Improving Test Coverage

This topic explains strategies for improving unit test coverage.

Sections include:

Understanding the Reasons for Low Coverage

Test coverage for C++test is measured using several predefined metrics (as described in Reviewing Coverage Information). In general, an attempt to "improve test coverage" should aim to raise all test coverage metrics. While these metrics will be individually affected by the techniques described here, in every case a simple logical connection can be made between the application of a given technique and its impact on a given coverage metric. Therefore, for simplicity, techniques are considered specifically in relation to line coverage. Extensions to other metrics are implicit.

The problem of increasing test coverage can typically be reduced to three scenarios:

1. Available tests do not utilize input values that will cause control expressions for conditionals in the code to exercise all branches.
2. Control expressions in a tested function depend on values returned by other functions, thus on the state of the program / code when the test is executing.
3. Available tests do not use the proper set up for code under test so running a test results in an exception, breaking up the execution flow at the point of the exception.

Depending on the structure of the code, control expressions may be trivial (e.g. one if/else statement per function) or not (nested loop conditionals with control expressions that are returns of function calls intermixed with branch expressions).

Based on these scenarios, different techniques for controlling conditions within the C++test framework are appropriate in each case.

Strategies for Increasing Code Coverage

The general techniques for increasing code coverage are the following:

- Adding new tests with specific input values or preconditions for the tested function (See Adding User-Defined Test Cases).
- Using specific constructors or constructor call sequences to create an object in a desired state (See Adding User-Defined Test Cases).
- Using user defined stubs (See Adding and Modifying Stubs).
- Using test case driven stubs (See Using Stubs Driven By Test Cases).

We recommend the following procedure when you want to improve test coverage.

1. Examine code coverage for the scope of concern (project or file).
2. If code coverage is below the desired level, analyze the statistics to rank files or functions based on
    - lowest code coverage, or
    - likely ROI in terms of increasing coverage for the effort, judged by looking at the tested code.
3. For all functions in the order of ranking, do the following for all control expressions blocking coverage and their conditional values:
    1. If the conditional is a direct function parameter or a data member of the function's class, add a test case applying a specific input value that causes the control expression to evaluate to a desired branch.
    2. Else, if the conditional is a simple function of a direct function parameter or a data member of the function's class, add a test case that creates a test object, then sets data members and input parameters for the tested function to specific values that case the control expression to evaluate to a desired branch
    3. Else, if the control expression seems to depend on a complex object (via a method call), create a complex test object in the appropriate state (see Complex Objects below)
    4. Else, if the coverage block is due to an exception breaking the execution flow:
        - Examine code to find out why the exception is thrown.
        - If the exception is thrown because of incorrect function/test case parameter value (NULL pointer dereference, etc.) create/modify a test case that will pass a correct value into the function.
        - If the state of a specific object is an issue, see Complex Objects below.
        - Create a user stub for the function that throws the exception or.
    5. Else, if the conditional is computed within function under test by a convoluted code sequence, continue to the next function.

Test driven stubs are usually good for functions with no or few preconditions or parameters, or for functions that encapsulate UI interaction and return values representing user actions. An example of such a function is GUIWidget::whichButtonWasPressed() (figuratively speaking).

If a conditional is a return value of a function, the order of preference of symbol substitution is a) original function b) user stub.

User Stubs

A user stub is typically written to return one of the following:

- The same value every time it is called.
- A different value every time it is called.
- A value depending on the name of the test case (test driven stubs).

Complex Objects

If a conditional depends on a state of a complex object (e.g. a function performing operations on a number of members of a list, such as List:: containsElement(Element&)), then the object needs to be put in the appropriate state as a precondition to the function test. Two important considerations in this case are:

1. What is the desired state of the object?
2. How can this state be attained?

Once the first part is understood, the desired state of an object in a C++test test case can generally be attained with the following approaches:

- Using member wise object initialization (practical only for simple classes). With the help of C++test instrumentation, all private data members of objects can be directly accessed from the body of the test cases. Thus, you can use direct assignments to the data members that affect execution of the specific test case.
- Using parameterized constructors with specific sets of arguments to create the test object(s).
- Using a specific initialization call sequence applied in the setUp method of the test suite. This is specifically effective when test objects always require non-trivial initialization. Using setUp method allows you to specify the initialization sequence once and automatically apply it to all test cases in the test suite.
- Using a test object factory that will supply test objects in known states using the factory class methods.