User Guide and Technical Reference Manual

wogtest

MicroConsult GmbH

Author: Remo Markgraf

Version: 2.0e

Date: 6th of February 2024

--- Intentionally left blank ---

Contents

1 Introduction

wogtest is a single include file embedded unit test framework for C and C++.

wogtest supports a substantial subset of the google test syntax.

wogtest aims at executing tests previously used under google test framework control on the PC on the target as well, without the need to migrate the google test framework to the target.

wogtest is provided free of charge to MicroConsult customers (please refer to License)

To comply with embedded requirements, wogtest is optimized to be applied

- without C++ streams (often too big for small controllers)
- without dynamic memory allocation on the heap (due to MISRA recommendations)
- also in a C-language subset of functionality in cases where no C++ compiler is available

2 Installation and Examples

wogtest can be downloaded at

https://www.microconsult.de/wogtest

The quarterly MicroConsult newsletter indicates when a new release is available for download. For your convenience, you are welcome to subscribe to the newsletter to avoid polling for new releases.

www.microconsult.de/newsletter

wogtest is provided as a single ZIP compressed file: wogtest.zip

wogtest.zip contains the following directory/file structure:

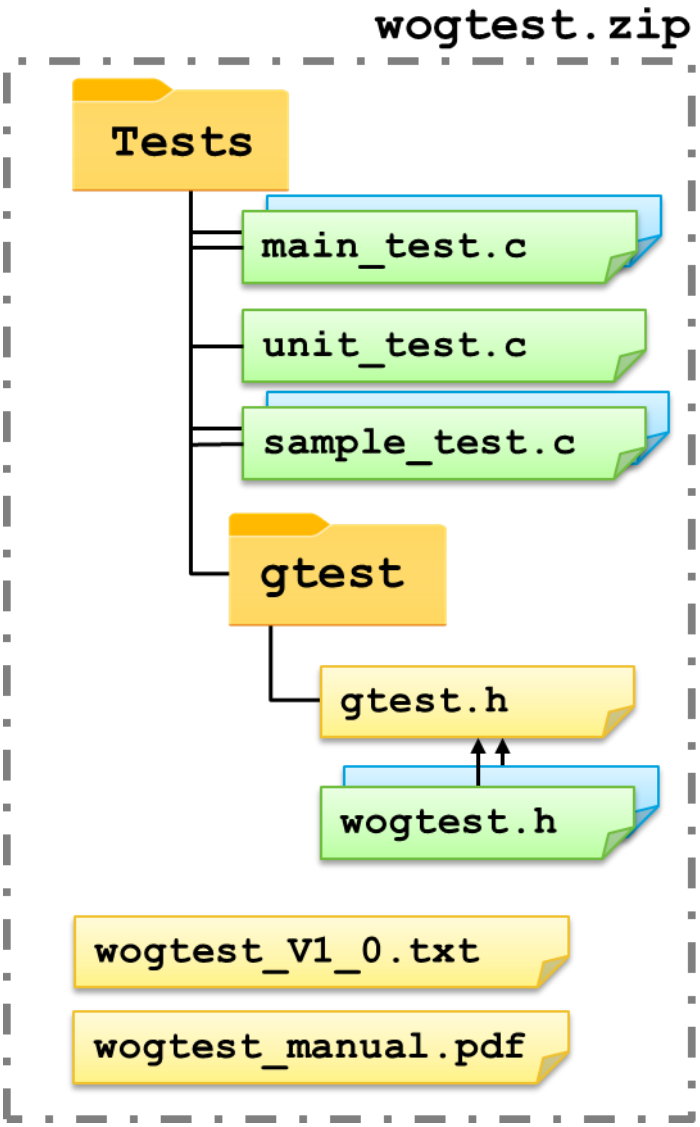


Figure 1: wogtest.zip file structure

The file wogtest_manual.pdf represents this document. Wogtest_V***r_vp***.txt is an ASCII file which is intentionally left blank and may contain last minute hints where ***r*** represents the release, ***v*** the version and ***p*** an optional patch level as a single lowercase character.

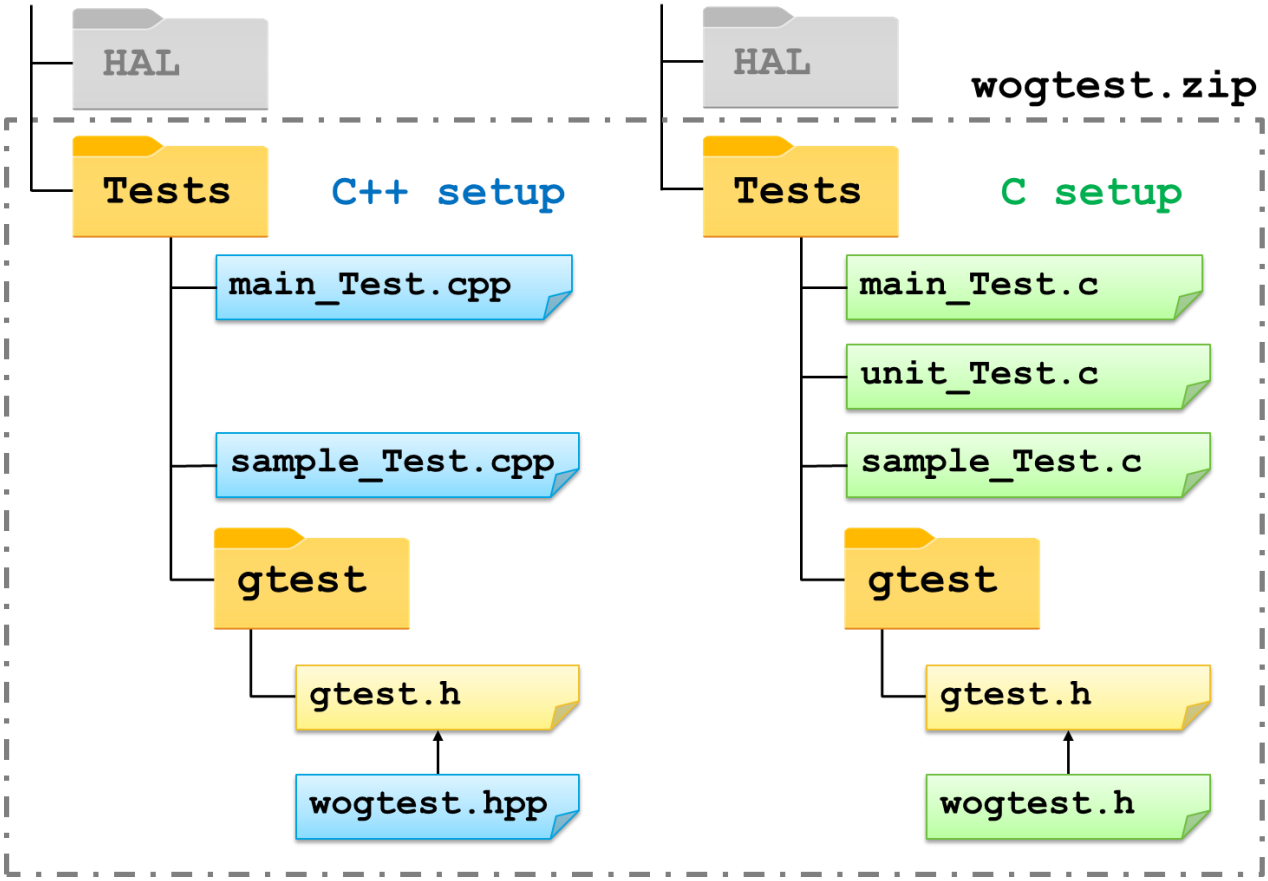


Figure 2: wogtest project structure

In the application context files may be copied into the project structure of the project under test. .cpp and .hpp files are used in the C++ setup, whereas .c and .h files are used in the C setup. The gtest/gtest.h file includes the wogtest.h/wogtest.hpp depending on the C/C++ compilation:

```
#ifndef GTEST_H_
#define GTEST_H_

#define ENABLE_TESTS

#ifdef __cplusplus
#include "wogtest.hpp"
#else
#include "wogtest.h"
#endif

#endif  //GTEST_H_
```

