Test Creation and Execution

In this section:

Creating and Executing Tests

In this section:

Generating Test Cases for Regression Testing and Exception Finding

This topic explains how to have C++test automatically generate unit test cases. These test cases can be used to capture a "functional snapshot" for regression testing, as well as to identify conditions that could result in exceptions, which could lead to system and application instability, security vulnerabilities (such as denial of service attacks), poor performance and application response time, and frequent down time.

Sections include:

- About Automated Test Case Generation
- Generating Test Cases
- Customizing Generation Options

About Automated Test Case Generation

C++test automatically generates test cases according to the parameters defined in the Test Configuration's Generation tab. These test cases use a format similar to the popular CppUnit format.

Generating Tests to Verify New Functionality

If you want to verify the functionality of new code, we recommend that you automatically generate 1-2 tests per function to start.

After you generate and execute these tests, you can then extend the test suite with user-defined test cases as described in "Extending and Modifying the Test Suite", page 406.

Generating Tests for Regression Testing

If you want to create a snapshot of the code's current behavior to establish a regression testing baseline (e.g., if you are confident that the code is behaving as expected), you can run a test using the "Unit Testing> Generate Regression Base" built-in Test Configuration. When this Test Configuration is run, C++test will automatically verify all outcomes.

These tests can then be run automatically, on a regular basis (e.g., every 24 hours) to verify whether code modifications change or break the functionality captured in the regression tests. If any changes are introduced, these test cases will fail in order to alert the team to the problem.

During subsequent tests C++test will report tasks if it detects changes to the behavior captured in the initial test. Verification is not required.

With the default settings, C++test generates one test suite per source/header file. It can also be configured to generate one test suite per function or one test suite per source file (see "Customizing Generation Options", page 350 for details).

Safe stub definitions are automatically-generated to replace "dangerous" functions, which includes system I/O routines such as `rmdir()`, `remove()`, `rename()`, etc. In addition, stubs can be automatically generated for missing function and variable definitions (see "Understanding and Customizing Automated Stub Generation", page 360 for details). User-defined stubs can be added as needed (see "Adding and Modifying Stubs", page 462 for details).

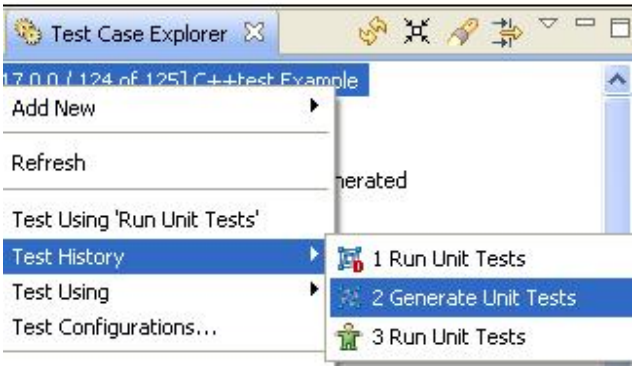
Generating Test Cases

The general procedure for test case generation is:

1. Identify or create a Test Configuration with your preferred test generation settings.
    - For a description of preconfigured Test Configurations, see "Built-in Test Configurations", page 713.
    - For details on how to create a custom Test Configuration, see the Parasoft Test User's Guide (Parasoft Test User's Guide> Configuration> Configuring Test Configurations and Rules for Policies). Details on C++test-specific options are available at "Test Configuration Settings", page 695.
2. Start the test using the preferred Test Configuration for test generation.
    - For details on testing from the GUI, see "Testing from the GUI", page 215.
    - For details on testing from the command line, see "Testing from the Command Line Interface (cpptestcli)", page 219.

Tip - Generating Tests from the Test Case Explorer

You can generate tests for a project directly from the Test Case Explorer (which can be opened by choosing **Parasoft> Show View> Test Case Explorer**). Just right-click the project node in the Test Case Explorer, then choose the desired test generation Test Configuration from the **Test History** or **Test Using** shortcut menu.



For details about the Test Case Explorer, see the Parasoft Test User's Guide (Parasoft Test User's Guide> Introduction> About the Parasoft Test UI).

3. Review the generated test cases.

- For details, see “Reviewing Automatically-Generated Test Cases”, page 381.

4. (Optional) Fine-tune test generation settings as needed.
    - For details, see “Generation Tab Settings: Defining How Test Cases are Generated”, page 701.

Customizing Generation Options

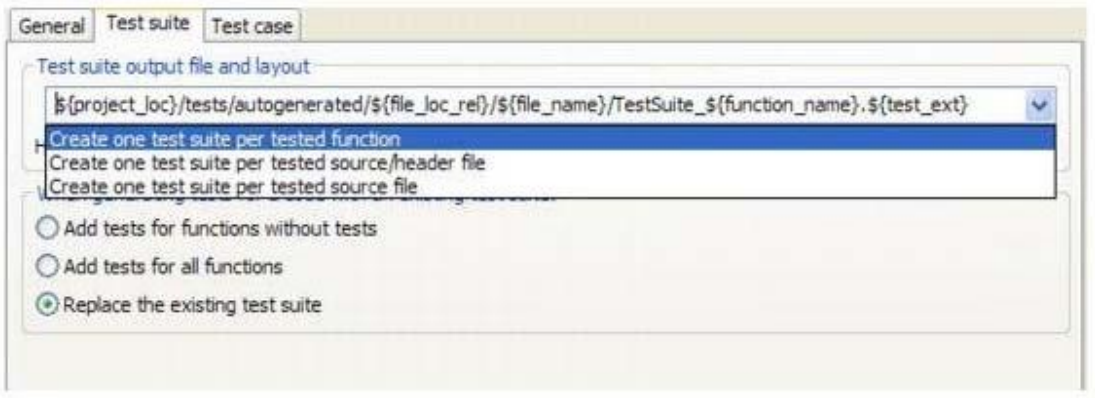
You can control a number of generation options by customizing the options in the Test Configuration’s Generation tab.

Controlling the Test Suite’s File Name, Location, and Layout

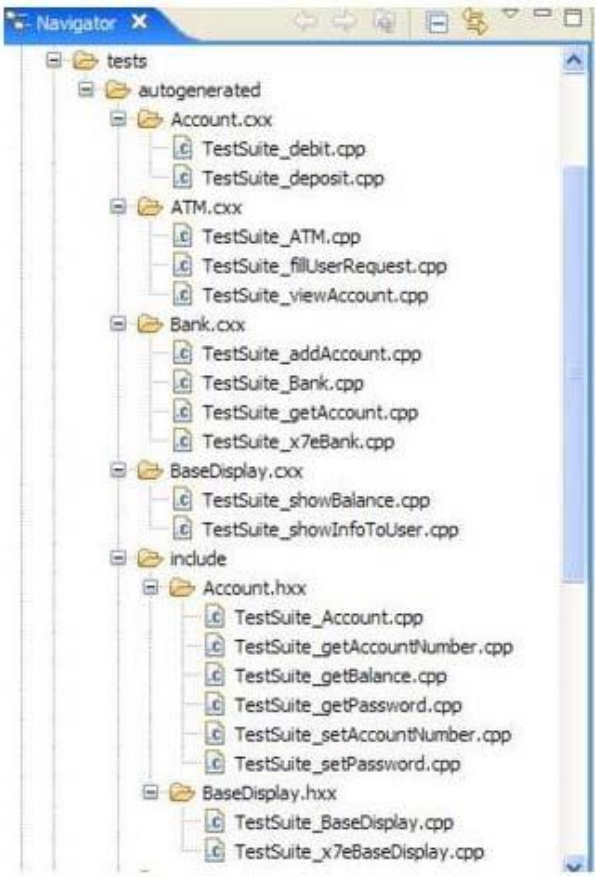
The generated test suite’s file name, location, and granularity/layout can be controlled by customizing options in the Test Configuration’s **Generation> Test suite** tab.

To change the default test suite output settings, first select one of the following three pre-defined output and layout options from the **Test suite output file and layout** box:

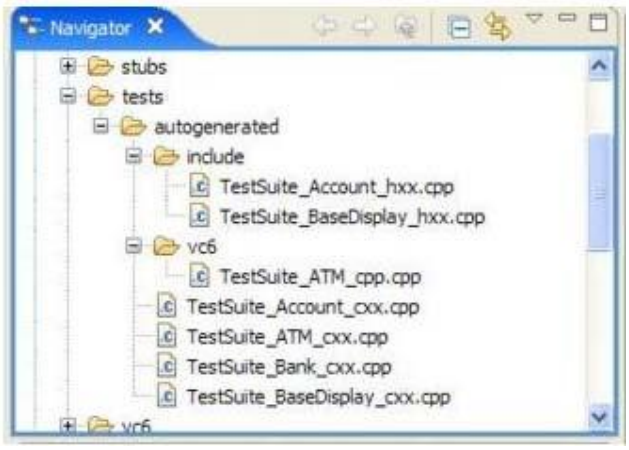
- Create one test suite per function
- Create one test suite per source/header file
- Create one test suite per source file



If the **Create one test suite per function** option is selected, a test suite generated for the sample ATM project (included in the examples directory) would look like this:



If the **Create one test suite per tested source/header** option is selected, a test suite generated for the sample ATM project (included in the examples directory) would look like this:



After selecting one of these options, you can customize the pattern as needed (for instance, to generate tests into the source location). You can use the following variables when you are customizing the pattern:

- ${test_ext} - C++test-specific extension of a test suite file (.cpp).
- ${file_name} - File name.
- ${file_base_name} - File name without extension.
- ${file_ext} - File extension.
- ${file_loc} - File location.
- ${file_loc_rel} - File location relative to the project root.
- ${file_uid} - File unique identifier.
- ${function_name} - Tested function name.
- ${function_uid} - Tested function unique identifier (hash-code computed from the function signature/mangled name).
- ${src_file_name} - Name of context (source) file. (A "context file" is a source file that describes the compilation unit in which the given tested function is defined).
- ${src_file_base_name} - Name of context (source) file without extension.
- ${src_file_ext} - Extension of context (source) file.
- ${src_file_loc} - Context (source) file location.
- ${src_file_loc_rel} -Context (source) file location relative to the project root.
- ${src_file_uid} - Source (context) Context file unique identifier (hash-code computed from the source file location).

Key

- file = The source/header file where the tested function is defined.
- source file = The source file that defines a compilation unit where the tested function is defined.

Warning

Removing some variables can lead to overlapping test suites. C++test will alert you to this by displaying the error message "Test suite output file pattern is ambiguous."

C++test uses the following internal checks and restrictions related to changing the basic patterns for location of generated tests:

- All automatically generated tests are of "included" type (the test suite file is "glued "together with the given source file/compilation unit).
- It tries to prevent cases where test cases for functions from different compilation units would be placed in the same test suite file (because it is not possible to correctly glue such a test suite file with original source file).
- C++test has different variables (which are resolved based on the file under test, its name, its location etc.) that can be used to make the test suite file pattern unambiguous (in terms of the item above).
- One commonly-used strategy is to generate a test suite file into a file/location that has the original file name/location in it. This the default pattern:

  ```
  ${project_loc}/tests/autogenerated/${file_loc_rel}/
  TestSuite_${file_base_name}_${file_ext}.${test_ext}
  ```

  This is not ambiguous because ${file_loc_rel} and ${file_base_name} variables are used (even though there are a number of files with the same name in the project, their location will be different—and that location will be a part of the test suite file name/location).
- There are also other available variables—for example, ${file_uid}, ${src_file_uid}—that can be used instead of the ${file_loc_rel} / ${file_base_name} pair while keeping the pattern unambiguous. These variables are resolved into a hash code of the original file location. For example, a pattern like

  ```
  ${project_loc}/tests/autogenerated/
  TestSuite_${file_base_name}_${file_uid}_${file_ext}.${test_ext}
  ```

  will result in the following test suite:

  ATM/tests/autogenerated/TestSuite_Account_d7a5efc6_hxx.cpp

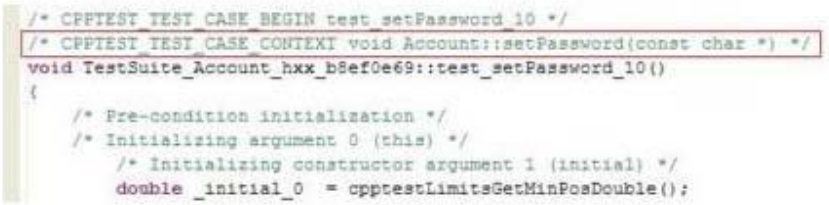
Appending or Replacing Existing Tests

You can also control whether C++test will append or replace existing tests if the generated test file has the same name and location as an existing test suite file. This behavior is determined by the **When generating tests for code with an existing test suite** setting, which provides the following options:

- **Add tests for functions without tests:** C++test will generate test cases for functions without tests. The existing tests will not be affected or modified.
- **Add tests for all functions:** C++test will generate test cases for all functions. The existing tests will not be affected or modified.
- **Replace the existing test suite:** C++test will generate test cases for all functions. The existing test suite will be removed and then replaced with the new one.

How does C++test determine if there are existing test cases for a given function?

It looks for the CPPTEST_TEST_CASE_CONTEXT marker inside the test suite file.



Choosing the Generation Options That Suit Your Goals

This section explains how to configure the Test Configuration's generation options to accomplish common test generation goals.

Options covered include the **Generation> General** tab's **Generate tests for code** option and the **Generation> Test suite** tab's **When generating tests for a code with an existing test suite** option.

To generate an initial set of tests

- For **Generate tests for code**, enable **Without test suites**.
- Specify additional parameters (function access level, output file location/name etc.).

To update an existing automatically-generated test suites with tests for new functions (do not generate new test suites)

- For **Generate tests for code**, enable **With up-to-date test suites With out-of-date test suites**.
- For **When generating tests for a code with an existing test suite**, enable **Add tests for functions without tests**.
- Specify additional parameters (function access level, output file location/name etc.).

To synchronize automatically-generated tests with the current code - append missing tests, create missing test suites

- For **Generate tests for code**, enable **Without test suites**, **With up-to-date test suites**, and **With out-of-date test suites**.
- For **When generating tests for a code with an existing test suite**, enable **Add tests for functions without tests**.
- Specify additional parameters (function access level, output file location/name etc.).

To fully reset existing automatically-generated tests

- For **Generate tests for code**, enable **Without test suites**, **With up-to-date test suites**, and **With out-of-date test suites**.
- For **When generating tests for a code with an existing test suite**, enable **Replace the existing test suite**.

- Specify additional parameters (function access level, output file location/name etc.).

Choosing the Layout Option That Suit Your Goals

This section explains how to configure the **Test suite output file and layout** option (in the Test Configuration's **Generation> Test suite** tab) to suit various layout needs. To help you understand how each option discussed translates to actual projects, we show how it would affect the following sample project:

```
MyProject
  headers
    MyClass.h // contains foo() definition
  sources
    MyClass.cpp // contains bar() and goo() definitions
```

To generate a single test suite file for each function, keep tests in a separate directory

Use `${project_loc}/tests/${file_loc_rel}/${file_name}/TestSuite_${function_name}.${test_ext}`

Sample layout:

```
MyProject
  headers
    MyClass.h
  sources
    MyClass.cpp
  tests
    headers
      MyClass.h
        TestSuite_foo.cpp // contains tests for foo()
    sources
      MyClass.cpp
        TestSuite_bar.cpp // contains tests for bar()
        TestSuite_goo.cpp // contains tests for goo()
```

Use `${project_loc}/tests/${file_loc_rel}/${file_name}/TestSuite_${function_name}.${test_ext}`

Sample layout:

```
MyProject
 Header Files
   MyClass.h
 Source Files
   MyClass.cpp
 tests
   Header Files
     MyClass.h
       TestSuite_foo.cpp // contains tests for foo()
   Source Files
     MyClass.cpp
       TestSuite_bar.cpp // contains tests for bar()
       TestSuite_goo.cpp // contains tests for goo()
```

To generate a single test suite file for each source/header file, keep tests in a separate directory

Use `${project_loc}/tests/${file_loc_rel}/TestSuite_${file_base_name}_${file_ext}.${test_ext}`

Sample layout:

```
MyProject
```

```
  headers
    MyClass.h
  sources
    MyClass.cpp
  tests
    headers
      TestSuite_MyClass_h.cpp // contains tests for foo()
    sources
      TestSuite_MyClass_cpp.cpp // contains tests for bar() and goo()
```

To generate a single test suite file for each source/header file, keep tests with the tested files

Use `${project_loc}/${file_loc_rel}/tests/TestSuite_${file_base_name}_${file_ext}.${test_ext}`

Sample layout:

```
MyProject
  headers
    MyClass.h
    tests
      TestSuite_MyClass_h.cpp // contains tests for foo()
  sources
    MyClass.cpp
    tests
      TestSuite_MyClass_cpp.cpp // contains tests for bar() and goo()
```

To keep auto-generated test suite files with original source files

Use `${project_loc}/tests/${file_loc_rel}/${file_name}/TestSuite_${function_name}.${test_ext}`

Sample layout:

```
MyProject
  module1
    sources
      MyClass.cpp
      tests
        TestSuite_MyClass_cpp.cpp
    headers
      MyClass.h
      tests
        TestSuite_MyClass_h.cpp
```

To keep the auto-generated test files in an intuitive structure (outside of my original files)

Use `${project}/tests/${source_loc_rel:/MyProject/module1}/TestSuite_${source_base_name}_${source_ext}.${test_ext}`

Sample layout:

```
MyProject
  module1
    sources
      MyClass.cpp
    headers
      MyClass.h
  tests
    sources
      TestSuite_MyClass_cpp.cpp
    headers
      TestSuite_MyClass_h.cpp
```

To keep test files flat in a single directory (if the project does not have duplicate names)

Use `${project}/tests/TestSuite_${source_base_name}_${source_ext}.${test_ext}`

Sample layout:

```
MyProject
  module1
    sources
      MyClass.cpp
    headers
      MyClass.h
  tests
    TestSuite_MyClass_cpp.cpp
    TestSuite_MyClass_h.cpp
```

To use "One test suite per function" mode

Use `${project}/tests/${source_base_name}_${source_ext}/TestSuite_${function_name}_${function_uid}.${test_ext}`

Sample layout:

```
MyProject
  module1
    sources
      MyClass.cpp
    headers
      MyClass.h
  tests
    MyClass_cpp
      TestSuite_foo_1234abcd.cpp
      TestSuite_foo_4321abcd.cpp
      TestSuite_goo_4321dcba.cpp
    MyClass_h
      TestSuite_bar_2143badc.cpp
```

Executing Test Cases

This topic explains how to use C++test to execute automatically-generated and/or user-defined C++test or legacy CppUnit test cases.

Sections include:

Executing Test Cases

C++test can run and report coverage information for any valid C++test or legacy CppUnit test case.

Handling of GCC -nostdinc option

If the GCC `-nostdinc` option is used, you need to perform one of the following steps to successfully create a test executable:

- Disable the data sources and streams support functionality. This can be achieved by adding the following options to the compiler command line (via Project properties> Build settings):
  -DCPPTEST_DATA_SOURCES_ENABLED=0
  -DCPPTEST_DISABLE_STREAMS_REDIRECTION=1
- Modify the compiler command line (in the project properties> build settings) to include the standard system directories for header files.

The general procedure for running test case execution is:

1. If you have not already done so, generate test cases as described in “Generating Test Cases for Regression Testing and Exception Finding”, page 348.
2. (Optional) If you want to automatically-generate stubs for missing function and variable definitions, run the built-in "Generate Stubs" Test Configuration, or a custom Test Configuration that is based on it. *This is not recommended for regularly-scheduled command line tests.*
   - See “Adding and Modifying Stubs”, page 462 for information on how stubs are generated, and instructions for customizing C++test’s stub generation.

3. (Optional) If you want to check whether auto-generated stubs and tests are compilable before you execute the tests, perform a trial build by running the built-in "Build Test Executable" Test Configuration, or a custom Test Configuration that is based on it. *This is not recommended for regularly-scheduled command line tests.*
    - See "Understanding Trial Builds", page 360 for information about trial builds.
4. Start the test using the built-in "Run Unit Tests" Test Configuration, or a custom Test Configuration that is based on it.

Tip - Executing Tests from the Test Case Explorer

You can execute tests directly from the Test Case Explorer (which can be opened by choosing **Parasoft> Show View> Test Case Explorer**). Just select the Test Case Explorer node(s) for resource(s) you want to test (projects, folders, test suites, or test cases), right-click the selection, then choose the desired test execution Test Configuration from the **Test History** or **Test Using** shortcut menu.

Executed tests will be color-coded to indicate their results status. Failed tests will be marked in red. Passed tests will be marked in green.

5. Review and respond to the test case execution results.
    - For details, see "Reviewing Test Execution Results", page 367.
6. (Optional) Fine-tune test execution settings as needed.
    - For details, see "Execution Tab Settings: Defining How Tests are Executed", page 703.

Testing Template Functions

C++test performs unit testing of instantiated function templates and instantiated members of class templates.

See "Support for Template Functions", page 739 for details.

Configuring Batch-Mode Regression Test Execution with cpptestcli

Regularly-schedule batch-mode regression tests should simply execute the built-in "Run Unit Tests" Test Configuration, or a custom Test Configuration that is based on it.

For example:

- cpptestcli -data /path/to/workspace -resource "ProjectToTest" -config team://ExecuteTests -publish

See "Testing from the Command Line Interface (cpptestcli)", page 219 for more details on configuring batch-mode tests.

Understanding Trial Builds

C++test can perform a trial build of the test executable, which includes test cases and user stubs, without executing the tests. This feature can be used to check whether auto-generated stubs and tests are compilable. You can perform a trial build even if there are not yet any test cases in the tested project.

The recommended way to perform a trial build is to run the built-in "Build Test Executable" Test Configuration.

Understanding and Customizing Automated Stub Generation

When you run the built-in "Generate Stubs" Test Configuration (or a custom Test Configuration that is based on it), C++test will automatically generate customizable stubs (or stub templates) for missing function and variable definitions. As described in “Executing Test Cases”, page 358, we recommend that you first generate test cases, then run the Generate Stubs Test Configuration, then run the Build Test Executable Test Configuration before you perform test case execution.

When a test is run using a Test Configuration set to generate stubs, C++test will create a stub file in the specified location. If C++test cannot automatically generate a complete stub definition, it will create a stub template that you can customize (by entering the appropriate return statement, adding include directives etc.). Stub templates will be saved in the stub file before complete stubs.

Automatically generated stubs will be used only if no other definition (user stub or original) is available.

Automatically-generated stubs and stub templates can be customized as described in “Adding and Modifying Stubs”, page 462. If you customize the stubs or stub templates, you need rerun the analysis to prompt C++test to use them.

To create a custom stub generation Test Configuration:

1. Open the Test Configurations panel by choosing **Parasoft> Test Configurations**.
2. Right-click the **Built-in> Unit Testing> Generate Stubs** Test Configuration, then choose **Duplicate** A new Generate Stubs Test Configuration will be added to the User-defined category.
3. Select **User-defined> Generate Stubs**.
4. Open the **Execution> Symbols** tab.
5. (Optional) If you do not want the generated stubs saved in the default location (${project_loc}/stubs/autogenerated), enter your preferred location in the **Auto-generated stubs output location** field.
6. Click **Apply**, then **Close**.

Defining a Custom Test Unit

By default, C++test computes the list of the tested files (project files to be tested) and test suites in the following way:

- All test suites that are selected and match the criteria specified in the **Execution> General** tab’s **Test suite file search patterns** option will be executed. If a test suite file uses the CPPTEST_CONTEXT and/or CPPTEST_INCLUDED_TO macro, the appropriate source and header files will become tested files.
- All project source and header files that are selected will become tested files. Additionally, C++test will search the **Test suite file search patterns** locations for test suite files with

CPPTEST_CONTEXT set to one of these tested files; if any such test suite files are found, they will also be executed.

If you want C++test to use any additional project source files to resolve the original definitions from the tested files, you can specify this in the Test Configuration's **Use symbols from additional project files** option (available in the **Execution> Symbols** tab). Additionally, you can use the **Use extra symbols from files found in** option to specify which stubs are used, and use the **Create separate test executable for each tested context** option to specify whether you want C++test to create a separate test executable for each context (a single source/header file or a project).

See "Symbols tab", page 708 for more details on how to specify the additional project files and stubs that you want to use, and how you want the test executable prepared.

Testing a Single File in Isolation

In certain situations, test policies require that unit tests be applied to code in isolation of the other related components. C++test allows you to perform such testing through the use of stubs.

To test a single file in isolation (on a different "test bed", including custom and automatic stubs):

1. Select the file you want to test.
2. Run a test using the built-in "Unit Testing> File Scope> Run Unit Tests (File Scope)" Test Configuration. This will typically result in an error status (with details displayed in the Console view) because C++test cannot locate definitions for a number of functions.
3. Provide the missing function or variable definitions by:
    - Creating user stubs manually (see "Adding and Modifying Stubs", page 462).
    - Using C++test to create stubs automatically using the built-in "Unit Testing> File Scope> Generate Stubs (File Scope)" Test Configuration (see "Understanding and Customizing Automated Stub Generation", page 360), then modifying their source as necessary (see "Adding and Modifying Stubs", page 462).
4. Run the built-in "Unit Testing> File Scope> Build Test Executable (File Scope)" Test Configuration to perform a trial build of the test executable. This is recommended to ensure that all necessary symbols are properly resolved, and there are no compilation errors introduced by custom stub code.
5. Run the built-in "Unit Testing> File Scope> Run Unit Tests (File Scope)" to execute the tests. This time, the run should successfully complete.

Ignoring Libraries and Object Files

When testing a single file in isolation, you might want to use the object and library files filter to prevent testing of specified libraries and objects. In the Test Configuration's **Ignore object/library files** field (in the Execution> Symbols tab), you can specify a semicolon-separated list of patterns of command line options. Only options from the linker command line are ignored. Standard compiler libraries and libraries included with pragmas are not filtered.

Executing Individual Test Cases

To execute a user-defined set of test cases:

1. Select the test function(s) you want to execute in one of the following ways:
    - Select the test cases in the Test Case Explorer.
    - Select the test case method in the editor for the test suite file.

- Select the related project tree node(s). (You can select and execute test cases from different test suites.)

CDT 4.x Note

Test functions are not available in the project tree for Managed C/C++ projects created with CDT 4.x. To execute individual test cases, select the test case name in the code editor.

2. Run a Test Configuration that is set to execute test cases.
    - For example, right-click the selection, then use the **Parasoft** shortcut menu to run the preferred Test Configuration.

Resolving Linker Errors from Unresolved Symbols

C++test may report linker errors if the code under test references symbols from additional files, but C++test cannot find those symbols.

To see which symbols are unresolved:

1. Enable the **Perform early check for potential linker problems** option in the Test Configuration's **Execution> Symbols** tab (see "Execution Tab Settings: Defining How Tests are Executed", page 703 for details).
2. Rerun the test.

C++test will report unresolved symbols (undefined functions) before the compilation and linking stage. To see the unresolved symbols, do one of the following:

- Open the Stubs view and look for symbols with the 'N/A' definition.
- Open the Console view and look for "Cannot configure stubs for function [function]" messages.

There are several ways to resolve symbols:

- If you have not already done so, automatically generate stubs for missing symbols. See "Understanding and Customizing Automated Stub Generation", page 360 for details.
- If the symbols are defined within the project, enable and enter an asterisk in the **Use symbols from additional project files** option in the Test Configuration's **Execution> Symbols** tab, then rerun the test. See "Execution Tab Settings: Defining How Tests are Executed", page 703 for details.
- If the symbols are in an external library, add the library to the linker command line, then rerun the test. See "Updating the Project", page 195 for details.
- Otherwise, provide user-defined stubs for the missing functions, then rerun the test. See "Adding and Modifying Stubs", page 462 for details.

Using a Debugger During Test Execution

See "Using a Debugger During Test Execution", page 378.

Testing Multi-Threaded Applications

This topic explains how to perform unit testing and data collection for multi-threaded applications. Sections include:

Prerequisites

To enable testing multi-threaded applications, add the following option to Build Settings' Compiler Options (described in "Setting Project and File Options", page 196):

"-DCPPTEST_THREADS=1"

This option will activate multi-thread support in the C++test's Runtime library, as well as disable use of Safe Stubs for thread-related routines.

Considerations

Considerations for testing multi-threaded applications include:

- C++test's test executable must be linked with the appropriate runtime library (i.e. the multi-threaded version of C Runtime Library for Visual C++ compilers or "pthread" library on UNIX). Also, on Solaris, -D_POSIX_PTHREAD_SEMANTICS should be added to the compiler options.
- C++test's Runtime does not control threads' life-time - we recommended you implement test cases so that all threads are terminated when the test case completes.

Known Problems and Limitations

Known problems and limitations:

- If multi-thread support is enabled (using -DCPPTEST_THREADS=1)—but no appropriate library is used during linking (or a multi- and single-threaded C Runtime is mixed)—then the test executable may produce corrupted test results, or may terminate unexpectedly.
- Unexpected behavior in a non-main thread (signal, unhandled exception, time-out) will cause the test executable to terminate.

Building the C++test Runtime Library

C++test's default Runtime Library has multi-thread support built-in. No additional action is required unless you are building a custom runtime library with multi-threaded support (e.g., for embedded testing).

If you need to build a custom Runtime Library with the multi-thread support, add "-DCPPTEST_THREADS_ENABLED=1" to the compiler command line.

Supported Thread APIs

The implementation of Runtime Library can use the following types of thread APIs:

- Windows Threads
- POSIX Threads
- VxWorks 5.4, 5.5, 6.x

Using Other Thread APIs

If you need to use another thread API (or a custom one), the following types and routines must be implemented.

Note that the CppTestThread.c file in the C++test Runtime sources contains definitions of thread support routines; it can be used as an example or as a base for changes.

