C++ Unit Testing Tool

Saurav Bhattarai (June 14, 2020)

Contents

1 Introduction

A unit testing tool will (obviously...) make unit testing easier.

Unit testing, in short, is one of the testing techniques which tests each individual units (example: functions, methods) to see if they are ready to use.

Read more on unit tests here.

For this document, I will be talking about an open source tool called Google Test (also known as gtest). It was created with Google's specific requirements and constraints in mind, so there's lot to discover. Let's get started.....

2 Installation and Prerequisites

As root do

```
dnf -y install gtest-devel
```

You can test your installation by doing

```
find /usr/include -name "gtest.h" -type f
```

If you see some results, you're good to go.

For this tutorial, I'll assume that you know C++ and you're familiar with classes and objects as well as pointers, however, first few sections won't require understanding of classes, objects and pointers – knowledge of functions will suffice.

3 Hello, Test!

Let's write a very simple C++ program that uses Google Test.

3.1 Initializing

First thing is to include the Google testing tool and initialize it in the main function. Following should be your template for all the programs:

```
#include <gtest/gtest.h>                // including the library

int main(int argc, char **argv)
{
    testing::InitGoogleTest(&argc, argv); // initializing it
    return RUN_ALL_TESTS();               // this will run all your tests
}
```

Compile this using:

```
g++ test.cpp -lgtest -lgtest_main -lpthread
```

Let's add a `factorial` function to the code.

```
#include <gtest/gtest.h>                // including the library

int factorial(int n)
{
    if (n < 0) return -1;   // invalid case
    int prod = 1;
    while (n != 0)
    {
        prod *= n--;
    }
    return prod;
}

int main(int argc, char **argv)
{
    testing::InitGoogleTest(&argc, argv); // initializing it
    return RUN_ALL_TESTS();               // this will run all your tests
}
```

3.2 Test a function

Now, let's test the `factorial`function. Add the `TEST` parts such that you final code looks like following:

```
#include <gtest/gtest.h>                // including the library

int factorial(int n)
{
    if (n < 0) return -1;   // invalid case
    int prod = 1;
    while (n != 0)
    {
        prod *= n--;
    }
    return prod;
}

TEST(FactorialTest, ZeroTest)  // for factorial of Zero
{
    ASSERT_EQ(1, factorial(0));
}
TEST(FactorialTest, PosTest)   // for factorial of Positive Numbers
{
    ASSERT_EQ(120, factorial(5));
    ASSERT_EQ(1, factorial(1));
    ASSERT_EQ(6, factorial(3));
}

int main(int argc, char **argv)
{
    testing::InitGoogleTest(&argc, argv); // initializing it
    return RUN_ALL_TESTS();               // this will run all your tests
}
```

Compile and run to see the results.

3.3 Breaking it down

Firstly, the general syntax of a `TEST` looks like this:

```
TEST(TestSuiteName, TestName)
{
    ... test body ...
}
```

Let's take one of the tests in our example and explain it line by line.

```
TEST(FactorialTest, ZeroTest)  // for factorial of Zero
{
    ASSERT_EQ(1, factorial(0));
}
```

First argument passed, `FactorialTest`, is the `TestSuiteName`, and the second argument, `ZeroTest`, is the `TestName`
A *test suite* groups related tests. In our case, we collected all tests related to the factorial function to a test suite called `FactorialTest`.

You will populate each of your *Test* with *assertions*, which are statements that will check whether a condition is true. A test is a failed test if it crashes or if it has a failed assertion. In our case, we assert that `factorial(0)` should be equal to 1.

Let's see the second test.

```
TEST(FactorialTest, PosTest)   // for factorial of Positive Numbers
{
    ASSERT_EQ(120, factorial(5));
    ASSERT_EQ(1, factorial(1));
    ASSERT_EQ(6, factorial(3));
}
```

Analyzing the header, this test belongs to the same test suite, `FactorialTest`, as the previous test.