The ENABLE_TESTS macro is not used within wogtest and is intended for user purposes to switch on or off conditional compilations in the tests.

```
#include "gtest/gtest.h"
void foo(void)
{
#ifdef ENABLE_TESTS
   //do something only in test code
#endif //ENABLE_TESTS
}
```

2.1 Executing the Sample Tests

Please refer to the chapter 4 for details.

Potential steps to be performed to run the sample tests in C++ setup:
(Please refer to chapter 6.1 for executing the sample tests in a pure C setup.)

- Setup your project by replacing the productive main function by the main_test.cpp function.
- If tests were already written for google test and shall be re-executed under wogtest control, either place them in the Tests directory (see figure 1 above) or make sure that the wogtest header files may be accessed via the include path.
- For new tests write them into xyz_test.cpp files.
- A sample_test.cpp file is provided as part of the wogtest sources and may be used as template for own tests.
- Copy the gtest directory with the gtest.h and wogtest.hpp header files included into the source path where your test sources main_test.cpp and xyz_test.cpp are located. Thus the include path “gtest/gtest.h” is resolved by the compiler.
- Make sure that in your project redirection of printf output to a terminal or debugger print viewer window properly works.
- Build and run the project and observe the test results.
- The test main function terminates with the return value 0 if all tests passed and 1 otherwise.

3 Supported Functionality

wogtest supports a substantial subset of the google test functions and macros while providing the same syntax.

Please refer to the list of macros/assertions/expectations for details.

The software under test can be written in C++, C or even Assembler.
It is assumed that the tests are written in C++ wherever possible.
In cases where no C++ compiler is available, a subset of functionality is available for tests in C (please refer to chapter 6 Testing in C for details).

Expectations and assertions can be extended by user output.

Example:

```
EXPECT_FALSE(true) << "assumed to fail" << " in line " << 14;
ASSERT_TRUE(false) << "assumed to fail in line " << 15;
```

3.1 User defined extensions

For strings and integers, the operator<< is predefined

```
Test_Assert& Test_Assert::operator<<(const char *str);
Test_Assert& Test_Assert::operator<<(const uint32_t num);
```

Further operator<< types may be added by the user in the wogtest.hpp include file.

If the USE_TEMPLATES is switched on

```
template<typename T> Test_Assert& operator<<(const T par)
{
    …
    gtests.append_message_buffer(par);
    …
```

A template is applied to generate the code and thus, only the gtests.append_message_buffer method which takes the new type as parameter needs to be provided to output the parameter to the output buffer.

3.2 Limitations

Currently not supported functions in comparison to google Test:

- Parameterized tests using TEST_P
  Not yet implemented, may be considered for future extensions.

- google Mock
  Since TESTs are expected to be executed on the target, the real HW and drivers can be used; absence of a mocking tool support therefore does not do any harm.
  Not yet implemented, may be considered for future extensions.

4 Designing Tests

4.1 TEST_MAIN

Since unit tests have to be executed in a specific test runner environment, you may either

- create your own test project including a main_test source and the units under test or
- replace the main function in your productive project with a main_test source

In either case, you have to add the sources containing your tests for the units under test.
wogtest main supports the same syntax as google test aiming at the possibility of using the test_main.cpp source for both google test and for wogtest.
The return value of the RUN_ALL_TESTS macro is also the same as in google test:

- 0 if all tests passed
- 1 otherwise

A sample main_test.cpp and main_test.c are included in the wogtest .zip file.

main_test.cpp sample:

```
#include "gtest/gtest.h"

GTEST_API_ int main(void)
{
    //initialize what ever is necessary to print out
    configHardware();

    testing::InitGoogleTest(); //remove this line in case of main_test.c

    return RUN_ALL_TESTS();
}
```

You may use as well

```
GTEST_API_ int main(int argc, const char *const argv[])
```

Please note: Parameters are ignored in the current version of wogtest.

4.2 Test Result Output

Your environment has to be configured for output redirection to display the ASCII test results.

Any terminal connected to a serail interface or even the debug print viewer is sufficient to display the test result. If evaluation boards are used at an early test stage, where the target probably does not exist yet, theses boards often feature a debugger chip providing tunneling of an UART interface to a virtual Com port via USB.

When printf works, the requirements to output wogtest results are fulfilled 😊

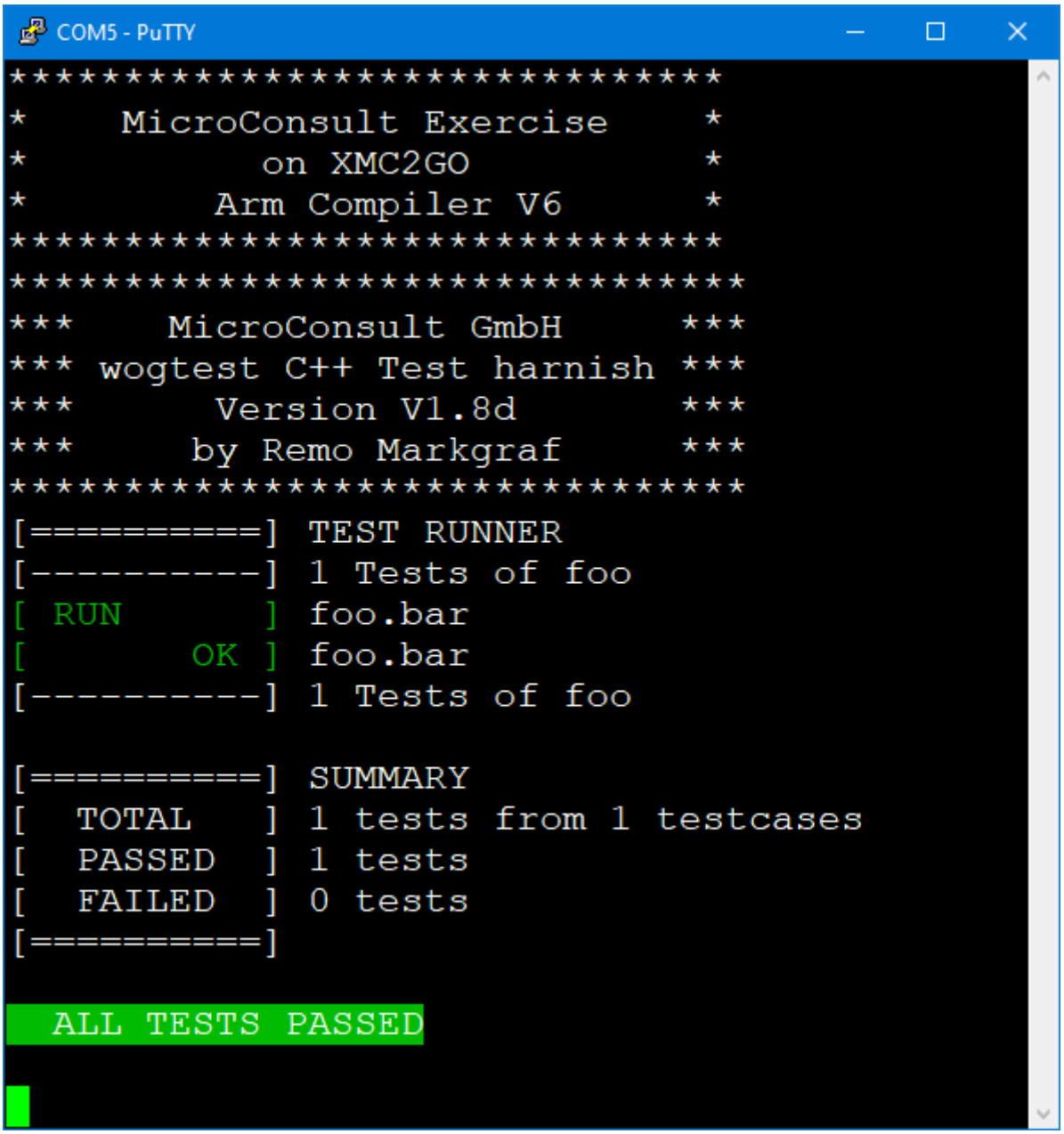
wogtest uses colored output, green for success and red for failure similar to google test.