Tls - Thread Local Storage

```
typedef IMPLEMENTATION_DEPENDENT_TYPE CppTestThreadKey;

/**
 * Creates key for thread local storage data
 * Returns 0 on success
 */
extern int localThreadKeyCreate(CppTestThreadKey* key);

/**
 * Deletes key for thread local storage data
 * Returns 0 on success
 */
extern int localThreadKeyDelete(CppTestThreadKey key);

/**
 * Returns a thread specific value associate with a key
 */
extern void* localThreadGetSpecific(CppTestThreadKey key);

/**
 * Associate a thread specific value with a key
 * Returns 0 on success
 */
extern int localThreadSetSpecific(CppTestThreadKey key, void* value);
```

Mutex

Note: C++test's Runtime assumes that a mutex can be statically initialized.

```
typedef IMPLEMENTATION_DEPENDENT_TYPE CppTestThreadMutex;

#define CPPTEST_THREADS_MUTEX_STATIC_INIT <IMPLEMENTATION_DEPENDENT_STATIC_INITIALIZER>

/**
```

```
 * Initializes mutex
 * @return 0 on success
 */
extern int localThreadMutexInit(CppTestThreadMutex* mutex);

/**
 * Destroy and release resources used by mutex
 * @return 0 on success
 */
extern int localThreadMutexDestroy(CppTestThreadMutex* mutex);

/**
 * Lock mutext and return when calling thread becames is owner of it.
 * @return 0 on success
 */
extern int localThreadMutexLock(CppTestThreadMutex* mutex);

/**
 * Releases mutex owned by calling thread.
 * @return 0 on success
 */
extern int localThreadMutexUnlock(CppTestThreadMutex* mutex);
```

Miscellaneous

```
/**
 * Exits calling thread
 * (never returns)
 */
extern void localThreadExit();

/**
 * @return non-zero if threads already finished execution
 */
extern int localThreadFinished(CppTestThread* thread);

/**
 * @return non-zero if threads are supported in current build
 * (proper macros, libraries, compiler options were used).
 */
extern int localThreadsSupported(void);

/**
 * Initializes given thread structure
 */
extern void localThreadInit(CppTestThread* thread);
```

Exploring Test Results

In this section:

Reviewing Test Execution Results

This topic covers how to analyze and address C++test's test execution results.

Sections include:

Reviewing Reported Tasks

After test execution, C++test generates a prioritized task list, organized by error categories and severities.

For tests run in the GUI, tasks are organized into the following categories in the Quality Tasks view:

- **Fix Unit Test Problems:** This category contains definite unit test problems—including functional test failures, unexpected exceptions, and timeouts—that need to be addressed.
- **Review Unit Test Outcomes:** This category contains unverified outcomes for test cases that were created during automated test case generation. Unverified outcomes are reported when C++test executes automatically-generated or user-defined test cases with postconditions that have not yet been converted to assertions The outcome might be the expected behavior, or it might indicate a problem. Further review and verification is required. If you determine that the outcome reflects the expected behavior, you verify it. If not, you specify the correct outcome.

For tests run from the command line interface, tasks are reported in the **Test Generation** and **Test Execution** section of the report. If results were sent to Team Server, results can be imported into the GUI as described in the Parasoft Test User's Guide (Parasoft Test User's Guide> Workflow and Usage> Importing Results into the UI). They will then be available in the Quality Tasks view—as if the tests were run from the GUI.

Viewing Test Configurations That Trigger Tasks

Test configurations that trigger tasks can be opened from the Quality Tasks view: Right-click on a task and choose **View Test Configuration**.

Quickly accessing test configuration from the from the Quality Tasks view is useful for group architects who are customizing tests and want to quickly disable settings that aren't applicable. Developers importing results from a server-based run may also need to open and review test configurations that trigger tasks.

Reviewing Stack Traces

For each unit testing problem reported, C++test reports the stack trace for the test case that caused the problem.

To review one of the lines of code referenced in the stack trace, double-click the node that shows the line number, or right-click that node and choose **Go to** from the shortcut menu. The editor will then open and highlight the designated line of code.

Reviewing Postconditions

The results of postcondition macros (all the asserted values reported) are displayed in the Quality Tasks view. These postconditions capture the state of test objects or global variables used in the test.

Any test case with reported post conditions can be automatically validated for use in regression testing. Verification changes the *_POST_CONDITION_* macros into assertions that will fail if a subsequent test does not produce the expected (validated) value. This is especially useful for automated generation of a regression base for legacy code. For details on verification, see “Verifying Test Cases for Regression Testing”, page 409.

Understanding and Customizing Task Severity Categorization

Within each category, tasks are organized according to severity to help you identify and focus on the most serious issues.

Tip

If you want the Quality Tasks view to display the severity of each task, go to the pull-down menu in the Quality Tasks view, then choose **Configure Contents** and set it to show **Severity**

Reviewing Test Case Execution Details

The Test Case Explorer helps you manage a project’s test cases, test suites, and related data sources. It provides detailed test statistics (executed/passed/failed/skipped) and allows you to search/filter the test case tree.

The Test Case Explorer is open by default, in the left side of the UI. If it is not available, you can open it by choosing **Parasoft> Show View> Test Case Explorer**.

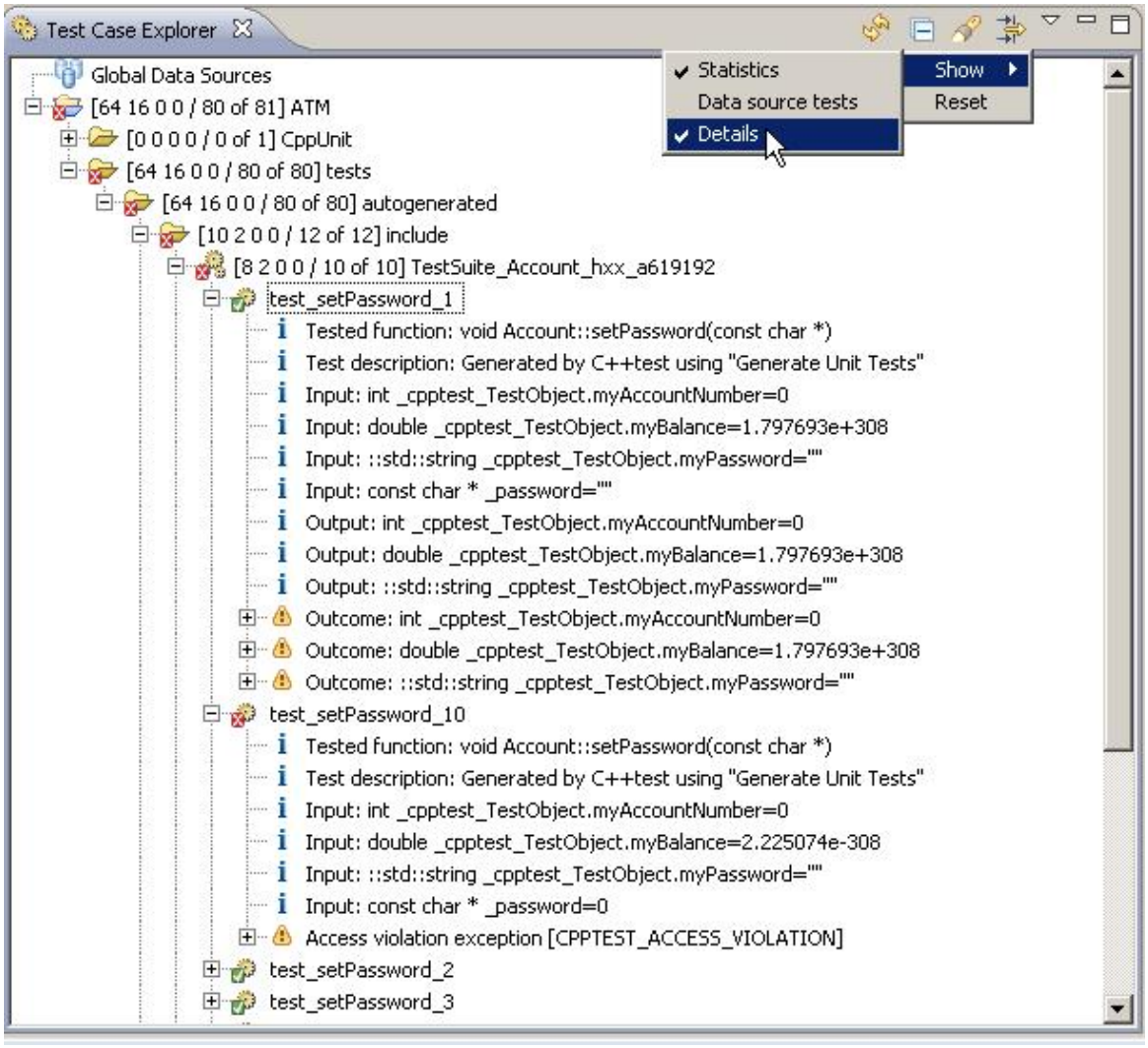
For details on the Test Case Explorer, see the Parasoft Test User’s Guide (Parasoft Test User’s Guide> Introduction> About the Parasoft Test UI).

Reviewing Test Case Details

To view test case details, enable the **Show> Details** option from the Test Case Explorer menu. This configures C++test to report the following additional information in the Test Case Explorer tree:

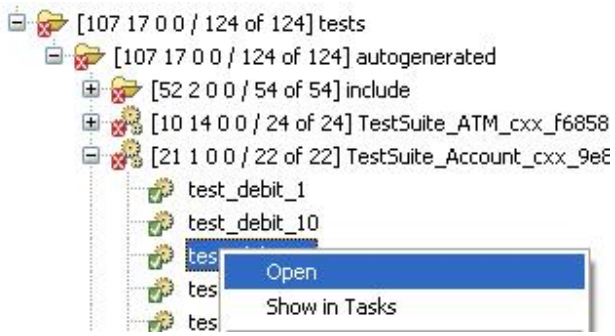
- Information about the function tested by a given test case (collected from the CPPTEST_TEST_CASE_CONTEXT comment). Such comments are always added by C++test for automatically-generated test cases and test cases created using Test Case Wizard.

- Information about the test case description (collected from the CPPTEST_TEST_CASE_DESCRIPTION comment). Such comments can be added by C++test for automatically-generated test cases (see “General tab”, page 703) and test cases created using Test Case Wizard (see “Test Case Configuration Tips”, page 415).
- Information about the test execution details. Depending on test configuration (see “Runtime tab”, page 709), this can include:
    - Test case reports / messages
    - Information about reported tasks (e.g., exceptions, failed assertions, unverified outcomes)
    - Information about checked and passed assertions



Opening the Source Code for a Test or Test Suite

To open the source code for a test suite or test case in the Test Case Explorer, right-click its Test Case Explorer node, then choose **Open**. Or, double-click its Test Case Explorer node.



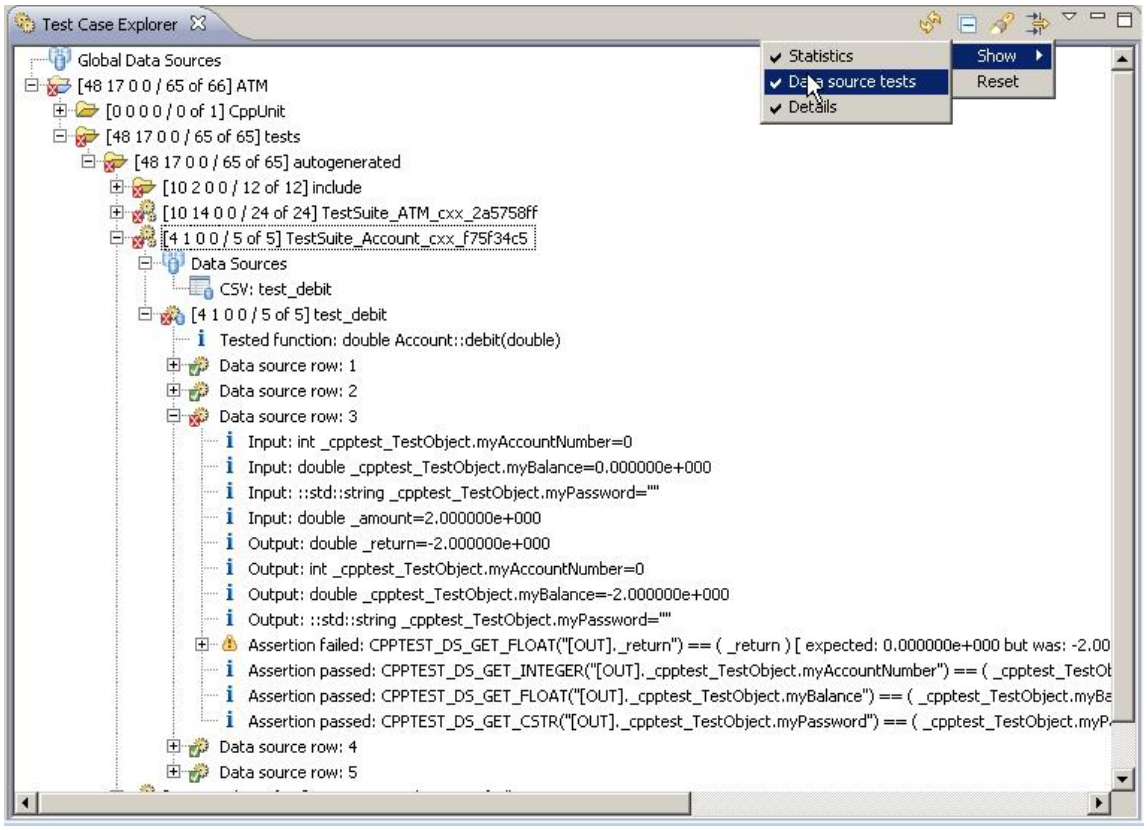
Correlating Test Case Explorer Nodes to Quality Tasks View Results

To open the test results (if available) for a test case in the Test Case Explorer, right-click its Test Case Explorer node, then choose **Show in Quality Tasks**. You can also perform the reverse action—to see the Test Case Explorer node that correlates to a result in the Quality Tasks view, right-click the Quality Tasks view node, then choose **Show in Tests**.

Reviewing Results of Data Source Test Cases

To view detailed information about executed data source test cases, enable **Show > Data source tests** from the Test Case Explorer menu. This will make C++test display information about each executed iteration of the data source test case.

The data source test case element of the tree presents statistics information about executed iterations (e.g. number of passed and failed iterations). Statistics shown for all parent nodes of the data source test case also include information about particular data source test case iterations.



Obtaining Traceability Details

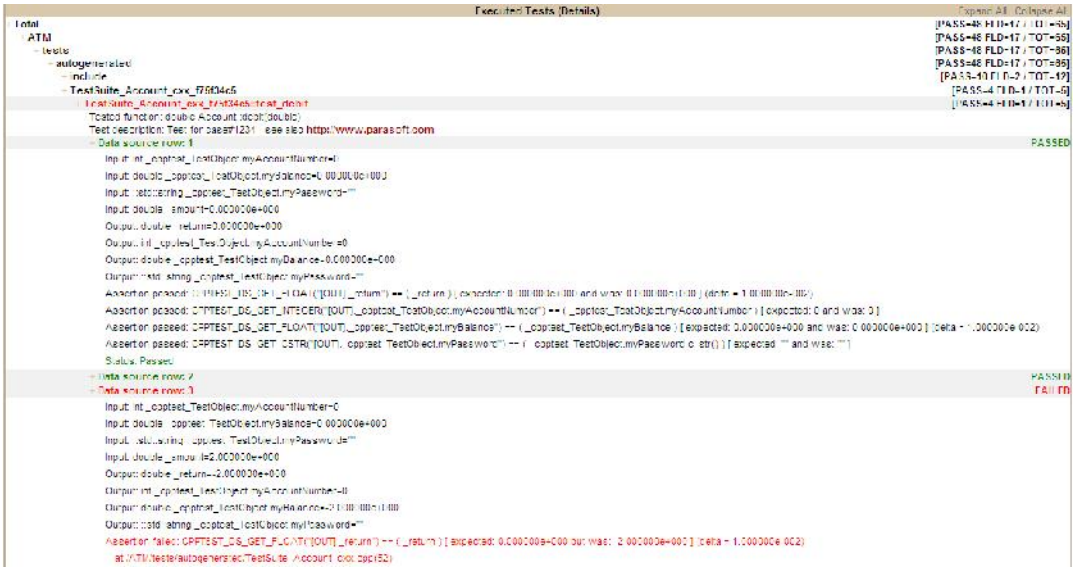
There are several ways to create a report containing the maximum amount of unit test execution and traceability information:

From here...	Perform these steps...
Generating test cases automatically	• Provide the test case description in the **Test Configuration > Generation> General> Test case description** field. • Enable **Test Configuration> Generation> Test case> Insert code to report test case inputs and outputs**.

From here...	Perform these steps...
Creating test cases using Test Case Wizard	• Provide a test case description. • Enable **Insert code to report test case inputs and outputs**.
Executing tests	• Enable **Test Configuration> Execution> Runtime> Report unit test execution details**. • Enable **Test Configuration> Execution> Runtime> Include tasks details**. • Enable **Test Configuration> Execution> Runtime> Include passed assertions details**.
Generating reports	• Enable **Parasoft> Preferences> Reports > Overview of checked files and executed tests**. • Choose **HTML (C++test's Unit Testing details)** as the Report format.
Generating reports from the command line	• Set the `report.contexts_details=true` option in the localsettings file to enable the Executed Test Cases section. • Set the `report.format=html_ut_details` option in the localsettings file to set up the report format.

Traceability Reporting Tips

- To report additional messages from test cases, use test case report macros (see “Test Case Report Macros”, page 764).
- For data source test cases, reported messages might be read from the data source. For example,
  CPPTEST_REPORT_CSTR("Requirement number", CPPTEST_DS_GET_CSTR("req_no"))
- If a test case description or test case message contains valid URLs (e.g. http:// or ftp://), they will be added into the generated HTML report as links.



Responding to Reported Tasks

Different types of findings require different response strategies. The following table lists the categories used to classify C++test's test execution findings, and links to sections that will help you understand and respond to them.

Category	Subcategory	Description and Recommended Response
Fix Unit Test Problems	Assertion Failures	See "Assertion Failures", page 375 and "Time-outs", page 376
	Runtime Exceptions	See "Runtime Exceptions", page 375
Review Unit Test Outcomes	Unverified Outcomes	See "Unverified Outcomes", page 377

Using Quick Fix (R) to Respond to Test Execution Findings

The Quick Fix (R) feature can be used to automate actions commonly performed while reviewing and responding to unit test findings.

To use Quick Fix to respond to a test execution finding:

1. Locate the detailed task message (the one with the line number) in the Quality Tasks view.
    - If you see the following icon next to the task message, it indicates that you can use Quick Fix to resolve this task:

2. Right-click that task message, then choose the appropriate Quick Fix option from the shortcut menu. C++test will then perform the specified action.
    - The available Quick Fix options are described below.
3. Save the modified file.

Understanding Available Quick Fix options

The following section explains the available Quick Fixes. Note that the Quick Fixes available for a specific task depends on the nature of the task.

Assertion Failure Quick Fixes

- **Verify assertion:** Indicates that the current value is correct (and the finding does not indicate a functional problem).
- **Ignore assertion:** Prevents the checking of this assertion in subsequent tests. The related test case will still be run. This action is recommended if the assertion is checking something that you don't want to check or something that changes over time (e.g., an outcome that checks the day of the month).
- **Remove Test Case:** Deletes the test case that produced given outcome—for example, if you are reviewing test results and see that a test case is invalid.

Unverified Outcome Quick Fixes

- **Verify outcome**: Indicates that the test case outcome is correct (and the finding does not indicate a functional problem).
- **Ignore outcome**: Prevents the checking of this outcome in subsequent tests. The related test case will still be run. This action is recommended if the assertion is checking something that you don't want to check or something that changes over time (e.g., an outcome that checks the day of the month).
- **Remove Test Case:** Deletes the test case that produced given outcome—for example, if you are reviewing test results and see that a test case is invalid.

Handling a Large Number of Reported Tasks (e.g., After Testing Legacy Code)

If you want to apply the same action to multiple reported problems:

1. Select the problems you want to apply the action to.
2. Right-click the selection, then choose the command for the desired response (for example, **Verify All Outcomes**).

Assertion Failures

This topic provides details about results in the Assertion Failure category.

Description

A Fix Unit Test Problems> Assertion Failures message indicates that a test case did not produce the correct outcome.

Recommended Response

Examine each failure and determine whether the code is functioning correctly. Unless the expected outcome has changed, a test case failure indicates a functional problem with the code.

- Use a debugger to investigate the reason for the failure. See “Using a Debugger During Test Execution”, page 378 for details.
- If the code is incorrect, correct the code.
- If the failure is intentional and describes an error reported by C++test, (i.e., the message contains a C++test error code such as [CPPTEST_EXIT_CALLED]), change the test case registration to use CPPTEST_TEST_ERROR. See “Test Suite/Test Case Registration Macros”, page 760 for details.
- If the expected outcome has changed, modify the test case's expected outcome by editing the test case in the test suite file. During subsequent test runs, C++test will check if the actual test case outcome matches this modified outcome.

Runtime Exceptions

This topic provides details about results in the Runtime Exception category.

Description

A Fix Unit Test Problems> Runtime Exceptions message indicates that a test case causes an unexpected exception (an exception that is not accounted for and handled in the test case code). An exception indicates code that is not sufficiently robust. If exceptions surface in the field, the resulting unexpected flow transfer and potential thread termination could lead to instability, unexpected results, or even crashes or security breaches.

If an exception occurs when a deployed application is running, it can cause system and application instability, security vulnerabilities (such as denial of service attacks), and frequent down time.

If an exception is reported during unit testing, it indicates that the execution of a test case creates conditions under which the code under test throws the exception. After the code under test is integrated into the application, the application might create these same conditions and force the code to throw the exception. As explained in the previous paragraph, any occurrence of this exception in a running deployed application is a critical problem.

Recommended Response

The exceptions that are reported in this category indicate incorrectly behaving code—code that should not throw an exception for the given arguments.

We recommend that you respond to all of the reported runtime exceptions before you work on other code. If the code requires modification, it is faster and easier to fix it as soon you learn about it than to wait until later in the development process. If the code is behaving correctly, you can prevent confusion and the introduction of errors by ensuring that this behavior is clearly documented; when other developers working with the code know exactly how the code is supposed to behave, they will be less likely to introduce errors.

When an exception is reported during unit testing, the recommended response is:

- Use a debugger to investigate the reason for the failure. See “Using a Debugger During Test Execution”, page 378 for details.
- Determine whether the exception indicates a bug in the code under test, or is caused by the test harness that is calling the code under test.
- If the exception is caused by a bug in the code under test, correct the code.
- If the exception is intentional, change the test case registration to use CPPTEST_TEST_EXCEPTION as described in “Test Suite/Test Case Registration Macros”, page 760.
- Often, the exception is not caused by the code under test (which is where the exception manifests itself), but rather caused by the code that calls it. Review the application code that calls the code under test to ensure that the application cannot produce the known exception-causing conditions exposed by the test harness.

Remember that the code under test might later be called by code that is not under your control— and which might create the conditions that could cause this exception. As a result, it’s essential to protect the code against the exception-causing inputs.

Timeouts

This topic provides details about results in the Timeout category.

Description

A Fix Unit Test Problems> Assertion Failures> Timeout message indicates that C++test timed out when trying to test a class. .

Recommended Response

Double-click the timeout message to determine the which test case is responsible for the timeout.

First, determine why the code under test is causing the test case to hang.

Response options are:

- If the behavior is correct and the tested code is just slow, change the timeout setting to a larger value in the Test Configuration's **Execution> Runtime** tab. See "Generation Tab Settings: Defining How Test Cases are Generated", page 701 for details.
- If it's possible to stub out the code that causes the test case to hang, you can write a stub method (see "Adding and Modifying Stubs", page 462 for details).
- Fix the code that hangs.

Unverified Outcomes

This topic provides details about results in the Unverified Outcome category.

Description

An unverified outcome message (a Review Unit Test Outcomes> Unverified Outcomes category message) is reported when C++test executes automatically-generated or user-defined test cases with postconditions that have not yet been converted to assertions The outcome might be the expected behavior, or it might indicate a problem. Further review and verification is required.

For details on reviewing the test case that produced this outcome, see "Reviewing Automatically-Generated Test Cases", page 381.

Recommended Response

Any test case that you want to use for regression testing should be verified. After an outcome is verified, the related test will fail if the same value is not achieved.

If you are confident that the outcomes represent correct code behavior, you can automatically verify all outcomes without reviewing them by right-clicking the Unverified Outcomes node, then choosing **Verify All** from the shortcut menu.

If you are not certain that the outcomes represent correct code behavior, reviewing and responding to unverified outcomes is recommended. Often, users discover unexpected behavior and other problems as they review unverified outcomes.

Right-click the message, then choose the appropriate Quick Fix option from the shortcut menu:

- If the test case outcome is correct (and the finding does not indicate a functional problem), choose **Verify outcome**.
- If you want to run the test case but not check the outcome in subsequent tests, choose **Ignore outcome**.

Using a Debugger During Test Execution

This topic explains how to step through C++test test cases with a debugger to better examine the code's internal state during a given test. For example, you might want to debug test cases to learn more about how C++test obtained an unexpected outcome, or to determine why a test case failed. You do not need to manually add breakpoints to the code. C++test will automatically set the breakpoints at the beginning of each test case that you select for execution. You can execute tests with your compiler's debugger, or debug directly in the Eclipse IDE (according to an Eclipse Debug Configuration you have configured).

Sections include:

Configuring Debugger Settings

The default debugger for GCC compilers is gdb. The default debugger for Sun CC compilers is dbx. The xxgdb and ddd debuggers are also supported.

C++test uses the following command-lines when launching the default debugger:

```
GNU GCC/Solaris: "/usr/openwin/bin/xterm -e gdb -x %s &"
GNU GCC/Windows: "cmd.exe /C gdb -x %s"
GNU GCC/Linux: "LC_ALL=C /usr/X11R6/bin/xterm -e gdb -x %s &"
Sun C++/Solaris: "/usr/openwin/bin/xterm -e dbx -c \"%s\" &"
```

To change the debugger or the debugger command line:

1. Right-click the project tree node for your project, then choose **Properties** from the shortcut menu. The Properties dialog will open.
2. Select the **Parasoft> C++test> Other Settings** category in the left pane.
3. In the Advanced Options table, click **Add** to add a new entry to the Advanced Options table, then add your preferred command line to the **Options** cell. Use the format `testrunner.debuggerCommandLine <COMMAND_LINE>`.
    - For example, to use the xxgdb debugger, use `testrunner.debuggerCommandLine /usr/X11R6/bin/xterm -e xxgdb -x %s &`.
    - To use the ddd debugger, use `testrunner.debuggerCommandLine ddd -x %s &`

Tip: Use the Standard Create Process Launcher

For the debug configurations, we recommend using the Standard Create Process Launcher (modified via **Windows> Preferences> Run/Debug> Default Launchers**) because this allows you to select a debugger. The default Eclipse launcher (GDB DSF) Create Process Launcher does not provide this option.

Executing Tests with a Debugger

To execute a test with your compiler's debugger:

1. Prepare a "Debug Unit Tests" Test Configuration.
    - Either use the built-in "Debug Unit Tests" Test Configuration or develop a custom Test Configuration by duplicating the built-in one, then customizing it as described in the Parasoft Test User's Guide (Parasoft Test User's Guide> Configuration> Configuring Test Configurations and Rules for Policies).

Debugging directly in Eclipse

If your compiler/debugger is supported by the Eclipse IDE, you can debug tests directly in the Eclipse IDE (according to an Eclipse Debug Configuration that you have configured) by creating a custom "Debug Unit Tests" Test Configuration. Set the following options in the Test Configuration's **Execution> Runtime** tab:

- Enable **Run tests in debugger**.
- Enable **Debug directly in Eclipse IDE**.
- Enter the name of the Eclipse Debug Configuration that should be used.

2. Run your preferred "Debug Unit Tests" Test Configuration as follows:
    a. Select test case(s) that you want to debug in one of the following ways:
        - Select the test case in the Test Case Explorer.
        - Select the test case function name in the code editor.
        - Select the test case function in the Project Explorer (not available for CDT 4.x+ Managed C/C++ projects).
    b. Launch your preferred "Debug Unit Tests" Test Configuration. For example, right-click the selection, then use the **Parasoft** shortcut menu to run the preferred Test Configuration. C++test will then launch appropriate debugger and automatically set the breakpoints at the beginning of each selected test case function.

Debugging in Microsoft Visual Studio

When using Microsoft Visual Studio 6 as the debugger environment, perform the following steps after the Visual Studio GUI opens:

- Select the test process in the **Attach to Process** dialog and click **OK**.
- Click **Break Execution** (or choose **Debug> Break** from the menu bar).
- Press the **F5 (Go)** button. The debugger will break at the beginning of the test case function.

When using Microsoft Visual Studio 2003 with Service Pack 1 as the debugger environment, perform the following steps after the Visual Studio GUI opens:

- Select the test process in the **Processes** dialog and click **Attach**.
- Confirm the application type (Native) by clicking **OK** in the Attach to Process dialog.
- Close the Processes dialog.

For other versions of Microsoft Visual Studio, no extra steps are needed.

3. Use standard debugger features to step through the test case.

Note

- Before you use your debugger with C++test, ensure that your debugger executable is included in the $PATH environmental variable.
- The debugger will be activated only if the debugging Test Configuration is run on a selection of one or more test cases. Otherwise, the following warning will appear in the console output and the execution will continue without debugging:

```
Warning: debugger will not be activated - no valid breakpoints found.
Select (test case) function definitions to set breakpoints.
```

Reviewing Automatically-Generated Test Cases

This topic explains how to access and explore C++test's automatically-generated test cases.

Sections include:

Accessing the Test Cases

The automatically-generated test cases are saved in test suite files. With the default settings, C++test generates one test suite per tested file. It can also be configured to generate one test suite per function (see "Test suite tab", page 701 for details).

To view the generated test cases:

1. In the project tree, locate the test suite file that C++test generated.
    - By default, automatically-generated test classes are saved in the `tests/autogenerated` directory within the tested project.
    - To check or change where C++test is saving the test suite files, open the Test Configurations dialog, select the Test Configuration that was used for the test run that generated the tests, then review the value in the **Generation> Test Suite** tab's **Test suite output file and layout** field (see "Test suite tab", page 701 for details).
2. Double-click the project tree node that represents the generated test class. The generated test class file will open in an editor.

Understanding the Test Cases

Test suites contain test cases, using the following architecture:

- **A Test Suite** is a kind of a framework for test cases. It helps in grouping test cases. Each test suite has a class definition that derives from the CppTest_TestSuite class, test case declarations, and definitions of the test case functions.
    - A test suite generated for a "C++" function or class method takes the form of a class and test cases are public methods of this class. If the tested method is a class member, then the test suite class is made a friend to the class that the tested method comes from. This association is very powerful, as it provides the ability to control private members of the tested class from inside a test case.
    - A test suite for C code is used only as a "container" (file) for declaring global functions.
- **A Test Case** is a Test Suite class' public method or a global function (depending on whether it is generated for C or C++) which contains three sections:
    - Arguments/Pre conditions assignments.
    - Tested function calls.
    - Results/Post conditions validations. C++test defines a set of macros for controlling and validating results. These macros can be used for passing messages/assertions

from the test executable to the C++test GUI. The available macros are listed at "C++test API Documentation", page 760.