Not surprisingly, a test can have multiple assertions. If any one of those assertion fails, the test `PosTest` fails as well.

3.4 ASSERT vs. EXPECT

You can also replace the `ASSERT_*` with `EXPECT_*`,however, there will be a slight difference in their performance. `ASSERT_*` will immediately abort the current test if any one of the `ASSERT_*` statement fails. On the contrary, `EXPECT_*` keeps on running even if one of the `EXPECT_*` fails. In our case, if `ASSERT_EQ(120, factorial(5))` fails, two assertion statements following this statement will not run, but they will run in the case of `EXPECT_*`.

4 Testing Pointers

Testing pointers is exactly similar to the previous syntax. Let's see an example.

```
void allocate(int *p)
{
    p = new int;
}
```

We have created an `allocate` function, which allocates memory to pointer `p` from the heap and now we want to test it if the memory was assigned.

The following will do the job:

```
TEST(AllocateTest, NullPtrReplacement)
{
    int *p = nullptr;
    allocate(p);
    EXPECT_NE(p, nullptr);
}
```

Note: While doing pointer comparisions, GoogleTest prefers `nullptr` over `NULL`. Read here for more details.

5 Testing methods in a class

This section will talk about how to test methods in a class.

5.1 Create a class

Let's create a simple class called `Person`. Paste the following code into a new *test.cpp* file:

```
#include <gtest/gtest.h> // including the library

// everything is made public to make this easy

class Person
{
public:
    Person()
      : age_(0)
    {}
    void add_age(int x)
    {
        age_ += x;
    }
    int age_;
};

int main(int argc, char **argv)
{
    testing::InitGoogleTest(&argc, argv); // initializing it
    return RUN_ALL_TESTS(); // this will run all your tests
}
```

Now let's test the methods of this class. You must have guessed that testing this would require us to create an object and call it's methods.

5.2 Testing constructors

For instance, the following would be one of the ways to test the constructor:

```
TEST(PersonTest, InitializationTest)
{
    Person p;
    EXPECT_EQ(p.age_, 0);
}
```

5.3 Testing other methods

For instance, the following would be one of the ways to test other methods:

```
TEST(PersonTest, AddAgeTest)
{
    Person p;
    p.add_age(10);
    EXPECT_EQ(p.age_, 10);
}
```

This will be it for testing methods in the class. Testing a customized object can be simplified further using *Test Fixture* which is discussed in the next section.

6 Test Fixture

In this section, we will talk about *Test Fixture*. As I have already mentioned, this is useful when we use same data configuration of objects for multiple tests.

6.1 Let's create and test a Singly Linked List class

Copy and paste the following code in a file called `linkedlist.cpp`:

```
#include <gtest/gtest.h> // including the library

// everything is made public to make this easy

template <typename T>
class Node
{
public:
    Node(T val, Node< T > *next=NULL)
        : val_(val), next_(next)
    {}
    T val_;
    Node< T > *next_;
};

template <typename T>
class LinkedList
{
public:
    LinkedList()
        : head_(NULL)
    {}
    ~LinkedList()
    {
        while (head_ != NULL)
        {
            Node < T > * head = head_->next_;
            delete head_;
            head_ = head;
        }
    }
```

```
    void print() const
    {
        Node< T > *cptr = head_;
        while (cptr != NULL)
        {
            std::cout << cptr->val_ << ' ';
            cptr = cptr->next_;
        }
        std::cout << std::endl;
    }
    void addtohead(T val)
    {
        head_ = new Node< T >(val, head_);
    }
    Node< T > * deletehead() // returns pointer to the deleted node
    {
        if (head_ == NULL) return NULL;
        Node< T > * head = head_;
        head_ = head_ -> next_;
        return head;
    }
    Node< T > *head_;
};

int main(int argc, char **argv)
{
    testing::InitGoogleTest(&argc, argv); // initializing it
    return RUN_ALL_TESTS(); // this will run all your tests
}
```

