SET-221
Software Testing Technologies

LAB # 04

LAB Title

Writing Assertions in Google Test

Assessment of CLO: 04, PLO: 03

Student Name:			
Roll No.			
Semester		Session	

S. No.	Perf. Level Criteria	Excellent (2.5)	Good (2)	Satisfactory (1.5)	Needs Improvement (0 ~ 1)	Marks Obtained
1	Project Execution & Implementation	Fully functional, optimized, and well-structured.	Minor errors, mostly functional.	Some errors, requires guidance.	Major errors, non-functional, or not Performed.	
2	Results & Debugging Or Troubleshooting	Accurate results with effective debugging Or Troubleshooting.	Mostly correct, some debugging Or Troubleshooting needed.	Partial results, minimal debugging Or Troubleshooting.	Incorrect results, no debugging Or Troubleshooting, or not attempted.	
3	Problem-Solving & Adaptability (VIVA)	Creative approach, efficiently solves challenges.	Adapts well, minor struggles.	Some adaptability, needs guidance.	Lacks innovation or no innovation, unable to solve problems.	
4	Report Quality & Documentation	Clear, structured, with detailed visuals.	Mostly clear, minor gaps.	Some clarity issues, missing details.	Poorly structured, lacks clarity, or not submitted.	
Total Marks Obtained Out of 10						

Experiment evaluated by

Instructor's Name	Engr.Bushra Aziz		
Date		Signature	

Objective: Exploring various types of assertions in Google Test and Learn how to apply assertions to test C++ code.

Introduction

Google Test (GTest) is a unit testing framework for C++ that provides a wide range of assertions to verify the behavior of code. This lab manual covers different types of assertions used in Google Test, their syntax, and examples.

Definition of Assertion

An assertion is a statement used in software testing that checks if a given condition is true. Assertions help verify the correctness of a program by ensuring that expected and actual values match. If an assertion fails, it indicates a bug or an issue in the code.

Difference between Fatal and Non-Fatal Assertions

Google Test provides two categories of assertions:

1. **Fatal Assertions (ASSERT_)**: If a fatal assertion fails, the test function is immediately aborted, and no further checks in that test function are executed.
2. **Non-Fatal Assertions (EXPECT_)**: If a non-fatal assertion fails, the test continues executing the remaining statements in the test function.

Example

```
TEST(FatalNonFatalTest, Example) {

int x = 5;

int y = 10;

EXPECT_EQ(x, 5);  // Non-fatal assertion, test continues even if it fails

ASSERT_EQ(y, 10); // Fatal assertion, test aborts if it fails

// This line will not execute if ASSERT_EQ fails

   EXPECT_GT(y, x);

}
```

Types of Assertion in Google Test

1. Basic Assertions

Basic assertions check for equality or inequality between values. These are fundamental in verifying the correctness of variables and expressions.

I. Equality Assertions

- ASSERT_EQ(val1, val2): Checks if val1 is equal to val2.
- ASSERT_EQ(val1, val2): Same as ASSERT_EQ but stops execution on failure.

- ASSERT_NE(val1, val2): Checks if val1 is not equal to val2.

II. *Relational Assertions*

- ASSERT_LT(val1, val2): Checks if val1 is less than val2.
- ASSERT_GT(val1, val2): Checks if val1 is greater than val2.
- ASSERT_LE(val1, val2): Checks if val1 is less than or equal to val2.
- ASSERT_GE(val1, val2): Checks if val1 is greater than or equal to val2.

2. Boolean Assertions

Boolean assertions ensure that a condition evaluates to true or false.

- ASSERT_TRUE(condition): Checks if the condition is true.
- ASSERT_FALSE(condition): Checks if the condition is false.

3. Floating-Point Assertions

Floating-point assertions handle precision-related comparisons for floating-point numbers.

- ASSERT_FLOAT_EQ(val1, val2): Compares floating-point values for equality.
- ASSERT_DOUBLE_EQ(val1, val2): Compares double precision values for equality.
- ASSERT_NEAR(val1, val2, abs_error): Checks if val1 and val2 are approximately equal within abs_error tolerance.

4. String Assertions

String assertions compare string values, either in a case-sensitive or case-insensitive manner.

