**Google Test**

* Based on xUnit architecture.
* Portable
* Reusable.
* Uses C++

**Installation**

* Clone the repository from GitHub.
* Install Google Test using CMake.

**Syntax**

Bare minimum for GTest

#include <gtest/gtest.h>

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

{

testing::InitGoogleTest(&argc, argv);

return RUN\_All\_TESTS();

}

TEST(TestName, SubTestName)

{

ASSERT\_TRUE(1==1);

}

**Additional Info**

* Assertions statements can result in success, non-fatal failure or fatal failure.
* Assert statements lead to fatal failure.

TEST(TestName, SubTestName)

{

ASSERT\_TRUE(1==2);

cout<<”Assertion Completed”<<endl; //cout statement doesn’t get printed due to fatal failure.

}

* Expect statements lead to non-fatal failure.

TEST(TestName, SubTestName)

{

EXPECT\_TRUE(1==2);

cout<<”Assertion Completed”<<endl; //cout statement gets printed due to non-fatal failure.

}

* It’s possible to write more than one assertion statements in test cases. But that can lead to serious repercussions. The test will fail even if any of the statements fail.
* Custom Output Message on Failure

ASSERT\_EQ(val1, val2)<<”Values are not same”; // The statement printed only if the condition is false.

**Assertion Types**

ASSERT\_TRUE(condition) & EXPECT\_TRUE(condition) checks if condition is true.

ASSERT\_FALSE(condition) & EXPECT\_FLASE(condition) checks if condition is false.

ASSERT\_EQ(val1, val2) & EXPECT\_EQ(val1, val2) checks if val1 == val2.

ASSERT\_NE(val1, val2) & EXPECT\_NE(val1, val2) checks if val1 != val2.

ASSERT\_LT(val1, val2) & EXPECT\_LT(val1, val2) checks if val1 < val2.

ASSERT\_LE(val1, val2) & EXPECT\_LE(val1, val2) checks if val1 <= val2.

ASSERT\_GT(val1, val2) & EXPECT\_GT(val1, val2) checks if val1 > val2.

ASSERT\_GE(val1, val2) & EXPECT\_GE(val1, val2) checks if val1 >= val2.

ASSERT\_THROW(statement, exception\_type) & EXPECT\_THROW(statement, exception\_type) verifies whether the statement throws the given exception\_type.

ASSERT\_ANY\_THROW(statement) & EXPECT\_THROW(statement) verifies whether the statement throws any exception\_type.

ASSERT\_NO\_THROW(statement) & EXPECT\_THROW(statement) verifies whether the statement does not throw any exception\_type.

ASSERT\_STREQ(str1, str2) & EXPECT\_STREQ(str1, str2) checks if contents of str1 and str2 are the same.

ASSERT\_STRNE(str1, str2) & EXPECT\_STRNE(str1, str2) checks if contents of str1 and str2 are different.

ASSERT\_STRCASEEQ(str1, str2) & EXPECT\_STRCASEEQ(str1, str2) checks if contents of str1 and str2 are the same, after ignoring the case.

ASSERT\_STRCASENE(str1, str2) & EXPECT\_STRCASENE(str1, str2) checks if contents of str1 and str2 are different, after ignoring the case.

ASSERT\_FLOAT\_EQ(val1, val2) & EXPECT\_FLOAT\_EQ(val1, val2) checks if val1 == val2.

ASSERT\_DOUBLE\_EQ(val1, val2) & EXPECT\_DOUBLE\_EQ(val1, val2) checks if val1 == val2.

ASSERT\_NEAR(val1, val2, abs\_error) & EXPECT\_NEAR(val1, val2, abs\_error) checks if modulus(val1-val2) < abs\_error.

ASSERT\_HRESULT\_SUCCEEDED(expression) & EXPECT\_HRESULT\_SUCCEEDED(expression) prints the expected result and the actual result with error details, if expression succeeds unexpectedly.

ASSERT\_HRESULT\_FAILED(expression) & EXPECT\_HRESULT\_FAILED(expression) prints the expected result and the actual result with error details, if expression fails unexpectedly.

ASSERT\_PRED1(pred, val1) EXPECT\_PRED1(pred, val1) checks if pred(val1) returns true

ASSERT\_PRED2(pred, val1, val2) EXPECT\_PRED2(pred, val1, val2) checks if pred(val1, val2) returns true

Maximum PRED5 possible.

**Useful Commands**

GTEST\_SKIP() //Skips test during runtime. Neither Success nor fail.

ADD\_FAILURE() // Generates a nonfatal failure with a generic message.

FAIL() // Generates a fatal failure with a generic message.

SUCCEED() // Generates a success with a generic message.

## Death tests

ASSERT\_DEATH, ASSERT\_EXIT are the death assertions. Useful in handling errors. They check if a proper error message is emitted in case of bad input to a routine or if the process exits with a proper exit code.

ASSERT\_DEATH(statement, expected\_message)

ASSERT\_EXIT(statement, predicate, expected\_message)

**Test Cases**

* It has to be fast.
* Run Independently
* No external dependency.
* 3 components: Arrange, Act, Assert.
* Arrange contains everything required to run the test.
* Act is the run.
* Assert is verification.