If more functions are later added to the file under test, then test generation is run on the updated file, additional test cases will be added to the original test suite file.

Context and Include Macros

Every test suite begins with the following two macros:

- `CPPTEST_CONTEXT`: Specifies which file the test suite tests. Only one source file can be specified. If no context is specified, the test suite will be executed whenever the project is executed.
  - When a source file or directory is selected in a project tree before test execution is started, C++test scans all test directories specified in the Test Configuration's test search path. All test suites that match the selected context will be executed.
  - If the entire project is selected, all test suites on the test path will be executed.
  - If a test suite or single test is selected, the CONTEXT macro is used to back-trace to the source file to which this test suite is related. Only the selected test suite(s) will be executed.
- `CPPTEST_TEST_SUITE_INCLUDED_TO`: Indicates that it's an included test suite and specifies which file the test suite source will be included when the test executable is built. When you want to include additional files, use the #include directive.

The files specified by these two macros should always match the name of the file under test. If the name of the file under test changes, the values of these macros must be modified accordingly.

Relative paths can be used in both of these macros. Relative paths must start with "./" or with "../".

Postcondition Macros

The following postcondition macros, are used to report the value that a variable or class member had during the test execution:

- CPPTEST_POST_CONDITION_BOOL(value_string, value)
- CPPTEST_POST_CONDITION_INTEGER(value_string, value)
- CPPTEST_POST_CONDITION_UINTEGER(value_string, value)
- CPPTEST_POST_CONDITION_FLOAT(value_string, value)
- CPPTEST_POST_CONDITION_CSTR(value_string, value)
- CPPTEST_POST_CONDITION_CSTR_N(value_string, value, max_size)
- CPPTEST_POST_CONDITION_MEM_BUFFER(value_string, value, size)
- CPPTEST_POST_CONDITION_PTR(value_string, value)

The results of these macros (all the asserted values reported) are displayed in the Quality Tasks view. Any test case with reported postconditions can be automatically validated for use in regression testing. Validation changes the *_POST_CONDITION_* macros into assertions that will fail if a subsequent test does not produce the expected (validated) value. This is especially useful for automated generation of a regression base for legacy code. For details on validation, see "Verifying Test Cases for Regression Testing", page 409.

Other Macros

Assertion macros are added to the test case when postconditions are verified.

Additionally, other macros may be added manually, based on your testing needs.

For a list of macros that can be used in test cases, see “C++test API Documentation”, page 760.

Test Functions

Automatically-generated test cases sometimes include special functions (e.g., for accessing minimum of maximum values).

To generate test cases using different numeric values, the C++test runtime library contains a set of functions that returns min/max values of integral and floating point types. The functions are described in the following table. All listed functions can be used in user-defined test cases.

Function	Purpose
`char cpptestLimitsGetMaxChar(void);`	Returns maximum value of char type.
`char cpptestLimitsGetMinChar(void);`	Returns minimum value of char type.
`signed char cpptestLimitsGetMaxSigned Char(void);`	Returns maximum value of signed char type.
`signed char cpptestLimitsGetMinSigned Char(void);`	Returns minimum value of signed char type.
`unsigned char cpptestLimitsGetMaxUnsigned Char(void);`	Returns maximum value of unsigned char type.
`short cpptestLimitsGetMaxShort(void);`	Returns maximum value of short type.
`short cpptestLimitsGetMinShort(void);`	Returns minimum value of short type.
`unsigned short cpptestLimitsGetMax UnsignedShort(void);`	Returns maximum value of unsigned short type.
`int cpptestLimitsGetMaxInt(void);`	Returns maximum value of int type.
`int cpptestLimitsGetMinInt(void);`	Returns minimum value of int type.
`unsigned int cpptestLimitsGetMaxUnsigned Int(void);`	Returns maximum value of unsigned int type.
`long cpptestLimitsGetMaxLong(void);`	Returns maximum value of long type.
`long cpptestLimitsGetMinLong(void);`	Returns minimum value of long type.
`unsigned long cpptestLimitsGetMaxUnsigned Long(void);`	Returns maximum value of unsigned long type.
`float cpptestLimitsGetMaxPosFloat(void);`	Returns maximum positive value of float type.
`float cpptestLimitsGetMinNegFloat(void);`	Returns minimum negative value of float type.
`float cpptestLimitsGetMaxNegFloat(void);`	Returns maximum negative value of float type.
`float cpptestLimitsGetMinPosFloat(void);`	Returns minimum positive value of float type.
`double cpptestLimitsGetMaxPosDouble(void);`	Returns maximum positive value of double type.
`double cpptestLimitsGetMinNegDouble(void);`	Returns minimum negative value of double type.

Function	Purpose
`double cpptestLimitsGetMaxNegDouble(void);`	Returns maximum negative value of double type.
`double cpptestLimitsGetMinPosDouble(void);`	Returns minimum positive value of double type.
`long double cpptestLimitsGetMaxPosLongDouble(void);`	Returns maximum positive value of long double type.
`long double cpptestLimitsGetMinNegLongDouble(void);`	Returns minimum negative value of long double type.
`long double cpptestLimitsGetMaxNegLongDouble(void);`	Returns maximum negative value of long double type.
`long double cpptestLimitsGetMinPosLongDouble(void);`	Returns maximum positive value of long double type.

Understanding Automatically-Generated Stubs

See “Understanding and Customizing Automatically-Generated Stubs”, page 470.

Coverage Analysis

In this section:

Generating Coverage Information

This topic explains how to generate the coverage information from test sessions and how to adjust the setttings to minimize the execution overhead

Enabling and Disabling Coverage Analysis

The instrumentation setting in the execution tab must be configured to collect code coverage information. C++test provides an option for setting which coverage metrics should be collected during tested code execution. See “Understanding Coverage Types”, page 390 for more information.

If performance is an issue, such as in embedded testing, you can also disable all code coverage monitoring. See “Execution Tab Settings: Defining How Tests are Executed”, page 703 for more information about controlling coverage analyasis.

Code coverage information can be collected from unit tests execution as well as from manual testing performed on your application built by C++test specifically for coverage monitoring.

Understanding Code Coverage Analysis overhead

By default, the code coverage information generated during tested code execution is immediately sent to C++test, which ensures accurate results. For example, if the test execution results in a crash, all coverage information up to the point where the crash occurred will be valid. The disadvantage of this approach is relatively high execution time overhead. The execution overhead for unit tests, however, is negligible in most cases. As a result, the default coverage mode is recommended.

Execution time overhead related to on-the-fly coverage data transfer (default mode), though, may not be acceptable for collecting coverage information from application level tests (application functional testing). For example, it is very common for coverage instrumentation to have an impact on application timing dependencies that affect the application logic in embedded systems.

If the default code coverage execution time overhead is unnaccetable, users can enable a special optimized mode to collect coverage from application level functional testing. In the optimized mode:

- Dedicated memory buffers are used to store information about covered C/C++ language constructions
- Coverage data is sent at application finalization; *not* during functional tests execution
- Code instrumentation executes the minimum required amount of machine instructions to store the information about covered C/C++ language statements.
- Therefore there is no requirement to assure a fast link between host and target to minimize overhead.

Optimized Coverage Metrics for Application Coverage Monitoring

The special optimized coverage instrumentation mode is only designed for application level testing and is not available for unit testing. Use the advanced settings from the Test Configuration's Execution Tab "Instrumentation features" panel to enable optimized coverage mode. These settings are described in “Execution Tab Settings: Defining How Tests are Executed”, page 703.

The Advanced settings section enables you to:

- Choose lower tested code execution time overhead or lower overhead on ram memory.

- Initialize coverage memory buffers
- Protect against coverage data buffers corruption

Dedicated initialization and finalization functions must be called to run an optimized code coverage monitoring session:

```
void CppTest_InitializeRuntime(void)
void CppTest_FinalizeRuntime(void)
```

By default, these functions are called at the beginning and end of the main function (C language application) or via the special global object constructor and destructor added for this purpose (C++ language application).

If your application does not have a "main" function or does not support construction/destruction of global objects, you can manually inject calls into the initialization and finalization functions in appropriate locations during the application startup and finalization.

Reviewing Coverage Information

This topic explains how to review coverage information from tests run with C++test. Coverage can be tracked for unit tests and manual or automated tests run at the application level.

Reviewing coverage statistics helps you measure the coverage of your current test suite and decide what additional test cases should be added. C++test can report a variety of code coverage types, including line, path, basic block, decision (branch), simple condition, and MCDC coverage metrics.

Sections include:

- Analyzing Coverage in the GUI
- Analyzing Coverage in Reports
- Understanding Coverage Types
- Increasing Coverage

Analyzing Coverage in the GUI

Coverage Summary

To view a summary of coverage information after C++test executes unit test cases:

1. Open the Coverage view.
    - If it is not available, choose **Parasoft> Show View> Coverage**.
2. From the pull-down menu in the Coverage view's toolbar, choose **Type> [Desired_Coverage_Type]**.
    - See "Understanding Coverage Types", page 390 for a description of the available coverage types.

Coverage statistics for the project and all analyzed files and functions will be displayed in the Coverage view. If no coverage is shown after test execution, check that 1) coverage was enabled and 2) you are displaying the type of coverage you configured C++test to calculate. The view's toolbar indicates the coverage metric being displayed.

You can sort and search through the results using the available GUI controls.

Annotated Source Code

When a tested file is opened in the editor, C++test will use green highlights to indicate which code was cover ed and pink highlights to indicate which code was not covered. Lines that are not executable will not be highlighted.

Note that you need to double-click on a function to start off path coverage view.

Note

There is a limit for the number of coverage elements (paths, blocks etc.) that can be computed by C++test. When the actual number of elements on a given level (function, file or a project) exceeds the limit, which is 2147483647, C++test will display "N/A" in the report for a given element. In such cases, the Coverage view will contain appropriate messages, e.g. [no paths], [no blocks].

Back Trace from Coverage Elements to Test Cases

Knowing which test cases are related to each coverage element can help you better assess how to extend test cases to improve coverage.

To see which test cases are related to a coverage element:

1. Select that coverage item in the editor.
    - C++test will consider the current selection / cursor position, not the mouse pointer location.
2. Right-click that selection, then choose **Parasoft> C++test> Show test case(s) for covered element**. All related test cases will be highlighted in the Test Case Explorer.

Coverage Data for Specific Test Cases

To analyze the coverage for individual test cases:

1. In the Test Case Explorer, select the test case(s) whose coverage you want to analyze.
    - The Test Case Explorer opens by default. If it is not available, you can open it by choosing **Parasoft> Show View> Test Case Explorer**. For details on understanding and navigating the Test Case Explorer, see the Parasoft Test User's Guide (Parasoft Test User's Guide> Introduction> About the Parasoft Test UI).
2. In the Coverage view, click the **Synchronize with selection in Test Case Explorer** filter (a double cog-wheel button).

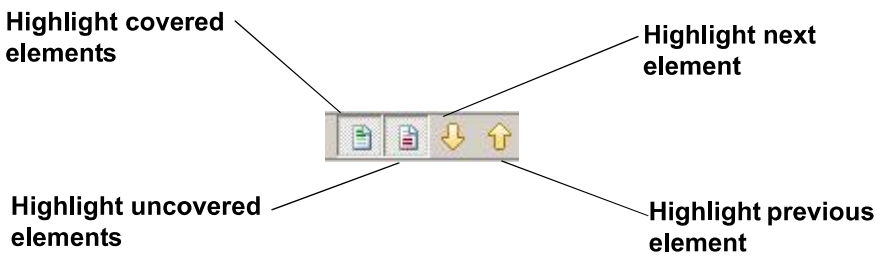
Coverage statistics and coverage highlights (in the code editor) will be computed/presented for the selected test cases only.

Tip: Using Toolbar Buttons and Menus to Explore Coverage

The Coverage view provides several buttons and menu commands to help you explore the coverage details reported.

The buttons on the right of the toolbar allow you to collapse, delete, search for results, and synchronize with the current selection in the Test Case Explorer.

In addition, the following toolbar buttons allow you to show/hide covered elements, show/hide uncovered elements, highlight the next path (for path coverage only), and highlight the previous path (for path coverage only).



The drop-down menu provides commands that allow you to sort results by ascending/descending name or coverage, as well as to select the desired coverage type.

Note: Coverage View Results Upon Startup

To optimize performance, the Coverage view starts in a uninitialized mode—displaying only project-level coverage summaries. This mode is indicated with grayed project icons:

Coverage in this mode may not be current—for instance, if the source code was modified outside of this IDE. You can update the coverage view by simply expand a project node. Or, you can wait for the view to be automatically updated during execution.

Analyzing Coverage in Reports

For details on configuring, generating, and reviewing reports, see “Reviewing Results”, page 269.

Understanding Coverage Types

C++test supports the following coverage types:

- Line Coverage
- Statement Coverage
- Block Coverage
- Path Coverage
- Decision (Branch) Coverage
- Modified Condition/Decision Coverage (MC/DC)
- Simple Condition Coverage

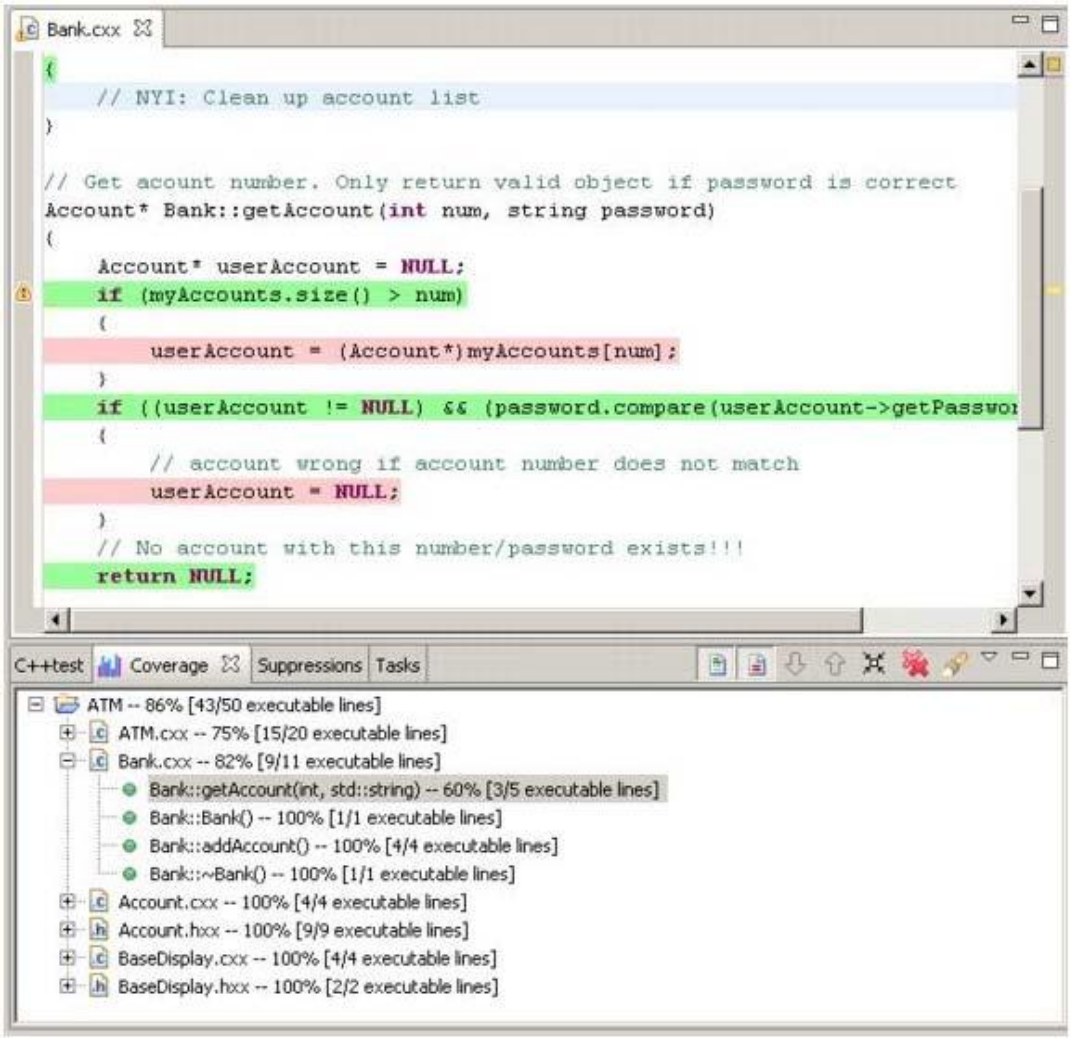
To help understand how these coverage types are handled by C++test, be sure to read and understand the terms in the following table:

Concept	Description
Basic Block	A sequence of non-branching statements; a linear sequence of code with no control flow route branchings.
Path	A unique sequence of basic blocks starting from the function entry to the point of exit.
Decision/ Branch	Decision/Branch is the possible control flow decision to be taken at the branching point in the code. C++test considers if-else, for, while, do-while, and switch instructions as the branching points. C++test does not take into account such dynamic branching points as exception handlers (throw-catch statements).

Concept	Description
Boolean expression	In C++, a boolean expression is simply an expression that has a 'bool' type. In C, C++test treats the following as boolean expressions: • A relational operator (<, <=, ==, >, >=, !=) with non boolean arguments. • Each argument of boolean operator (||, &&, !). • Condition in if, for, while and do-while instructions. • Condition in ? operator.
MC/DC Decision	A top-level boolean expression composed of conditions and zero or more boolean operators. C++test computes MC/DC and SCC on all boolean non-constant expressions in the source code except constructor initializers and function default arguments.
Condition	An atomic boolean non-constant expression that is a part of the MC/DC decision. A sub-expression of the MC/DC decision is considered to be a condition if it does not contain boolean operators (&&, ||, !). If a given atomic expression appears more than once in a decision, each occurrence is a distinct condition.

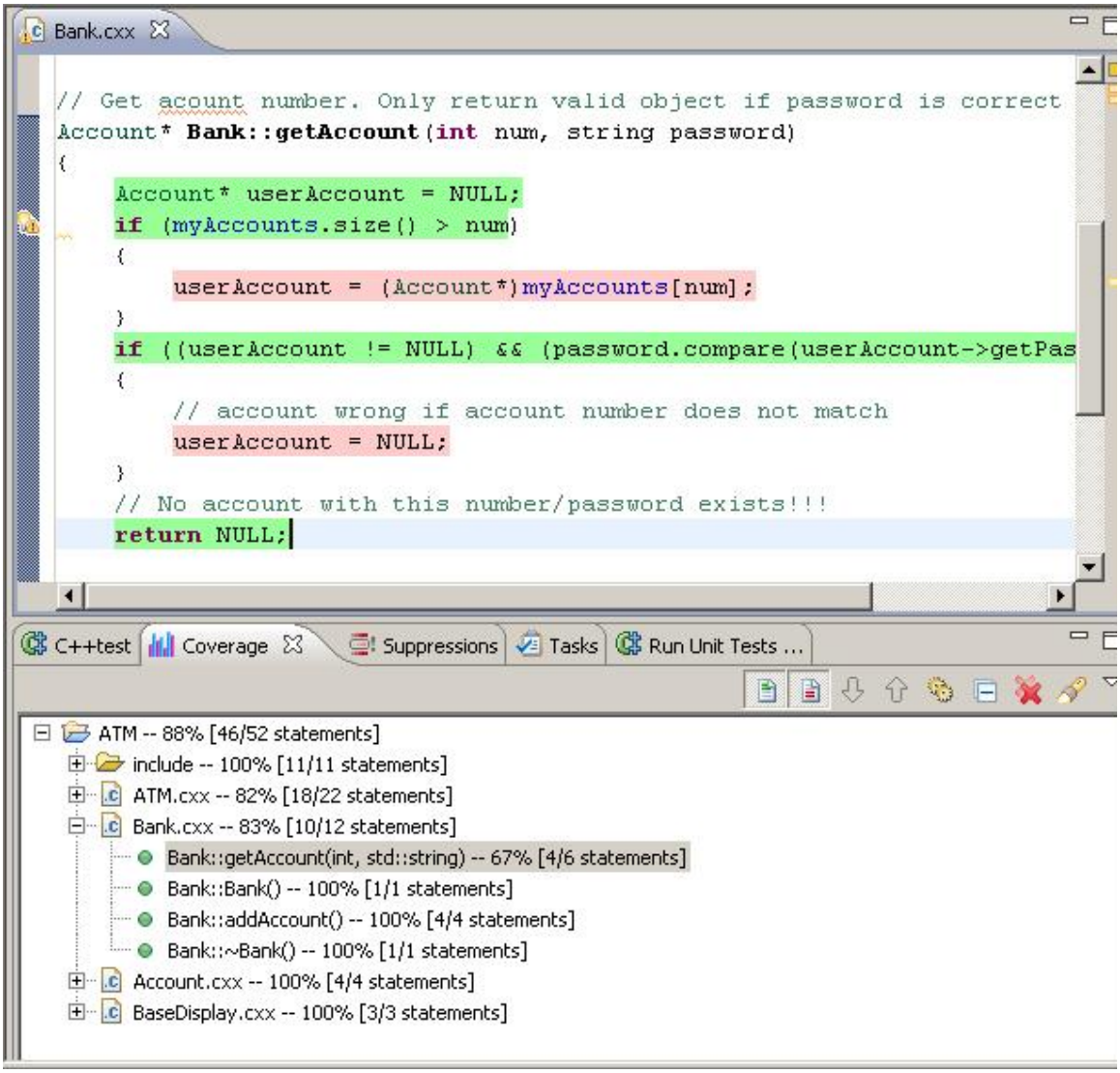
Line Coverage

Indicates how many executable lines of source code were reached by the control flow at least once. Complete, 100% line coverage is obtained if all executable lines are reached at least once.



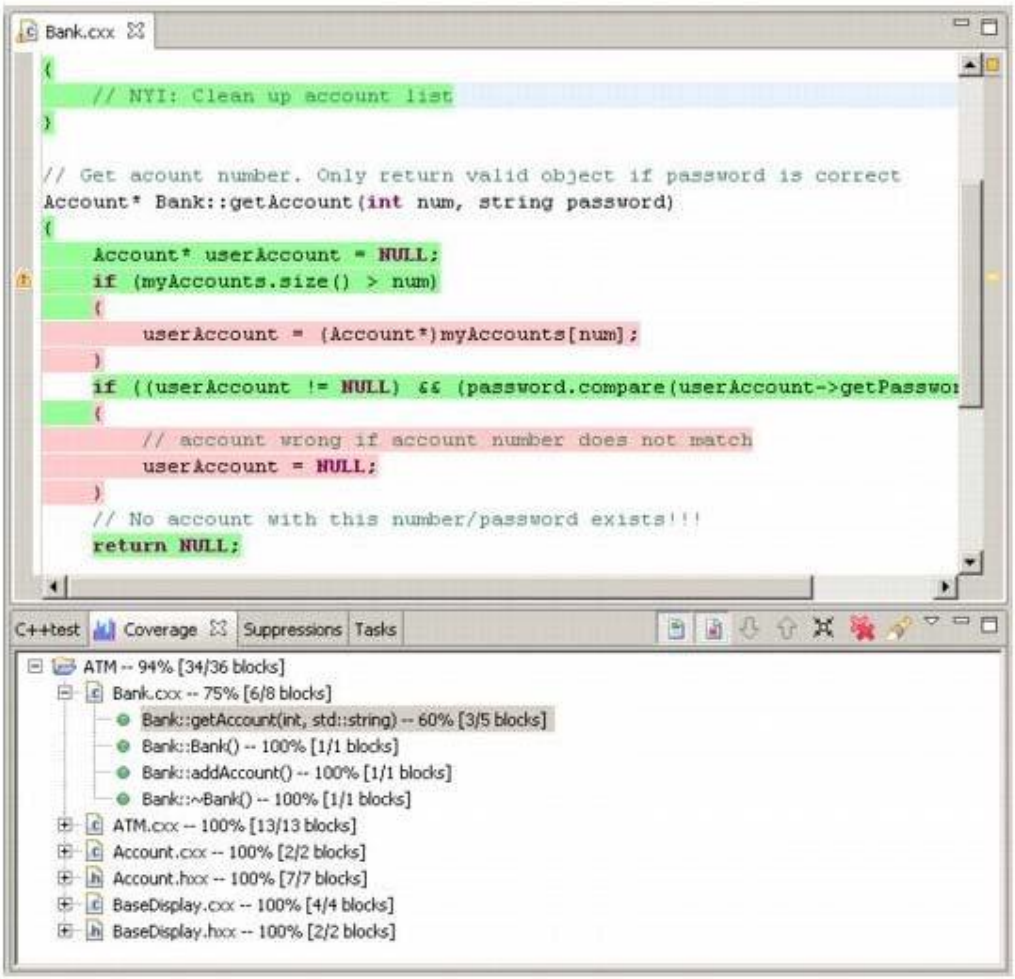
Statement Coverage

Indicates how many executable source code statements were reached by the control flow at least once. Complete, 100% statement coverage is obtained if all executable statements are reached at least once.



Block Coverage

Similar to Line Coverage—except that with Block Coverage, the unit of measured code is a basic block (see the definition of this term in the previous table). This indicates how many basic blocks in the source code were reached by the control flow at least once.



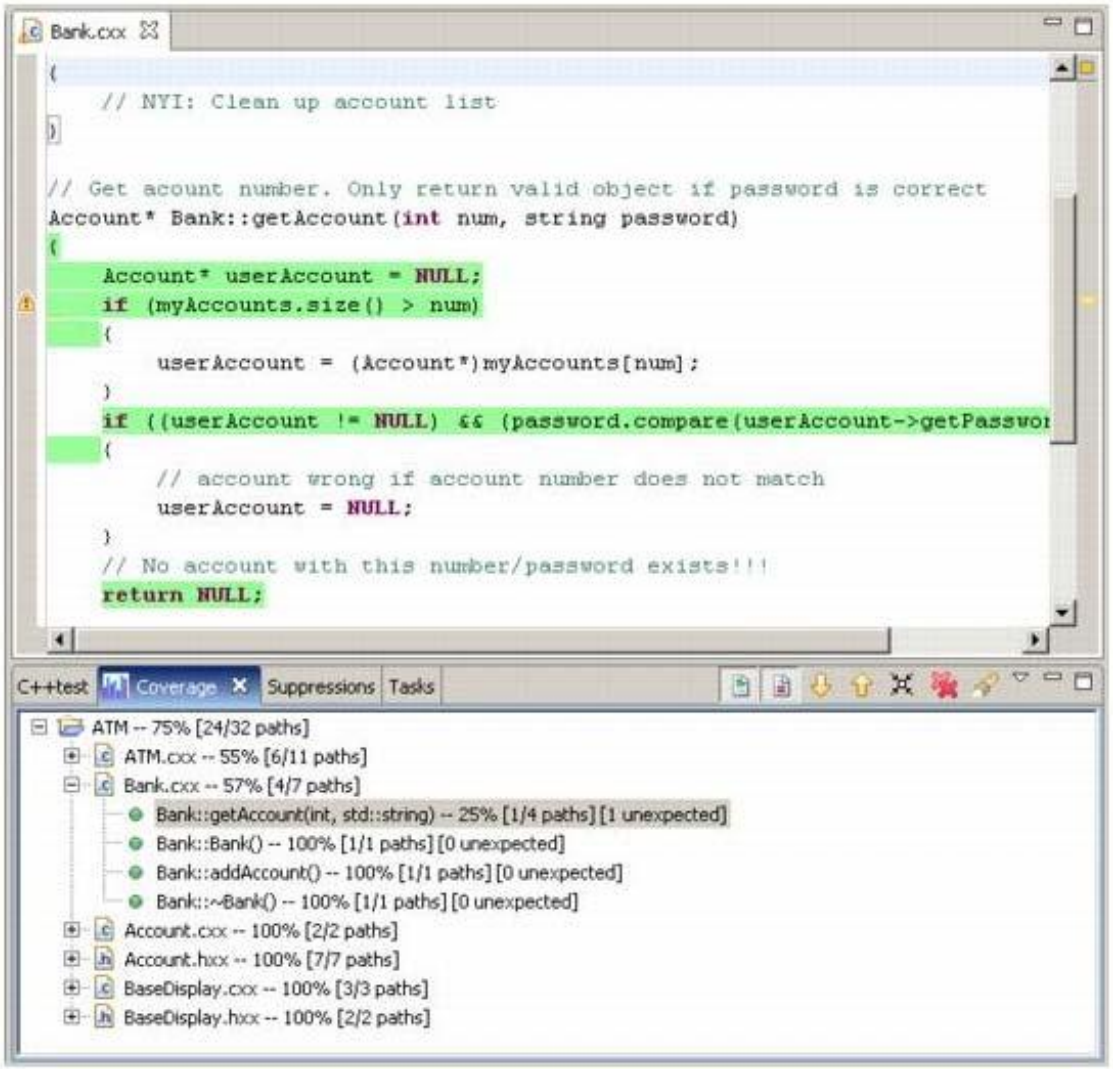
Path Coverage

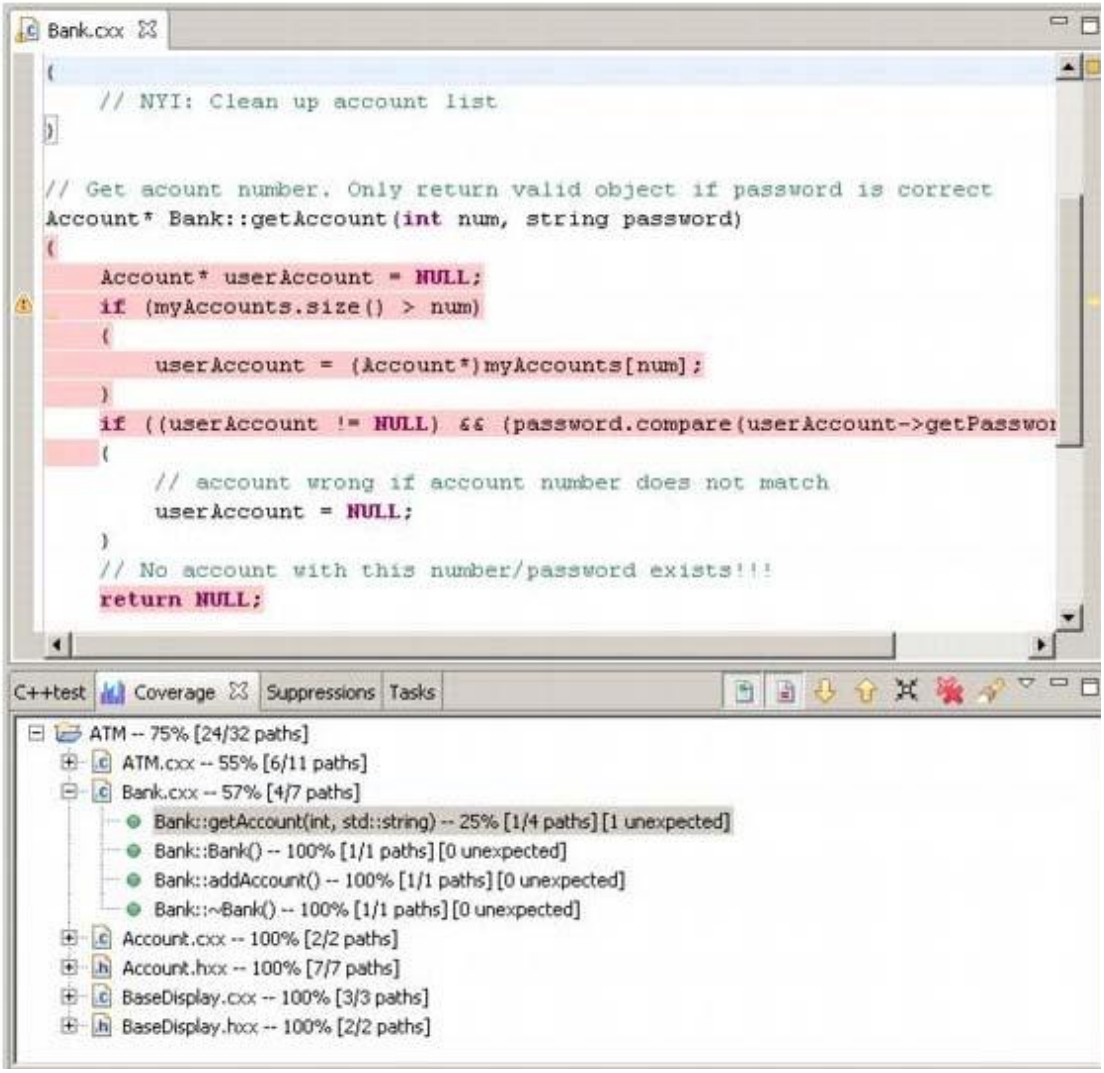
Indicates if each possible path in a given function was followed by the control flow. Branching points used to single out paths (see an explanation of this term in the previous table) are the same as in Decision (Branch) Coverage.