Example:

```
#include "gtest/gtest.h"

TEST(foo, bar) {
  ASSERT_TRUE(1);
}
```

Result of test execution on putty:

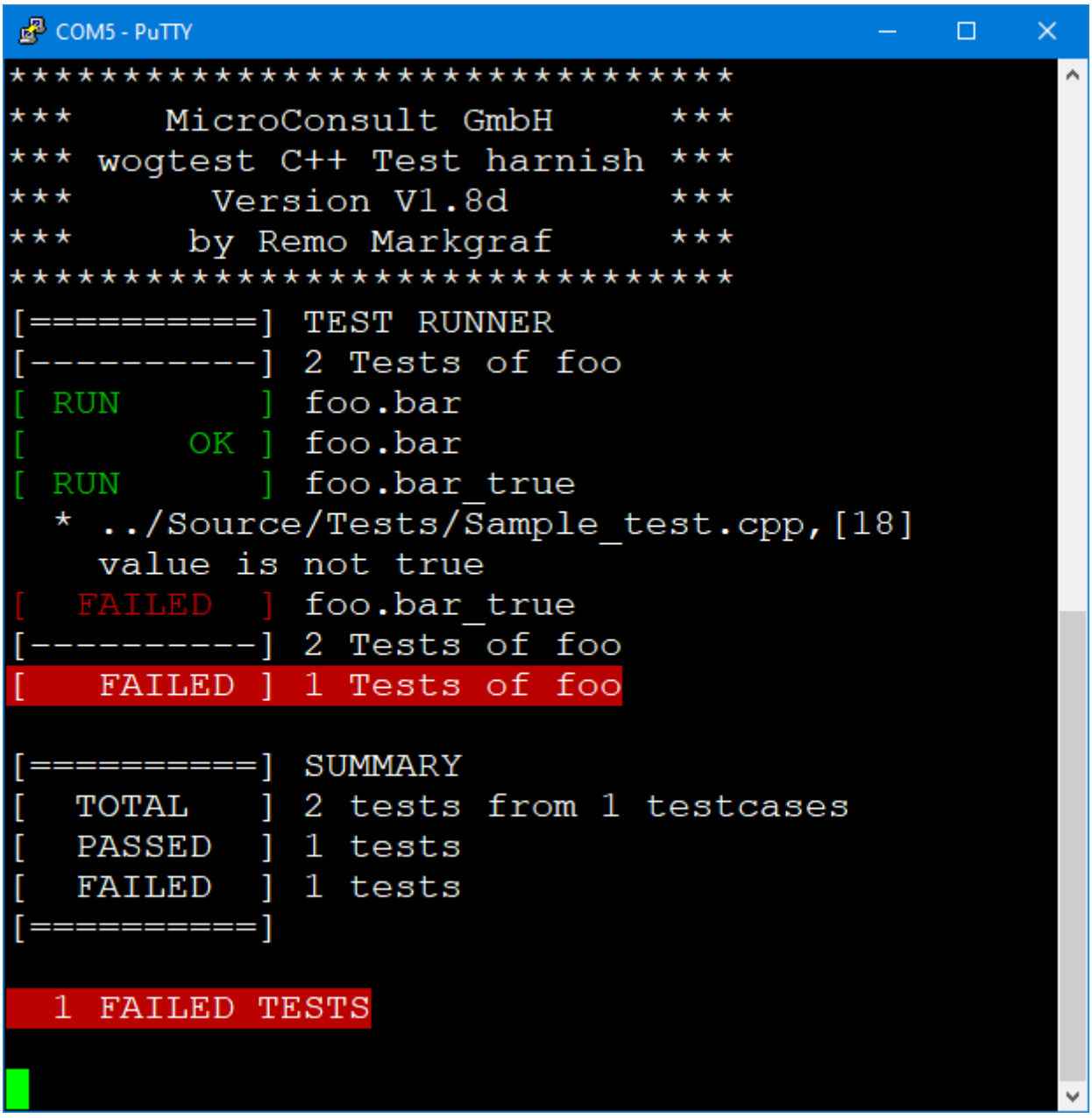


Example:

```
#include "gtest/gtest.h"

TEST(foo, bar) {
  ASSERT_TRUE(1);
}
TEST(foo, bar_true) {
  uint32_t i = 1;
  ASSERT_TRUE(i);  // pass!
  ASSERT_TRUE(42); // pass!
  ASSERT_TRUE(0);  // fail!
}
```

Result of test execution on putty:



4.3 Define a Test

To define a test to run, perform the following;

```
#include "gtest/gtest.h"

TEST(foo, bar) {
  ASSERT_TRUE(1);
}
```

A sample_test.cpp and sample_test.c are included in the wogtest .zip file.

The TEST macro takes two parameters - the first is the name of the test case, the second is the unique name of the test. A test case is a group of tests to which the test belongs.

Neither name has to be declared in advance, just type in any name you like to, as long as the name complies with C/C++ naming conventions for variables.

You will find a summary for every test case in the test output, and, inside the test case, a test result for every test belonging to that test case.

4.4 wogtest Configurations

wogtest offers the possibility of configuring several parameters.
The default configuration can be changed by editing constants at the beginning of the include files wogtest.hpp for C++ environments, resp. wogtest.h for C environments. Since the wogtest.h file is only used in a pure C setup, changes do not influence the wogtest C++ behaviour defined in wogtest.hpp and vice versa.