It should be clear that to test a *non-empty linked list* we need to create a linked list and add elements to it, and then test if the member functions work correctly.
Testing `addtohead` in a non-empty list:

```
TEST(LinkedListTest, AddToNonEmptyListTest)
{
    LinkedList< int > ll;
    ll.addtohead(5);
    ASSERT_NE(ll.head_, nullptr);
    EXPECT_EQ(5, ll.head_->val_);
}
```

6.2 Example of ASSERT vs. EXPECT

Notice that I used `ASSERT` for checking if the head is `nullptr`, because there is no point on continuing if the head is `nullptr`, because you'll be dereferencing a `nullptr` later.
Testing `deletehead` from a non-empty list:

```
TEST(LinkedListTest, DeleteFromNonEmptyListTest)
{
    LinkedList< int > ll;
    ll.addtohead(5);
    Node< int > *n = ll.deletehead();
    ASSERT_NE(n, nullptr);
    EXPECT_EQ(n->val_, 5);
    delete n;
}
```

However, the customization can be collected up and simplified. See next section.

6.3 Creating a test fixture class

We need to create a test fixture class and make our customized objects as the member of that class. To create a test fixture clas, we derive a class from `::testing::Test`.
Let's look at an example. For testing non-empty linked list, like our example in last section, we will do this:

```
class NonEmptyLinkedListTest : public ::testing::Test
{
public:
    NonEmptyLinkedListTest()
      : ll1(new LinkedList< int >)
    {
        ll1->addtohead(5);
    }
    ~NonEmptyLinkedListTest()
    {
        delete ll1;
    }
    LinkedList< int > * ll1;
};
```

6.4 Syntax and usage

Now, syntax for creating a test using test fixture looks like the following:

```
TEST_F(TestFixtureName, TestName)
{
    ... test body ...
}
```

Note: The TestFixtureName argument should be same as the Test Fixture class name that we define for customizing object.

In our case, our test looks like:

```
TEST_F(NonEmptyLinkedListTest, AddToHeadTest)
{
    ASSERT_NE(ll1->head_, nullptr);
    EXPECT_EQ(5, ll1->head_->val_);
}
```

Using `TEST_F()` will automatically create the text fixture object for you, therefore, you can directly refer to the members in it.

For testing `deletehead`,

```
TEST_F(NonEmptyLinkedListTest, DeleteHeadTest)
{
    Node< int > *n = ll1->deletehead();
    ASSERT_NE(n, nullptr);
    EXPECT_EQ(n->val_, 5);
    delete n;
}
```

Note: For each tests define with `TEST_F()`, google test creates a fresh test fixture object. In our case, an object of `NonEmptyLinkedLinkedTest` was freshly created for both `AddToHeadTest` and `DeleteHeadTest`.

Furthermore, we can also define any methods inside test fixture class we want our tests to use.

7 Wrapping Up

Let's wrap up with some extra information.

You should not be surprised to see different version of `ASSERT_*` and `EXPECT_*`. The table below mentions some others:

`ASSERT_*`	`EXPECT_*`	VERIFIES
`ASSERT_EQ(val1, val2)`	`EXPECT_EQ(val1, val2)`	`val1 == val2`
`ASSERT_NE(val1, val2)`	`EXPECT_NE(val1, val2)`	`val1 != val2`
`ASSERT_LT(val1, val2)`	`EXPECT_LT(val1, val2)`	`val1 < val2`
`ASSERT_GT(val1, val2)`	`EXPECT_GT(val1, val2)`	`val1 > val2`
`ASSERT_STREQ(val1, val2)`	`EXPECT_STREQ(val1, val2)`	the two C strings have the same content

This is all I have to say for Google Test. Of course, there are a lot of other features like parameterized testing, typed testing which you can learn more about. For more details, see the documentation for Google Test.
(General documentation: Link)
(Advanced documentation: Link)