Since loops introduce an unbounded number of paths, this measure considers only a limited number of looping possibilities. C++test considers two possibilities for while-loops and for-loops: zero and at least one repetition.

In the source editor, C++test highlights one path at a time. To view a path in the source code editor, double-click on the appropriate function node in the Coverage view. To navigate between paths for the function, use the **Highlight next element** and **Highlight previous element** buttons in the Coverage view toolbar.

To prompt C++test to show only uncovered paths, disable the **Highlight covered elements** button in the Coverage view toolbar. To prompt C++test to show only covered paths, disable the **Highlight not covered elements**. Note that the **Highlight next/prev element** buttons iterate through unexpected paths only when the **Highlight not covered elements** button is disabled.


Bank.cxx
{
// NYI: Clean up account list
}
// Get acount number. Only return valid object if password is correct
Account* Bank::getAccount(int num, string password)
{
Account* userAccount = NULL;
if (myAccounts.size() > num)
{
userAccount = (Account*)myAccounts[num];
}
if ((userAccount != NULL) && (password.compare(userAccount->getPasswo
{
// account wrong if account number does not match
userAccount = NULL;
}
// No account with this number/password exists!!!
return NULL;
C++test
Coverage
Suppressions
Tasks
ATM -- 75% [24/32 paths]
ATM.cxx -- 55% [6/11 paths]
Bank.cxx -- 57% [4/7 paths]
Bank::getAccount(int, std::string) -- 25% [1/4 paths] [1 unexpected]
Bank::Bank() -- 100% [1/1 paths] [0 unexpected]
Bank::addAccount() -- 100% [1/1 paths] [0 unexpected]
Bank::~Bank() -- 100% [1/1 paths] [0 unexpected]
Account.cxx -- 100% [2/2 paths]
Account.hxx -- 100% [7/7 paths]
BaseDisplay.cxx -- 100% [3/3 paths]
BaseDisplay.hxx -- 100% [2/2 paths]

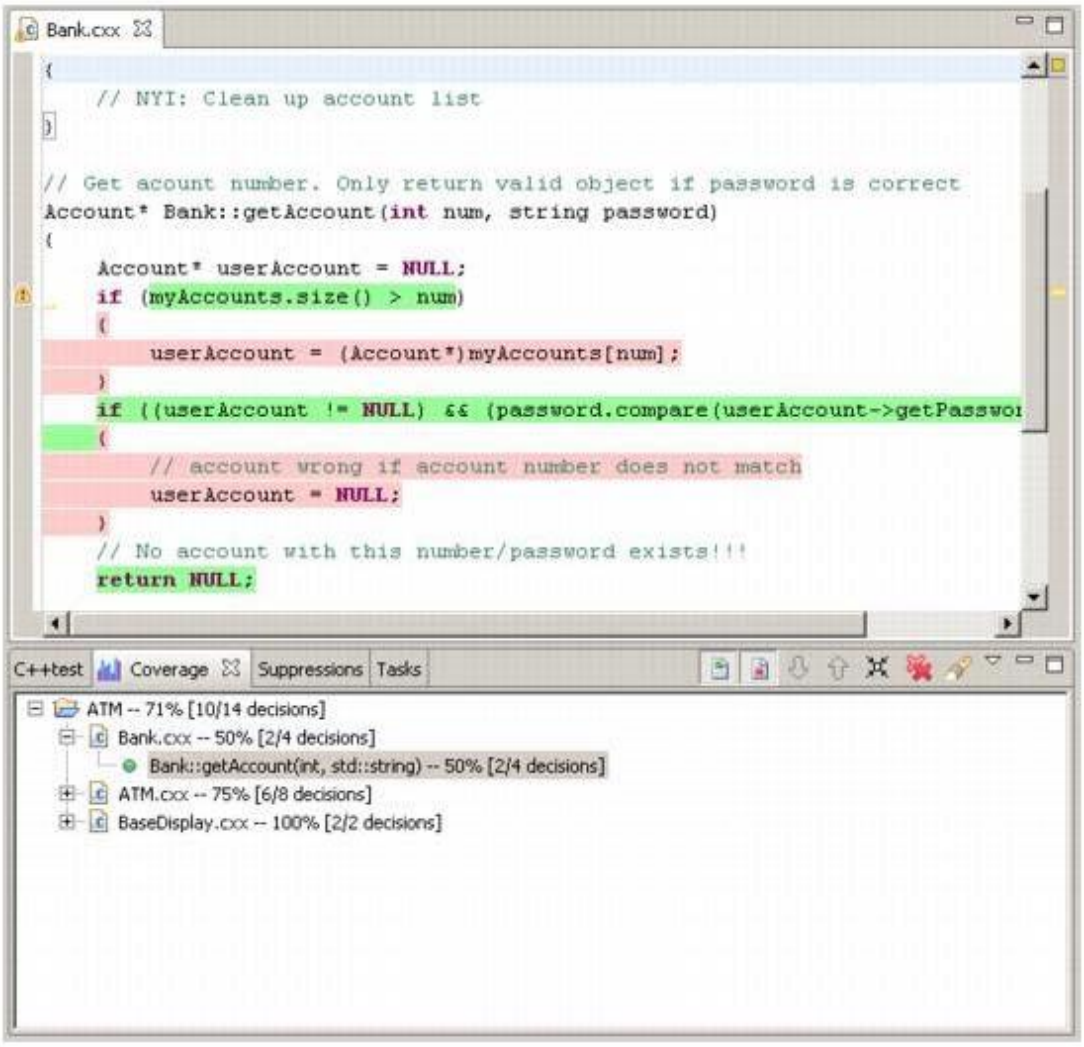


Decision (Branch) Coverage

Indicates how many branches in source code control flow passed through. Complete, 100% coverage is obtained when every decision at all branching points took all possible outcomes at least once.

C++test considers the following statement types branching points in source code: `if-else`, `for`, `while`, `do-while`, and `switch`. C++test does not take into account such dynamic branching points as exception handlers (`throw-catch` statements).

If there are no decisions in a file, C++test reports that this metric is not available (using an "N/A" label).



Modified Condition/Decision Coverage (MC/DC)

MC/DC conforms to the international technical standard DO-178B (RTCA), which specifies the software certification criteria for mission-critical equipment and systems within the aviation industry. This includes real time embedded systems.

According to the DO-178B standard, the following three conditions must be satisfied to obtain complete (100%) MC/DC coverage:

a. Every decision has taken all possible outcomes at least once.

b. Every condition in a decision has taken all possible outcomes at least once.

c. Every condition in a decision has been shown to independently affect that decision's outcome.

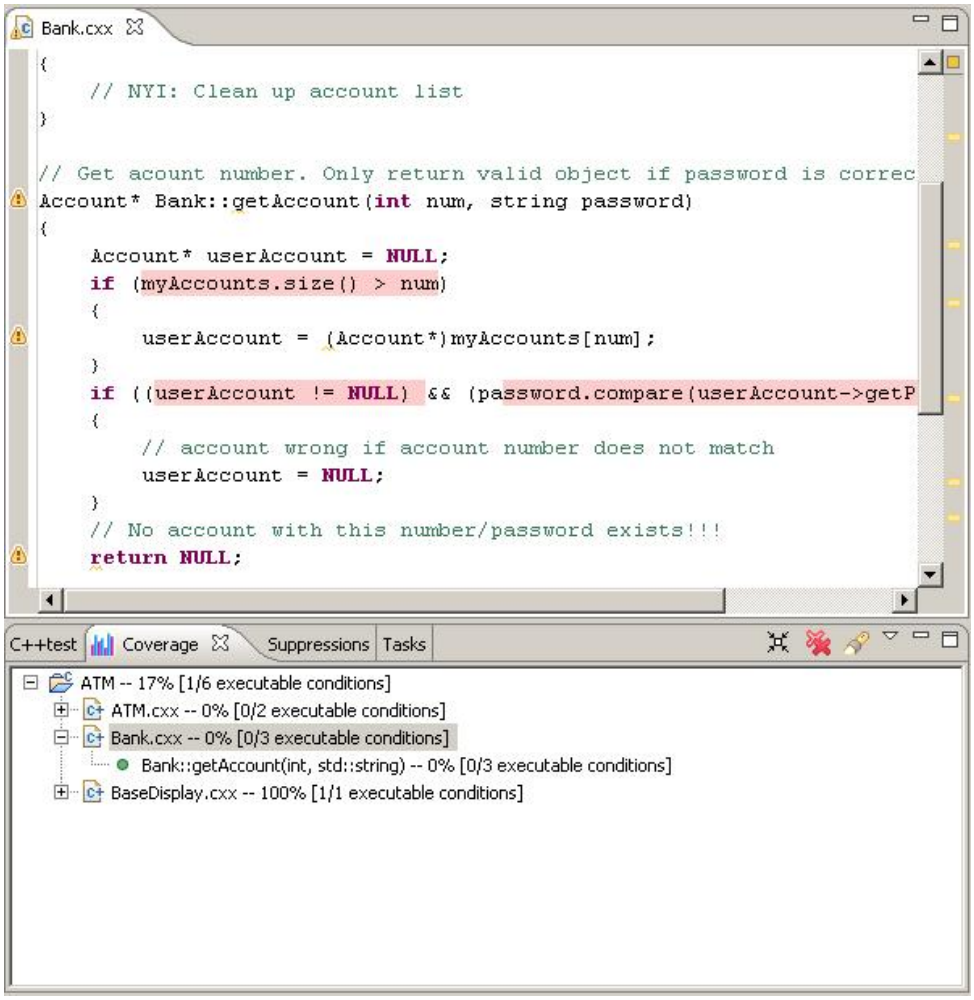
Since C++test considers that every condition and decision can have only two outcomes for MC/DC coverage (true or false), it checks only for the third option (c) listed immediately above—since point (c) implies conditions (a) and (b). A condition is shown to independently affect the outcome of a decision by varying only that particular condition while holding fixed all other possible conditions. To test one given condition, C++test looks for test cases where:

- The tested condition have both true and false outcomes.
- Other conditions in a decision do not change (or are not evaluated because operators in C/C++ are short-circuit logical).
- The outcome of a decision changes.

Thus, to calculate the MC/DC ratio, C++test uses the formula
`MC/DC[%] = m/n`

where `m` is the number of boolean conditions proven to independently affect a decision's outcome, and `n` is the total number of conditions in a decision.

Note that in order to make a single condition covered, at least two test cases need to be executed: one with the condition evaluated to `true` and a second with this condition evaluated to `false`.



MC/DC Example

For example, consider the following code:

```
if (a && (b || c))
   // [...]
```

There are three simple conditions here: 'a', 'b' and 'c'. As a result, `n` (in the `m/n` MC/DC formula) will equal 3.

Now, let's assume that the following test cases were executed:

id	a	b	c	a && (b || c)
1	true	true	false	true
2	false	true	false	false
3	true	false	false	false

To compute MC/DC, C++test looks for the pairs of test cases where 1) the given decision was evaluated to a different value and 2) the value of only one condition changes (all other conditions remain unchanged).

In our example, there is one pair where independently changing the value of `a` affects the value of the complete decision:

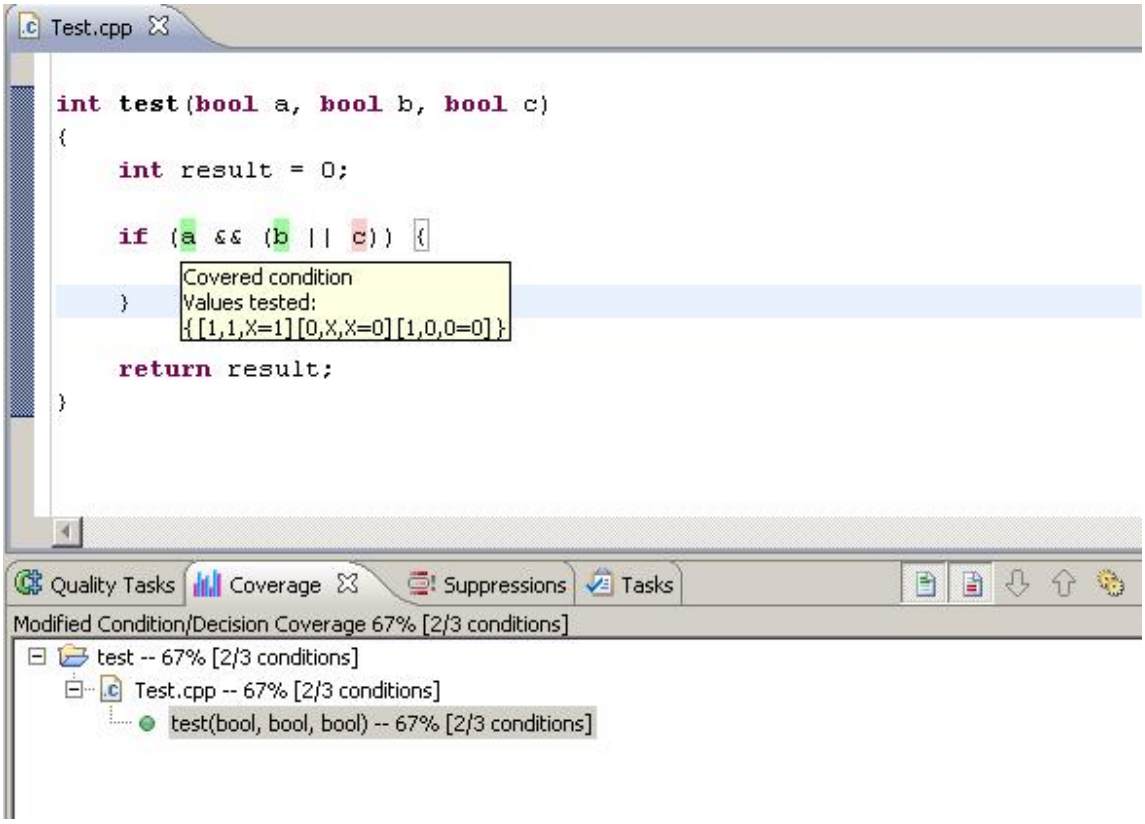
id	a	b	c	a && (b || c)
1	true	true	false	true
2	false	true	false	false

Another pair shows that independently changing the value of `b` affects the value of the complete decision:

id	a	b	c	a && (b || c)
1	true	true	false	true
3	true	false	false	false

This means that our test cases proved that `a` and `b` independently affect the value of the complete decision. In terms of MC/DC coverage, they are covered and the `c` condition is not covered.

C++test will report the MC/DC coverage for such an example to be 67% [2/3 conditions covered]. You can see the actual values that the conditions and decision evaluated to in a tool tip by placing your cursor above on the condition.

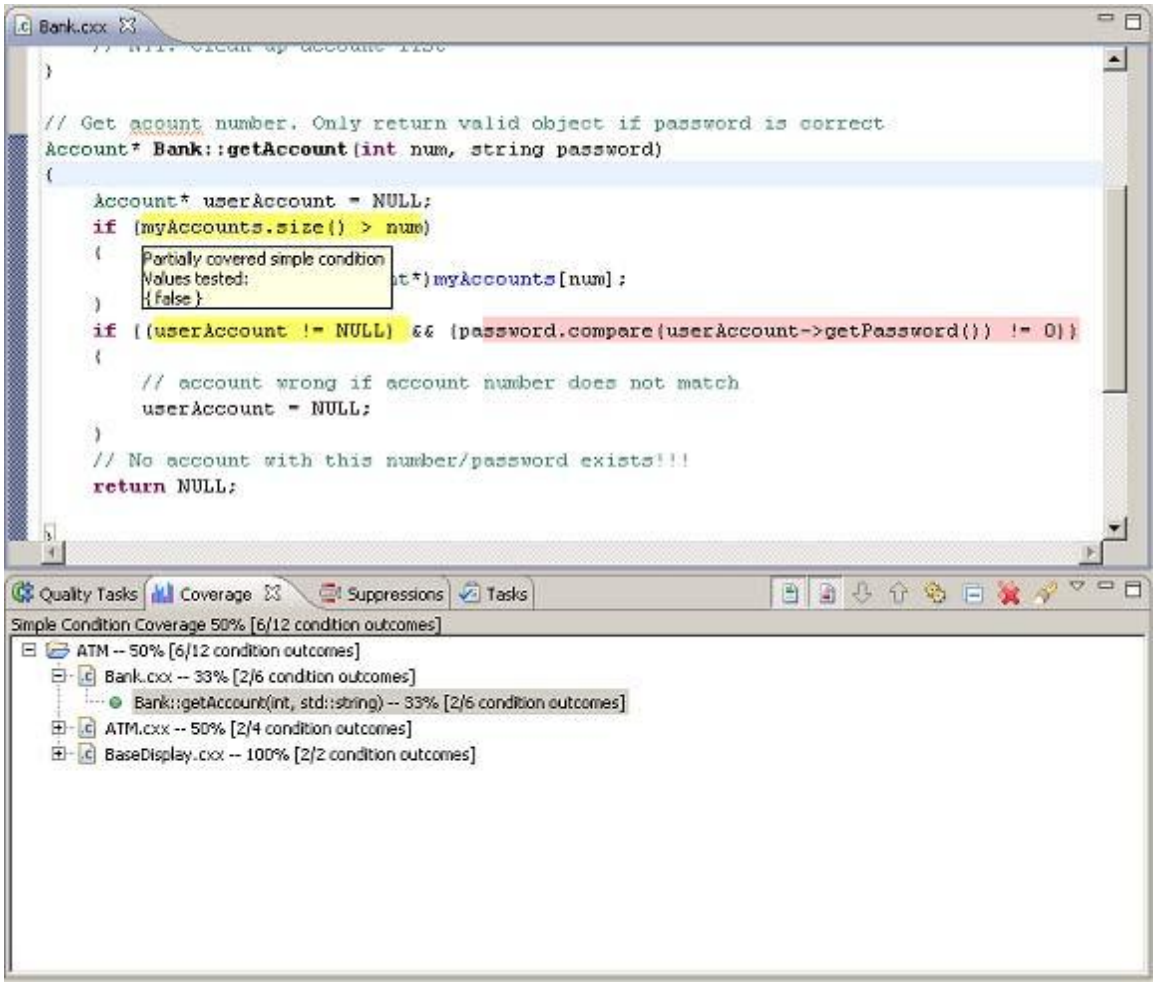


Simple Condition Coverage

Indicates the coverage for the outcomes of all decisions' conditions. The overall number of outcomes for a decision equals 2 * n, where n is the number of conditions in a decision. Therefore, to obtain 100% coverage, all conditions must take all possible outcomes. However, to obtain non-zero coverage, one condition only needs to take one outcome.

Each condition is a boolean condition. If it evaluated to both true and false, it is marked in green. If it evaluated to either true or false, it is marked in yellow. It if did not evaluate to true or false, it is marked in pink.

To see the actual boolean value that the condition evaluated to, place your cursor above it. The value will be shown in a tool tip.



Increasing Coverage

Strategies for improving coverage are discussed in “Improving Test Coverage”, page 403.

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”, page 388). 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”, page 411).
- Using specific constructors or constructor call sequences to create an object in a desired state (See “Adding User-Defined Test Cases”, page 411).
- Using user defined stubs (See “Adding and Modifying Stubs”, page 462).
- Using test case driven stubs (See “Using Stubs Driven By Test Cases”, page 472).

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:
    a. 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.
    b. 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
    c. 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)
    d. 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.
    e. 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.

Extending and Modifying the Test Suite

In this section:

Extending and Modifying the Test Suite: Overview

This topic provides an overview of the different ways that you can extend and modify the automatically-generated unit test suite.

Use the following table as a reference to determine which test suite extension/modification methods are best suited to your goals

To achieve this	Do this	Reference
Add new test cases to check specific unit-level functionality requirements or to improve coverage	Use the Test Case Wizard to create test cases graphically, or add code to test case templates	“Adding User-Defined Test Cases”, page 411
Modify automatically-generated test cases to check specific unit-level functionality requirements or to improve coverage	Edit the related test files	“Adding User-Defined Test Cases”, page 411
Modify test generation or execution settings	Configure settings in the Test Configuration panel’s Generation and Execution tabs	“Generation Tab Settings: Defining How Test Cases are Generated”, page 701 and “Execution Tab Settings: Defining How Tests are Executed”, page 703
Remove test cases and disable outcome checks or test cases that are not currently of concern to you	To remove a test suite: Right-click its Test Case Explorer node and choose **Delete** To remove a test case: Right-click its Test Case Explorer node and choose **Delete** To disable checking of a specific outcome: Right-click the unverified outcome in the Quality Tasks view, then choose **Ignore Outcome** from the shortcut menu To disable a complete test case: Right-click its Test Case Explorer node and choose **Disable**	“Deleting and Disabling Tests”, page 459
Prevent the testing of certain classes or methods	Specify the resources that you want to include or exclude	“Testing a User-Defined Set of Resources”, page 217

To achieve this	Do this	Reference
Convert automatically-generated tests into a "functional snapshot" for regression testing (to identify changes/problems introduce by code modifications)	If the code is behaving correctly, right-click the unverified outcome node, then choose **Verify Outcome** from the shortcut menu	“Verifying Test Cases for Regression Testing”, page 409
Access data source values during testing	Configure test cases to access values stored in a data source.	“Using Data From Data Sources to Parameterize Test Cases”, page 429
Use standard I/O data in test cases	Add C++test Stream API calls for redirecting standard input/output stream	“Using Data From Standard IO”, page 454
Define custom stubs (to specify what values an external method/function returns to the class under test)	Use the Stub wizard to create a stubs framework, then customize it.	“Adding and Modifying Stubs”, page 462
Execute existing unit test cases (i.e., CppUnit test cases) using C++test	Ensure that your preferred test execution Test Configuration can find the test cases	“Executing Manually-Written CppUnit Test Cases”, page 474

Verifying Test Cases for Regression Testing

This topic explains how to convert test cases into regression test cases through test case verification.

Sections include:

Tip: Generating Regression Tests that Don’t Require Verification

If you want to create a snapshot for regression testing (e.g., if you are confident that the code is behaving as expected), use the "Unit Testing> Generate Regression Base" built-in Test Configuration.

When this Test Configuration is run, C++test will automatically verify all outcomes.

During subsequent tests C++test will report tasks if it detects changes to the behavior captured in the initial test. Verification is not required.

About Verification

When C++test automatically generates unit tests for code that behaves as expected, the generated test cases form a "functional snapshot": a suite of unit tests that capture the code's current behavior, which is assumed to be correct. This test suite is essentially an executable specification. It can be used to establish a baseline that can be used to identify problems and changes introduced by new/modified code. When your goal is to establish a baseline rather than verify the application's current functionality, you do not need to review the outcomes for these test cases. Moreover, because you want to maintain the integrity of this baseline, you do not want or need to recreate unit tests for this same code. As the application evolves, you can test new and modified code against this baseline to ensure that changes have not impacted or "broken" the previously-verified functionality.

Any test case that you want to use for regression testing should be verified. When a test case is verified, postconditions (which capture the actual value that a variable or class member had during the test execution) are converted to assertions that will be checked in all subsequent tests. After an outcome is verified, the related test will fail if the same value is not achieved.

We recommended verifying user-defined test cases that use postconditions, as well as any automatically-generated test cases that you have reviewed and deemed useful for regression testing. When reviewing automatically-generated test cases, it is helpful to expand the test case node to display the postcondition details, which are automatically added for the function return value and direct member variables for object under test.

Verifying Test Cases

Automated Verification of Test Cases with Unverified Outcomes

Test cases with unverified outcomes (postconditions rather than assertions) can be verified automatically.

To automatically verify one or more test case outcomes that are marked as Unverified Outcomes:

- Right-click the unverified outcome (or a node representing a set of unverified outcomes) in the Quality Tasks view, then choose **Verify Outcome** from the shortcut menu.

All post condition macros (whether added manually or automatically) from selected context(s) will be converted to appropriate assertions. For example:

CPPTEST_POST_CONDITION_INTEGER("{int}_return=", _return) is converted to CPPTEST_ASSERT_EQUAL(0, _return),

CPPTEST_POST_CONDITION_FLOAT("{float}_return=", _return) is converted to CPPTEST_ASSERT_DOUBLES_EQUAL(1.000000e+000, _return, 0.00001),

CPPTEST_POST_CONDITION_PTR("{int *}_return=", _return) is converted to CPPTEST_ASSERT(_return != 0).

Manual Verification

To manually verify test cases:

1. In the project tree, locate the test suite file that C++test generated.
    - By default, automatically-generated test classes are saved in the `tests/autogenerated` directory within the tested project.
    - To check where C++test is saving the test suite files, open the Test Configurations dialog, select the Test Configuration that was used for the test run that generated the tests, then review the value in the **Generation> Test Suite** tab's **Test suite output file and layout** field (see "Test suite tab", page 701 for details).
2. Double-click the project tree node that represents the generated test class. The generated test class file will open in an editor.
3. Use the available macros to convert the postconditions to assertions.
    - See "C++test API Documentation", page 760 for a list of macros.
4. Make any additional modifications needed.
5. Save the modified file.

Adding User-Defined Test Cases

This topic explains how you can extend the test suite adding new test cases via a graphical editor.

Sections include:

The test suite can also be extended with your existing unit test cases (i.e., CppUnit test cases); this is described in “Executing Manually-Written CppUnit Test Cases”, page 474.

Overview of Available Options

C++test provides several ways to add user-defined test cases. Available options are as follows (in the recommended order):

- Test Case Wizard - This is designed to facilitate black box testing of new code. It provides a simple way to specify inputs and expected outputs, and it offers a better analysis of the tested code’s data dependencies (e.g. global variables), which facilitates test case setup. Using the Test Case Wizard is the only way to automatically generate a data source template for a test case.
    - Potential drawback: Can only create one test at a time.
- Automated test case generation - This is designed to facilitate rapid development of a large number of customizable/extensible tests in bulk. This approach supports characterization testing on volumes of existing code, where tests are generated for many functions at once, and results of tests need to be captured automatically.
    - Potential drawback: Test case editing is required to control test case values.
- Test Suite Wizard - This enables you to quickly create a form for writing complex test code that complies with the C++test test framework.
    - Potential drawback: Code needs to be added to the template.

Adding Test Suites and Test Cases with the Graphical Test Case Wizard

The Test Case Wizard allows you to select a function to test, then graphically specify test case preconditions and postconditions. These test cases can be parameterized with data source values (or automatically-generated values) to rapidly create test case scenarios that ensure broad, thorough test coverage, and that test the code's response to a wide range of inputs.

Test cases created in the Test Case Wizard are saved in source code, using the standard C++test test format (similar to CppUnit).

Adding Test Suites

All test cases must exist within a test suite. Before you can add test cases with the graphical Test Case Wizard, you must first have a test suite that can contain them.

Generating a Test Suite

To prompt C++test to generate a set of empty test suites for each testable context (file or function):

1. Select the resource(s) for which want to generate a test suite.
2. Run the "Unit Testing> Generate Test Suites" Test Configuration or a custom team Test Configuration that is based on it.
    - Any custom Test Configuration used for this purpose must be set to generate tests, but have the **Max. number of generated test cases (per function)** parameter set to zero.

Specifying a New Test Suite

If you want more control over the nature of the test suite than the automated generation (described above) allows, add a new test suite as follows:

1. In the Test Case Explorer, right-click the node for the project that the test suite will test, then choose **Add New> Test Suite** from the shortcut menu
    - The Test Case Explorer is open by default, in the left side of the UI. If it is not available, you can open it by choosing **Parasoft> Show View> Test Case Explorer**. For details on understanding and navigating the Test Case Explorer, see the Parasoft Test User's Guide (Parasoft Test User's Guide> Introduction> About the Parasoft Test UI).
2. Enter the test suite parameters in the Test Suite Wizard dialog. Available parameters are:
    - **Test suite name:** Determines the test suite's name.
    - **Test suite location:** Determines the test suite's location.
    - **Test suite file(s):** Reports the path to the test suite file(s).
    - **Test suite language:** Determines whether the test suite is implemented in C or C++.
    - **Tested file**: Sets the test suite context to the specified tested source file. The specified file will be set as the `CPPTEST_CONTEXT` macro, which associates a given test suite file with a specified source file.
        - If a file was selected when you started the wizard, that file will be specified here.
        - If a project was selected when you started the wizard, no context will be specified, and the test suite will be at the project scope (which means it will only get executed if all tests for the project are run, or if it is selected as the test file). When it is selected as the test file (and does not have the CONTEXT) C++test

assumes it has the project context, and prepares all files in the project to link against.

