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. Generate test cases as described in Generating Test Cases for Regression Testing and Exception Finding.
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 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 compliable 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 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.
6. (Optional) Fine-tune test execution settings as needed.
   - For details, see Execution Tab Settings: Defining How Tests are Executed.

Testing Template Functions

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

See Support for Template Functions 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 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 can be compiled. 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, 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. 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 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).
- 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), then modifying their source as necessary (see Adding and Modifying Stubs).

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

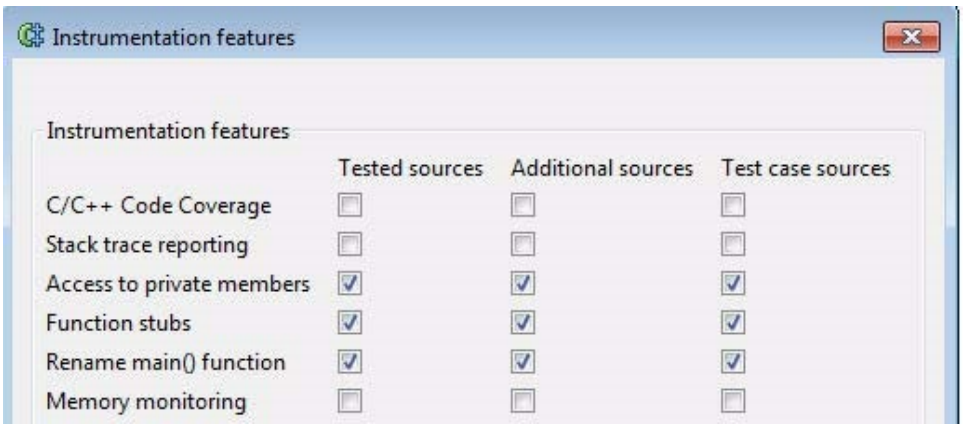
When testing a single file in isolation, you might want to use the object and library files filter to prevent testing of specified libraries and objects. In the Test Configuration's **Ignore object/library files** field (in the Execution> Symbols tab), you can specify a semicolon-separated list of patterns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

Testing Functions in Isolation

You can test functions in isolation in many ways. This section describes the approach used frequently for testing safety-critical source code. A fundamental requirement for this kind of testing is that the tested function source code should not be modified by the instrumentation module. This is to ensure that test results reflect function execution in a real production environment. This is to ensure that test results reflect function execution in a real production environment.

To ensure that a tested function is not instrumented, you should verify the instrumentation settings in the test configuration:

1. Open the test configuration and choose the **Execution> General** tabs
2. Enable the **Unit Testing** execution mode and click **Edit** in the Execution details section at the Instrumentation mode field.



Only the following features should be enabled in the Tested sources category:

- Access to private members
- Functions stubs (with appropriate configuration of stubs instrumentation mode)
- Rename main() function

The following sections will provide additional details on configuring the stubbing mode, managing stubs per test suite configuration, and test executable configuration for testing functions in isolation.

Configuring Stubs Instrumentation

You must enable stubs instrumentation and select the stubbing mode to use function stubs for tested code. Enabling stubs instrumentation activates the stubbing engine. Selecting a stubbing mode defines the way in which stub calls are applied. There are two stubbing modes available:

- Instrument function calls
- Instrument stubbed functions (function calls not modified)

The following examples illustrate how these modes differ in the way the stub function or method is invoked.

```
/* Original definition of function to be stubbed */
int Func(void)
{
     // Function body
 }
/* tested function definition */
int testedFunction(void)
{
     int val = Func ();
     /* tested function body */
     return val;
}
```

The following tables shows the instrumented version of the example source code for both stubbing modes:

Instrument Function Calls	Instrumented Stubbed Functions

`/* Stub function definition */`
`int CppTest_Stub_StubbedFunc(void)`
`{`
`  // Stub body`
`  return 0`
`}`
`/* Original definition of stubbed function */`
`int Func(void)`
`{`
`  // Function body`
`}`
`/* tested function definition */`
`int testedFunction(void)`
`{`
`  int val = CppTest_Stub_StubbedFunc();`
`  /* tested function body */`
`  return val;`
`}`
	`/* Stub function definition */`
	`int CppTest_Stub_StubbedFunc(void)`
	`{`
	`  // Stub body`
	`  return 0`
	`}`
	`/* Original definition of stubbed function */`
	`int Func(void)`
	`{`
	`  return CppTest_Stub_StubbedFunc();`
	`  // Function body removed`
	`}`
	`/* tested function definition */`
	`int testedFunction(void)`
	`{`
	`  int val = Func ();`
	`  /* tested function body */`
	`  return val;`
	`}`

The Instrument stubbed functions mode is usually preferred when testing safety-critical code because it does not change the body of tested function. To set the stubs mode:

1. Open the test configuration and choose the **Execution>General** tab
2. Click **Edit** next to **Instrumentation mode** drop-down menu and enable the stubbing mode

