Googletest Primer

★ Introduction: Why google test?

⇒ googletest helps you write better C++ tests.

⇒ googletest is a testing framework developed by the Testing Technology team with Google's specific requirements and constraints in mind.

⇒ It supports any kind of tests, not just unit tests.

⇒ We believe:

⇒ Tests should be independent and repeatable.

- → It's a pain to debug a test that succeeds or fails as a result of other tests.
- → googletest isolates the tests by running each of them on a different object.
- → When a test fails, googletest allows you to run it in isolation for quick debugging.

⇒ Tests should be well organized and reflect the structure of the tested code.

⇒ When tests fail, they should provide as much information about the problem as possible.

- → googletest doesn't stop at the first test failure.
- → Instead, it only stops the current test and continues with the next.
- → You can also set up tests that report non-fatal failures after which the current test continues.
- → Thus, you can detect and fix multiple bugs in a single run-edit-compile cycle.

★ Basic Concepts

⇒ When using googletest, you start by writing assertions, which are statements that check whether a condition is true.

⇒ An assertion's result can be:

- → Success
- → Nonfatal Failure
- → Fatal Failure → If a fatal failure occurs, it aborts the current function; otherwise the program continues normally.

⇒ Tests use assertions to verify the tested code's behavior.

- → If a test has a failed assertion, then it fails; otherwise it succeeds.

⇒ A test suite contains one or many tests.

- → You should group your tests into test suites that reflect the structure of the tested code.
- → When multiple tests in a test suite need to share common objects and subroutines, you can put them into a test fixture class.

⇒ A test program can contain multiple test suites.

★ Assertions

⇒ googletest assertions are macros that resemble function calls.

⇒ You test a class or function by making assertions about its behavior.

⇒ When an assertion fails, googletest prints the assertion's source file and line number location, along with a failure message.

- → You may also supply a custom failure message which will be appended to googletest's message

⇒ The assertions come in pairs that test the same thing but have different effects on the current function.

- → ASSERT_* ~ generate fatal failures when they fail, and abort the current function
- → EXPECT_* ~ generate nonfatalfailures, which don't abort the current function.

To provide a custom failure message, simply stream it into the macro using the << operator or a sequence of such operators.

```
ASSERT_EQ(x.size(), y.size()) << "Vectors x and y are of unequal length";
```

★ Basic Assertions

These assertions do basic true/false condition testing.

Fatal assertion	Nonfatal assertion	Verifies
`ASSERT_TRUE(condition);`	`EXPECT_TRUE(condition);`	`condition` is true
`ASSERT_FALSE(condition);`	`EXPECT_FALSE(condition);`	`condition` is false

★ Binary Comparision

This section describes assertions that compare two values.

Fatal assertion	Nonfatal assertion	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_LE(val1, val2);`	`EXPECT_LE(val1, val2);`	`val1 <= val2`

`ASSERT_GT(val1, val2);`	`EXPECT_GT(val1, val2);`	`val1 > val2`
`ASSERT_GE(val1, val2);`	`EXPECT_GE(val1, val2);`	`val1 >= val2`

⇒ ASSERT_EQ() does pointer equality on pointers.

- → If used on two C strings, it tests if they are in the same memory location, not if they have the same value.
- → Therefore, if you want to compare C strings (e.g. const char*) by value, use ASSERT_STREQ().
- → To compare two string objects, you should use ASSERT_EQ .

⇒ When doing pointer comparisons use *_EQ(ptr, nullptr) and *_NE(ptr, nullptr) instead of *_EQ(ptr, NULL) and *_NE(ptr, NULL) .

- → This is because nullptr is typed, while NULL is not.

⇒ If you're working with floating point numbers, you may want to use the floating point variations of some of these macros in order to avoid problems caused by rounding.

★ String Comparison

⇒ The assertions in this group compare two C strings.

Fatal assertion	Nonfatal assertion	Verifies
`ASSERT_STREQ(str1,str2);`	`EXPECT_STREQ(str1,str2);`	the two C strings have the same content
`ASSERT_STRNE(str1,str2);`	`EXPECT_STRNE(str1,str2);`	the two C strings have different contents
`ASSERT_STRCASEEQ(str1,str2);`	`EXPECT_STRCASEEQ(str1,str2);`	the two C strings have the same content, ignoring case
`ASSERT_STRCASENE(str1,str2);`	`EXPECT_STRCASENE(str1,str2);`	the two C strings have different contents, ignoring case

★ Simple Tests

⇒ To create a test, use the TEST() macro to define and name a test function.

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

{ Both names must be valid C++ identifiers, and they should not contain any underscores (_). }

⇒ A test's full name consists of its containing test suite and its individual name.

⇒ Tests from different test suites can have the same individual name.

⇒ googletest groups the test results by test suites, so logically related tests should be in the same test suite.

★ Test Fixtures

⇒ If you find yourself writing two or more tests that operate on similar data, you can use a test fixture.

⇒ This allows you to reuse the same configuration of objects for several different tests.

⇒ To create a fixture:

① Derive a class from ::testing::Test
   → Start its body with protected: , as we'll want to access fixture members from sub-classes.

② Inside the class, declare any objects you plan to use.

③ If necessary, write a default constructor or SetUp() function to preparethe objects for each test.
   → A common mistake is to spell SetUp() as Setup() with a small u.
   → Use override in C++11 to make sure you spelled it correctly.

④ If necessary, write a destructor or TearDown() function to release any resources you allocated in SetUp()

⑤ If needed, define subroutines for your tests to share.

⇒ When using a fixture, use TEST_F() instead of TEST() as it allows you to access objects and subroutines in the test fixture:

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

⇒ Like TEST() , the first argument is the test suite name, but for TEST_F() this must be the name of the test fixture class.

⇒ For each test defined with TEST_F()
- googletest will create a fresh test fixture at runtime
- immediately initialize it via SetUp()
- run the test
- clean up by calling TearDown()
- and then delete the test fixture.

⇒ Note that different tests in the same test suite have different test fixture objects, and googletest always deletes a test fixture before it creates the next one.

⇒ By convention, you should give fixture class the name FooTest where Foo is the class being tested.

★ Invoking the tests

⇒ TEST() and TEST_F() implicitly register their tests with googletest.

⇒ After defining your tests, you can run them with RUN_ALL_TESTS() , which returns 0 if all the tests are successful, or 1 otherwise.

⇒ Your main() function must return the value of RUN_ALL_TESTS()

★ Writing main function

```
int main(int argc, char **argv) {
  ::testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

⇒ The ::testing::InitGoogleTest() function parses the command line for googletest flags, and removes all recognized flags.