Example 1

TEST(TestName, SubTestName)

{

//Arrange

int val = 100;

int inc = 5;

//Act

val = val + inc;

//Assert

ASSERT\_EQ(val, 105);

cout<<”Assertion Completed”<<endl;

}

Example 2

TEST(TestName, increment\_by\_5)

{

//Arrange

int val = 100;

int inc = 5;

//Act

val = val + inc;

//Assert

ASSERT\_EQ(val, 105);

cout<<”Assertion Completed”<<endl;

}

TEST(TestName, increment\_by\_10)

{

//Arrange

int val = 100;

int inc = 10;

//Act

val = val + inc;

//Assert

ASSERT\_EQ(val, 110);

cout<<”Assertion Completed”<<endl;

}

*In Example 2, the test cases are independent of one another (Even though the TestName is same, SubTestName is different). They require no external input as the values are set in Arrange.*

Example 3

#include<iostream>

#include<gtest/gtest.h>

using namespace std;

class MyClass

{

string id;

public:

MyClass(string \_id) : id(\_id) {}

string GetId() { return id; }

};

TEST(TestName, SubTest)

{

//Arrange

MyClass mc(“root”)

//Act

string value = mc.GetId();

//Assert

ASSERT\_STREQ(value.c\_str(), “root”);

}

int main()

{

testing::InitGoogleTest(&argc,argv);

return RUN\_All\_TESTS();

}

**Test Fixture**

* Avoids writing of similar Arrange part by maintaining a structure for Arrange.
* Structure will be automatically used every time a test fixture runs.
* The structure publicly inherits from testing::Test
* TEST\_F is used in place of TEST.

Example 1

#include<iostream>

#include<gtest/gtest.h>

using namespace std;

class MyClass

{

int baseValue;

public:

MyClass(int \_bv) : baseValue(\_bv) {}

void Increment(int byValue)

{

baseValue += byValue;

}

int getValue()

{

return baseValue;

}

};

struct MyClassTest : public testing::Test

{

MyClass \*mc;

void SetUp()

{

cout<<”SetUp”<<endl;

mc = new MyClass(100);

}

void TearDown()

{

cout<<”TearDown”<<endl; delete mc;

}

};

TEST\_F(MyClassTest, Increment\_By\_5)

{

//Act

mc->Increment(5);

//Assert

ASSERT\_EQ(mc->getValue(),105);

}

TEST\_F(MyClassTest, Increment\_By\_10)

{

//Act

mc->Increment(10);

//Assert

ASSERT\_EQ(mc->getValue(),110);

}

int main()

{

testing::InitGoogleTest(&argc,argv);

return RUN\_All\_TESTS();

}

**Invoking the Tests**

RUN\_ALL\_TESTS() should be invoked only once in main(). It  automatically detects and runs all the tests defined using the TEST macro. When invoked, the RUN\_ALL\_TESTS() macro:

* Saves the state of all googletest flags.
* Creates a test fixture object for the first test.
* Initializes it via SetUp().
* Runs the test on the fixture object.
* Cleans up the fixture via TearDown().
* Deletes the fixture.
* Restores the state of all googletest flags.
* Repeats the above steps for the next test, until all tests have run.
* If a fatal failure happens the subsequent steps will be skipped.

Command Prompt Usage

* --gtest\_list\_tests: Lists the names of all the tests available.
* --gtest\_filter=pattern: Run only the tests whose names match the regex pattern. For example: `--gtest\_filter="ConvolutionLayerTest.\*"
* --gtest\_shuffle: Run unit tests in random order.

--gtest\_output="xml:report.xml"  dump the output into XML format

--gtest\_repeat=2 --gtest\_break\_on\_failure on the command line, the same test is repeated twice. If the test fails, the debugger is automatically invoked.

--gtest\_filter=<test string>. The format for the test string is a series of wildcard patterns separated by colons (:) For example, --gtest\_filter=\* runs all tests while --gtest\_filter=SquareRoot\* runs only the SquareRootTest tests. If you want to run only the positive unit tests from SquareRootTest, use --gtest\_filter=SquareRootTest.\*-SquareRootTest.Zero\*. Note that SquareRootTest.\* means all tests belonging to SquareRootTest, and -SquareRootTest.Zero\* means don’t run those tests whose names begin with Zero.

--gtest\_also\_run\_disabled\_tests This runs the disabled tests also.

|  |  |
| --- | --- |
| --gtest\_shuffle --gtest\_filter=FooTest.\* | Runs everything in test case FooTest. And shuffle the test order. |

* ./foo\_test It has no flag and thus runs all its tests.
* ./foo\_test --gtest\_filter=\* Also runs everything, due to the single match-everything \* value.
* ./foo\_test --gtest\_filter=FooTest.\* Runs everything in a test suite FooTest .
* ./foo\_test --gtest\_filter=\*Null\*:\*Constructor\* Runs any test whose full name contains either "Null" or "Constructor" .
* ./foo\_test --gtest\_filter=-\*DeathTest.\* Runs all non-death tests.
* ./foo\_test --gtest\_filter=FooTest.\*-FooTest.Bar Runs everything in the test suite FooTest except FooTest.Bar.
* ./foo\_test --gtest\_filter=FooTest.\*:BarTest.\*-FooTest.Bar:BarTest.Foo Runs everything in the test suite FooTest except FooTest.Bar and everything in the test suite BarTest except BarTest.Foo

## Temporarily disabling tests

Simply add the DISABLED\_prefix to the logical test name or the individual unit test name and it won’t execute.

TEST(DISABLED\_ TestName, SubTestName)

**Command Line Argument**

#include "gtest/gtest.h"

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

{

::testing::InitGoogleTest(&argc, argv);

for (int i = 1; i < argc; ++i) {

printf("arg %2d = %s\n", i, argv[i]);

}

return RUN\_ALL\_TESTS();

}

//Run $ ./argrm\_test foo bar baz --gtest\_filter=foo