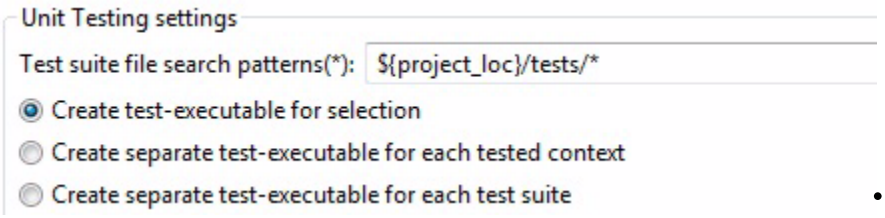
Configuring the instrumentation settings in this way introduces some limitations:

- If a tested function requires a stub for another function, the stubbed function becomes untest-able in the same test session (if defined in tested source code)
- Test suites should group test cases for functions or methods with the same stubs configuration
- A single test binary cannot contain test suites that have conflicting stubs configurations. This is true when one test suite assumes that there is a stub for a function under test in another test suite

The following sections describe how to work with these limitations.

Configuring Test Executable Build

1. Open a test configuration and choose the **Execution> General** tab
2. In the Unit Testing settings section, enable a rule for dividing the selected test suite into test executable(s)



The following settings are available:

- **Create test-executable for selection**: Default value for all non-File Scope test configurations.
- **Create separate test-executable for each tested context:** Default setting for File Scope test configurations.
- **Create separate test-executable for each test suite:** Enable this option to create a separate test binary for each test suite from your selection. This may be required if your test suites group test cases, which requires a specific selection of stubs that cannot be mixed.

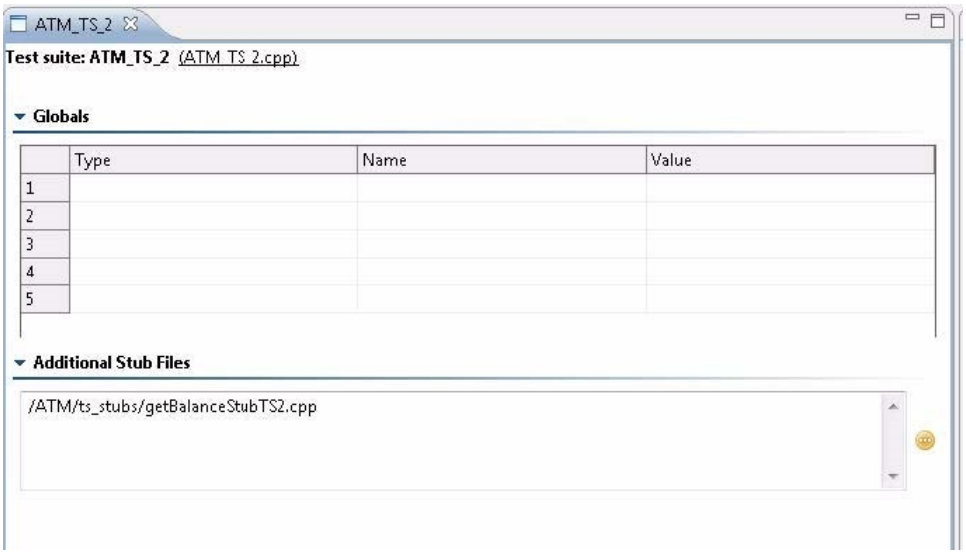
3. Click **Apply**

Managing Test Suite-specific Stubs

Unit testing strategies sometimes require organizations to use separate sets of stubs for each test suite. This can be accomplished with either a dedicated test configuration for each test suite, or, preferably, by specifying desired stub file(s) directly in the test suite.

To specify stub files for the test suite:

1. Open the Test Case Explorer View and double click a test suite
2. Drag the desired stub file into the Additional Stub Files field
   Alternatively, you can browse for stub files by click the file browser button located to the right of Additional Stub Files field



Test suite-specific stub files can also be added to the test suite without using the GUI by modifying the test suite file directly. To add a stub file to the test suite, open the test suite in a text/code editor and insert the following macro for each stub file. You can also use a root directory path to include all stubs from a given location.

```
CPPTEST_ADDITIONAL_STUB_FILES(<stub file path or root directory>);
```

The macro can be specified in any valid C/C++ macro location, but it must be added after the cpptest.h header file. All stub files specified at the test suite level are appended to the stub files affected by the test configuration settings.

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 Editor containing the test case and make sure that the Editor window is “in focus” (see Generating Test Cases, for information about using the Test Case Editor)
   - 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 (e.g., Builtin> Unit Testing> Run Unit Tests).
   - 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 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 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 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 for details.
- Otherwise, provide user-defined stubs for the missing functions, then rerun the test. See Adding and Modifying Stubs for details.

Using a Debugger During Test Execution

See Using a Debugger During Test Execution.