Execution Tab Settings - Defining How Tests are Executed

During a test, C++test will execute generated and user-defined tests (or run the application with runtime coverage monitoring enabled) based on the parameters defined in the selected Test Configuration's Execution tab

The Execution tab has the following settings:

- **Enable Test Execution:** Determines whether C++test executes available unit tests or runs the application with runtime coverage monitoring enabled. If this option is not checked, all other test execution parameters are irrelevant.

General tab

- **Execution mode> Application Monitoring:** Configures C++test to prepare an instrumented version of the application executable and then run it (e.g., for runtime error detection, as described in Runtime Error Detection).
- **Execution mode> Unit Testing:** Configures C++test to execute unit tests.
- **Execution details> Instrumentation mode:** Determines the degree to which test cases/application are instrumented during execution. Available presets include:
  - **Full:** Provides full instrumentation, including coverage, function stubs, stack trace reporting, access to private members, and renaming "main" function. These instrumentation options are described below, under the description of the Custom instrumentation option.
  - **Full runtime with memory monitoring:** Provides full instrumentation mode and also instruments the test executable for runtime error detection during unit test execution.
  - **Full runtime with line coverage:** Provides full instrumentation mode with line coverage only. Line hit count is disabled.
  - **Full runtime w/o coverage:** Provides partial instrumentation, including function stubs, stack trace reporting, access to private members, and renaming "main" function—but not coverage.
  - **Full runtime w/o stubs:** Provides partial instrumentation, including coverage, stack trace reporting, access to private members, and renaming "main" function—but not function stubs.
  - **Application full monitoring:** Instruments the complete application with support for both runtime error detection and coverage tracking. If you will be running tests on an embedded device with limited memory, you may not be able to use both coverage and memory monitoring together; in this case, you can alternate between the following two modes.
  - **Application coverage monitoring:** Instruments the complete application with support for coverage tracking.
  - **Application memory monitoring:** Instruments the complete application with support for runtime error detection and coverage monitoring.
  - **No instrumentation:** No instrumentation will be performed. Coverage information will not be tracked and function stubs will not be used.
  - **Custom instrumentation:** Click **Edit** to customize the level of instrumentation and select Custom instrumentation from the drop-down menu to enable your settings.
    - The following table describes the **Instrumentation features** you can enable/disable for tested sources, additional sources, and test case sources:

      a. **C/C++ Code Coverage:** Determines whether coverage information is tracked. Coverage instrumentation for "test case sources" (the standalone test suites) is needed for the inline functions from included headers. C++test never instruments test case functions for coverage.
      b. **Stack trace reporting:** Determines whether stack trace callbacks are reported. If disabled, the reported stack trace will not show the exact lines being executed and will miss non-stubbed function calls.
      c. **Access to private members:** Determines whether private members are used in test cases (for example, calling private functions, using private constructors to create objects, setting up private fields in test case preconditions, and checking values of private fields in postconditions).
      d. **Function stubs:** Determines whether safe definitions (provided by C++test) and user-defined stubs are used in unit testing.
      e. **Rename main() function:** Determines whether code that contains a "main()" function is tested. If enabled, C++test will rename this function during instrumentation. This is needed because C++test provides its own "main()" function that does test case execution. If this option is disabled and the code under test contains a "main()" function, there will be an error during the test executable linking phase caused by multiply-defined "main" symbols. When you are working in "Application Monitoring" mode, this mode should be left disabled."
      f. **Memory monitoring:** Determines whether runtime error detection for memory-related problems is enabled.

      g. **Assembly Code Coverage:** Determines whether assembly coverage information is tracked. See Assembly Code Coverage.