4.4.1 wogtest Configurations in a C++ environment

The default configuration can be changed by editing constants at the beginning of the include file wogtest.hpp for C++ environments:

```
//---------------------------------------
//select one of
// <stdint.h> to use "printf" or
// <iostream> to use "cout <<"
//---------------------------------------
#include <stdio.h> //use "printf"
//#include <iostream>  //use "cout <<"

//---------------------------------------
//define max number of characters of error messages
//per ASSERT/EXPECT including your << addons
//no overflow check is performed
//one byte on stack required per character
//---------------------------------------
#define WOGTEST_MESSAGEBUFFERSIZE 1024
//---------------------------------------
//configure if heap should be used for testcases,
//if yes, define TESTCASES_NUM to DYNAMIC
//else,   define TESTCASES_NUM to the number of maximal possible testcases
//in this case a static array of testcases is declared.
//12 bytes for every array element are required in RAM (not Stack)
//default is 20 testcases, thus 240 bytes of RAM
//Note: testcases is the number of groups of TESTS or TEST_F
//---------------------------------------
//#define WOGTESTCASES_NUM DYNAMIC
#define WOGTESTCASES_NUM 20
//---------------------------------------
//configure if float/double should be testable,
//if yes, remove comment from define WOGTEST_FLOAT
//in this case
//---------------------------------------
#define WOGTEST_FLOAT
//---------------------------------------
//Apply templates for operator<< output extensions
//if yes, remove comment from define USE_TEMPLATES
//---------------------------------------
#define USE_TEMPLATES
//---------------------------------------
//Apply printf for output of messages
//otherwise putchar is used
//only relevant when streams are NOT used
//if yes, remove comment from define USE_PRINTF
//---------------------------------------
//#define USE_PRINTF
```

4.4.2 wogtest Configurations in a pure C setup

The default configuration can be changed by editing constants at the beginning of the include file wogtest.h for pure C environments:

```
//---------------------------------------
#define WOGTEST_VERSION V2.0
//---------------------------------------
//---------------------------------------
//define max number of characters of error messages per ASSERT/EXPECT
//no overflow check is performed
//one byte on stack required per character
//---------------------------------------
#define WOGTEST_MESSAGEBUFFERSIZE 256

//---------------------------------------
//configure if float/double should be testable,
//if yes, remove comment from define WOGTEST_FLOAT
//in this case
//---------------------------------------
#define WOGTEST_FLOAT

//---------------------------------------
//for porting purposes to compiler/linker tool chains
//the WEAKFUNC macro may be adopted accordingly
//---------------------------------------
#define WEAKFUNC __attribute__((weak))
```

5 Testing Macros

Matching with the google test syntax, two variants of each error checking condition are provided - ASSERTs and EXPECTs. If an ASSERT fails, the test will cease execution and wogtest will continue with the next test to be run. If an EXPECT fails, the remaining test will still be executed, allowing for further checks to be performed.

Currently, the following macros are provided to be used within wogtests:

5.1 TEST

The TEST macro takes two parameters - the first is the name of the test case, the second is the unique name of the test. A test case is a group of tests to which the test belongs.

Neither name has to be declared in advance, just type in any name you like to, as long as the name complies with C/C++ naming conventions for variables.

You will find a summary for every test case in the test output, and, inside the test case, a test result for every test belonging to that test case.

Example:

```
TEST(foo1, bar1) {
    // TEST Body
}
TEST(foo1, bar2) {
    // TEST Body
}
TEST(foo2, bar1) {
    // TEST Body
}
```

Like google test, wogtest defines and instantiates a class for every TEST.
The class name is built from the test case name, the test name and _test.
In the above example:

```
foo1_bar1_test
foo1_bar2_test
foo2_bar1_test
```

These classes may be used in your test code, for example, in friend declarations to access private class attributes for testing purposes.
The test classes are derived from the class testing::Test and thus inherit all members of the class Test. Inheritance in the above example:

```
testing::Test -> foo1_bar1_test
testing::Test -> foo1_bar2_test
testing::Test -> foo2_bar1_test
```

5.2 TEST_F

The TEST_F macro takes two parameters - the first is the name of the test case, the second is the unique name of the test. A test case is a group of tests to which the test belongs.

The first parameter is also the name of a fixture class that needs to be declared before the first TEST_F macro referencing to that fixture is called.

Example:

```
class foo1 : public testing::Test
{
   //setup of test environment for foo1 tests
};
```

```
class foo2 : public testing::Test
{
   //setup of test environment for foo2 tests
};

TEST_F(foo1, bar1) {
    // TEST Body
}
TEST_F(foo1, bar2) {
    // TEST Body
}
TEST_F(foo2, bar1) {
    // TEST Body
}
```

5.2.1 Fixture Classes

The fixture classes may be used in your test code, for example, in friend declarations to access private class attributes for testing purposes.
A new instance of the fixture is instantiated for every TEST_F execution.

The test classes are derived from the corresponding fixture class, and the fixture class is derived from the class testing::Test. It therefore inherits all members of the class Test.
Inheritance in the above example:

```
testing::Test -> foo1 -> foo1_bar1_test
testing::Test -> foo1 -> foo1_bar2_test
testing::Test -> foo2 -> foo2_bar1_test
```

For the sake of separating testcase outputs, you may insert additional hierarchical levels:

```
class foo1 : public testing::Test
{
   //setup of test environment for foo1 tests
};
class group1 : public foo1 {};
class group2 : public foo1 {};

testing::Test -> foo1 -> group1 -> group1_bar1_test
testing::Test -> foo1 -> group2 -> group2_bar1_test
```

foo1 should thus include the “real” fixture, containing the code to initialize the tests while group1 and 2 are empty classes which split the output into a group1 summary and a group2 summary.

TEST and TEST_F macros can be mixed in the same xyz_test.cpp source file.

5.2.2 SetUp

The Test class contains an empty virtual method

```
virtual void SetUp(void) {}
```

This method may be overridden by the derived fixture.

After the constructor of the fixture is called, the SetUp method will be called during every execution of a corresponding TEST_F.

Example:

```
class foo1 : public testing::Test
{
   void SetUp(void) override {
        //setup for foo1 tests
   }
}
```

5.2.3 TearDown

The Test class contains an empty virtual method

```
virtual void TearDown(void) {}
```

This method may be overridden by the derived fixture.

Before the destructor of the fixture is called, the TearDown method will be called during every execution of a corresponding TEST_F.

Example:

```
class foo1 : public testing::Test
{
   void SetUp(void) override {
        //setup for foo1 tests
   }
   void TearDown(void) override {
        //Clean up of foo1 tests
   }
}
```

5.2.4 Order of SetUp and TearDown calls

The constructor, the destructor, the virtual SetUp and the virtual TearDown methods will be called in a fixed order during every test execution of a corresponding TEST_F fixture class.

Example:

```
class foo1 : public testing::Test
{
   foo1(void)
   {
      //constructor for foo1 tests
      //will be called 1st if defined
   }
   void SetUp(void) override
   {
      //Setup for foo1 tests
      //will be called 2nd if defined
   }

      //The run method of foo1 tests
      //is defined by the TEST_F macro,
      //contains the body of the corresponding TEST_F and
      //will be called 3rd (always defined)
      //Prototype of run is:
      //void foo1_testname_test::run(void)

   void TearDown(void) override
   {
      //CleanUp of foo1 tests
      //will be called 4th if defined
   }
   virtual ~foo1()
   {
      //destructor for foo1 tests
      //will be called 5th (last) if defined
   }

}
```

5.3 Boolean Testing

A failing **assertion** raises a **fatal failure**.
Execution of the current TEST is terminated, and the TEST is considered failed.

