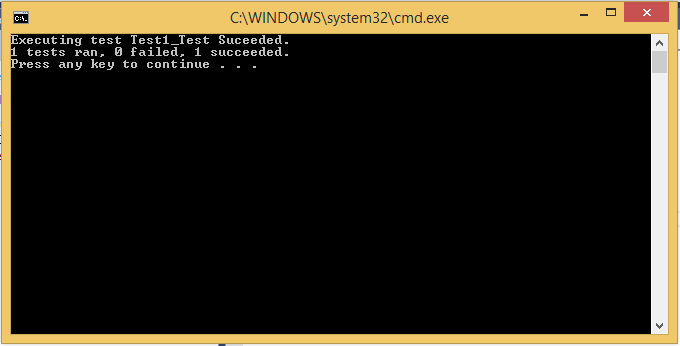
{

CTEST\_RUN\_TEST\_SUITE(SimpleTestSuiteOneTest);

return 0;

}

The results would look like:



# Test suites

One test suite is supported per translation unit.

The beginning of a test suite is marked by the macro CTEST\_BEGIN\_TEST\_SUITE(suiteName).

The end of a test suite is marked by the macro CTEST\_END\_TEST\_SUITE(suiteName).

To run all the tests in a test suite the following macro can be used:

CTEST\_RUN\_TEST\_SUITE(suiteName{,failedTestCount});

The execution order of the tests in a test suite is not guaranteed. Tests are executed sequentially.

The failedTestCount for CTEST\_RUN\_TEST\_SUITE is optional. If specified, the number of failed tests will be summed up in the failedTestCount variable, that is passed as argument.

# Fixtures

## CTEST\_SUITE\_INITIALIZE

This special fixture is executed before all the tests in the test suite. All resources allocated in CTEST\_SUITE\_INITIALIZE should be freed in CTEST\_SUITE\_CLEANUP.

CTEST\_SUITE\_INITIALIZE()

{

/\* Some init code \*/

}

## CTEST\_SUITE\_CLEANUP

This special fixture is executed after all the tests in the test suite.

CTEST\_SUITE\_CLEANUP()

{

/\* Free resources allocated in CTEST\_SUITE\_INITIALIZE \*/

}

## CTEST\_FUNCTION\_INITIALIZE

This special fixture is executed before calling each test function in the test suite. All resources allocated in CTEST\_FUNCTION\_INITIALIZE should be freed in CTEST\_FUNCTION\_CLEANUP.

CTEST\_FUNCTION\_INITIALIZE()

{

/\* Initialize specific things for each test function \*/

}

## CTEST\_FUNCTION\_CLEANUP

This special fixture is executed after each test in the test suite.

CTEST\_FUNCTION\_CLEANUP()

{

/\* Free resources allocated in CTEST\_FUNCTION\_INITIALIZE \*/

}

# Assert macros

Assert macros allow asserting various results and failing the tests if the asserted values/expressions fail.

By default if no assert macro fails a test, the test is reported as succesfull.

Two sets of assert macros are available: with and without a supplied assert text. The macros suffixed with \_WITH\_MSG behave the same like their counterpart without an assert text, but additionally they print the supplied assert text (message) into the test run output.

The following assert macros without a supplied assert text are supported by CTEST:

## CTEST\_ASSERT\_FAIL(message);

This macro fails the test and displays the message argument.

## CTEST\_ASSERT\_ARE\_EQUAL(type, expected, actual, [format, …]);

This macro compares the expected and actual values, assuming they are of type “type” and fails the test if the values are different.

CTEST\_FUNCTION(Assert\_Are\_Equal\_2\_Ints\_Fails)

{

// arrange

// act

int x = SomeFunction();

// assert

CTEST\_ASSERT\_ARE\_EQUAL(int, 42, x);

}

## CTEST\_ASSERT\_ARE\_NOT\_EQUAL(type, expected, actual, [format, …]);

This macro compares the expected and actual values, assuming they are of type “type” and fails the test if the values are equal.

CTEST\_FUNCTION(Assert\_Are\_Not\_Equal\_2\_Ints\_Fails)

{

// arrange

// act

int x = SomeFunction();

// assert

CTEST\_ASSERT\_ARE\_NOT\_EQUAL(int, 0, x);

}

When using the [format, …] optional arguments to specify an assert message:

CTEST\_FUNCTION(Assert\_Are\_Not\_Equal\_2\_Ints\_Fails)

{

// arrange

// act

int x = SomeFunction();

// assert

CTEST\_ASSERT\_ARE\_NOT\_EQUAL(int, 0, x, "{ Message with int: %d }", 42);

}

## CTEST\_ASSERT\_IS\_NULL(value, [format, …]);

This macro fails the test if the value argument is not NULL.

CTEST\_FUNCTION(Assert\_Is\_NULL)

{

// arrange

// act

void\* x = SomeFunction();

// assert

CTEST\_ASSERT\_IS\_NULL(x);

}

## CTEST\_ASSERT\_IS\_NOT\_NULL(value, [format, …]);

This macro fails the test if the value argument is NULL.

CTEST\_FUNCTION(Assert\_Is\_Not\_NULL)

{

// arrange

// act

void\* x = SomeFunction();

// assert

CTEST\_ASSERT\_IS\_NOT\_NULL(x);

}

## CTEST\_ASSERT\_IS\_TRUE(expression, [format, …]);

This macro fails the test if expression evaluates to zero.

CTEST\_FUNCTION(Assert\_Is\_True)

{

// arrange

// act

int x = SomeFunction();

// assert

CTEST\_ASSERT\_IS\_TRUE(x == 4);

}

## CTEST\_ASSERT\_IS\_FALSE(value, [format, …]);

This macro fails the test if expression evaluates to something different than zero.

CTEST\_FUNCTION(Assert\_Is\_False)

{

// arrange

// act

int x = SomeFunction();

// assert

CTEST\_ASSERT\_IS\_FALSE(x != 4);

}

# Specialized comparer and string conversion function

## CTEST\_COMPARE(niceType, type)

In order to allow comparing of specialized types (i.e. structures, etc.), the CTEST\_COMPARE macro can be used:

typedef struct mystruct\_tag

{

unsigned char x;

} mystruct;

CTEST\_COMPARE(mystruct\_ptr, mystruct\*)

{

return (left->x != right->x)

}

A comparer is responsible for comparing 2 values (passed as arguments): *left* and *right.*

*left* and *right*, are of the type “type”. The comparer function should return a value different than zero if the *left* and *right* values are different; otherwise, if the values are equal it should return zero.

The *niceType* is a typedef (cannot contain any invalid characters like space, \*, etc).

## CTEST\_TOSTRING(niceType, type, string, bufferSize, value)

CTEST\_TOSTRING(mystruct\_ptr, mystruct\*, string, bufferSize, value)

{

(void)snprintf(string, bufferSize, "{ %d }", (int)value->x);

}

The function should print in *dst* the desired representation of *value*, where *value* is of type “type”, and *dst* is of type char\*.

The *niceType* is a typedef (cannot contain any invalid characters like space, \*, etc).

# Out of the box supported types

The CTest harness supports out of the box string formatting and comparers for the following types:

* int
* char
* short
* long
* uint8\_t
* int8\_t
* uint16\_t
* int16\_t
* uint32\_t
* int32\_t
* uint64\_t
* int64\_t
* size\_t
* float
* double
* long double
* char\* (char\_ptr)