    - The **C/C++ Code Coverage metrics** section allows you to select the type of coverage metric(s). The **Line hit count** option determines whether the number of line hits is calculated. For additional information about types of metrics, see Understanding Coverage Types.
    - Select the **Enablecoverage for C++ templates** option to enable the collection of code coverage information from C++ templates. See Support for Template Functions.
    - Select the **Optimize coverage (Application Monitoring Only)** option to enable special optimized coverage modes, which are described in the following table:

Option	Purpose
Coverage instrumentation> Size optimized	The memory buffers used by coverage instrumentation for storing metrics data are optimized to take the lowest possible amount of RAM. This mode is suitable for embedded devices withlimitedamount of memory.

Coverage instrumentation> Speed optimized	The memory buffers used by code coverage instrumentation for storing metrics data are optimized to assure the lowest possible execution time overhead. This mode is suitable for measuring coverage on application withlowtolerance for additional execution time overhead.
Enable data consistency check	This option enables additional algorithms for detecting coverage buffer corruptions. Enable this option if coverage buffer corruption is likely caused by faulty behavior in the application under test. Enabling this option imposes some additional execution time overhead.
Enable initialization of the coverage memory buffer	This option enforces the initialization of memory buffers used by code coverage instrumentation. This option should be selected if your compiler does not initialize global and static variables to 0 before the program begins running (required by ISO C standard). This option is enabled automatically (and can not beunabled) if "Size optimized" coverage instrumentation mode and "Enable data consistency check" are enabled simultaneously.

- **Enable enum data autogeneration**: Determines whether enum data required by enumeration-related API macros is collected (see Handling Enum Values for details).

Customizing the test execution flow

When you run a Test Configuration set to execute tests, C++test performs a series of actions which usually leads to the unit testing results being loaded into the C++test UI. These actions are defined in a test flow file, which is stored in XML format and saved as part of the Test Configuration, which makes it possible to share it across the team.

C++test provides a default execution flow that is designed specifically for host-based testing.

C++test also allows you to define a custom test flow which can include C++test internal actions or external utilities started as processes in the operating system. The default test flow can be modified by:

- Customizing the parameters of existing steps - see Customizing the Test Execution Flow
- Removing existing steps
- Adding new steps

To define a custom test flow:

1. Open the Test Configurations panel by choosing **Parasoft> Test Configurations**.
2. Select the Test Configuration that you plan to use for test execution.
3. Open the **Execution> General** tab.
4. Click the **Edit** button
5. Enter your modified test flow or adapt existing one
    - If you want to adapt an existing test flow, choose that test flow from the **Available built-in test execution flow (s)** box, then click **Restore**. The XML for that test flow will then be displayed, and can be edited as needed.
6. Click **OK** to save the modified file.

- **Execution details> Test execution flow:** Determines whether C++test uses the default flow for host-based unit testing, for building a test executable (e.g., for trial builds without execution), for generating stubs for missing function and variable definitions, or a custom flow (e.g., for embedded or other cross-platform testing). Details on defining a custom flow are discussed in Customizing the Test Execution Flow.
- **Execution details> Quick execution mode:** Configures C++test to be less strict when verifying whether the test executable subproducts are current (e.g., whether they have changed since the previous run).

Tip: When to use Quick Execution Mode

"Quick execution mode" can be used to reduce the time required to prepare the test executable. Since C++test will be less strict when verifying whether the test executable subproducts are current, the test preparation phase is faster.

In this mode, C++test assumes that the following have not been modified since the previous run:

- Compiler and linker options.
- Header files.
- Stub configuration.

We recommended using the quick execution mode as follows:

1. Create a "quick" execution Test Configuration by duplicating a Test Configuration with the "normal" mode, and then enabling the **Quick execution mode** checkbox.
2. Use the two Test Configurations as follows:
    - When running tests on the server machine (nightly tests), use the "normal" Test Configuration.
    - When running tests on the developer machine (daily work), use "quick" Test Configuration.

Normal execution mode is required on the developer machine when:

- Compiler / linker flags in the project have changed (e.g., a new macro definition was added to the compiler options or a new external library was added to the list of the linker flags).
- One of the header files included by the tested source / test suite source / stubs source was modified.
- A new user stub was added to the project.
- A stub definition was removed from the project.
- Coverage data needs to be reported.