- **Test suite mode:** Determines whether the test suite is implemented as an included test suite, or a standalone test suite. C++test instruments both types of test suites; it can access private/protected class members.
    - Included test suites are physically included in one of the generated test harness source files. "Included" test suites are combined with instrumented code and compiled to form a single object file. All automatically-generated test suites are included test suites. Additional header files can be included and macro definitions can be defined in original headers. However, included test suites do not need any headers included, unless there are types used that are not visible in the original tested source file. Typically, this is only necessary when you are modifying generated tests (e.g., to include a test factory, etc.).
    - Standalone test suites are compiled separately and linked in with the test executable for the testable context. Any additional headers must be included directly in these test suites. Coverage information can be tracked for included headers.
- **When referencing tested file in test suite:** Determines whether the CPPTEST_CONTEXT and CPPTEST_TEST_SUITE_INCLUDED_TO macros use the full project path or relative paths.
    - In most cases, using full paths is recommended. Relative paths can be helpful in special cases, such as when you want to generate tests for code that is used in different locations, and you want to use the tests in multiple locations as well. For example, assume that you have source files for a library that can be used by many different projects, and you have some tests connected with that source code. In that case, no matter where that source code is placed in the projects, the connected tests should work with it (because no full paths are used).

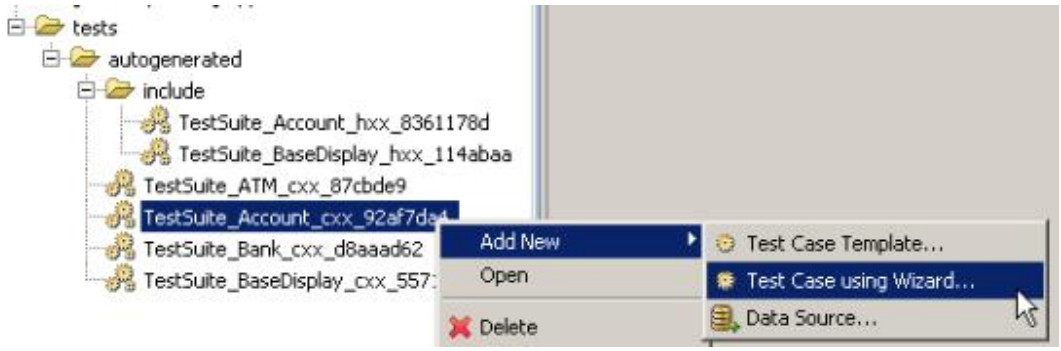
Adding Test Cases Using the Graphical Wizard

You can use the Test Case Wizard to specify new test cases graphically, using GUI controls. C++test generates test case code representing the specified tests, and adds this code to the corresponding test suite. These test cases can be executed along with the other test cases, and they can be modified/extended as needed.

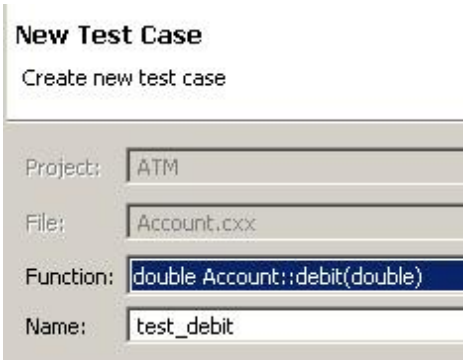
To add a new test case using the Test Case Wizard:

1. In the Test Case Explorer, select the node for the test suite that you want to include the new tests.
    - The Test Case Explorer is open by default, in the left side of the UI. If it is not available, you can open it by choosing **Parasoft> Show View> Test Case Explorer**. For details on understanding and navigating the Test Case Explorer, see the Parasoft Test User's Guide (Parasoft Test User's Guide> Introduction> About the Parasoft Test UI).

2. Right-click the selection, then choose **Add New> Test Case using Wizard** from the shortcut menu.

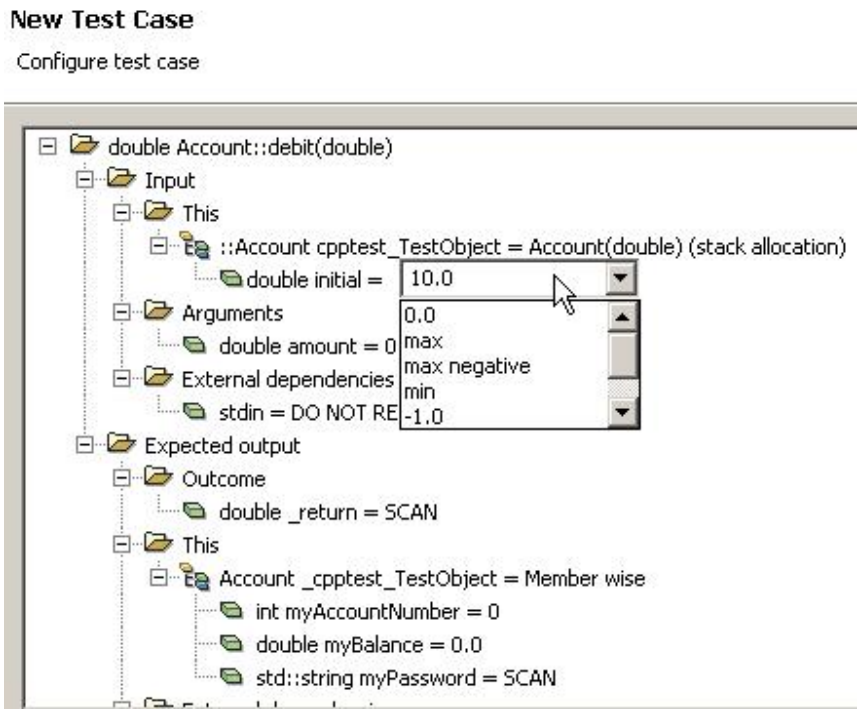


3. On the first page, specify the source file (compilation unit) and the function for which you want to add a test case, then enter a name for the test case.
    - The option to specify a source file is available only if your test suite was configured as a "standalone" test suite.
    - If the test suite was configured as "included", you must choose among functions from the compilation unit that the Test Suite is included to.
    - If the test suite has a context file specified, you must choose among functions defined in the given source/header file.



4. Click **Next** to open the next wizard page.
5. Configure the test case by specifying its input and expected output values using GUI controls.

- See “Test Case Configuration Tips”, page 415 for details.

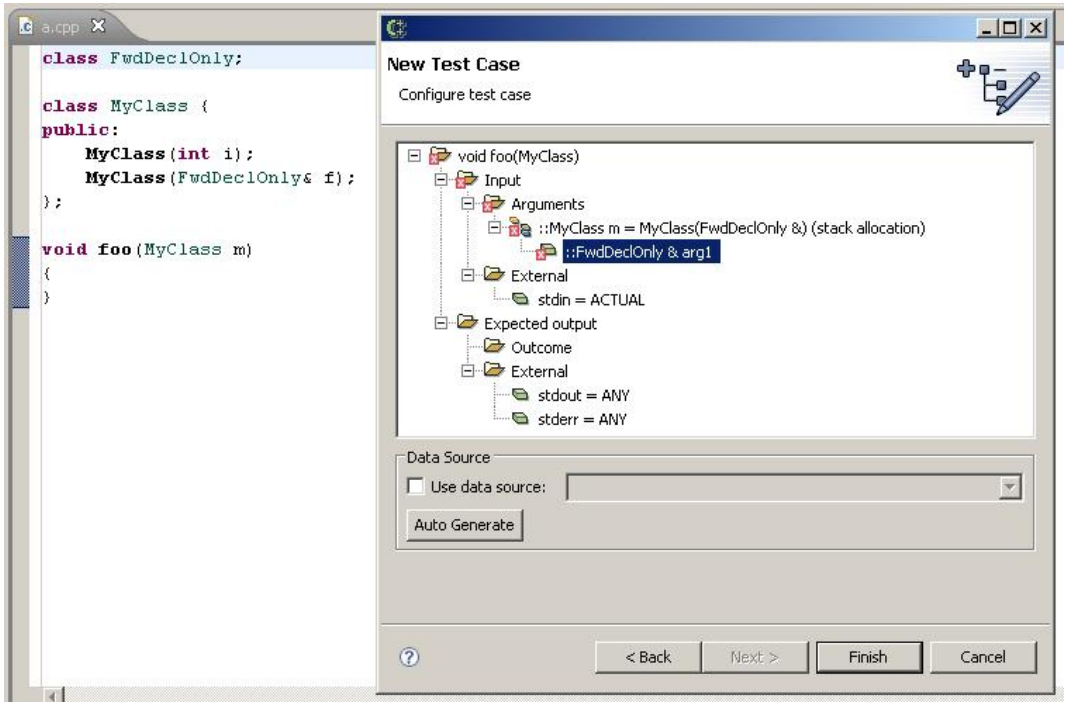


6. Click **Finish** to generate the test case. The new test case will be added to the test suite and the generated source code will be opened in the editor.

Test Case Configuration Tips

- You can specify the following pre-conditions for the test case:
  - arguments for the tested function
  - (for the non-static member functions) value of the tested object ('this')
  - values of global variables used by the tested function
  - value of the standard input stream (see “Using Data From Standard IO”, page 454)
- In addition, you can specify the following expected post-conditions:
  - return value of the tested function
  - (for non-const reference and pointer types) values of the tested function arguments
  - (for the non-static member functions) state of the tested object ('this')
  - values of global variables used by the tested function
  - value of the standard output / standard error streams (see “Using Data From Standard IO”, page 454)
- To specify a description for a test case, add it in the **Additional settings> Test case description** field. It will be saved with the generated test case.
- To have C++test insert macros that report the values of test case inputs and outputs, enable the **Additional settings > Insert code to report test case inputs and outputs** option.

- To change the value for a given pre- or post-condition, double-click it then select the appropriate initializer type.
- For simple type nodes (boolean, integer, floating point, string types), you can provide an appropriate value by editing the node value.
- Values that do not need to be initialized (e.g., global variables or the value of the standard input stream) can be left uninitialized by selecting the 'LEAVE NOT INITIALIZED' or 'ACTUAL' value.
- To exempt a post-condition node from verification, select the 'ANY' value. This will prevent C++test from generating any assertion macros for the given outcome.
- For each simple type post-condition node (boolean, integer, floating point, string types) as well as for post-conditions of pointers to simple types (e.g. char* or int*), you can have C++test scan the actual value by selecting the 'ACTUAL' value. This will prompt C++test to generate a post-condition macro instead of the assertion for the given outcome in the resulting test case.
- To learn how to use data sources values in test cases, see “Using Data Source Values to Parameterize Test Cases”, page 432.
- If C++test is unable to provide any valid initializer for a given test case object (e.g. to create a reference to a forward-declared class), then the related test case tree node will be marked with a red X icon. This icon will also mark all parent nodes up to the top-level node representing the tested function. Generating a test case in this state will result in C++test creating an incomplete test case that will need to be modified manually.



Adding Test Suites and Test Cases Manually

Adding New Test Suites

To add user-defined test cases:

1. If you have not already done so, create a new directory for your tests.
    - The test directory can be located anywhere, as long as it is visible in the project tree. By default, C++test expects tests to be stored in a subdirectory of the project's `tests` directory. However, you can use a different location, as long as you modify the Test Configuration's **Test suite file search patterns** setting (in the **Execution> General** tab) accordingly.
    - If you do not want to store your tests within the project directory, you can add a folder that links to files stored elsewhere in your file system. To do this:
        a. Choose **File> New> Folder** (if this is not available, choose **File> New> Other**, select **General> Folder**, then click **Next)**.
        b. Click the **Advanced** button.
        c. Enable the **Link to folder in file system** option.
        d. Enter or browse to the location of your source files.
        e. Click **Finish**.
2. Open the Test Suite Wizard as follows:
    a. Select your tests directory in the project tree.
    b. Right-click the selection, then choose **New> Other** from the shortcut menu. A wizard will open.
    c. Select **Parasoft> C++test> Test Suite**, then click **Next**.
3. Enter the test suite parameters in the Test Suite Wizard dialog. Available parameters are:
    - **Test suite name:** Determines the test suite's name.
    - **Test suite location:** Determines the test suite's location.
    - **Test suite file(s):** Reports the path to the test suite file(s).
    - **Test suite language:** Determines whether the test suite is implemented in C or C++.
    - **Tested file**: Sets the test suite context to the specified tested source file. The specified file will be set as the `CPPTEST_CONTEXT` macro, which associates a given test suite file with a specified source file.
        - If a file was selected when you started the wizard, that file will be specified here.
        - If a project was selected when you started the wizard, no context will be specified, and the test suite will be at the project scope (which means it will only get executed if all tests for the project are run, or if it is selected as the test file). When it is selected as the test file (and does not have the CONTEXT) C++test assumes it has the project context, and prepares all files in the project to link against.
    - **Test suite mode:** Determines whether the test suite is implemented as an included test suite, or a standalone test suite. C++test instruments both types of test suites; it can access private/protected class members.
        - Included test suites are physically included in one of the generated test harness source files. "Included" test suites are combined with instrumented code and compiled to form a single object file. All automatically-generated test suites are included test suites. Additional header files can be included and macro definitions can be defined in original headers. However, included test

suites do not need any headers included, unless there are types used that are not visible in the original tested source file. Typically, this is only necessary when you are modifying generated tests (e.g., to include a test factory, etc.).

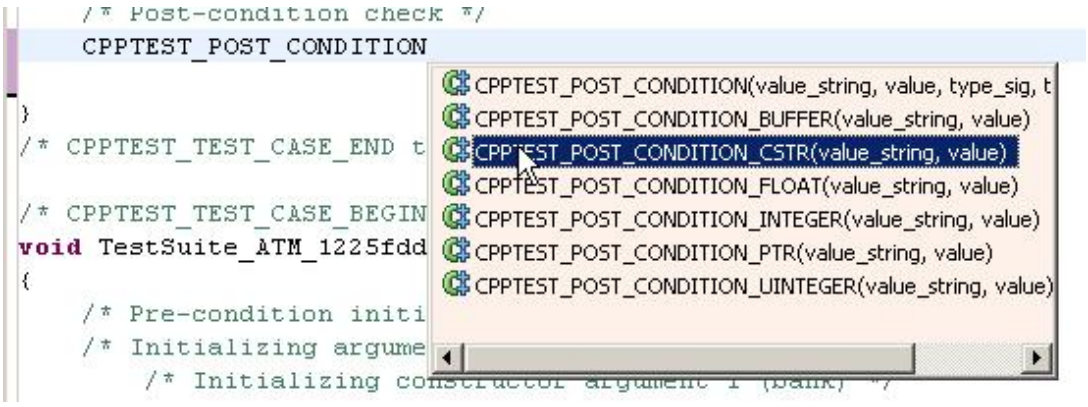
- Standalone test suites are compiled separately and linked in with the test executable for the testable context. Any additional headers must be included directly in these test suites. Coverage information can be tracked for included headers.

4. (Optional) If you want to specify test cases at this point (i.e., so C++test registers them in the test suite and adds a skeletal framework for the tests), click **Next**, then use the controls to add test case names and specify the order in which you want them added.
5. Click **Finish**.

Adding Test Cases to an Existing Test Suite

To add a new test case to an existing test suite:

1. Select the test suite file in the project tree, or open it in the editor, then right-click the selection, then choose **Parasoft> C++test> Add Test Case Template** from the shortcut menu. Or, right-click a the test suite in the Test Case Explorer and choose **Add New> Test Case Template**.
2. Enter a name for the test case. The new test case will be added, and the test suite will be modified accordingly.
3. Open the test suite source file in the editor, enter test case definitions, and make additional modifications as needed. You can...
    - Use standard C or C++ code, the macros described in "C++test API Documentation", page 760, the routines described in "Available Test Functions", page 426, and the postconditions described in "Test Case Post-Condition Macros", page 765.
        - The Content Assist feature help you add macros and postconditions. For instance, to add a postcondition template, type `CPPTEST_POST_CONDITION`, place your cursor after the N, press **Ctrl + Space**, then select the desired postcondition.To add an assertion template, type `CPPTEST_ASSERT`, place your cursor after the T, press **Ctrl + Space**, then select the desired assertion. Be sure to customize the added templates.



- Use include directives and macro definitions defined in header files.

- Prompt C++test to include additional files when building a test executable by using an #include directive.
- Change the test suite context by changing the `CPPTEST_CONTEXT` macro, which associates a given test suite file with a specified source file. Only one source file can be specified. If no context is specified, the test suite will be executed whenever the project is executed.
    - When a source file or directory is selected in a project tree (e.g. a source file) before test execution is started, C++test scans all test directories specified in the Test Configuration's test search path. All test suites that match the selected context will be executed.
    - If the entire project is selected, all test suites on the test path will be executed.
    - If a test suite or single test is selected, the CONTEXT macro is used to backtrace to the source file to which this test suite is related. Only the selected test suite(s) will be executed.
    - If relative paths are used, they must start with "./" or with "../".

4. Save the modified file.

Test Driven Development (TDD) Tip

The test cases in the generated template are set to fail. For TDD, you can write the source for the implementation, then remove the failure macros when you are satisfied with the test.

cout Statements that Spawn New Processes

If you have unit tests with `cout` statements that spawn other processes, use `endl` instead of the newline character to end the `cout` statements. This way, the output gets flushed immediately to the console.

For example, use this:

- cout << "xyz" << endl;

not this:

- cout << "xyz\n";

Modifying Automatically-Generated Test Cases

User-defined test cases can often be defined by modifying automatically-generated test cases.

To modify an existing test case:

1. In the Test Case Explorer or in the project tree, locate the test suite file that C++test generated.
    - By default, automatically-generated test classes are saved in the `tests/autogenerated` directory within the tested project.
    - To check where C++test is saving the test suite files, open the Test Configurations dialog, select the Test Configuration that was used for the test run that generated the tests, then review the value in the **Generation> Test Suite** tab's **Test suite output file and layout** field (see "Customizing Generation Options", page 350 for details).

2. Double-click the node that represents the generated test class you want to modify. The generated test class file will open in an editor.
3. Modify the test case as needed. You can...
    - Use standard C or C++ code, the macros described in “C++test API Documentation”, page 760, the routines described in “Available Test Functions”, page 426, and the postconditions described in “Test Case Post-Condition Macros”, page 765.
        - The Content Assist feature help you add macros and postconditions. For instance, to add a postcondition template, type `CPPTEST_POST_CONDITION`, place your cursor after the N, press **Ctrl + Space**, then select the desired postcondition.To add an assertion template, type `CPPTEST_ASSERT`, place your cursor after the T, press **Ctrl + Space**, then select the desired assertion. Be sure to customize the added templates.
    - Use include directives and macro definitions defined in header files.
    - Prompt C++test to include additional files when building a test executable by using an #include directive. Included test suites (the test suite type used for autogenerated test suites) do not need any headers included, unless there are types used that are not visible in the original tested source file. For example, additional headers might need to be included when you are modifying generated tests to include a test factory, etc.
    - Change the test suite context by changing the `CPPTEST_CONTEXT` macro, which associates a given test suite file with a specified source file. Only one source file can be specified. If no context is specified, the test suite will be executed whenever the project is executed.
        - When a source file or directory is selected in a project tree (e.g. a source file) before test execution is started, C++test scans all test directories specified in the Test Configuration’s test search path. All test suites that match the selected context will be executed.
        - If the entire project is selected, all test suites on the test path will be executed.
        - If a test suite or single test is selected, the CONTEXT macro is used to backtrace to the source file to which this test suite is related. Only the selected test suite(s) will be executed.
4. Save the modified file.

Using Setup and Teardown Functions

You can perform test case setup and cleanup actions using setUp and tearDown functions. setUp and tearDown are called before and after each test case, respectively.

To perform setup or cleanup actions for the whole test suite (called before and after each test suite), define custom setup/cleanup functions and register them with the following macros:

```
CPPTEST_TEST_SUITE_SETUP(setup_function_name)
CPPTEST_TEST_SUITE_TEARDOWN(cleanup_function_name)
```

The registration should be added within the test suite declaration section (before the first test case registration). For example:

```
class MyTestSuite : public CppTest_TestSuite
{
  public:
      CPPTEST_TEST_SUITE(MyTestSuite);
      CPPTEST_TEST_SUITE_SETUP(myTestSuiteSetUp);
      CPPTEST_TEST_SUITE_TEARDOWN(myTestSuiteTearDown);
```

```
      CPPTEST_TEST(test1);
      CPPTEST_TEST(test2);
      CPPTEST_TEST_SUITE_END();

      void setUp();
      void tearDown();

      static void myTestSuiteSetUp();
      static void myTestSuiteTearDown();

      void test1();
      void test2();
};

void MyTestSuite::myTestSuiteSetUp()
{
  // Test suite setup
}

void MyTestSuite::myTestSuiteTearDown()
{
  // Test suite cleanup
}
```

Notes

- Test suite setup/cleanup functions take no parameters.
- We recommend using the static storage class specifier for C++ test suite setup/cleanup functions.
- We do not recommend adding constructors/destructors to the test suite and attaching setup/teardown code to them because the test suite could be instantiated multiple times to run a subset of test cases each time—or even one per instance of the test suite. It is not correct to assume that the constructor is called only once before any test suite tests are executed, or that the destructor is correspondingly called only after all tests have completed.

Using Different Tests and/or Stubs for Different Contexts

If you want to use different test cases and/or stubs for different testing contexts, you can create a different set of test cases and/or stubs for each testing context, then create a Test Configuration that uses each set of test cases and/or stubs. This is helpful in a number of situations; for example:

- If you want to use one set of tests and stubs when testing on the host system and another set when testing on the target device).
- If you want team members to perform fast "sanity checks" on the developer workstation, then have C++test server run a comprehensive regression test on the entire code base each night.

Test Driven Development (TDD) Tip

Typically, teams following TDD have Test Configurations for the following scenarios:

- A "sandbox" test scenario for tests to run before checking in code. These tests should take 20 minutes or less to execute.
- A "continuous integration" test scenario for executing a slightly larger set of tests. These tests may take over an hour to run.
- An "overnight testing" test scenario for executing all available tests.

In this case, you can set up Test Configuration with nested inclusion: for instance, the sandbox configuration will execute only the tests in dir1, the "continuous integration" configuration will execute tests in dir1 and dir2, then the "overnight testing" configuration will execute tests in dir1, dir2, and dir3.

If you want to use different tests and/or stubs for different contexts:

1. Store each set of test cases and stubs in a separate directory.
2. Create one Test Configuration for each testing context, and configure it to use the appropriate set of test cases and stubs for the given context.
    - The test case location is specified in the **Execution> General** tab's **Test suite location patterns** field.
    - The stub location is specified in the **Execution> Symbols** tab's **Use extra symbols from files found in** field.

Specifying Custom Compiler Options for Standalone Test Suites

You can set custom compiler options (for instance, to use C++test-specific flags) for each standalone test suite as described in "Specifying Custom Compiler Settings and Linker Options for Testing with C++test", page 208

Measuring and Validating Response Time in Real-Time Systems

To measure and validate response time in real-time systems, use the following macros:

Name	Details
`CppTest_Time CppTest_TimeInit(int seconds, int nanoseconds);`	Initializes the CppTest_Time structure with the given values. Nanoseconds should be between -999999999 and 999999999

Name	Details
`CppTest_Time CppTest_TimeCurrent();`	Initializes the CppTest_Time structure with the current time. Time is stored as number of seconds (and nanoseconds) since 00:00:00 UTC, January 1, 1970. Accuracy depends on platform used.
`CppTest_Time CppTest_TimeDiff(CppTest_Time t1, CppTest_Time t2);`	Returns the difference between t1 and t2.
`int CppTest_TimeCompare(CppTest_Time t1, CppTest_Time t2);`	Compares two CppTest_Time structures. Returns: • < 0: If t1 is smaller/earlier than t2 • == 0: If t1 and t2 are equal • > 0: If t1 is bigger/later than t2

Here is an example of how to use these macros to measure and report time:

```
   ...
   /* Collect start time */
   CppTest_Time start = CppTest_TimeCurrent();
   /* Tested function call */
   int _return  = ::foo();
   /* Compute time diff */
   CppTest_Time stop = CppTest_TimeCurrent();
   CppTest_Time diff = CppTest_TimeDiff(stop, start);
   /* Report time diff (assuming positive values) */
   CPPTEST_MESSAGE(cpptestFormat("Call time for foo(): %d.%09d", diff.seconds, diff.nanosec-
onds));
   ...
```

Here is an example of how to use these macros to check and compare time:

```
   ...
   /* Verify execution time */
   CppTest_Time limit5ms = CppTest_TimeInit(0, 5000000);
   CPPTEST_ASSERT(CppTest_TimeCompare(diff, limit5ms) < 0)
   ...
```

Available Macros

C++test provides macros for controlling how source code test results are collected and validated; the available macros and routines are defined in the following interfaces:

- **Registration:** Defines test case registration macros. Within the test suite declaration, each test case is "registered" with a macro of the form CPPTEST_TEST*. For example, CPPTEST_TEST, CPPTEST_TEST_EXCEPTION, CPPTEST_TEST_ERROR, CPPTEST_TEST_FAIL. If a test case that is registered using CPPTEST_TEST_FAIL(), CPPTEST_TEST_ERROR() or CPPTEST_TEST_EXCEPTION() does not produce expected result, C++test reports a test case failure.

- **Test Case API:** Defines macros that can be used in the test case code (for instance, to validate results). Note that the available C++test assertion macros are similar to CppUnit macros; however, the C++test macro names are prepended with CPPTEST_ (instead of CPPUNIT_).

For information on these macros, see "C++test API Documentation", page 760.

Testing Functions/Methods with Endless Loops

Executing unit tests for functions that contain endless loops poses a problem when checking assertions. This is because test cases require the tested function to exit in order to perform assertion validation. In some situations, however, users can implement test cases for a function containing an endless loop. This is accomplished by using special macros and functions that allow conditional breaks of endless loop code execution:

Name	Details
`CPPTEST_REGISTER_JMP (expression)`	Sets an internal jump buffer using setjmp or sigsetjmp and evaluates the passed expression. The buffer can be jumped to using the CPPTEST_JMP API call. This macro is typically used inside the test case where it wraps the call to the tested function (see example below).
`CPPTEST_JMP (value)`	Executes a longjmp call (longjmp or siglongjmp is used), which restores the exectuion state to the most recent invocation of the CPPTEST_REGISTER_JMP call. This macro is typically used inside the stub body. It accepts an integer argument that will be returned inside the test case via the cpptestGetJmpReturn function. The integer argrument can be used to verify the correctness of the performed jump (see example below).
`int CDECL_CALL cpptestGetJmpReturn();`	Returns the return value of the most recent CPPTEST_JMP call, i.e. the argument of longjmp/siglongjmp, which is a return value from setmp/sigsetjmp.

These macros are intended to be used in conjunction and are intended to provide users with the ability to break the execution of the tested code (e.g. inside the stub body) and return to the test case to perform assertion checks. They function similarly to the setjmp and longjmp calls declared in the standard setjmp.h header.

This feature is useful for testing typical events-processing embedded functions that contain endless loops. Breaking test code execution is accomplished by a call to CPPTEST_JMP. The recommended technique for injecting a call to CPPTEST_JMP is through the stub body.

The following example illustrates the most common use for these macros and functions:

```
/* Function to be tested */
void signalData(void)
{
    while(1) {
```

```
        // Get the returning frequency from the frequency sensor.
        if (scanFrequencySensor(returningFrequency) == FAILURE) {
            // If sensor has an error, don't give semaphore,
            // report error, call recovery and retry
            log_error("The sensor has encountered an error, retrying...\n");
            recover();
        } else {
            semGive(dataSemId);
            taskDelay(DELAY_TICKS);
        }
    }
}
/* Test case */

/* CPPTEST_TEST_CASE_BEGIN test_signalData */
/* CPPTEST_TEST_CASE_CONTEXT void signalData(void) */
void TestSuite_main_c_883563c3_test_signalData()
{
    /* Pre-condition initialization */
    /* Initializing global variable returningFrequency */
        returningFrequency  = 0;
    /* Tested function call */
     CPPTEST_REGISTER_JMP(signalData())
    /* Post-condition check */
   CPPTEST_POST_CONDITION_INTEGER("cpptestGetJmpReturn()=", (

   cpptestGetJmpReturn() ));
    CPPTEST_ASSERT_INTEGER_EQUAL(0, ( returningFrequency ));
}
/* CPPTEST_TEST_CASE_END test_signalData */
/* Stub */

/** User stub definition for function: int scanFrequencySensor(int) */
EXTERN_C_LINKAGE int scanFrequencySensor (int returningValue) ;
EXTERN_C_LINKAGE int CppTest_Stub_scanFrequencySensor (int returningValue)
{
    static unsigned int counter = 0;
    unsigned int ret = 0;
    counter++;
    //Simulate failure every 5 scans of the sensor
    if (counter == 5) {
        CPPTEST_REPORT("Simulating sensor scan failure")
        ret = FAILURE;
    } else if (counter == 10) {
       CPPTEST_REPORT("Testing accomplished - going back to test case for assertions
check")
        CPPTEST_JMP(counter)
    } else {
        ret = scanFrequencySensor(returningValue);
    }
    return ret;
}
```

When C++test executes the TestSuite_main_c_883563c3_test_signalData test case, it saves the jump buffer prior to calling the tested function. When the tested code executes, it performs the call to CPPTEST_JMP from the CppTest_Stub_scanFrequencySensor stub body, which transfers code execution back to the test case to perform assertions checks.

Available Test Functions

To generate test cases using different numeric values, the C++test runtime library contains set of functions that returns min/max values of integral and floating point types. These same functions can be used in user-defined test cases. For a description of the available functions, see “Test Functions”, page 383.

Using Data From Data Sources

See “Using Data From Data Sources to Parameterize Test Cases”, page 429.

Using Data from Standard IO

See “Using Data From Standard IO”, page 454.

