SET-221
Software Testing Technologies

LAB # 06

LAB Title

Parameterized Testing with Google Test Running the same test with different inputs.

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: To gain a comprehensive understanding of parameterized testing using Google Test, enabling efficient and thorough testing of the calculator class with various input combinations.

Theory:

Introduction to Parameterized Testing

Parameterized tests allow you to run the same test code with different input values. Instead of writing multiple test cases for each input combination, you define a single test case and provide a set of parameters. Google Test will then execute the test for each parameter set. This significantly reduces code duplication and improves test coverage.

Core Concepts of Google Test Parameterized Testing

1. **::testing::TestWithParam<ParamType>:** Base class for parameterized tests. ParamType defines parameter type (single value, tuple, custom).
2. **GetParam():** Retrieves the current parameter set within the test case.
3. **INSTANTIATE_TEST_SUITE_P(prefix, test_case_name, parameter_generator):** Instantiates the test suite.

    - prefix: Generated test name prefix.
    - test_case_name: Test fixture class name.
    - parameter_generator: Parameter set generator.

Built-in Parameter Generators:

1. **::testing::Values(val1, val2, ...):**

This generator produces a sequence of values directly specified as arguments. It's a simple, direct way to provide a fixed set of parameters. It essentially creates a static list of values, which are then iterated over in the order they are provided.

2. **::testing::ValuesIn(container):**

This generator produces a sequence of values from the elements of a container (e.g., std::vector, std::array).It leverages the iterator interface of the container to traverse its elements sequentially. This allows for dynamic parameter sets that can be modified at runtime.

```
//Example of ValuesIn

class ExampleValuesInTest : public ::testing::TestWithParam<std::string> {};

TEST_P(ExampleValuesInTest, ValuesInTest) {

 std::string str = GetParam();

 ASSERT_FALSE(str.empty());

                }

INSTANTIATE_TEST_SUITE_P(

                   ValuesInExample,

                  ExampleValuesInTest,
```

::testing::ValuesIn(std::vector<std::string>{"apple", "banana", "cherry"}));

3. **::testing::Range(begin, end, step):**

This generator produces a sequence of integer values within a specified range.It implements an arithmetic progression generator, starting from begin and incrementing by step until reaching (but not including) end. This is useful for generating sequences of numbers for loop-based tests or index-based tests.

```
//Example of Range

class ExampleRangeTest : public ::testing::TestWithParam<int> {};

TEST_P(ExampleRangeTest, RangeTest) {

int value = GetParam();

  ASSERT_TRUE(value % 2 == 0);

                }

INSTANTIATE_TEST_SUITE_P(

   RangeExample,

   ExampleRangeTest,

::testing::Range(0, 10, 2));
```

4. **::testing::Combine(gen1, gen2, ...):**

This generator produces the Cartesian product of the values generated by multiple input generators.It implements a nested iteration strategy, where each iteration of the outer generators is combined with all iterations of the inner generators. This is essential for exhaustive testing of functions with multiple input parameters.

```
class ExampleCombineTest : public ::testing::TestWithParam<std::tuple<int, std::string>> {};

TEST_P(ExampleCombineTest, CombineTest) {

                int num = std::get(GetParam());

                std::string str = std::get(GetParam());

                ASSERT_FALSE(str.empty());

                ASSERT_TRUE(num >= 0);

              }


              INSTANTIATE_TEST_SUITE_P(

                CombineExample,
```

```
ExampleCombineTest,

::testing::Combine(

   ::testing::Values(1, 2, 3),

   ::testing::Values("A", "B")));
```

5. **Custom Parameter Generators:**

   Google Test allows you to create custom parameter generators by implementing a generator class or function.This enables the generation of complex or dynamic parameter sets that are not easily produced by the built-in generators. Custom generators can:

   - Read data from files or databases.
   - Generate random or pseudo-random data.
   - Implement complex data transformations or combinations.
   - Implement user defined patterns.

   Implementation:

   - Custom generators typically involve defining an iterator-like interface that produces the next parameter set on each invocation.
   - This allows for the use of complex logic while still allowing for simple iteration by the google test framework.

6. **Tuples:** std::tuple groups multiple parameters.

Advantages of Parameter Generators:

- **Data-Driven Testing:** Generators facilitate data-driven testing by separating test data from test logic.
- **Test Coverage:** Generators enable the creation of comprehensive test data sets, improving test coverage.
- **Maintainability:** Generators simplify the management and modification of test data.
- **Reusability:** Generators can be reused across multiple test cases or test suites.
- **Readability:** Generators make test code more readable and easier to understand.

Example: Testing calculator application using Parameterized testing

// pt.cpp(main)

```
#include <iostream>
using namespace std;
#include <stdexcept>
#include "pt.h"
int main()
{
    calculator calc;
    cout << "Hello World!\n";
    cout << calc.Add(4.7, 2.5) << endl;
    cout << calc.Subtract(60, 90) << endl;
    cout << calc.Multiply(20, 12) << endl;
    cout << calc.Divide(10, 2) << endl;
    cout << calc.Divide(60, 0) << endl;


   }
```

// pt.h

```
class calculator {
public:
       double Add(double a, double b);
       double Subtract(double a, double b);
       double Multiply(double a, double b);
       double Divide(double a, double b);

   };
```

// ptest.cpp

```
#include "pt.h"
#include <stdexcept>
double calculator::Add(double a, double b) {
       return a + b;
}

double calculator:: Subtract(double a, double b)
{ return a - b;
}
double calculator:: Multiply(double a, double b) {
    return a * b;
}
double calculator:: Divide(double a, double b) {
    if (b == 0) {
        throw std::invalid_argument("Division by zero");
    }
    return a / b;
}
```

Test file:

//CalP_Test.cpp

```
#include "pch.h"
#include "C:\Users\baziz\source\repos\ft\fixt.cpp"
#include "gtest/gtest.h"
using namespace testing;
// calculator_test.cpp


// Define a parameterized test for the add function
class CalculatorTest : public ::testing::TestWithParam<std::tuple<int, int, int>> {
protected:
    calculator calc; // Instance of Calculator
};

// Test using the parameter set
TEST_P(CalculatorTest, Addition) {
    int a = std::get<0>(GetParam());  // First input
    int b = std::get<1>(GetParam());  // Second input
    int result = std::get<2>(GetParam()); // Expected result
    EXPECT_EQ(calc.Add(a, b), result); // Validate the addition
}

// Define the parameter list for testing
INSTANTIATE_TEST_CASE_P(AdditionT,CalculatorTest,
    ::testing::Values(
        std::make_tuple(1, 2, 3),
        std::make_tuple(3, 4, 7),
        std::make_tuple(-1, 1, 0),
        std::make_tuple(-5, -6, -11)
```

```
    ));

class CalculatorDivideTest : public ::testing::TestWithParam<std::tuple<double, double,
double>> {};

TEST_P(CalculatorDivideTest, Divide) {
    calculator calc;
    double a = std::get<0>(GetParam());
    double b = std::get<1>(GetParam());
    double expected = std::get<2>(GetParam());
    EXPECT_DOUBLE_EQ(expected, calc.Divide(a, b));
}
INSTANTIATE_TEST_CASE_P(DivideValues,CalculatorDivideTest,

    ::testing::Values(
        std::make_tuple(4.0, 2.0, 2.0),
        std::make_tuple(10.0, 5.0, 2.0),
        std::make_tuple(-9.0, 3.0, -3.0),
        std::make_tuple(5.0, 2.0, 2.5)
    ));
```

Task:

Test a function that checks whether a string is a palindrome. Use Google Test's parameterized testing features to test multiple cases efficiently.