A failing **expectation** raises a **non-fatal failure**.
In this case the execution of the current TEST is continued, and the TEST is considered failed.

5.3.1 ASSERT_TRUE(x)

Asserts that x evaluates to true (e.g. non-zero).

Example:

```
TEST(foo, bar) {
  uint32_t i = 1;
  ASSERT_TRUE(i);  // pass!
  ASSERT_TRUE(42); // pass!
  ASSERT_TRUE(0);  // fail!
}
```

5.3.2 ASSERT_FALSE(x)

Asserts that x evaluates to false (e.g. zero).

Example:

```
TEST(foo, bar) {
  uint32_t i = 0;
  ASSERT_FALSE(i); // pass!
  ASSERT_FALSE(1); // fail!
}
```

5.3.3 EXPECT_TRUE(x)

Expects that x evaluates to true (e.g. non-zero).

Example:

```
TEST(foo, bar) {
  uint32_t i = 1;
  EXPECT_TRUE(i);  // pass!
  EXPECT_TRUE(42); // pass!
  EXPECT_TRUE(0);  // fail!
}
```

5.3.4 EXPECT_FALSE(x)

Expects that x evaluates to false (e.g. zero).

Example:

```
TEST(foo, bar) {
  uint32_t i = 0;
  EXPECT_FALSE(i); // pass!
  EXPECT_FALSE(1); // fail!
}
```

5.4 Integer Testing

A failing **assertion** raises a **fatal failure**.
Execution of the current TEST is terminated, and the TEST is considered failed.

A failing **expectation** raises a **non-fatal failure**.
In this case the execution of the current TEST is continued, and the TEST is considered failed.

5.4.1 ASSERT_EQ(x, y)

Asserts that x and y are equal.

Example:

```
TEST(foo, bar) {
  uint32_t a = 42;
  uint32_t b = 42;
  ASSERT_EQ(a, b);     // pass!
  ASSERT_EQ(a, 42);    // pass!
  ASSERT_EQ(42, b);    // pass!
  ASSERT_EQ(42, 42);   // pass!
  ASSERT_EQ(a, b + 1); // fail!
}
```

5.4.2 ASSERT_NE(x, y)

Asserts that x and y are not equal.

Example:

```
TEST(foo, bar) {
  uint32_t a = 42;
  uint32_t b = 13;
  ASSERT_NE(a, b);   // pass!
  ASSERT_NE(a, 27);  // pass!
  ASSERT_NE(69, b);  // pass!
  ASSERT_NE(42, 13); // pass!
  ASSERT_NE(a, 42);  // fail!
}
```

5.4.3 ASSERT_LT(x, y)

Asserts that x is less than y.

Example:

```
TEST(foo, bar) {
  uint32_t a = 13;
  uint32_t b = 42;
  ASSERT_LT(a, b);   // pass!
  ASSERT_LT(a, 27);  // pass!
  ASSERT_LT(27, b);  // pass!
  ASSERT_LT(13, 42); // pass!
  ASSERT_LT(b, a);   // fail!
}
```

5.4.4 ASSERT_LE(x, y)

Asserts that x is less than or equal to y.

Example:

```
TEST(foo, bar) {
  uint32_t a = 13;
  uint32_t b = 42;
  ASSERT_LE(a, b);   // pass!
  ASSERT_LE(a, 27);  // pass!
  ASSERT_LE(a, 13);  // pass!
  ASSERT_LE(27, b);  // pass!
  ASSERT_LE(42, b);  // pass!
```

```
  ASSERT_LE(13, 13); // pass!
  ASSERT_LE(13, 42); // pass!
  ASSERT_LE(b, a);   // fail!
}
```

5.4.5 ASSERT_GT(x, y)

Asserts that x is greater than y.

Example:

```
TEST(foo, bar) {
  uint32_t a = 42;
  uint32_t b = 13;
  ASSERT_GT(a, b);   // pass!
  ASSERT_GT(a, 27);  // pass!
  ASSERT_GT(27, b);  // pass!
  ASSERT_GT(42, 13); // pass!
  ASSERT_GT(b, a);   // fail!
}
```

5.4.6 ASSERT_GE(x, y)

Asserts that x is greater than or equal to y.

Example:

```
TEST(foo, bar) {
  uint32_t a = 42;
  uint32_t b = 13;
  ASSERT_GE(a, b);   // pass!
  ASSERT_GE(a, 27);  // pass!
  ASSERT_GE(a, 13);  // pass!
  ASSERT_GE(27, b);  // pass!
  ASSERT_GE(42, b);  // pass!
  ASSERT_GE(13, 13); // pass!
  ASSERT_GE(42, 13); // pass!
  ASSERT_GE(b, a);   // fail!
}
```

5.4.7 EXPECT_EQ(x, y)

Expects that x and y are equal.

Example:

```
TEST(foo, bar) {
  uint32_t a = 42;
  uint32_t b = 42;
  EXPECT_EQ(a, b);     // pass!
  EXPECT_EQ(a, 42);    // pass!
  EXPECT_EQ(42, b);    // pass!
  EXPECT_EQ(42, 42);   // pass!
  EXPECT_EQ(a, b + 1); // fail!
}
```

5.4.8 EXPECT_NE(x, y)

Expects that x and y are not equal.

Example:

```
TEST(foo, bar) {
  uint32_t a = 42;
  uint32_t b = 13;
```

```
  EXPECT_NE(a, b);   // pass!
  EXPECT_NE(a, 27);  // pass!
  EXPECT_NE(69, b);  // pass!
  EXPECT_NE(42, 13); // pass!
  EXPECT_NE(a, 42);  // fail!
}
```

5.4.9 EXPECT_LT(x, y)

Expects that x is less than y.

Example:

```
TEST(foo, bar) {
  uint32_t a = 13;
  uint32_t b = 42;
  EXPECT_LT(a, b);   // pass!
  EXPECT_LT(a, 27);  // pass!
  EXPECT_LT(27, b);  // pass!
  EXPECT_LT(13, 42); // pass!
  EXPECT_LT(b, a);   // fail!
}
```

5.4.10 EXPECT_LE(x, y)

Expects that x is less than or equal to y.

Example:

```
TEST(foo, bar) {
  uint32_t a = 13;
  uint32_t b = 42;
  EXPECT_LE(a, b);   // pass!
  EXPECT_LE(a, 27);  // pass!
  EXPECT_LE(a, 13);  // pass!
  EXPECT_LE(27, b);  // pass!
  EXPECT_LE(42, b);  // pass!
  EXPECT_LE(13, 13); // pass!
  EXPECT_LE(13, 42); // pass!
  EXPECT_LE(b, a);   // fail!
}
```

5.4.11 EXPECT_GT(x, y)

Expects that x is greater than y.

Example:

```
TEST(foo, bar) {
  uint32_t a = 42;
  uint32_t b = 13;
  EXPECT_GT(a, b);   // pass!
  EXPECT_GT(a, 27);  // pass!
  EXPECT_GT(27, b);  // pass!
  EXPECT_GT(42, 13); // pass!
  EXPECT_GT(b, a);   // fail!
}
```

5.4.12 EXPECT_GE(x, y)

Expects that x is greater than or equal to y.