Example: Test Case Modification

For example, assume that C++test generated the following test case:

```
void TestSuite_getData_0::test_getData_0()
{
    /* Pre-condition initialization */
    /* Initializing argument 0 (this) */
        /* Initializing constructor argument 1 (fill) */
        char _cpptest_TestObject_arg_fill = cpptestLimitsGetMaxChar();
    ::Data _cpptest_TestObject (_cpptest_TestObject_arg_fill);
    /* Tested function call */
    const char * _result = _cpptest_TestObject.getData();
    /* Post-condition check */
    CPPTEST_POST_CONDITION_CSTR("const char* _result", _result)
}
```

There is no need for additional precondition initialization other than creating an object to perform a function call on it. However, for the sake of this example, assume that we want to introduce the following modifications:

- Change the constructor used for creating ::Data object to: ::Data _cpptest_TestObject('g');. This will fill the internal char buffer with g.
- Replace the postcondition macro with a section with validation for controlling test results.

The modified test case would look like this:

```
void TestSuite_getData_0::test_getData_0()
{
    /* Pre-condition initialization */
    /* Initializing argument 0 (this) */
    ::Data _cpptest_TestObject('g') ;
    /* Tested function call */
    const char * _result = _cpptest_TestObject.getData();
    /* Post-condition check */
    bool res = true;
    for (int i = 0; i < _cpptest_TestObject.getSize() - 1; i++) {
        if (_result[i] != 'g') {
            res = false;
        }
    }
    // To see what is in the buffer:
    CPPTEST_MESSAGE(_result);
```

```
    // To control test case status:
    CPPTEST_ASSERT_MESSAGE("Test failed : Buffer should be filled with 'g'", res == true);
}
```

Note: The CPPTEST_MESSAGE macro will send the content of "_result" buffer as the test case output, and the CPPTEST_ASSERT_MESSAGE will assert when the test condition fails.

Maintaining the Test Suite

Renaming Test Suites

Test suite files should not be renamed. Doing so could prevent C++test from identifying and running the test suite.

Renaming Test Cases

To rename a test case:

1. In the project tree, locate the test suite file that contains the test case(s) you want to rename.
    - By default, automatically-generated test classes are saved in the `tests/autogenerated` directory within the tested project.
    - To check or change where C++test is saving the test suite files, open the Test Configurations dialog, select the Test Configuration that was used for the test run that generated the tests, then review the value in the **Generation> Test Suite** tab's **Test suite output file and layout** field (see "Customizing Generation Options", page 350 for details).
2. Double-click the project tree node that represents the test suite file. The file will open in an editor.
3. Modify the test suite header as shown in the following examples:
    - Test case registration section:
        - Before:
          ```
          CPPTEST_TEST(test_processData_123);
          ```
        - After:
          ```
          CPPTEST_TEST(test_processData_regression);
          ```
    - Test case declaration section:
        - Before:
          ```
          /* CPPTEST_TEST_CASE_DECLARATION test_processData_123 */
          void test_processData_123();
          ```
        - After:
          ```
          /* CPPTEST_TEST_CASE_DECLARATION test_processData_regression */
          void test_processData_regression();
          ```
4. Modify the test suite definition as shown in the following examples:
    - Before:
      ```
      /* CPPTEST_TEST_CASE_BEGIN test_processData_123 */
      void TestSuite_processData_1::test_processData_123()
      {
      ...
      ```

```
}
/* CPPTEST_TEST_CASE_END test_processData_123 */
```

- After:

```
/* CPPTEST_TEST_CASE_BEGIN test_processData_regression */
void TestSuite_processData_1::test_processData_regression()
{
...
}
/* CPPTEST_TEST_CASE_END test_processData_regression */
```

5. Save the modified file.

Updating Test Suites for a Renamed Source File

If the name of the tested file changes:

1. In the project tree, locate the test suite file that contains the test case(s) you want to rename.
   - By default, automatically-generated test classes are saved in the `tests/autogenerated` directory within the tested project.
   - To check or change where C++test is saving the test suite files, open the Test Configurations dialog, select the Test Configuration that was used for the test run that generated the tests, then review the value in the **Generation> Test Suite** tab's **Test suite output file and layout** field (see "Customizing Generation Options", page 350 for details).
2. Double-click the project tree node that represents the test suite file. The file will open in an editor.
3. Modify the `CPPTEST_CONTEXT` and `CPPTEST_TEST_SUITE_INCLUDED_TO` macro values.
4. Save the modified file.

Using Data From Data Sources to Parameterize Test Cases

This topic explains how to use pre-defined or automatically-generated data source values during testing. Having the ability to run your tests on different sets of data allows you to increase the amount of testing and coverage with very little effort.

Sections include:

- Adding Data Sources
- Automatically Generating New Data Sources
- Using Data Source Values to Parameterize Test Cases
- Exploring Available Data Sources
- Sharing Data Sources
- Ensuring Data Source Portability (Portable Mode)
- Configuring Tests to Use "Unmanaged" Array or CSV file Data Sources - Advanced

Tip: Using data sources in stubs

Any data source that you configure for use in C++test can be used in stubs. For details, see “Using Data Sources in Stubs”, page 471

Adding Data Sources

Data sources can be defined in C++test using a GUI wizard. The wizard allows you to specify a data source from a comma separated values file (.csv), Excel spreadsheet (.xls), database query, or a C++test managed data source table. During the test case execution process, values collected from the given data source are used by the test case.

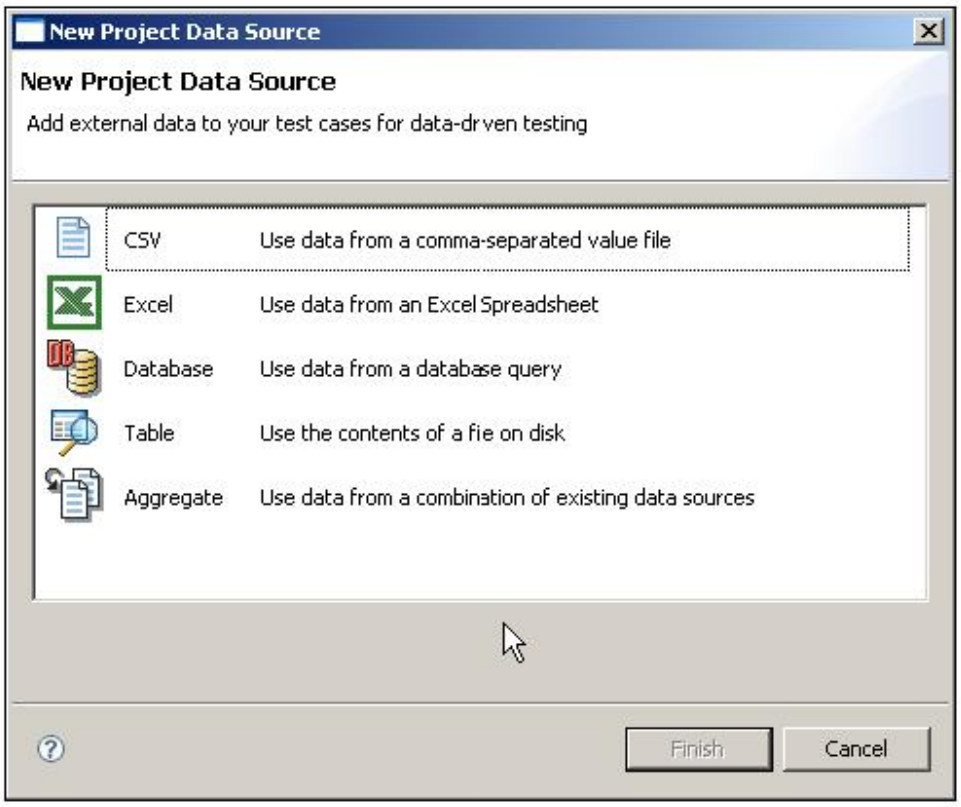
Managed data sources can be defined from the Test Case Explorer at the project level (visible by all project test suites) or at the single test suite level (to be used by the given test suite only). Additionally, managed data sources can be created at the workspace level).

Adding Existing Data Sources at the Project/Test Suite Level

To create a managed data source at the project/test suite level:

1. Select the project or test suite node in the Test Case Explorer view.
2. Right-click the selection, then choose **Add New> Data Source** from the shortcut menu.

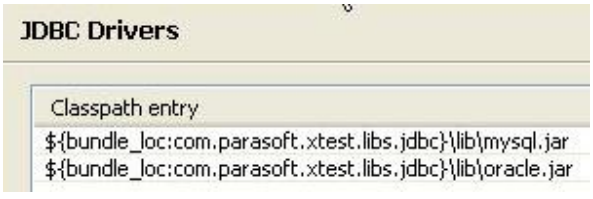
3. Specify the data source type.



4. Configure the data source (e.g., specifying the path to the .csv file, or database parameters and an SQL query).

Tip: Specifying JDBC drivers

JDBC drivers are needed to connect to databases. You can add drivers through the Preference Panels' JDBC Drivers page (**Parasoft> Preferences> JDBC Drivers**).



Adding Existing Data Sources at the Workspace Level

To create a managed data source at the workspace level:

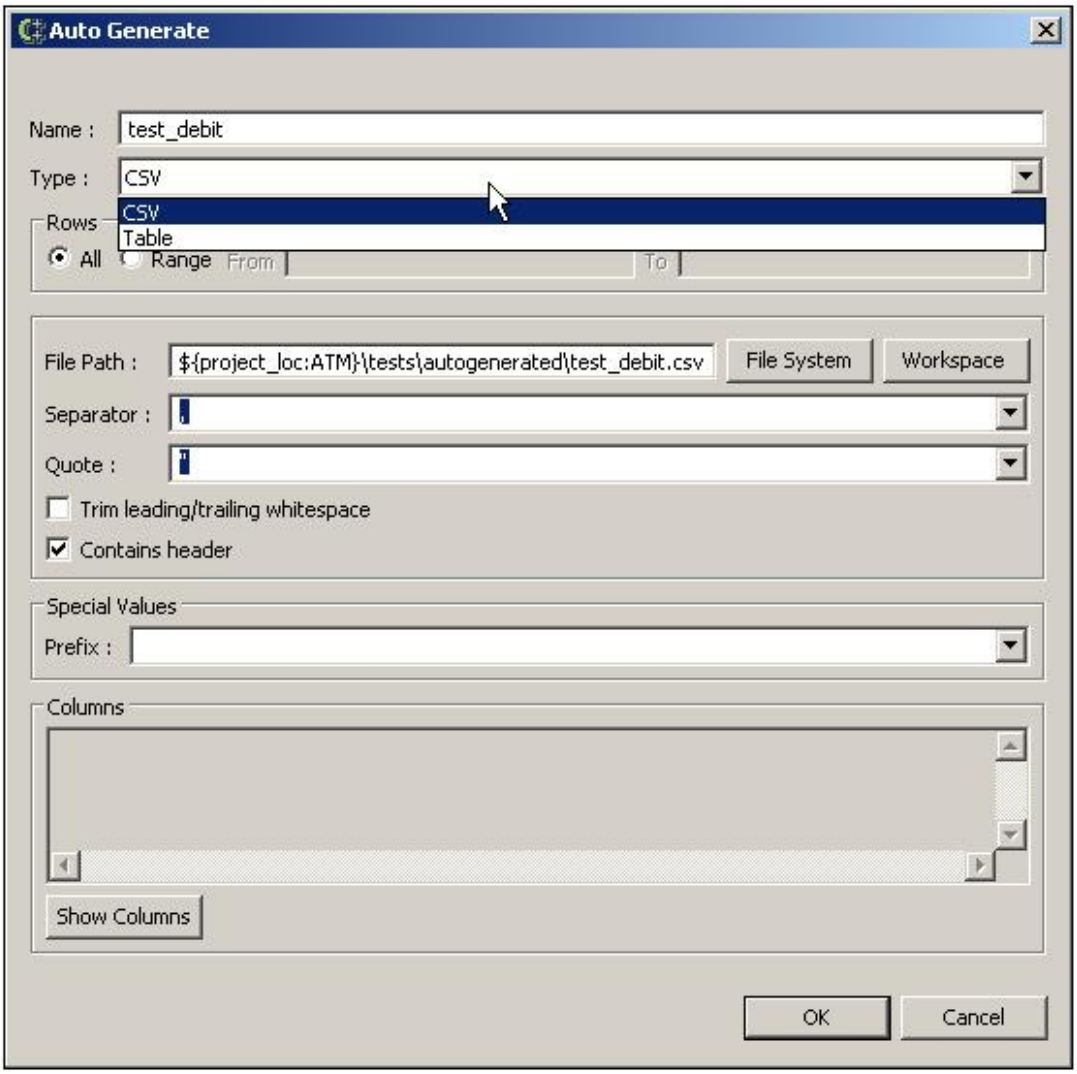
1. Right-click the **Global Data Sources** node in the Test Case Explorer view, then choose **Add New> Data Source**.

2. Define the new managed data source.

Automatically Generating New Data Sources

C++test can generate a comma separated value file (.csv) or a C++test managed table data source with automatically-generated data source values.

1. When configuring a test case in the Test Case Wizard's editor page, click the **Auto Generate** button.
2. Specify the desired settings in the Auto Generate window that opens.



When generating the data source, C++test creates a separate column for each pre- and post-condition that has editable values (boolean, integer, floating-point and string types). The data source is generated at the test suite level.

Once it is generated, the generated data source can be used to parameterize other test cases in the test suite.

Using Data Source Values to Parameterize Test Cases

You can use data source values to parameterize existing (automatically-generated or user-defined) test cases, as well as to define test case values as you write test cases in the Test Case Wizard. Data source values can be taken from any data source that is defined in C++test as described above.

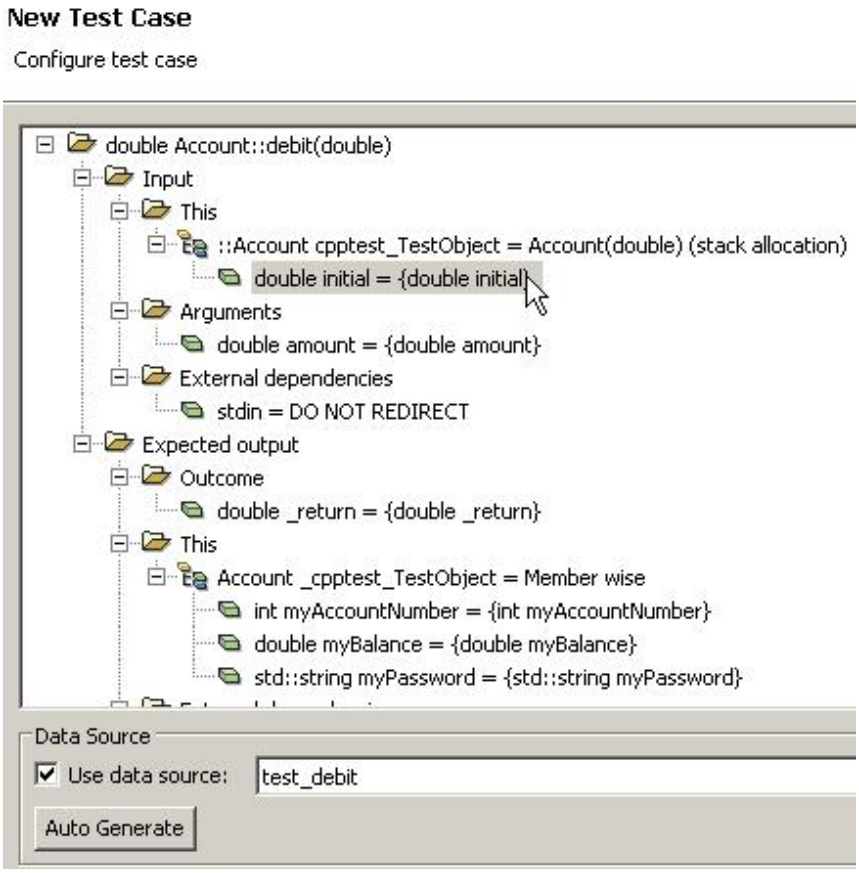
Note

C++test cannot automatically verify the unverified outcomes and failed assertions that result from the execution of a test case that was parameterized with data source values. This is because it typically requires using an expected outcome that is also parameterized with the data source.

Configuring Data Source Usage from the Test Case Wizard

To use the Test Case Wizard to add a new test case that pulls values from a previously-defined managed data source:

1. In the Test Case Explorer, right-click the test suite node, then choose **Add New> Test Case using Wizard** from the shortcut menu.
2. On the first page, specify the source file (compilation unit) and the function for which you want to add a test case, then enter a name for the test case.
3. Click **Next** to open the next wizard page.
4. Check the **Use data source** check box and choose the appropriate data source.



5. Configure the test case by specifying its input and expected output values using GUI controls.

- To use a data source value for a given pre- or -post-condition, double-click the related node then select the appropriate data source column name. Data source values are listed in the combo-box as the column name surrounded with curly braces—for example, "{myColumnName}".

6. Click **Finish** to generate the test case. The new test case will be added to the test suite and the generated source code will be opened in the editor.

Configuring Data Source Usage from the Test Case Code

To parameterize an existing (automatically-generated or user-defined) test case with a previously-defined managed data source:

1. Specify which data source to use with the registration macro CPPTEST_TEST_DS(<TEST_CASE_NAME>, CPPTEST_DS("<MANAGED_DATA_SOURCE_NAME>"));
2. Specify how to use the data source values with the `CPPTEST_DS*` macros (explained in “Macros for Accessing Data Source Values”, page 435).

Example

For example, to parameterize an existing test case with values from an Excel sheet which we will call "MyDataSourceForSum":

1. Open the project.
2. If you have not already done so, add the Excel sheet as a managed data source:
   a. In the Test Case Explorer, right-click the project node and choose **Add New> Data Source** from the shortcut menu.
   b. Choose **Excel**, then click **Finish**.
   c. Enter a meaningful name for the Data Source (e.g. "MyDataSourceForSum").
   d. Enter the location of your .xsl file.
   e. (Optional) Click **Show Columns** and quickly verify the column names.
   f. Click **OK**. The new data source will be shown in the Test Case Explorer.
3. In the Test Case Explorer, double-click the test case that you want to parameterize.
4. Find test case registration line (in the test suite file). For example:
   ```
   ...
   CPPTEST_TEST(sumTest1);
    ...
   ```
5. Change the registration type to use Data Source (your_Data_Source_name):
   ```
   ...
   CPPTEST_TEST_DS(sumTest1, CPPTEST_DS("MyDataSourceForSum"));
   ...
   ```
6. Go to test case definition and add code that tells C++test how to use values from data source columns. Use the macros described in “Macros for Accessing Data Source Values”, page 435. For example:

```
...
/* CPPTEST_TEST_CASE_BEGIN sumTest1 */
void MyTestSuite::sumTest1()
{
    int iVal = CPPTEST_DS_GET_INTEGER("i");
    int jVal = CPPTEST_DS_GET_INTEGER("j");
    int expectedResult = CPPTEST_DS_GET_INTEGER("result");
    CPPTEST_ASSERT_INTEGER_EQUAL(expectedResult, sum(iVal, jVal));
```

```
}
/* CPPTEST_TEST_CASE_END sumTest1 */
...
```

When the test case is executed, it will be parameterized with values from the "MyDataSourceForSum" Data Source of Excel type.

Macros for Accessing Data Source Values

The following macros can be used to access values from a data source. Each macro takes the parameter NAME, which specifies a unique identifier for a data source column.

To do this...	Use this macro...	Notes
Return a null-terminated string value	`const char* CPPTEST_DS_GET_CSTR(const char* NAME)`	N/A
Return an integer value	`long long CPPTEST_DS_GET_INTEGER(const char* NAME)`	N/A
Return an unsigned integer value	`unsigned long long CPPTEST_DS_GET_UINTEGER(const char* NAME)`	N/A
Return a floating point value	`long double CPPTEST_DS_GET_FLOAT(const char* NAME)`	N/A
Return a boolean value	`int CPPTEST_DS_GET_BOOL(const char* NAME)`	N/A
Return the memory buffer	`const char* CPPTEST_DS_GET_MEM_BUFFER (const char* NAME, unsigned int* SIZE_PTR)`	If SIZE_PTR is not null, the size of the buffer will be stored there. For null-terminated strings, the ending null is counted.
Return a value from SOURCE array	`CPPTEST_DS_GET_VALUE(SOURCE)`	A variable of array type should be specified as the SOURCE parameter: The number of the row from which the values are extracted will be automatically increased after each subsequent test case execution.
Return the current iteration (row number)	`unsigned int CPPTEST_DS_GET_ITERATION()`	Can be used to determine the current iteration/row number.

To do this...	Use this macro...	Notes
Return a non-zero value if column 'name' exists in the current iteration of the current data source.	`int CPPTEST_DS_HAS_COLUMN(const char* name)`	Can be used in stubs for test case specific behavior (see “Using Data Sources in Stubs”, page 471). Zero is returned if data source is not used.

Format of Data Source Values

The following table provides information about the supported formats of the data source values for particular types:

Data source type	C++test API function	Supported format
bool	CPPTEST_DS_GET_BOOLEAN()	For true: • true • non-zero integer constant; e.g. 1 For false: • false • 0

Data source type	C++test API function	Supported format
signed integer	CPPTEST_DS_GET_INTEGER()	Decimal representation of the integer (optionally prefixed with - or + sign); e.g.: • 0 • 1 • -200 • +55 Hexadecimal representation of the integer prefixed with 0x or 0X (optionally prefixed with - or + sign); e.g.: • 0x1a • -0XFF • +0x12 Octal representation of the integer prefixed with 0 (optionally prefixed with - or + sign); e.g.: • 0711 • -0123 • +055 Binary representation of the integer prefixed with 0b or 0B, (optionally prefixed with - or + sign); e.g.: • 0b01010101 • -0B1111 • +0b11
unsigned integer	CPPTEST_DS_GET_UINTEGER()	Decimal representation of the integer; e.g.: • 0 • 1 • 200 Hexadecimal representation of the integer prefixed with 0x or 0X; e.g.: • 0x1a • 0XFF Octal representation of the integer prefixed with 0; e.g.: • 0711 • 0123 Binary representation of the integer prefixed with 0b or 0B; e.g.: • 0b01010101 • 0B1111

Data source type	C++test API function	Supported format
floating point	CPPTEST_DS_GET_FLOAT()	The actual format of the floating point value depends on the configuration of the compiler used—the actual implementation of the CPPTEST_STR_TO_FLOAT / CPPTEST_SCANF_FLOAT / CPPTEST_SCANF_FLOAT_FMT macros. In most cases, the sequence of decimal digits possibly containing a radix character (decimal point, locale dependent, usually "."). Optionally, this can be followed by a decimal exponent. A decimal exponent consists of an "E" or "e", followed by an optional plus or minus sign, followed by a non-empty sequence of decimal digits, and indicates multiplication by a power of 10. Examples: • 100 • 1.5 • -3.14159 • 0.44E7 • 99.99e-15
char	CPPTEST_DS_GET_CHAR()	Character to be used; e.g.: • a • 0 • $ Additionally, the following C-like escape sequences should be used to handle special characters: • \n • \t • \v • \b • \r • \f • \a • \\ • \' • \" For non-printable characters, an octal (1-3 digits prefixed with \) or hexadecimal (1-2 digits prefixed with \x) representation of the character can be used; e.g.: • \13 • \017 • \x0A

Data source type	C++test API function	Supported format
string	CPPTEST_DS_GET_CSTR()	Character string to be used; e.g.: • abcdefgh • qaz123 • Hello world! Additionally, the following C-like escape sequences should be used to handle special characters: • \n • \t • \v • \b • \r • \f • \a • \\ • \' • \" Example: • Hello, \"world\"! To use non-printable characters in the string, an octal (1-3 digits prefixed with \) or hexadecimal (1-2 digits prefixed with \x) representation of the character can be used; e.g.: • \13 • \017 • \x0A To define the null as the C-string value in a data source, a managed data source can be configured to recognize some character; e.g. $, as a Special Value Prefix and then the NULL prefixed with the Prefix character can be used in the data source; e.g.: • $NULL Note that this can be used with the UI-managed data sources only (test cases registered with CPPTEST_DS("ds_name") macro).
data buffer	CPPTEST_DS_GET_MEM_BUFFER()	Use the same format as for the string type (see above).

Data Source Types

The following table provides information about formatting data source values using particular data source types:

Data source type	Data source registration macro	Additional information
CSV file (managed data source)	CPPTEST_DS()	Data source values might be surrounded with quotes (quote sign can be defined in the data source configuration); e.g.: • "Hello world!" This is necessary if a separator character needs to be used in the value or if the quote sign itself needs to be used in the value. In the latter case, the quote sign to be used as an element of the data source value needs to be doubled; e.g.: • "Hello, world!" • "Hello, \""world\""!"
Excel spreadsheet (managed data source)	CPPTEST_DS()	No Excel-specific value preparation is needed. Note that the data needs to be formatted to be used for particular types as described above. To make sure that C++test processes values properly, you can use Text format for all cells in the spreadsheet. For example, this way you can be sure that octal representation of the integer value will be handled correctly.
Table (managed data source)	CPPTEST_DS()	No table-specific value preparation is needed. Note that the data needs to be formatted to be used for particular types as described above.
Database (managed data source)	CPPTEST_DS()	No database-specific value preparation is needed. Note that the data needs to be formatted to be used for particular types as described above.
CSV file	CPPTEST_DS_CSV()	Data source values might be surrounded with quotes (quote sign can be defined in the data source configuration); e.g.: • "Hello world!" This is necessary if a separator character needs to be used in the value or if the quote sign itself needs to be used in the value. In the latter case, the quote sign to be used as an element of the data source value needs to be doubled; e.g.: • "Hello, world!" • "Hello, \""world\""!"

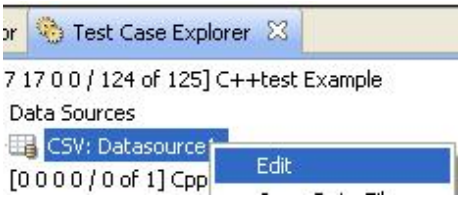
Data source type	Data source registration macro	Additional information
Source code array	CPPTEST_DS_ARRAY()	No array-specific value preparation is needed. Note that the data needs to be formatted to be used for particular types as described above.

Exploring Available Data Sources

Viewing/Modifying a Data Source Configuration

To view or edit an existing data source's configuration:

- Right-click the Test Case Explorer node for that data source, then choose **Edit** from the shortcut menu.



You can then view the data source configuration in the window that opens, and modify it as needed.

Opening a Data Source File

To open a data source file:

- Right-click the Test Case Explorer node for that data source, then choose **Open Data File** from the shortcut menu.

Sharing Data Sources

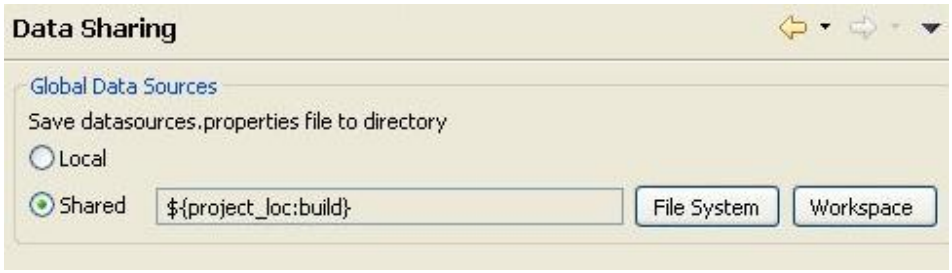
Information about data sources added at the project/test suite level is stored in the .parasoft properties file inside each project. As a result, you can share information about data sources through source control (by committing and checking out .parasoft properties files).

Global data sources—data sources added at the workspace level—can also be shared this way, but you need to define where (e.g., inside which project) to create an additional .datasource data file that contains information about the global data sources. By default, global data source information is stored locally.

To specify a shared location for global data sources:

1. Choose **Parasoft> Preferences**. The Preferences dialog will open.
2. In the left pane, choose **Team> Data Sharing**.

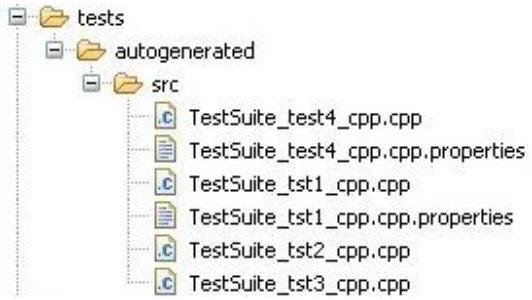
3. Specify where you want the datasources.properties file saved.



Ensuring Data Source Portability (Portable Mode)

The data source portable mode is designed to handle specific development environments where test suite files need to be moved to different locations. In standard data sources mode, if you move a test suite file to a different location, that data sources association will be lost. In portable mode, you can easily move data sources along with the test suite.

To ensure data source portability, all test suite level data source information is stored in a file with the same name as the test suite file name and a .properties extension. As a result, each test suite has its own data source settings file and can be moved along with these data.

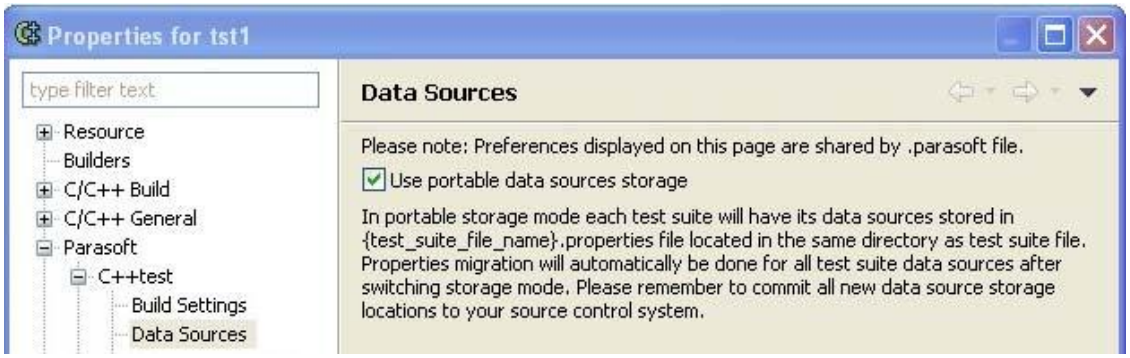


Portable data source mode applies only to data sources created at test suite level—it does not apply to data sources created at the project or workspace level.

Changing Data Source Modes

To change data source settings:

1. Right-click the project node and choose **Properties** from the shortcut menu.
2. Expand the **Parasoft> C++test> Data Sources** category in the left pane.
3. Modify the **Use portable data sources storage** setting, then click **Apply** or **Ok** button.

