**4-2 Milestone - Unit Testing**

The goal of this Milestone was to use a Google Test fixture to validate the functionality of the std::vector class. A total of 15-unit tests were created, including both positive and negative tests to thoroughly check the behavior of the vector under various conditions. The project was done using Google Test in Visual Studio 2022, with precompiled headers.

**Changes Made to the Code**

**Custom Exception Class:**

* A new class, CustomException, was added. It throws a custom error message ("Custom exception occurred!") when needed. This helps us manage specific error scenarios more effectively.

**Exception Handling in Application Logic:**

* The function do\_even\_more\_custom\_application\_logic() now throws a standard exception if something goes wrong. The do\_custom\_application\_logic() function tries to run it and catches any exceptions that occur, logging the error and then throwing our custom exception. This ensures that errors are handled correctly within the application.

**Division Logic:**

* A function called divide() handles division and throws an error if we try to divide by zero. The do\_division() function wraps this logic and handles any division errors, like catching the "divide by zero" problem.

**Collection Test Setup:**

* The CollectionTest fixture sets up a std::vector<int> to run various tests on the vector’s behavior, like adding elements and checking its size. We used SetUp() to prepare the vector before each test and TearDown() to clear it afterward, making sure every test starts with a clean slate.

**Positive Tests:**

* Tests like IsEmptyOnCreate and CanAddToEmptyVector were written to check if the vector behaves normally. For instance, we confirmed that a new vector starts empty and that elements can be added correctly.

**Negative Tests:**

* Negative tests, like ThrowsOutOfRangeWhenAtInvalidIndex, were created to see how the vector handles errors. These tests ensure the vector throws exceptions when you try to access or erase elements outside of its valid range, using ASSERT\_THROW() to check for these errors.

**Custom Tests:**

* Two custom tests were added:
  + CanInsertValueAtPosition: Ensures that we can insert a value at a specific spot in the vector.
  + AccessingInvalidIteratorThrows: Confirms that trying to access an invalid iterator throws an exception, and that erasing a valid range works as expected.

**Break Down of Screenshots**

1. **Test Environment Setup:**
   * The project was set up in Visual Studio 2022, and the **Google Test Adapter** was installed to run the tests within the IDE. Precompiled headers (pch.h) were included to manage project dependencies.
   * The test fixture, CollectionTest, was created to manage a std::vector<int> stored in a smart pointer (std::unique\_ptr). Each test within the fixture validated different behaviors of the vector.

*A screenshot of a computer program

Description automatically generated*

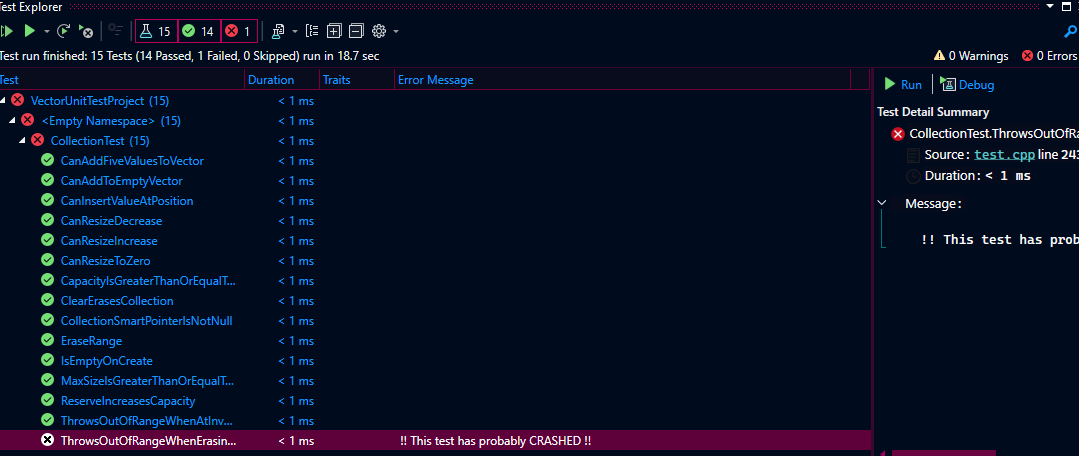
**Explanation:** This screenshot shows how the test environment was configured in Visual Studio, with the use of precompiled headers to optimize build times. The test class and functions were organized for easy management and execution of the unit tests.