Example:

```
TEST(foo, bar) {
```

```
  uint32_t a = 42;
  uint32_t b = 13;
  EXPECT_GE(a, b);   // pass!
  EXPECT_GE(a, 27);  // pass!
  EXPECT_GE(a, 13);  // pass!
  EXPECT_GE(27, b);  // pass!
  EXPECT_GE(42, b);  // pass!
  EXPECT_GE(13, 13); // pass!
  EXPECT_GE(42, 13); // pass!
  EXPECT_GE(b, a);   // fail!
}
```

5.5 String Testing

To test 0-terminated character strings dedicated EXPECT_STR… and ASSERT_STR… macros are provided.

When passing a null pointer as parameter to one of the EXPECT_STR… or ASSERT_STR… macros they return as failed.

5.5.1 ASSERT_STREQ(x, y)

Asserts that strings x and y are equal and have the same length.
Comparison is case sensitive: Upper and lower case characters are considered unequal.

Example:

```
TEST(foo, bar) {
  const char* a = "foo";
  const char* b = "bar";
  const char* c = "bar";
  const char* d = "bAr";
  const char* e = "barrel";
  ASSERT_STREQ(a, a); // pass!
  ASSERT_STREQ(b, b); // pass!
  ASSERT_STREQ(b, c); // pass!
  ASSERT_STREQ(a, b); // fail!
  ASSERT_STREQ(b, d); // did fail if previous line did not abort testing!
  ASSERT_STREQ(b, e); // did fail if previous line did not abort testing!
}
```

5.5.2 ASSERT_STRNE(x, y)

Asserts that strings x and y are not equal or have different lengths.
Comparison is case sensitive: Upper and lower case characters are considered unequal.

Example:

```
TEST(foo, bar) {
  const char* a = "foo";
  const char* b = "bar";
  ASSERT_STRNE(a, b); // pass!
  ASSERT_STRNE(a, a); // fail!
}
```

5.5.3 ASSERT_STRCASEEQ(x, y)

Asserts that strings x and y are equal and have the same length.
Comparison is NOT case sensitive: Upper and lower case characters are considered equal.

Example:

```
TEST(foo, bar) {
  const char* a = "foo";
```

```
  const char* b = "bar";
  const char* c = "bAr";
  ASSERT_STRCASEEQ(a, a); // pass!
  ASSERT_STRCASEEQ(b, b); // pass!
  ASSERT_STRCASEEQ(b, c); // pass!
  ASSERT_STRCASEEQ(a, b); // fail!
}
```

5.5.4 ASSERT_STRCASENE(x, y)

Asserts that strings x and y are not equal or have different lengths.
Comparison is NOT case sensitive: Upper and lower case characters are considered equal.

Example:

```
TEST(foo, bar) {
  const char* a = "foo";
  const char* b = "bar";
  const char* c = "bAr";
  ASSERT_STRCASENE(a, b); // pass!
  ASSERT_STRCASENE(a, a); // fail!
  ASSERT_STRCASENE(b, c); // did fail if previous line did not abort!
}
```

5.5.5 EXPECT_STREQ(x, y)

Expects that the strings x and y are equal and have the same length.
Comparison is case sensitive: Upper and lower case characters are considered unequal.

Example:

```
TEST(foo, bar) {
  const char* a = "foo";
  const char* b = "bar";
  const char* c = "bar";
  const char* d = "bAr";
  const char* e = "barrel";
  EXPECT_STREQ(a, a); // pass!
  EXPECT_STREQ(b, b); // pass!
  EXPECT_STREQ(b, c); // pass!
  EXPECT_STREQ(a, b); // fail!
  EXPECT_STREQ(b, d); // fail!
  EXPECT_STREQ(b, e); // fail!
}
```

5.5.6 EXPECT_STRNE(x, y)

Expects that the strings x and y are not equal or have different lengths.
Comparison is case sensitive; Upper and lower case characters are considered unequal.

Example:

```
TEST(foo, bar) {
  const char* a = "foo";
  const char* b = "bar";
  EXPECT_STRNE(a, b); // pass!
  EXPECT_STRNE(a, a); // fail!
}
```

5.5.7 EXPECT_STRCASEEQ(x, y)

Expects that the strings x and y are equal and have the same length.
Comparison is NOT case sensitive: Upper and lower case characters are considered equal.

Example:

```
TEST(foo, bar) {
  const char* a = "foo";
  const char* b = "bar";
  const char* c = "bAr";
  EXPECT_STRCASEEQ(a, a); // pass!
  EXPECT_STRCASEEQ(b, b); // pass!
  EXPECT_STRCASEEQ(b, c); // pass!
  EXPECT_STRCASEEQ(a, b); // fail!
}
```

5.5.8 EXPECT_STRCASENE(x, y)

Expects that the strings x and y are not equal or have different lengths.
Comparison is NOT case sensitive: Upper and lower case characters are considered equal.

Example:

```
TEST(foo, bar) {
  const char* a = "foo";
  const char* b = "bar";
  const char* c = "bAr";
  EXPECT_STRCASENE(a, b); // pass!
  EXPECT_STRCASENE(a, a); // fail!
  EXPECT_STRCASENE(b, c); // fail!
}
```

5.6 Floating Point Testing

Please note: Testing of float and double is only available when #define WOGTEST_FLOAT is configured at the top of the wogtest header include file. In this case wogtest adds its own functionality to compare 32-bit float and 64-bit double precision floating point numbers and to convert them into ASCII strings for output purposes.

Floats are output with 7 digits on the righthand side of the dot and a maximum of 13 digits on the lefthand side plus a signment in case of negative values, summing up to maximal 22 digits in total.
<-><max13digits>.<7digits>

Example: 42.0000991

Double are output with 15 digits on the right hand side of the dot and a maximum of 14 digits on the lefthand side plus an signment in case of negative values, summing up to maximal 31 digits in total.
<-><max14digits>.<15digits>

Example: 42.000011444091796

Exponential floating point format of output is not provided, thus in cases where the unsigned integer value of a floating point does not fit into an unsigned 32-bit resp. 64-bit integer an error message is displayed instead of the value. In case of positive overflow:
“ERROR:FLOAT_OVERFLOW" resp. “ERROR:DOUBLE_OVERFLOW" and in case of negative underflow "ERROR:FLOAT_UNDERFLOW” resp. "ERROR:DOUBLE_UNDERFLOW".
In the unlikely event that the buffer of the conversion overflows "ERROR:BUFFER_OVERFLOW" is returned instead of the floating point number in ASCII format.

Please note: Only the output functionality of failing tests is affected by these size limitations. Assertions and expectations do work (fail or pass) in full value range.

5.6.1 ASSERT_FLOAT_EQ(x, y)

Asserts that the single precision floating point values x and y are within 4 ULPs distance.

1 ULP refers to the least significant digit of the mantissa.
32-bit single precision floating points are composed of:

1 bit signment,

8 bit exponent and

23 bits of mantissa, where the 24th bit is always considered 1.

Example:

```
TEST(foo, bar) {
  float a = 42.0f;
  float b = 42.00001f;
  float c = 42.0001f;
  ASSERT_FLOAT_EQ(a, b); // pass!
  ASSERT_FLOAT_EQ(a, b); // pass!
  //a evaluates to 42.0000000
  //c evaluates to 42.0000991
  ASSERT_FLOAT_EQ(a, c); // fail!
}
```

5.6.2 ASSERT_DOUBLE_EQ(x, y)

Asserts that the double precision floating point values x and y are within 4 ULPs distance.