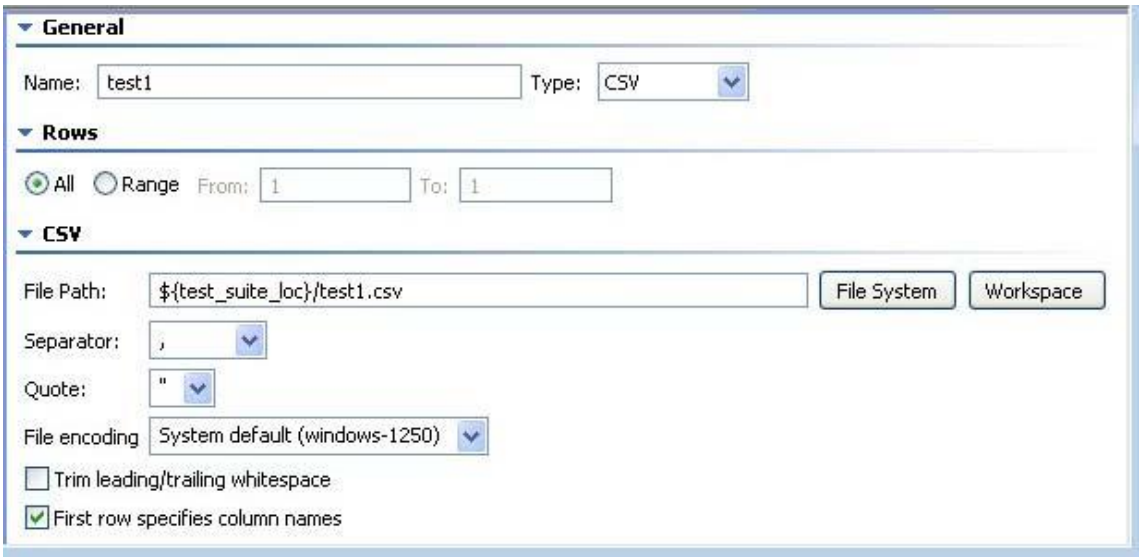


Each time the mode is switched, all existing data sources associated with test suites are automatically converted to the selected mode. Switching to portable mode causes data source settings to be transferred from the .parasoft file to the corresponding <test_suite_name>.properties files. A message will inform you when the conversion is completed.

You can share information about data sources through source control by committing and checking out all or selected *.properties files.

Using Relative Paths to Excel or CSV Data Source Files

To ensure better portability for Excel and CSV data source types, you can use the C++test ${test_suite_loc} variable in the Excel/CSV file path. This variable resolves to the absolute path to the test suite file location.



Configuring Tests to Use "Unmanaged" Array or CSV file Data Sources - Advanced

In addition to working with managed data sources (data sources defined in C++test as described above), you can also use values from arrays or CSV files that are not added as managed data sources.

Using Array Data Source Values

One implementation of data sources in C++test is based on typed arrays, which are then used by a single test to step through rows of data.

To create a test case that can access an array data source:

- Register the test case with the `CPPTEST_TEST_DS(testCaseName, dataSourceConfig)` macro.

Replace `testCaseName` with the name of the test case function that you want to access the data source value, and replace `dataSourceConfig` with a macro for data source configuration.

Two helper macros can be used for `dataSourceConfig`: `CPPTEST_DS_ARRAY(ARRAY_NAME, ROW, COLUMN)` and `CPPTEST_DS_REPEAT(NUMBER)`.

With `CPPTEST_DS_ARRAY(ARRAY_NAME, ROW, COLUMN)`, note that:

- This macro uses ARRAY_NAME as a data source. ARRAY_NAME should be of type `const char* data[ROW][COLUMN]`.
- The first row in the table must contain columns names.
- The test case will be executed (ROW - 1) times.
- The data source access macros can be used to extract values as described in the following sections.

With `CPPTEST_DS_REPEAT(NUMBER)`, note that:

- The test case will be executed NUMBER times
- The test case can be used to access values from custom data arrays as explained in the CPPTEST_DS_GET_VALUE and CPPTEST_DS_GET_ITERATION descriptions in “Macros for Accessing Data Source Values”, page 435.

Tips

- Data source arrays can be declared outside of a test suite class (this is the only option for C test suites) or as a test suite class member (for C++ test suites). Note that C++ test suites can be used for the C language (assuming that you have a C++ compiler to build them).
- When data source arrays are declared as a class member, scope of access is not important because the arrays are only accessed by test case functions and setup/teardown, which are member functions.
- Declaring typed arrays and initializing them in setUp method is a very powerful way to initialize data source array elements to arbitrary data. This data can be dynamically allocated objects, factory test objects, etc. For example:

```
#include "cpptest.h"
int plus_one(int);

class TestSuite_3 : public CppTest_TestSuite {
public:
    CPPTEST_TEST_SUITE(TestSuite_3);
    CPPTEST_TEST_DS(test_ds_repeat, CPPTEST_DS_REPEAT(2));
    CPPTEST_TEST_SUITE_END();

    void setUp();
    void tearDown();

    void test_ds_repeat();

private:
    int _dsRepeat_arg[2];
    int _dsRepeat_return[2];
};

CPPTEST_TEST_SUITE_REGISTRATION(TestSuite_3);

void TestSuite_3::setUp()
{
    _dsRepeat_arg[0] = 1;
    _dsRepeat_arg[1] = 2;

    _dsRepeat_return[0] = 2;
    _dsRepeat_return[1] = 3;
}

void TestSuite_3::tearDown()
{
}

void TestSuite_3::test_ds_repeat()
{
    int index = CPPTEST_DS_GET_ITERATION() - 1;
    int _value = _dsRepeat_arg[index];
    int _expected = _dsRepeat_return[index];
```

```
    int _return = plus_one(_value);
    CPPTEST_ASSERT_INTEGER_EQUAL(_expected, _return);
}
```

Using Unmanaged CSV Data Sources

In addition to supporting CSV data sources as described earlier in this section, C++test can also use "unmanaged" CSV data sources when the CSV data satisfies the following criteria:

- CSV data is stored in an external file.
- CSV data contains records (rows) of values separated with a (customizable) separator character.
- CSV data can contain a header.
- CSV data values can contain space characters that can be optionally ignored.

See "CSV File Support Details", page 447 for more information on the file types supported.

To create a test case that can access a CSV data source:

- Register the test case with the `CPPTEST_TEST_DS(testCaseName, dataSourceConfig)` macro.

Replace `testCaseName` with the name of the test case function that you want to access the data source value, and replace `dataSourceConfig` with CPPTEST_DS_CSV(FILE_NAME, SEPARATOR, HAS_HEADER, TRIM).

CPPTEST_DS_CSV is defined in the C++test Data Source API. Its parameters are:

- **FILE_NAME -** Name of the data source file with data source. It may be specified with a full or relative file path.
  Note: If relative path is used, it's relative to the test executable's working directory, which can be customized in the Test Configurations's **Test executable run directory**. field (in the **Execution> Runtime** tab).
- **SEPARATOR -** Field separator. It should be specified as a character constant (ex: ',').
- **HAS_HEADER -** If the first record (row) is a header, this should be set to 1. Otherwise, it should be set to 0.
  Note: If there is no header in the CSV file, then each column name is its ordinal number (starting with 0).
- **TRIM -** If spaces before/after a value should be omitted, this should be set to 1. Otherwise, it should be set to 0.

If there is no header in the CSV file, then each field name is its ordinal number (starting with 0). Values should be quoted (string type). For example: CPPTEST_DS_GET_FLOAT("0");

FILE_NAME may be a full or relative file path. The latter is relative to the current directory.

CPPTEST_DS_CSV Examples

CPPTEST_DS_CSV("/home/project/testdata.csv", ',', 1, 0)

- "/home/project/testdata.csv" will be used as a data file.
- Values are separated with the ',' character.
- The first record will be treated as a header.
- Spaces before/after value will not be trimmed.

CPPTEST_DS_CSV("testdata.csv", ';', 0, 1)

- "testdata.csv" (located in the current working directory) will be used as a data file.
- Values are separated with the ';' character.
- There is no explicit header - the first record (row) will be used as data.
- Spaces before/after value will be trimmed.

CSV File Support Details

- Each record is located on a separate line, delimited by a line break (CRLF). For example:
  aaa,bbb,ccc CRLF
  zzz,yyy,xxx CRLF
- The last record in the file may or may not have an ending line break. For example:
  aaa,bbb,ccc CRLF
  zzz,yyy,xxx
- There may be an optional header line appearing as the first line of the file with the same format as normal record lines. This header will contain names corresponding to the fields in the file and should contain the same number of fields as the records in the rest of the file (the presence or absence of the header line should be indicated via the optional "header" parameter of this MIME type). For example:
  field_name,field_name,field_name CRLF
  aaa,bbb,ccc CRLF
  zzz,yyy,xxx CRLF
- Within the header and each record, there may be one or more fields, separated by commas. Each line should contain the same number of fields throughout the file. Spaces are considered part of a field and should not be ignored. The last field in the record must not be followed by a comma. For example:
  aaa,bbb,ccc
- Each field may or may not be enclosed in double quotes (however, some programs [such as Microsoft Excel] do not use double quotes at all). If fields are not enclosed with double quotes, then double quotes may not appear inside the fields. For example:
  "aaa","bbb","ccc" CRLF
  zzz,yyy,xxx
- Fields containing line breaks (CRLF), double quotes, and commas should be enclosed in double-quotes. For example:
  "aaa","b CRLF
  bb","ccc" CRLF
  zzz,yyy,xxx
- If double-quotes are used to enclose fields, then a double-quote appearing inside a field must be escaped by preceding it with another double quote. For example:
  "aaa","b""bb","ccc"
- LF (instead of CRLF) can be used to separate records
- Spaces before and after value can be omitted
- An equal number of values in records is not required
- Non-comma characters can be used as field separators

Examples

Using CPPTEST_DS_ARRAY

In this example, an array data source is used. First, we define the array of data:

```
const char* MyData[][3] = {
    { "arg1",  "arg2", "return" },
    {    "1",   "one",    "111" },
    {    "0",   "two",    "222" },
    {    "2", "three",    "333" },
};
```

It has three columns and four rows. The first row contains the names of columns (`arg1`, `arg2`, and `return`). Three other rows keep data for test cases (so there will be three test cases executed). After we have the data prepared, we have to register the test case as follows:

```
    CPPTEST_TEST_DS(testFoo2, CPPTEST_DS_ARRAY(MyData, 4, 3));
```

The test case `testFoo2` will be executed three times, and will be fed with data from the `MyData` array.

Finally, we need to write the test case. The tested function takes two parameters and returns an integer. We'll take both parameters from the data source as follows:

```
    int arg1 = CPPTEST_DS_GET_INTEGER("arg1");
    const char * arg2 = CPPTEST_DS_GET_CSTR("arg2");
```

If a column with the specified name is not available or if its value type does not match, test case execution will be aborted.

For postcondition checking, we'll get the expected value from data source as follows:

```
    int expectedReturn = CPPTEST_DS_GET_INTEGER("return");

    /* check return value with value from data source */
    CPPTEST_ASSERT_INTEGER_EQUAL(expectedReturn, actualReturn);
```

Here is the completed test:

```
class TestWithDataSource : public CppTest_TestSuite
{

public:
    CPPTEST_TEST_SUITE(TestWithDataSource);

    /* register standard test case */
    CPPTEST_TEST(testFoo1);

    /* register test case with "MyData" data source access */
    CPPTEST_TEST_DS(testFoo2, CPPTEST_DS_ARRAY(MyData, 4, 3));

    CPPTEST_TEST_SUITE_END();
...
...
...

/* test case using values from data source */
```

```
void TestWithDataSource::testFoo2()
{

    /* extract arguments from data source */
    int arg1 = CPPTEST_DS_GET_INTEGER("arg1");
    const char * arg2 = CPPTEST_DS_GET_CSTR("arg2");

    /* call tested function */
    int actualReturn = foo(arg1, arg2);

    /* extract expected return from data source */
    int expectedReturn = CPPTEST_DS_GET_INTEGER("return");

    /* check return value with value from data source */
    CPPTEST_ASSERT_INTEGER_EQUAL(expectedReturn, actualReturn);
}
```

Using CPPTEST_DS_REPEAT

In this example, we will use the `repeat` data source:

```
/* data source for arg1 */
int _arg1[] = {
   1,
   0,
   2,
};


/* data source for arg2 */
const char * _arg2[] = {
   "one",
   "two",
   "three",
};

/* data source for return */
int _return[] = {
   111,
   222,
   333,
};
```

This is a special data source: it doesn't give access to any data, it just repeats test case execution the requested number of times. The user specifies what data should be used during each run. In this example, we'll use three separate arrays: `_arg1`, `_arg2`, and `_return`.

We register the test case as follows:

```
    CPPTEST_TEST_DS(testFoo2, CPPTEST_DS_REPEAT(3));
```

The `CPPTEST_DS_REPEAT` parameter indicates how may times the test case `testFoo2` should be executed.

In the test case, we'll get data directly from arrays. To get the value from the proper row, we can use the value of `CPPTEST_DS_GET_ITERATION()` as an array index, or have the `CPPTEST_DS_GET_VALUE(SOURCE)` macro perform indexing for us as follows:

```
    int arg1 = CPPTEST_DS_GET_VALUE(_arg1);
```

```
    const char * arg2 = CPPTEST_DS_GET_VALUE(_arg2);
```

There is no range or type checking in the CPPTEST_DS_GET_VALUE macro. It is just an index (given an array with a proper index).

Here is the completed test:

```
class TestWithDataSource : public CppTest_TestSuite
{

public:
    CPPTEST_TEST_SUITE(TestWithDataSource);

    /* register standard test case */
    CPPTEST_TEST(testFoo1);

    /* register test case that will be executed number of times */
    CPPTEST_TEST_DS(testFoo2, CPPTEST_DS_REPEAT(3));

    CPPTEST_TEST_SUITE_END();
...
...
...

/* test case using values from custom arrays */
void TestWithDataSource::testFoo2()
{

    /* extract arguments from data source */
    int arg1 = CPPTEST_DS_GET_VALUE(_arg1);
    const char * arg2 = CPPTEST_DS_GET_VALUE(_arg2);

    /* call tested function */
    int actualReturn = foo(arg1, arg2);

    /* extract expected return from data source */
    int expectedReturn = CPPTEST_DS_GET_VALUE(_return);

    /* check return value with value from data source */
    CPPTEST_ASSERT_INTEGER_EQUAL(expectedReturn, actualReturn);
}
```

Using CPPTEST_DS_CSV

Here is a data source declaration for data that is stored in the file `/home/test/t2.data`, where header and field are separated with a comma (','). No trimming is used.

```
class TestSuite : public CppTest_TestSuite
{
        public:
        CPPTEST_TEST_SUITE(TestSuite);
        CPPTEST_TEST_DS(test_fooc_1, CPPTEST_DS_CSV("/home/test/t2.data", ',', 1, 0));
        CPPTEST_TEST_SUITE_END();

        void setUp();
        void tearDown();

        void test_fooc_1();
};
```

Here is a definition of a test case that uses the standard functions from the Data Source API:

```
void TestSuite::test_fooc_1()
{
        int _boo  = CPPTEST_DS_GET_INTEGER("first");
        float _goo  = CPPTEST_DS_GET_FLOAT("second");

        int _return  = fooc(_boo, _goo, _g);

        CPPTEST_ASSERT_INTEGER_EQUAL(CPPTEST_DS_GET_INTEGER("result"), ( _return ))
}
```

Here is the CSV data source file (`/home/test/t2.data`), which has a header and three test cases.

```
first,second,result
1,2,3
2,2,4
2,2,5
```

Creating Data-Source Based Regression Tests

To create two data-source based regression tests—one using string array data, and the other using typed array data—for the sample ATM project, which is shipped with C++test (in the examples directory):

1. Right-click the project node, then choose **New> Other**, select **C++test> Test suite**, then click **Next**. This opens up a New Test Suite Wizard.
2. Specify `TestSuiteAccount` as the test suite name (this will be the class name for C++ test suites) and `/ATM/tests/user` as the test suite location (click **Create Directory** if the directory does not exist).
3. For **Test suite language**, select **C++**.
4. Leave all the other options unchecked.
5. Click **Next** and add two tests for the Account class by clicking the **Add** button twice. Name one test `setPasswordTest`, the other `depositTest`. Completing the Wizard sequence will create a skeleton test suite with these tests appropriately registered.
6. Open the test suite source file.
7. Create a data source array of strings (shown below) for testing the password setting, and put it in the source file of the test suite, above the test suite class declaration. Generally, data source arrays can be declared at the file scope or as static members of a test suite class, following standard language idioms:

```
const char* passwordData [] [2] = {
          { "arg1", "result" },
          { "", "" },
          { "a1", "a1" },
          { "really_long_password", "really_long_password" },
          { "foo", "goo" }
};
```

8. Write a test case for setPassword method. The test should set a password and check that the password returned conforms to what is specified in the data source:

```
void TestSuiteAccount::setPasswordTest()
{
          const char* password = CPPTEST_DS_GET_CSTR("arg1");
          const char* result = CPPTEST_DS_GET_CSTR("result");
          Account _cpptest_object;
          _cpptest_object.setPassword(password);
          CPPTEST_ASSERT_CSTR_EQUAL(result, _cpptest_object.getPassword())
```

```
}
```

9. Add `#include "Account.hxx"` to the top of the file so we get a hold of the Account class declaration.
10. Modify the test case registration in the test suite declaration `CPPTEST_TEST(setPasswordTest)` to look like this:

```
CPPTEST_TEST_DS(setPasswordTest, CPPTEST_DS_ARRAY(passwordData, 5, 2));
```

11. Create a custom Test Configuration that will execute the user-defined tests as follows:
    a. Choose **Parasoft> Test Configurations**.
    b. Right-click **Builtin> Run Unit Tests**, then choose **Duplicate**.
    c. Rename the new User-defined Test Configuration to "Run DS Tests."
    d. Open the **Execution> General** tab.
    e. Change the test suite location pattern to include only tests in `/tests/user/*` .
    f. Click **Apply**, then **OK** to save your changes.
12. Select the project node, then run the new "Run DS Tests" Test Configuration.
13. Open the Quality Tasks view after tests complete. In the Fix Unit Tests category, you should see one unit test assertion pointing to row 4 of the data source, which contained mismatched data on purpose (simulated regression failure), as well as another assertion from the second test case.
14. Create a second test case using data in typed arrays. Create one data source array of type double with deposit values, and another one with expected account balance values. This time, let's make the arrays members of the test suite. The end of the test suite declaration should look like the following:

```
                void depositTest();

        private:
                static double deposit [];
                static double balance [];
};

                double TestSuiteAccount::deposit [] = {
                        0.0,
                        1.0,
                        -2.0,
                        1e06
                };
                double TestSuiteAccount::balance [] = {
                        1.0,
                        2.0,
                        -1.0,
                        1000001.0
        };
```

15. Write a test for the deposit function and check the final balance for regression purposes:

```
void TestSuiteAccount::depositTest()
{
        double dep_value = CPPTEST_DS_GET_VALUE(deposit);
        double balance_value = CPPTEST_DS_GET_VALUE(balance);
        Account _cpptest_object(1.0);  // create account with initial deposit
        _cpptest_object.deposit(dep_value); // additional deposit
        CPPTEST_ASSERT_DOUBLES_EQUAL(balance_value, _cpptest_object.getBalance(), 1e-6)
}
```

16. Change the registration of the test CPPTEST_TEST(depositTest) to run 4 times:

```
CPPTEST_TEST_DS(depositTest, CPPTEST_DS_REPEAT(4));
```

17. Rerun the "Run DS Tests" Test Configuration on the project.
18. Review the results reported under the Fix Unit Tests category of the Quality Tasks view. No additional tasks should be reported since all values in the new test match.

Using Data From Standard IO

This topic explains how you can have tests examine data which is sent to standard output or standard error streams. This is accomplished by interacting with the C++test Stream API, which is designed for using standard input/output streams. This API allows manipulation and content verification of streams.

Sections include:

Using the C++test Stream API

To have tests examine data which is sent to standard output or standard error streams:

1. To redirect the stream before test case execution, use one of the following functions:

   ```
   CppTest_StreamRedirect* CppTest_RedirectStdOutput()

   CppTest_StreamRedirect* CppTest_RedirectStdError()
   ```

   Both `CppTest_StreamRedirect* CppTest_RedirectStdOutput()` and `CppTest_StreamRedirect* CppTest_RedirectStdError()` return a pointer to a special internal C++test structure or NULL (in case of error). This pointer can be used later in other stream redirection functions, as illustrated in "Example", page 455.

2. To compare the content of streams with a null-terminated string or a data buffer, use the following functions:

   ```
   CppTest_StreamCompare(CppTest_StreamRedirect* redirect, const char* value);

   CppTest_StreamNCompare(CppTest_StreamRedirect* redirect, const char* value,
   unsigned int size);
   ```

   Both functions return 0 if the value matches the stream value, an integer less than zero if the value is less than the stream value, or an integer greater than zero if the value is greater than the stream value.

   `CppTest_StreamCompare` behaves like strcmp.

   To get the whole data buffer from the stream using a function, use `char* CppTest_StreamReadData(CppTest_StreamRedirect* redirect, unsigned int* len);` This function returns a pointer to a buffer with data from the stream. The size of the buffer will be passed back in the `len` parameter. The buffer is allocated with the `malloc` function. You are responsible for freeing this buffer.

3. To set values for the input stream, use the following initialization functions:

   ```
   CppTest_StreamRedirect* CppTest_RedirectStdInput(const char* value);

   CppTest_StreamRedirect* CppTest_RedirectNStdInput(const char* value, unsigned
   int size)
   ```

These functions set the input stream to the value of the supplied string (using N characters in the second case).

4. At the end of the test case, reset the C++test internal stream to the default state with `void CppTest_StreamReset(CppTest_StreamRedirect* redirect);`

Example

```
/* CPPTEST_TEST_CASE_BEGIN foo_test */
void TestSuite::foo_test()
{
  /* Pre-condition initialization */
  CppTest_StreamRedirect* output_stream = CppTest_RedirectStdOutput();

  /* Tested function call */
  ::foo();

  /* Post-condition check */
  /* This assert validates that "Output string" is put by foo() on standard
     output. */
  CPPTEST_ASSERT(0 == CppTest_StreamCompare(output_stream, "Output
  string"));
  CppTest_StreamReset(output_stream);
}
/* CPPTEST_TEST_CASE_END foo_test */

/* CPPTEST_TEST_CASE_BEGIN bar_test */
void TestSuite_a_0::bar_test()
{
  /* Pre-condition initialization */
  CppTest_StreamRedirect* input_stream = CppTest_RedirectStdInput("Hello
  World!\n");

  /* Tested function call */
  /* bar() reads standard input and returns the string */
  std::string ret = ::bar();

  /* Post-condition check */
  CPPTEST_ASSERT(ret == "Hello World!");
  CppTest_StreamReset(input_stream);
}
/* CPPTEST_TEST_CASE_END bar_test */
```

Using Factory Functions

This topic explains how you can define and use factory functions that return specific values of a given type. This allows you to establish repositories of valid objects that can be used in automatically-generated tests and in test cases created with the Test Case Wizard.

Sections include:

About Factory Functions

A factory function is a function (global, namespace level or a static class method) that returns a given type and has a CppTest_Factory_ name prefix. There can be more than one factory function defined for a given type.

When C++test sees a declaration of such a factory function in the compilation unit for which it generates test cases, it considers this factory function to be one of the possible initialization options for a given type. If the factory function has a non-empty parameters list, C++test will initialize factory function arguments just as it would for arguments of a constructor used to create an object of a given type.

Defining Factory Functions

To create a factory function for the type Type, write the function with the following signature:

```
Type CppTest_Factory_<factory_function_name>(<factory_arguments>);
```

For example:

```
int CppTest_Factory_generateBooleanInt(bool b)
{
    return b ? 1 : 0;
}

MyClass* CppTest_Factory_createMyClass(int size, int value)
{
    MyClass* m = new MyClass;
    m->initialize(size);
    m->setValue(value);
    return m;
}
```

Since C++test needs to see the declarations of the factory functions in the context of the compilation unit for which it generates test cases, we recommend the following approach to writing factory functions:

1. Create a header file with the factory function declarations to be used in test cases. For instance, such a header can be created in the 'factory' folder under the tested project root.
2. In all tested source files for which factory functions should be used, add an include directive for the created factory functions header. We recommend having it guarded with #ifdef PARASOFT_CPPTEST.

- You can quickly add such an include directive to the tested source file by going to the source code editor, typing `ffi`, then pressing **CTRL+SPACE**.

3. Create a source file with the factory functions implementation. Such a source file should be created in one of the folders specified in the Test Configuration's **Execution> Symbols> Use extra symbols from files found in** field, which is set to `${cpptest:cfg_dir}/safestubs;${project_loc}/stubs;${project_loc}/factory` by default.
    - You can quickly add a factory function section to the factory functions source file by going to the source code editor, typing `ffs` in the source code, then pressing CTRL+SPACE.
    - You can quickly add an additional factory function template to the code by going to the source code editor, typing `ff` in the source code, then pressing CTRL+SPACE.

Notes

- When using factory functions in standalone test suites—both for test cases that were written manually and those added using the Test Case Wizard—make sure you add include directives for factory function headers that contain declarations of the factory functions used.
- Source files created to contain factory function definitions need to be excluded from the regular build. See "Excluding Test Cases from the Build", page 475 for instructions.

Using Factory Functions

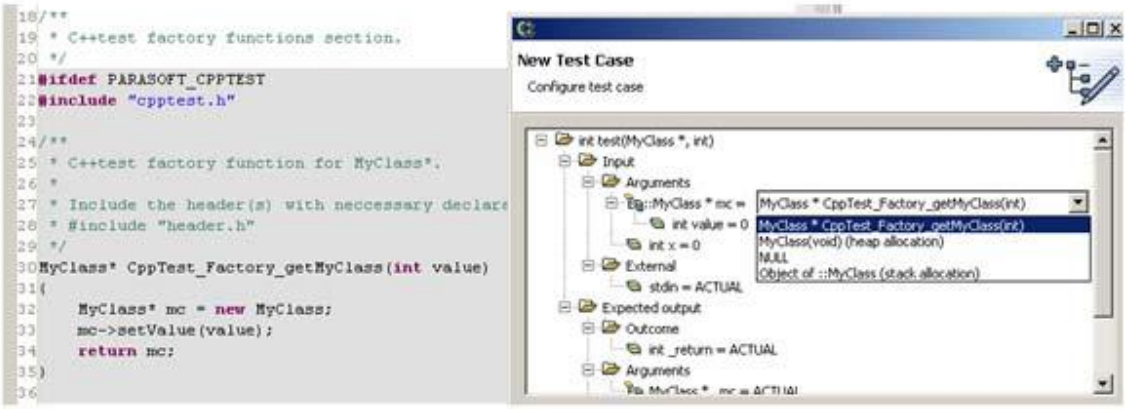
With Automatically-Generated Test Cases

To configure C++test to use factory functions in automatically-generated test cases:

1. In the Test Configuration manager's **Generation> Test Case** tab, enable the **Use factory functions** option.
2. If you want to use factory functions exclusively, also enable **Do not use other initializers for types with factory functions**.

With the Test Case Wizard

Factory functions found in the tested compilation unit can also be used when creating test case with the Test Case Wizard. In this case, factory functions will appear as additional initialization methods for a given type.



Using Data Sources with Factory Functions

Any data source that you configure for use in C++test (as described in “Using Data From Data Sources to Parameterize Test Cases”, page 429) can be used in factory functions. You can use the C++test Data Source API to access values from a data source.

Here is an example:

```
MyClass CppTest_Factory_getMyClassObjectFromDS (void)
{
  MyClass obj;
  if (CPPTEST_DS_HAS_COLUMN("MyClass.int")) {
    obj.initialize(CPPTEST_DS_GET_INTEGER("MyClass.int"));
  } else {
    // Data Source not available in this test case
    obj.initialize(0);
  }
  return obj;
}
```

Notes

- Ensure that the given data source column exists in the context of the currently-executed test case.
- If you want to use the C++test API in a factory function, ensure that the "cpptest.h" header is included in the factory function definition file. We recommend including this header before any other header files.

Deleting and Disabling Tests

This topic explains how you can focus your test suite on the test cases that are most important for your current goals and project phase. This is accomplished by removing test cases and/or disabling outcome checks that are not currently of concern to you.

Sections include:

- Disabling Test Cases
- Disabling the Checking of Specific Test Case Outcomes
- Deleting Test Cases
- Deleting Test Suites

Disabling Test Cases

If you do not want to use a test case at the current time, but think you might want to use it sometime in the future, you can disable it. Disabled test cases are not executed and their status is reported as "Test case execution disabled".

When a test case is disabled, its macros are changed as follows:

Macro for enabled test case	Macro for disabled test case
CPPTEST_TEST(testcasename)	CPPTEST_TEST_DISABLED(testcasename)
CPPTEST_TEST_FAIL(testcasename)	CPPTEST_TEST_FAIL_DISABLED(testcasename)
CPPTEST_TEST_EXCEPTION(testcase-name,ExcType)	CPPTEST_TEST_EXCEPTION_DISABLED(testcasename,ExcType)
CPPTEST_TEST_ERROR(testcase-name,ErrorCode)	CPPTEST_TEST_ERROR_DISABLED(testcase-name,ErrorCode)

Test cases can be disabled (as well as later enabled) from the Test Case Explorer or from the project tree.

From the Test Case Explorer

To disable a test case from the Test Case Explorer.

1. In the Test Case Explorer, select the resource(s) you want to disable. You can select folder(s), test suite(s), or test case(s).
2. Right-click the selection, then choose **Disable** from the shortcut menu.

Disabled tests will be skipped during execution.

The Test Case Explorer’s filters can be used while enabling/disabling tests. If a test is “filtered out,” its status will not be modified.

If you later want to re-enable disabled test cases, right-click the related Test Case Explorer nodes, then choose **Enable** from the shortcut menu.

From the Code Editor

To disable a test case from the code editor:

1. Open the source code for the test you want to disable. You can do this by double-clicking the related project tree node.
    - By default, automatically-generated test classes are saved in the `tests/autogenerated` directory within the tested project.
    - To check or change where C++test is saving the test suite files, open the Test Configurations dialog, select the Test Configuration that was used for the test run that generated the tests, then review the value in the **Generation> Test Suite** tab's **Test suite output file and layout** field (see "Test suite tab", page 701 for details).

> **CDT 4.x Note**
>
> Test functions are not available in the project tree for Managed C/C++ projects created with CDT 4.x. To disable a test case, select the test case name in the code editor.