1. **Positive Unit Tests:**
   * Several tests were written to validate that the vector behaves as expected when interacting with its size, capacity, and max size. For example:
     + **IsEmptyOnCreate:** Confirms that a newly created vector is empty.
     + **CanAddToEmptyVector:** Verifies that a value can be added to an empty vector.
     + **MaxSizeIsGreaterThanOrEqualToSize:** Ensures that the vector's maximum size is always greater than or equal to its current size.

A screenshot of a computer

Description automatically generated**Explanation:** This screenshot confirms that all tests, including positive and custom tests, passed successfully in the Test Explorer. The vector operations, including resizing, inserting values, and managing capacity, all worked as expected, providing validation that the code is functioning correctly.

1. **Negative Unit Tests:**
   * Negative tests were designed to check the behavior of the vector under invalid conditions, such as accessing an out-of-bounds index. For example:
     + **ThrowsOutOfRangeWhenAtInvalidIndex:** This test confirmed that calling at() with an index beyond the size of the vector throws a std::out\_of\_range exception.
     + **AccessingInvalidIteratorThrows:** Tested invalid iterator access and ensured that exceptions were thrown when attempting to access elements beyond valid bounds.



**Explanation:** This screenshot shows an issue encountered during testing, specifically when trying to erase elements using invalid iterators, leading to a crash. This shows the initial debugging step where the failure was identified.

1. **Custom Tests:**
   * Two additional custom tests were implemented to extend the test coverage:
     + **CanInsertValueAtPosition:** Verified that a value could be inserted at a specific position in the vector.
     + **AccessingInvalidIteratorThrows:** This test confirmed that accessing an invalid iterator triggers an appropriate exception without causing the program to crash.

*A screen shot of a computer code

Description automatically generated*

**Explanation:** This screenshot shows the successful implementation of the custom test for inserting a value into a specific position in the vector. It demonstrates how the test verifies correct behavior when inserting values into the vector.

A screen shot of a computer code

Description automatically generated

**Explanation:** This screenshot explains the custom test designed to check the vector’s behavior when attempting to access invalid iterators and ensuring that erasing valid ranges does not cause unexpected issues. It demonstrates exception handling for invalid access and validates proper range management.

1. **Debugging and Challenges:**
   * One of the main challenges was addressing crashes caused by undefined behavior, specifically when erasing elements using invalid iterators. This resulted in a test failure, but the issue was resolved by ensuring that valid ranges were used when erasing from the vector.
   * Another challenge was managing the size and capacity of the vector in various tests. The ASSERT\_NO\_THROW and ASSERT\_THROW macros were particularly useful in ensuring that operations like erasing within a valid range didn’t cause unexpected exceptions, while invalid access did.

*A screenshot of a computer program

Description automatically generated***Explanation:** The screenshot confirms that after fixing the issues with invalid iterator access and out-of-range exceptions, all 15 tests passed successfully. This provides validation that the vector’s functionality was thoroughly tested, including both positive and negative scenarios.

1. **Final Test Results:**
   * After addressing all issues, the final test run was executed, and all 15 tests passed successfully. The results confirmed that the vector’s functionality was thoroughly tested and worked as expected, both for positive cases (e.g., adding values, resizing) and negative cases (e.g., out-of-range access, invalid iterator usage).

A screenshot of a computer program

Description automatically generated

**Explanation:** This screenshot shows the final output of the test execution. It confirms that all tests were run without errors or crashes, including both positive and negative tests. The total runtime of 20 milliseconds demonstrates the efficiency of the test.

**References:**

Google Test Primer. (n.d.). Retrieved from https://google.github.io/googletest/primer.html

Microsoft Visual Studio Documentation. (n.d.). Unit Testing in C++. Retrieved from <https://docs.microsoft.com/en-us/visualstudio/test/writing-unit-tests-for-c-cpp>