- ASSERT_STREQ(str1, str2): Checks if two C-style strings are equal.
- ASSERT_STRNE(str1, str2): Checks if two C-style strings are not equal.
- ASSERT_STRCASEEQ(str1, str2): Case-insensitive comparison of two strings.
- ASSERT_STRCASENE(str1, str2): Case-insensitive inequality comparison of two strings.

5. Exception Assertions

Exception assertions check if a particular piece of code throws or does not throw an expected exception.

- ASSERT_THROW(statement, exception_type): Checks if statement throws an exception of type exception_type.
- ASSERT_ANY_THROW(statement): Checks if statement throws any exception.
- ASSERT_NO_THROW(statement): Checks if statement does not throw any exception.

Calculator Program

Before testing, we need a simple calculator program that performs basic arithmetic operations.

Calculator Implementation

Calculator.cpp (main file)

```
#include <iostream>
#include "Cal.h"
#include <stdexcept>
using namespace std;
```

```
int main()
{
    std::cout << "Sample Calculator Application!\n";
    std::cout << "This is " << name() << std::endl;

        std::cout << "2 + 3 = " << add(2, 3) << std::endl;
        std::cout << "5 - 1 = " << subtract(5, 1) << std::endl;
        std::cout << "4 * 6 = " << multiply(4, 6) << std::endl;
        std::cout << "10 / 2 = " << divide(10, 2) << std::endl;
        std::cout << "10 / 0 = " << divide(10, 0) << std::endl;
        std::cout << "10 % 3 = " << modulus(10, 3) << std::endl;
        std::cout << "5 ^ 4 = " << power(5, 4) << std::endl;
        return 0;

    }
```

Cal.h

```
#pragma once
#include <cmath> // For pow()
#include <string>
using namespace std;
     string name();
    double add(double a, double b);
    double subtract(double a, double b);
    double multiply(double a, double b);
    double divide(double a, double b);
    double modulus(int a, int b);

    double power(double base, double exponent);
```

Cal.h

```
#include "Cal.h"
#include <stdexcept>
#include<string>
string name() {
    return "LabFour";
}

double add(double a, double b) {
    return a + b;
}

double subtract(double a, double b) {
    return a - b;
}

double multiply(double a, double b) {
    return a * b;
}

double divide(double a, double b) {
    if (b == 0) {
        //return 0;
        throw runtime_error("Division by zero");
    }
    return a / b;
}

double modulus(int a, int b) {
    if (b == 0) {
        return 0; // Or throw an exception
    }
    return a % b;
}

double power(double base, double exponent) {
    return std::pow(base, exponent);

}
```

Google Test for Calculator

Now, we will write a test program to verify the correctness of our calculator functions using different assertions.

Test File

```
#include "pch.h"
#include "C:\Users\baziz\source\repos\Calculator\Cal.cpp"

TEST(CalculatorTests, Testname) {

    ASSERT_EQ("LabFour", name());

}
TEST(CalculatorTest, Addition) {

    ASSERT_EQ(add(2, 3), 5);
    ASSERT_NE(add(4, 4), 9);
    ASSERT_GT(add(3, 2), 4);
}


// Test subtraction function
TEST(CalculatorTest, Subtraction) {

    ASSERT_EQ(subtract(10, 5), 5);
    ASSERT_LE(subtract(3, 3), 0);
    ASSERT_LT(subtract(2, 5), 0);
}

// Test multiplication function
TEST(CalculatorTest, Multiplication) {

    ASSERT_EQ(multiply(3, 4), 12);
    ASSERT_NE(multiply(-2, 5), 10);
    ASSERT_GE(multiply(5, 2), 10);
}

// Test division function
TEST(CalculatorTest, Division) {
    ASSERT_DOUBLE_EQ(divide(10, 2), 5.0);
    ASSERT_NEAR(divide(7, 2), 3.5, 0.01);
    ASSERT_TRUE(divide(9, 3) == 3.0);
}

// Test division by zero exception
TEST(CalculatorTest, DivisionByZero) {

    ASSERT_THROW(divide(5, 0), runtime_error);
    ASSERT_ANY_THROW(divide(1, 0));
}
```

Lab Exercise

1. Write a test case for modulus and power function.
2. Add more edge cases for division, including large numbers and floating-point precision.
3. Write any program that contain string data then test it using different string assertions such as ASSERT_STREQ and ASSERT_STRNE.