2. Right-click the related source code, then choose **Parasoft> C++test> Test Suite> Disable Test Case(s)**.

You can later re-enable the test cases in the same manner (just choose **Enable** instead of **Disable**).

Disabling the Checking of Specific Test Case Outcomes

If you do not want to check a test case outcome right now, but think you might want to check it sometime in the future, you can add comments prevent C++test from checking it. If you later want to check that test case outcome, you can simply remove the comments:

To prevent the checking of a particular test case outcome:

1. In the Quality Tasks view, right-click the Unverified Outcome for that test case.
2. Choose **Ignore Outcome** from the shortcut menu. C++test will then comment out the source code for checking that outcome.

Alternatively, you could manually comment out the source code for checking that outcome.

If you later want to re-enable checking of a disabled outcome, remove the comments.

Deleting Test Cases

Test cases can be deleted from the Test Case Explorer, project tree or from the Quality Tasks view.

To permanently delete one or more test cases from the Test Case Explorer:

- Right-click the test case(s) you want to delete, then choose **Delete**.

To permanently delete one or more test cases from the project tree:

1. In the project tree, select the test case(s) you want to delete.

- By default, automatically-generated test classes are saved in the `tests/autogenerated` directory within the tested project.
- To check or change where C++test is saving the test suite files, open the Test Configurations dialog, select the Test Configuration that was used for the test run that generated the tests, then review the value in the **Generation> Test Suite** tab's **Test suite output file and layout** field (see "Test suite tab", page 701 for details).

CDT 4.x Note

Test functions are not available in the project tree for Managed C/C++ projects created with CDT 4.x. To delete a test case, select the test case name in the code editor.

2. Right-click the selection, then choose **C++test> Remove Test Case(s)** from the shortcut menu.

To permanently delete one or more test cases from the Quality Tasks view:

1. Select the node(s) related to the test cases you want to delete. You can select:
   - Test case nodes
   - Violation nodes
   - Children of violation nodes (including stack trace items
2. Right-click the selection, then choose **Remove Test Case(s)** from the shortcut menu.

Deleting Test Suites

Test suites can be deleted from the Test Case Explorer, project tree or from the Quality Tasks view.

To permanently delete one or more test suites from the Test Case Explorer:

- Right-click the test suite(s) you want to delete, then choose **Delete**.

To permanently delete an entire test suite from the project tree:

1. In the project tree, locate the test suite file that you want to delete.
   - By default, automatically-generated test classes are saved in the `tests/autogenerated` directory within the tested project.
   - To check where C++test is saving the test suite files, open the Test Configurations dialog, select the Test Configuration that was used for the test run that generated the tests, then review the value in the **Generation> Test Suite** tab's **Test suite output file and layout** field (see "Test suite tab", page 701 for details).
2. Right-click the related test suite node, then choose **Delete**.

Adding and Modifying Stubs

This topic explains how to add user-defined stubs to replace calls to resources that you cannot (or do not want to) access during testing, as well as how to modify automatically-generated stubs.

Sections include:

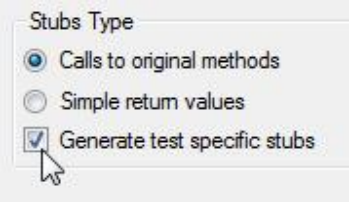
Note

- When you are working with stubs, ensure that your Test Configuration's **Instrumentation mode** setting (in the **Execution> General** tab) is set to **Full**, **Full runtime w/o coverage**, or a custom option that includes stub instrumentation.
- C++test prioritizes stubs in the following order: user-defined stub, automatically-generated safe stub, original function, auto stub. Thus, automatically-generated stubs will be used only if no other definition (user stub or original) is available.
- C++test does not stub destructors if the original definition is available anywhere in the code or library. The C++test stub for the destructor will be used only when the original destructor is not available.

About Stubs

See "Stubs", page 56.

Viewing Stubs in the Stubs View



The Stubs view provides details on the stub configuration based on the most recent run of a unit testing Test Configuration. It allows you to modify the configuration by adding user-defined stubs or automatically generating stubs for missing symbols.

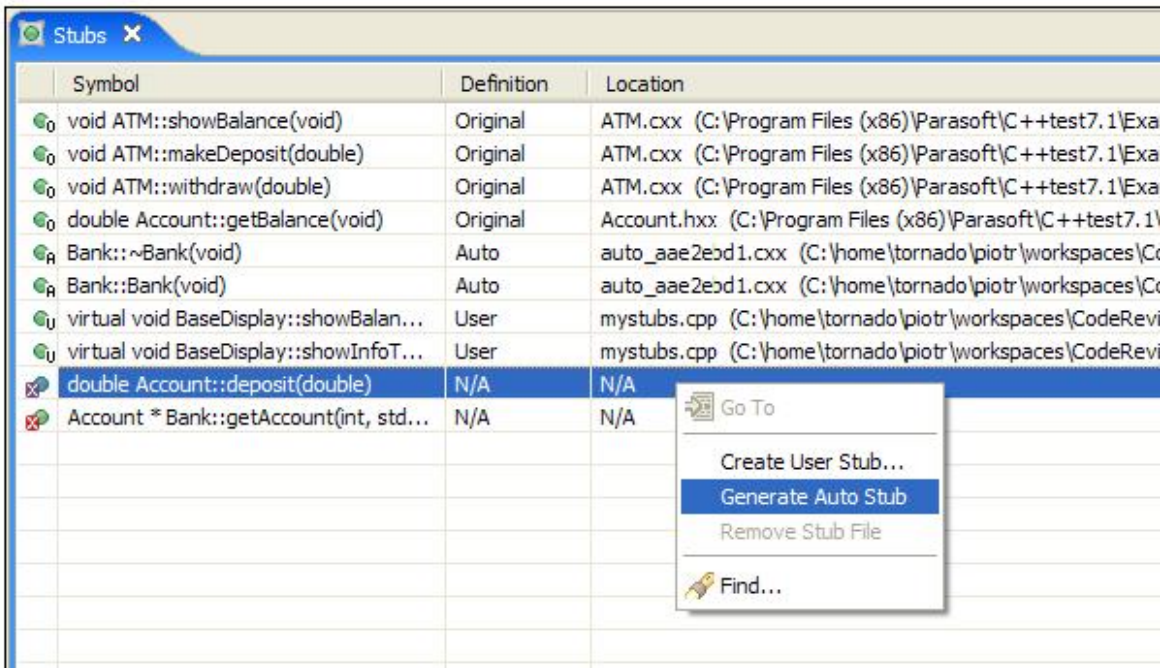
Accessing the Stubs View

To access the Stubs view:

- Choose **Parasoft> Show View> Stubs**.

Note

- When you first open the Stubs view, it will be empty. A message stating "Symbols data not collected" will be displayed.



Understanding the Stubs View

The Stubs view contains a table with the following columns:

- **Symbol:** Function or global variable name.
- **Definition:** The current definition/stub type:
  - **User:** User provided definition/stub will be used.
  - **Safe:** C++test's safe definition/stub will be used.
  - **Original:** Original definition will be used.
  - **Auto:** C++test's auto definition/stub will be used.
  - **N/A (not required):** Definition is not available, not needed by the linker.
  - **N/A:** Definition is not available, but is needed by the linker (in most cases, this will result in a linker error when building the test executable).
- **Location:** Location of the current definition (source file, library, N/A if not found).

Updating

The Stubs view updates its contents based on data collected during unit testing, and in response to specific actions available from the Stubs view (Create User Stub, Generate Auto Stub). Note that it does not update its contents in response to external actions that you perform (manually adding/removing stubs, etc.).

Unused Definitions

The Stubs view presents information about the most-recently used stub configuration, as well as other available (but not used) definitions. For instance, it shows information about existing original definitions for functions with user stubs defined. Such definitions display `(not used)` in the 'Definition' column—for example: `Original (not used)`. To hide such definitions in the Stubs view, click the **Filter** button in the Stubs view tool bar, then check **Hide unused definitions**.

Multiple Definitions

If a single function has more than one stub definition (for example, if there are two user stub definitions for a particular function), then the Stubs view will show both of them, but place an error mark on the stub icon and display `(conflict)` in the 'Definition' column—for example: `User (conflict)`.

Collecting/Refreshing Symbols Data

To collect or refresh symbols data:

1. In the project tree, select the files to be tested.
2. Run a Test Configuration that will build the test executable (for example, the "Collect Stub Information" or "Build Test Executable" Test Configuration).

Tips for Using the Stubs View

- To jump to a symbol definition/stub (in the code editor), double-click the related table row. Or, right-click it and choose **Go to** from the shortcut menu (available only for definitions located in files that are part of the current project).
- To remove a file with stub definitions, select that file in the table, then choose **Remove Stub File** from the shortcut menu. All stub definitions located in this file will be removed.
- To sort the table by one column's values, click on the column's header.
- To search table contents, right-click the table, choose **Find** from the shortcut menu, then specify search criteria.

Adding User-Defined Stubs to a Wizard-Generated Stub File

To add user-defined stubs by creating and then editing a wizard-generated stub file:

1. If you have not already done so, create a new directory for your stubs.
    - The stubs directory can be located anywhere within the project. By default, C++test expects stubs to be stored in a subdirectory of the project's `stubs` directory. However, you can use a different location as long as you modify the Test Configuration's **Use extra symbols from files found in** setting (in the **Execution> Symbols** tab) accordingly.

Tip

If you do not want to store your stubs within the project directory, you can add a folder that links to files stored elsewhere in your file system. To do this:

a. Choose **File> New> Folder** (if this is not available, choose **File> New> Other**, select **General> Folder**, then click **Next)**.

b. Click the **Advanced** button.

c. Enable the **Link to folder in file system** option.

d. Enter or browse to the location of your source files.

e. Click **Finish**.

2. Open the Stub Wizard in one of the following ways:
    - In the Stubs view, right-click the function for which you want to create a stub, then choose **Create User Stub**.
    - Select your stubs directory in the project tree, right-click the selection, then choose **New> Other** from the shortcut menu. A wizard will open. Select **C++test> User stub**, then click **Next**. Then, in that wizard's functions table, select the function for which you want to create a stub, and click **Next**.

Note

- The "Symbols data not collected" warning indicates that the required symbols data was not yet collected. See "Collecting/Refreshing Symbols Data", page 464 for details on how to collect/update symbols data.
- The "No symbols to create stubs for" warning indicates that there are no known functions for which user-defined stubs can be created.

3. In the User Stub File dialog that opens, enter a name and location for the new stub file. The user-defined stub file will then be created and opened in the code editor. C++test will automatically add the appropriate definition and the required #include directives.
4. Click **Finish**. The stub file will be automatically opened in the editor.
5. Examine/modify the stub definition and/or #include directives as needed.
6. Save the modified file.

Note

User-defined stubs...

- Can be created for multiple functions at once. To do this, select multiple functions in the table, then right-click the selection and choose **Create User Stub** from the shortcut menu. All user stubs will be added to the same stub file.
- Can be created for any function.
- Have the highest priority. A user-defined stub will be used even if the original definition is available.
- Cannot be created for global variables. Auto Stubs should be used instead.

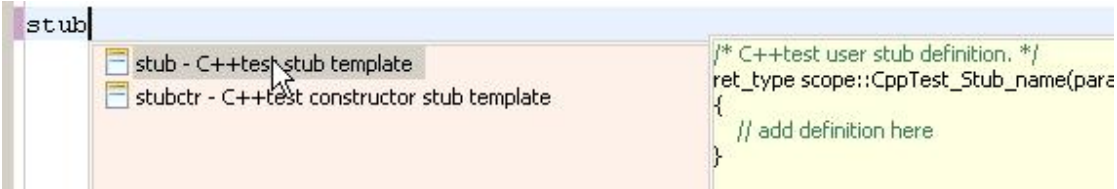
Tip

- To quickly add stubs for all functions from a given library, sort the table by Location so it is easier to select all functions from that library.

Adding User-Defined Stubs to an Empty Stub File

To add user-defined stubs to an empty stub file:

1. Open the Stub Wizard in one of the following ways:
   - In the Stubs view, right-click the function for which you want to create a stub, then choose **Create User Stub**.
   - Select your stubs directory in the project tree, right-click the selection, then choose **New> Other** from the shortcut menu. A wizard will open. Select **C++test> User stub**, then click **Next**. Then, in that wizard's functions table, select the function for which you want to create a stub, and click **Next**.
2. Do not make any selection in the functions table (in order to create an empty stub file).
3. Click **Next**.
4. Enter the stub file name/location.
5. Click **Finish**. The stub file will be automatically opened in the editor.
6. Create a stub template by typing `stub`, placing your cursor immediately after the "b" in "stub", pressing **Ctrl+ Space**, then choosing the appropriate template (either a standard stub template for a C or C++ function, or a constructor/destructor stub template).



7. Insert the appropriate values for the ret_type, scope, and name, parameters.
    - Note that C++test's stubs (except for constructor stubs) take the same values as the original functions.

Tip

- You can use the **Tab** key to move between ret_type, scope, name, and parameters.

8. Enter the stub body/definition.
    - User-defined stubs can interact with C++test API functions as described in "C++test API Functions for User-Defined Stubs", page 471.
9. Save the modified file.

Stubbing symbols from a library (outside of your project)?

Be sure to set the Test Configuration's Function stubs mode to **Stub all calls** as follows:

1. Choose **Parasoft> Test Configurations**.
2. Select the Test Configuration you will be using to execute tests with these stubs (or create a new one).
3. Open the **Execution> General** tab.
4. Under **Instrumentation mode**, choose **Custom instrumentation**.
5. Click the **Edit** button to the right of **Instrumentation mode**.
6. At the bottom of the Instrumentation features dialog, set the Function stubs mode to **Stub all calls**.

Stubbing virtual function calls?

When stubbing virtual function calls, be sure to create a stub for a function from a class that the given pointer or reference points to at compilation time.

For example, in the following code

```
void example(Base* ptr)
{
    // ...
    ptr->doSth();  // (*)
    // ...
}
```

the call marked with (*) will be stubbed *only* if a stub for a Base::doSth() method is created. The actual runtime type of the object pointed to by the pointer does not matter.

To create user stubs for operators, use the following stub function names:

Operator	Function Name
new	CppTest_Stub_operator_new
delete	CppTest_Stub_operator_delete
new[]	CppTest_Stub_operator_array_new
delete[]	CppTest_Stub_operator_array_delete
+	CppTest_Stub_operator_plus
-	CppTest_Stub_operator_minus
*	CppTest_Stub_operator_star
/	CppTest_Stub_operator_divide
%	CppTest_Stub_operator_remainder
^	CppTest_Stub_operator_excl_or
&	CppTest_Stub_operator_ampersand
|	CppTest_Stub_operator_or
~	CppTest_Stub_operator_or
!	CppTest_Stub_operator_not
=	CppTest_Stub_operator_assign
<	CppTest_Stub_operator_lt
>	CppTest_Stub_operator_gt
+=	CppTest_Stub_operator_plus_assign
-=	CppTest_Stub_operator_minus_assign
*=	CppTest_Stub_operator_times_assign
/=	CppTest_Stub_operator_divide_assign
%=	CppTest_Stub_operator_remainder_assign
^=	CppTest_Stub_operator_excl_or_assign
&=	CppTest_Stub_operator_and_assign
|=	CppTest_Stub_operator_or_assign
<<	CppTest_Stub_operator_shift_left
>>	CppTest_Stub_operator_shift_right
>>=	CppTest_Stub_operator_shift_right_assign
<<=	CppTest_Stub_operator_shift_left_assign
==	CppTest_Stub_operator_eq

Operator	Function Name
!=	CppTest_Stub_operator_ne
<=	CppTest_Stub_operator_le
>=	CppTest_Stub_operator_ge
&&	CppTest_Stub_operator_and_and
||	CppTest_Stub_operator_or_or
++	CppTest_Stub_operator_plus_plus
--	CppTest_Stub_operator_minus_minus
->*	CppTest_Stub_operator_arrow_star
->	CppTest_Stub_operator_arrow
()	CppTest_Stub_operator_function_call
[]	CppTest_Stub_operator_subscript
<?	CppTest_Stub_operator_gnu_min
>?	CppTest_Stub_operator_gnu_max
,	CppTest_Stub_operator_comma

Generating Stubs for Symbols with Missing Definitions

To generate an Auto Stub file for a symbol with missing definitions:

- In the Stubs view, right-click the function in the table, then choose **Generate Auto Stub** from the shortcut menu.

An Auto Stub file will be created and opened in the editor.

C++test will automatically add the appropriate definition and required #include directives.

Note

Auto Stubs...

- Can be created for multiple symbols at once. To do this, select multiple functions in the table, then right-click the selection and choose **Generate Auto Stub** from the shortcut menu. All Auto Stubs will be added to the same stub file.
- Can be created only for symbols without available definitions. For other symbols, use User Stubs instead.
- Have the lowest priority. An Auto Stub will not be used if any other definition is available.

Understanding and Customizing Automatically-Generated Stubs

C++test can be used to automatically generate customizable stubs for missing function and variable definitions as described above.

Automatically-generated stubs have the same functionality as user-defined stubs, but are marked with the `CppTest_Auto_Stub_` prefix (instead of the `CppTest_Stub_` prefix). This allows you to have multiple stubs for the same function the in scope.

If C++test cannot automatically generate a complete stub definition, it will create a stub template that you can customize (by entering the appropriate return statement, adding include directives etc.). Stub templates will be saved in the stub file before complete stubs.

Automatically generated stubs will be used only if no other definition (user stub or original) is available.

Automatically-generated stubs and stub templates can be customized as needed—for instance, to change the value being returned from the stub, or to use the C++test stubs API (e.g. CppTest_IsCurrentTestCase) described in "C++test API Functions for User-Defined Stubs", page 471.

To customize these stubs or stub templates:

1. Open the related stub file, which is saved in the location indicated in the Test Configuration's **Auto-generated stubs output location** (in the **Execution> Symbols** tab).
2. Modify the ret_type, scope, name, and parameters as needed.
    - Note that C++test's stubs (except for constructor stubs) take the same values as the original functions.
    - You can use the **Tab** key to move between ret_type, scope, name, and parameters.
3. Modify the stub body/definition as needed.
    - User-defined stubs can interact with C++test API functions as described in "C++test API Functions for User-Defined Stubs", page 471.
4. Save the modified file.
5. Rerun the analysis.

Disabling Automatically-Generated Safe Stubs

Safe definitions are automatically-generated to replace "dangerous" functions. Safe definitions are available for most system I/O routines (`rmdir()`, `remove()`, `rename()`, etc.). If a safe definition is used, the originals are not called—even if they are available. We recommend that you use safe definitions when they are available; using these definitions prevents problems during unit testing. These stubs cannot be modified.

If you do not want to use the automatically-generated safe definitions, delete the `${cpptest:cfg_dir}/safestubs` entry from the **Use extra symbols from files found in** field in the Test Configuration's **Execution> Symbols** tab.

If you want to disable usage of a safe stub for a particular function (and use the original definition instead), write the user stub that will work as a wrapper for the original function call. For example:

```
int CppTest_Stub_mkdir(const char* p)
{
    return mkdir(p);
}
```

C++test API Functions for User-Defined Stubs

```
void CppTest_Assert(bool test, const char * message)
```

This function works in a similar fashion to a standard assert function. Whenever the value of the "test" parameter is false, test case execution is stopped. A "User-defined assertion failed" message will be reported as the test case result. In addition, the value of the "message" parameter will be shown as a detailed failure description, along with location and stack trace details.

```
void CppTest_Break()
```

This function allows you to unconditionally stop the test case execution. Test cases stopped in this manner result in a "User-defined break called" message. In addition, the location and stack trace information is available.

```
bool CppTest_IsCurrentTestCase(const char* id;
```

This function allows you to query the currently executed test case. It will return `true` if the specified `id` equals the name of the currently executed test case, otherwise it will return `false`. This feature is useful for functions that use conditional statements based on the external function calls. See "Using Stubs Driven By Test Cases", page 472 for an example.

```
bool CPPTEST_DS_HAS_COLUMN(const char* name)
```

This function allows you to query data source columns in user/auto stubs. Note that data sources are test case specific, so the data is available only when the stub is called in the context of a given test case (it is not available when the stub is called during global initialization, etc).

```
const char* CppTest_GetCurrentTestCaseName();
const char* CppTest_GetCurrentTestSuiteName();
```

These functions get the name of the currently-executed test case and test suite. Using these functions, you can write a stub that applies to a specific test case from a specific test suite.

See "Using Stubs Driven By Test Cases", page 472 for an example.

Using Data Sources in Stubs

Any data source that you configure for use in C++test can be used in stubs.

To configure a data source, use the instructions provided in "Adding Data Sources", page 429.

To use a data source in a stub, use the bool CPPTEST_DS_HAS_COLUMN(const char* name) API function. You can query a data source column in a stub as follows:

```
int CppTest_Stub_goo (void)
{
        if (CPPTEST_DS_HAS_COLUMN("stub_goo_return")) {
            return CPPTEST_DS_GET_INTEGER("stub_goo_return");
        } else {
                return 0; // Data Source not available
        }
}
```

Note that data sources are test case specific, so the data is available only when the stub is called in the context of a given test case (it is not available when the stub is called during global initialization etc).

The `bool CPPTEST_DS_HAS_COLUMN(const char* name)` macro can also be combined with other API functions—such as CppTest_isCurrentTestCase().

Using Different Tests and/or Stubs for Different Contexts

See "Using Different Tests and/or Stubs for Different Contexts", page 421.

Using Stubs Driven By Test Cases

C++test allows you to create stubs that can be "driven" by the currently executed test case. The C++test API provides the following function:

```
bool CppTest_IsCurrentTestCase(const char* id);
```

This function can be used in the "user stub" definition to query a currently executed test case. It will return `true` if specified `id` equals the name of the currently executed test case, otherwise it will return `false`.

This feature is useful for functions that use conditional statements based on the external function calls. For example:

```
void foo()
{
  if (goo() == 1) {
     //code
  } else {
     //code
  }
}
```

To achieve 100% line coverage, you could create two test cases for the `foo()` function and then create a user stub for the `goo()` function:

```
int ::CppTest_Stub_goo()
{

  if (CppTest_IsCurrentTestCase("TestCase1")) {
    return 1;
  } else {
    return 0;
  }

}
```

This stub is now "driven" by test cases. In this example, the return value depends on the currently executed test case: it depends upon the `TestCase1` test case. For `TestCase1`, it will return 1, and for any other test case(s) it will return 0. In this way, you could achieve 100% coverage for the `foo()` function.

Other API functions that are useful for writing stubs driven by test cases are `const char* CppTest_GetCurrentTestCaseName();` and const char* CppTest_GetCurrentTestSuiteName();. These functions get the name of the currently-executed test case and test suite. Using these functions, you can write a stub that applies to a specific test case from a specific test suite. For example, the following stub behaves differently for all test cases that have a name containing "nomemory" and that are from the "AllocTestSuite" test suite:

```
#include <string>
#include <stdlib.h>
```

```
EXTERN_C_LINKAGE void* CppTest_Stub_malloc(size_t size)
{
    std::string testSuiteName = CppTest_GetCurrentTestSuiteName();
    std::string testCaseName = CppTest_GetCurrentTestCaseName();
    if ((testSuiteName == "AllocTestSuite") &&
        (testCaseName.find("nomemory") != std::string::npos))
    {
        // Simulate no memory situation.
        return 0;
    }

    return malloc(size);
}
```

Specifying Custom Compiler Options for User-Defined Stub Files

You can set custom compiler options (for instance, to use C++test-specific flags) for each user-defined stub file as described in “Specifying Custom Compiler Settings and Linker Options for Testing with C++test”, page 208.

Executing Manually-Written CppUnit Test Cases

This topic explains how to prepare C++test to execute your existing manually-written CppUnit test cases. Running CppUnit test cases in C++test allows you to centralize unit testing and reporting. C++test's reporting and authorship calculation capabilities help the team track which test cases failed, since when, and who is responsible for fixing each failure.

The team can run these tests in command-line mode each night. For instant feedback on whether their code changes broke the existing functionality, each developer can import the regression failures caused by their modifications. Since the regression failures are directed to the developers responsible for them, the overall process of fixing them is much more streamlined than it would be if all developers were looking at the same list of regression failures.

Additionally, C++test provides test coverage information for CppUnit test cases and can also perform runtime error detection as they execute.

Sections include:

Accessing the Tests

To configure C++test to access your manually-written CppUnit test cases:

1. Ensure that the CppUnit test case files are available within the project tree.
    - The test directory can be located anywhere, as long as it is visible in the project tree. By default, C++test expects tests to be stored in a subdirectory of the project's `tests` directory. However, you can use a different location, as long as you modify the Test Configuration's **Test suite file search patterns** setting (in the **Execution> General** tab) accordingly.
    - If you do not want to store your tests within the project directory, you can add a folder that links to files stored elsewhere in your file system. To do this:
        a. Choose **File> New> Folder** (if this is not available, choose **File> New> Other**, select **General> Folder**, then click **Next**).
        b. Click the **Advanced** button.
        c. Enable the **Link to folder in file system** option.
        d. Enter or browse to the location of your source files.
        e. Click **Finish**.
2. Choose **Parasoft> Test Configurations** to open the Test Configurations dialog.
3. Select the Test Configurations category that represents the user-defined Test Configuration you want to execute the CppUnit tests.
4. Open the **Execution** tab.
5. In the **General** subtab, modify the **Test suite file search patterns** setting as needed so that C++test will locate and test the CppUnit tests.

- Be sure to add an asterisk (*) at the end of the directory path.

6. Click either **Apply** or **Close** to commit the modified settings.

The tests will be executed when you run a test using this Test Configuration.

7.

Requirements/Limitations

The imported test cases must satisfy the following conditions:

- The CppUnit test class must have CPPUNIT_NS::TestFixture or CPPUNIT_NS::TestCase as its base class.
- The CppUnit test class cannot be a template class.
- The CPPUNIT_TEST_SUITE_REGISTRATION(TestSuiteName) macro should be inserted into a CppUnit source file (.cpp).
- The following CppUnit macros are supported:
  - CPPUNIT_TEST_SUITE(ATestFixtureType)
  - CPPUNIT_TEST_SUITE_END()
  - CPPUNIT_TEST_SUITE_NAMED_REGISTRATION
  - CPPUNIT_TEST(testMethod)
  - CPPUNIT_TEST_EXCEPTION(testMethod, ExceptionType)
  - CPPUNIT_TEST_FAIL(testMethod) CPPUNIT_ASSERT(condition)
  - CPPUNIT_ASSERT_MESSAGE(message, condition)
  - CPPUNIT_FAIL(message)
  - CPPUNIT_ASSERT_EQUAL(expected, actual)
  - CPPUNIT_ASSERT_EQUAL_MESSAGE(message, expected, actual)
  - CPPUNIT_ASSERT_DOUBLES_EQUAL(expected, actual, delta)
  - CPPUNIT_ASSERT_THROW(expression, ExceptionType)
  - CPPUNIT_ASSERT_NO_THROW(expression)
  - CPPUNIT_TEST_SUITE_REGISTRATION(ATestFixtureType)
- The context of the imported CppUnit test cases will be set to 'project'.

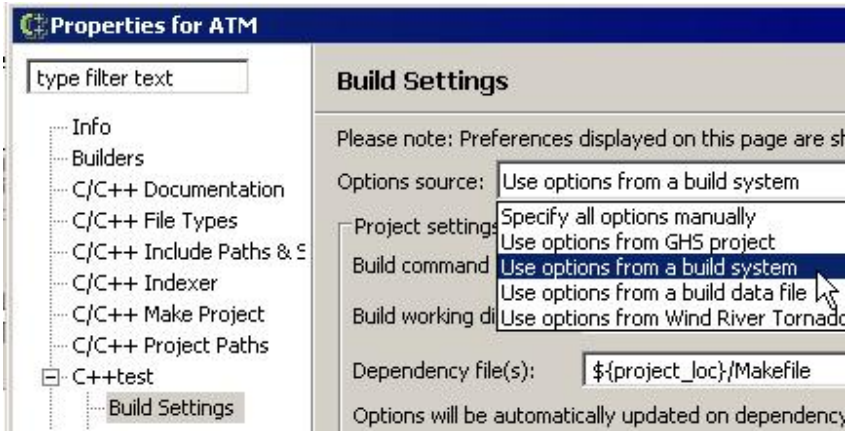
Excluding Test Cases from the Build

If a given CppUnit file is not excluded from build, then:

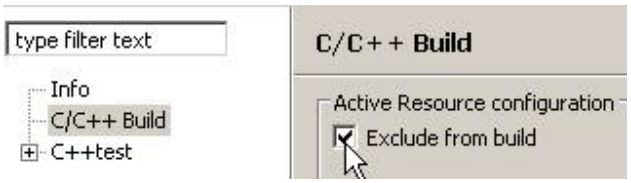
- Static analysis will be performed for that test file.
- Coverage will be reported from that test file.
- Function calls will be stubbed in that test file.
- Auto-generated tests will be created for that test file.

Thus, we recommend excluding CppUnit files from the build. However, it is not required.

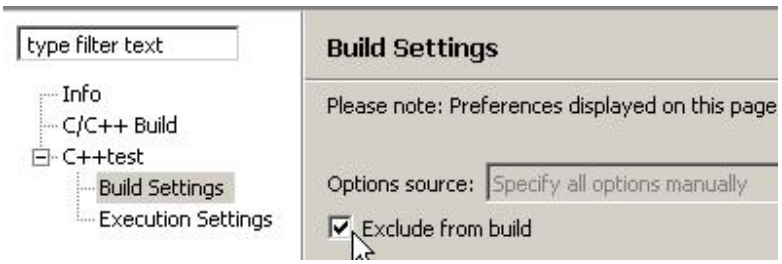
The procedure for excluding CppUnit files from the build depends on your project's options source (set in the C++test Build Settings):



- If the project is set to "Use Options from Managed Make C/C++ Project", right-click the source file containing the CppUnit method definitions, choose **Properties**, then go to **C/C++ Build> Exclude from build**. Then, repeat the same configuration for the header file containing the CppUnit class.



- If the project is set to "Specify All Options Manually", right-click the source file containing the CppUnit method definitions, choose **Properties**, then go to **Parasoft> C++test> Build Settings> Exclude from build**. Then, repeat the same configuration for the header file containing the CppUnit class.



- For any other option source, the file will be excluded only if its build options are not available in the original build system (Makefile, .dsp etc.). For example, when you open the ATM project that is shipped with C++test, the CppUnit test suite is automatically excluded from build because it is not a part of the ATM project's Makefile.