1 ULP refers to the least significant digit of the mantissa.
64-bit double precision floating points are composed of:

1 bit signment,

11 bit exponent and

52 bits of mantissa, where the 53rd bit is always considered 1.

Example:

```
TEST(foo, bar) {
  double a = 42.0f;
  double b = 42.000001f;
  double c = 42.00001f;
  ASSERT_DOUBLE_EQ(a, b); // pass!
  //a evaluates to 42.00000000000000
  //c evaluates to 42.00001144491796
  ASSERT_DOUBLE_EQ(a, c); // fail!
}
```

5.6.3 ASSERT_NEAR(x, y, epsilon)

Asserts that the double precision floating point values x and y are within epsilon distance of each other.

Example:

```
TEST(foo, bar) {
  double a = 42.0f;
  double b = 42.01f;
     //     a evaluates to 42.00000000000000
     //     b evaluates to 42.00999832133203
     // 0.01f evaluates to  0.00999999976482
  ASSERT_NEAR(a, b, 0.01f);  // pass!
     //     a evaluates to 42.00000000000000
     //     b evaluates to 42.00999832133203
```

```
    //0.001f evaluates to  0.00100000047497
  ASSERT_NEAR(a, b, 0.001f); // fail!
}
```

5.6.4 EXPECT_FLOAT_EQ(x, y)

Expects that the single precision floating point values x and y are within 4 ULPs distance.

1 ULP refers to the least significant digit of the mantissa.
32-bit single precision floating points are composed of:

1 bit signment,

8 bit exponent and

23 bits of mantissa, where the 24th bit is always considered 1.

Example:

```
TEST(foo, bar) {
  float a = 42.0f;
  float b = 42.00001f;
  float c = 42.0001f;
  EXPECT_FLOAT_EQ(a, b); // pass!
  EXPECT_FLOAT_EQ(a, b); // pass!
  //a evaluates to 42.0000000
  //c evaluates to 42.0000991
  EXPECT_FLOAT_EQ(a, c); // fail!
}
```

5.6.5 EXPECT_DOUBLE_EQ(x, y)

Expects that the double precision floating point values x and y are within 4 ULPs distance.

1 ULP refers to the least significant digit of the mantissa.
64-bit double precision floating points are composed of:

1 bit signment,

11 bit exponent and

52 bits of mantissa, where the 53rd bit is always considered 1.

Example:

```
TEST(foo, bar) {
  double a = 42.0f;
  double b = 42.000001f;
  double c = 42.00001f;
  EXPECT_DOUBLE_EQ(a, b); // pass!
  //a evaluates to 42.00000000000000
  //c evaluates to 42.00001144491796
  EXPECT_DOUBLE_EQ(a, c); // fail!
}
```

5.6.6 EXPECT_NEAR(x, y, epsilon)

Expects that the double precision floating point values x and y are within epsilon distance of each other.

Example:

```
TEST(foo, bar) {
  double a = 42.0f;
  double b = 42.01f;
    //     a evaluates to 42.00000000000000
```

```
    //     b evaluates to 42.00999832133203
    // 0.01f evaluates to  0.00999999976482
  EXPECT_NEAR(a, b, 0.01f);  // pass!
    //     a evaluates to 42.00000000000000
    //     b evaluates to 42.00999832133203
    //0.001f evaluates to  0.00100000047497
  EXPECT_NEAR(a, b, 0.001f); // fail!
}
```

5.7 Exception Testing

Exceptions need to be configured in your development environment and compiler.
Please refer to your compiler user manual.

For Arm µVision Keil, you need to add the option **-fexceptions** in the Compiler Misc Controls

5.7.1 ASSERT_THROW(x, exception_type)

Asserts that exception_type will be thrown when code x is executed.

Example:

```
class foo
{
public:
    class fooExc { };
    class barExc { };
    void bar(int value) { if( value == 1) throw fooExc(); }
};

TEST(foo, bar)
{
    foo fooObj;
    ASSERT_THROW(fooObj.bar(1), foo::fooExc); // pass!
    ASSERT_THROW(fooObj.bar(1), foo::barExc); // fail!
}
```

5.7.2 ASSERT_NO_THROW(x)

Asserts that no exception will be thrown when code x is executed.

Example:

```
class foo
{
public:
    class fooExc { };
    class barExc { };
    void bar(int value) { if( value == 1) throw fooExc(); }
};

TEST(foo, bar)
{
    foo fooObj;
    ASSERT_NO_THROW(fooObj.bar(0) ); // pass!
    ASSERT_NO_THROW(fooObj.bar(1) ); // fail!
}
```

5.7.3 ASSERT_ANY_THROW(x);

Asserts that an exception of any exception_type will be thrown when code x is executed.

Example:

```
class foo
```

```
{
public:
    class fooExc { };
    class barExc { };
    void bar(int value) { if( value == 1) throw fooExc(); }
};

TEST(foo, bar)
{
    foo fooObj;
    ASSERT_ANY_THROW(fooObj.bar(1) ); // pass!
    ASSERT_ANY_THROW(fooObj.bar(0) ); // fail!
}
```

5.7.4 EXPECT_THROW(x, exception_type)

Expects that exception_type will be thrown when code x is executed.

Example:

```
class foo
{
public:
    class fooExc { };
    class barExc { };
    void bar(int value) { if( value == 1) throw fooExc(); }
};

TEST(foo, bar)
{
    foo fooObj;
    EXPECT_THROW(fooObj.bar(1), foo::fooExc); // pass!
    EXPECT_THROW(fooObj.bar(1), foo::barExc); // fail!
}
```

5.7.5 EXPECT_NO_THROW(x)

Expects that no exception will be thrown when code x is executed.

Example:

```
class foo
{
public:
    class fooExc { };
    class barExc { };
    void bar(int value) { if( value == 1) throw fooExc(); }
};

TEST(foo, bar)
{
    foo fooObj;
    EXPECT_NO_THROW(fooObj.bar(0) ); // pass!
    EXPECT_NO_THROW(fooObj.bar(1) ); // fail!
}
```

5.7.6 EXPECT_ANY_THROW(x);

Expects that an exception of any exception_type will be thrown when code x is executed.

Example:

```
class foo
{
public:
```

```
    class fooExc { };
    class barExc { };
    void bar(int value) { if( value == 1) throw fooExc(); }
};

TEST(foo, bar)
{
    foo fooObj;
    EXPECT_ANY_THROW(fooObj.bar(1) ); // pass!
    EXPECT_ANY_THROW(fooObj.bar(0) ); // fail!
}
```

5.8 Test Execution Macros

5.8.1 SUCCEED

Always passes, is intended for future google test extensions and may currently be used to output information into the test result.

Example:

```
TEST(foo, bar)
{
    SUCCEED() << "Hello World"; // pass!
}
```

5.8.2 FAIL

Generates a fatal failure, further test execution is therefore aborted.

Example:

```
TEST(foo, bar)
{
    FAIL() << "Start here"; // fail!

    //never each this line
}
```

5.8.3 ADD_FAILURE

Generates a non-fatal failure; the further test is therefore executed.

Example:

```
TEST(foo, bar)
{
    ADD_FAILURE() << "Start here"; // fail!

    //this line is reached
}
```

5.8.4 ADD_FAILURE_AT(file,line)

Generates a non-fatal failure at the file and line number specified.

Example:

```
TEST(foo, bar)
{
    ADD_FAILURE_AT("myfile",123) << "Start here"; // fail!

    //this line is reached
}
```

5.8.5 HasFatalFailure

Returns true if the current test has a fatal failure.

Prototype: `bool HasFatalFailure(void);`

Example:

```
void foo(void)
{
    ASSERT_EQ(1,2); //fails!
}

TEST(foo, bar)
{
    foo();
    if( HasFatalFailure() )
        return;

    //this line is reached only in case of no fatal failure
}
```

5.8.6 HasNonfatalFailure

Returns true if the current test has a non-fatal failure.

Prototype: `bool HasNonFatalFailure(void);`

Example:

```
void foo(void)
{
    EXPECT_EQ(1,2); //fails!
}

TEST(foo, bar)
{
    foo();
    if( HasNonfatalFailure() ) return;

    //this line is reached only in case of no nonfatal failure
}
```

5.8.7 HasFailure

Returns true if the current test has any failure, either fatal or non-fatal.

Prototype: `bool HasFailure(void);`

Example:

```
void foo(void)
{
    EXPECT_EQ(1,2); //fails!
    ASSERT_EQ(1,2); //fails!
}

TEST(foo, bar)
{
    foo();
    if( HasFailure() ) return;

    //this line is reached only in case of no failure
}
```

6 Testing in C

In cases where there is no C++ compiler available on the intended target, wogtest provides a subset of functionality in a pure C environment. The tests and the software under test may thus be written in C, without the need for C++. Nevertheless, the application scenario to write tests under control of google test on the PC and the re-run of the tests under wogtest control on the target is supported without the need to edit the tests. In this case it is recommended to take the later use in a pure C environment into account from the very beginning to restrict the used google test features to those applicable in a C environment and supported by wogtest as depicted in this chapter.

Consequently, features as described in chapter 6.4 which utilize C++ capabilities are not available in this pure C setup of wogtest.

6.1 Writing a TEST in pure C setup

Please refer to the chapter 4.

List of potential steps to be performed to run tests in a pure C setup:

- Setup your project by replacing the productive main function by the main_test.c function.
- Add a unit_test.c module where the used testcases are defined by TESTCASE macros.
- If tests were already written for google test and shall be re-executed under wogtest control in a pure C setup, rename the xyz_test.cpp files which contain the written tests into xyz_test.c files.
- For new tests write them into xyz_test.c files.
- A sample_test.c file is provided as part of the wogtest sources and may be used as template for own tests.
- Copy the gtest directory with the gtest.h and wogtest.h header files included into the source path where your test sources main_test.c, unit_test.c and xyz_test.c are located. Thus the include path “gtest/gtest.h” is resolved by the compiler.
- Make sure that in your project redirection of printf output to a terminal or debugger print viewer window properly works.
- Build and run the project and observe the test results.
- The test main function terminates with the return value 0 if all tests passed and 1 otherwise.

6.2 Supported features in pure C setup

A substantial subset of wogtest features are also supported in a pure C setup.

These features are supported in a pure C setup:

- TEST macro as described in chapter 5.1 with the exception that the instruction `testing::InitGoogleTest`(); in the main function needs to be removed
- Boolean ASSERT and EXCEPT Macros as described in chapter 5.3
- Integer ASSERT and EXCEPT Macros as described in chapter 5.4
- String ASSERT and EXCEPT Macros as described in chapter 5.5
- Floating point ASSERT and EXCEPT Macros as described in chapter 5.6
- Test execution Macros as described in in chapter 5.8

6.3 Additional features

A few features are added to wogtest in a pure C setup to compensate shortcomings of the restricted C language capabilities.

These features are: added in a pure C setup:

- As a substitution to the non-supported fixtures dedicated SetUp and TearDown functions are automatically called before and after every test execution.
- SetUp and TearDown functions may optionally be defined per testcase and therefore shared between several tests.
- The TESTCASE macro needs to be used to define a new testcase in a source file, which does not include TEST macros. Hence a separate Unit_test.c source file is recommended.

6.3.1 SetUp Functions

SetUp functions may optionally be defined per testcase and therefore be shared between several tests of the same testcase. The name of the corresponding function is composed from the testcase name and _SetUp. The SetUp function is automatically called before every test execution and thus does not need to be explicitly called by the tests.

Eample:

```
void foo1_SetUp(void){};
void foo2_SetUp(void){};

TEST(foo1, bar)
{
    //foo1_SetUp() is automatically called, no need to add it here
    EXPECT_...
}
```

6.3.2 TearDown Functions

TearDown functions may optionally be defined per testcase and therefore be shared between several tests of the same testcase. The name of the corresponding function is composed from the testcase name and _TearDown. The TearDown function is automatically called after every test execution and thus does not need to be explicitly called by the tests.

Eample:

```
void foo1_TearDown(void){//TearDown body executed after test eecution};
void foo2_TearDown(void){//TearDown body executed after test eecution };

TEST(foo1, bar)
{
    EXPECT_...
    //foo1_TearDown() is automatically called, no need to add it here
}
```

6.3.3 TESTCASE Macro

Every usage of a new testcase name in TEST macros need be accompanied by a corresponding testcase definition by a TESTCASE macro.

Example for file Unit_test.c:

```
#include "gtest/gtest.h"
TESTCASE(foo1);
TESTCASE(foo2);
```

Example for file foo_test.c:

```
#include "gtest/gtest.h"
TEST(foo1, bar1){ //body of foo1_bar1_test
}
TEST(foo1, bar2){ //body of foo1_bar2_test
}
```

```
TEST(foo2, bar1){ //body of foo2_bar1_test
}
```

6.4 Limitations of a pure C setup

Not supported features in a pure C setup:

- Test fixture classes and the corresponding TEST_F macro including the Constructor, SetUp, TearDown and Destructor methods.
- Exception Testing including the corresponding assertions and expectations
- << output extensions by user messages
- Automatic detection of testcases is replaced by the TESTCASE macro defined in chapter 6.3.3. If testcases are not defined by the TESTCASE macro, their corresponding tests are not executed during the test run.
- Automatic detection of tests defined by the TEST macro is based on weak attributes which need to be considered by the linker. For any testcase, defined by the TESTCASE macro, 100 tests are created as empty weak functions, which may be overridden by the body of tests defined by TEST macros. The storage space for testcases is by default limited to 20 testcases. The wogtest.h file may be adapted by the user to deviate from the default values of 100 tests per testcase and 20 testcases. Since the wogtest.h file is only used in a pure C setup, changes do not influence the wogtest C++ behaviour defined in wogtest.hpp and vice versa.
- For porting purposes the weak attribute is defined as macro in the configuration section at the top of the wogtest.h header file. Please refer also to chapter 4.4.2

7 License

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT TO THIS SOFTWARE**.** IN NO EVENT SHALL THE AUTHORS OR MicroConsult GmbH BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

8 History of Change

V2.0c 9th of January 2024

V2.0d 2nd of February 2024
HoC added as chapter 8.
All new lines in text outputs are now preceded by an explicit carriage return.
Terminal SW which is not adding CR automatically is now supported as well.

V2.0e 6th of February 2024
inlining of some static functions to get rid of warnings in case of GCC
Destructor of class Test made virtual to ensure proper destruction of fixture classes

MicroConsult GmbH

Remo Markgraf