After you use the "normal" configuration to re-build the test-executable, you can return to the "quick" mode.

- **Unit Testing> Test suite file search patterns:** Determines where C++test looks for test suite files to be executed in the current test. To specify multiple locations (for instance, if you have different tests for different levels of testing or for host vs. target), enter a semicolon before each additional location.
    - The following variables are supported (examples are based on a tested file /MyProject/module1/src/MyClass.cpp)
        - ${source_name} - Full name of the tested file (MyClass.cpp)
        - ${source_base_name} - Base name of the tested file (MyClass)
        - ${source_ext} - Extension of the tested file (cpp)
        - ${source_loc} - Workspace-oriented location of the tested file (/MyProject/module1/src)
        - ${source_loc_rel} - Project relative location of the tested file (module1/src)
        - ${source_loc_rel:<path>} - <path> Relative location of the tested file (src, for <path>=/MyProject/module1)
        - ${test_ext} - C++test specific extension of a test suite file (.cpp)

        • ${project_loc} - The path to the Eclipse project location (the location that contains the .project file). Do not use this variable if your project has "linked" source folders—source files that do not actually live within your project location.• ${resource_loc} - The path to the specific project resource. For instance, ${resource_loc:/MyProject/linked_src_dir/source.cpp} will be resolved into C:\src\source.cpp. You can use this variable if your project has "linked" source folders—source files that do not actually live within your project location. For example: Do not use this variable if your project has "linked" source folders—source files that do not actually live within your project location.
    - Test suites must be available from the project tree.

      If you do not want to store your tests within the project directory, you can add a folder that links to files stored else-where in your file system. To do this:
      • Choose **File> New> Folder** (if this is not available, choose **File> New> Other**, select **General> Folder**, then click **Next**).
        - Click the **Advanced** button.
        - Enable the **Link to folder in file system** option.
        - Enter or browse to the location of your source files.
        - Click **Finish**.

- **Unit Testing> Create separate test executable for each tested context:** Determines whether C++test uses a separate test executable for each tested context (e.g., file).
    - If this option is enabled, C++test will create and execute a separate executable for each source file (but it will still use additional project sources and stubs as defined on the **Symbols** tab). An additional test executable will be prepared and run for all tests that do not have CPPTEST_CONTEXT set. If you prefer smaller test executables, enable this option.
    - If this option is disabled and the complete project is selected for unit test execution, C++test will build one large executable with all of the source files and run it once.

Symbols tab

Library symbols identification mode

This section specifies how C++test creates a list of original symbol definitions (e.g., for functions or global variable) that should be available (for example, in external libraries) when preparing a test executable.

- **Off:** C++test does not try to find symbol definitions and assumes that all necessary symbols will be available in the linking phase.
- **Automatic:** C++test automatically tries to create a list of original symbol definitions that should be used during linking.

Symbol Sources

- **Use symbols from additional project files:** Defines any project source files that C++test should use—in addition to those used by default (as specified in the first bullet below)—to resolve the original definitions from the tested files.
    - 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.
    - You can use the following C++test variables to specify the path to additional project files you want to use:
        - `${project_loc}` resolves to the path to theEclipseproject location(the location that contains the `.project` file). *You cannot use this variable if your project has "linked" source folders—source files that do not actually live within your project location.*
        - `${resource_loc}` resolves to the path to the specific project resource.For instance, `${resource_loc:/MyProject/linked_src_dir/source.cpp}` will be resolved into `C:\src\source.cpp`.
    - To use symbols from a specific subset of project files, enter that subset (for example,`${project_loc}/src/core/*`)
    - To build the test executable using all project sources, simply enter an asterisk (*) in this field. This ensures that all symbol definitions from the project will be available during the tests.
    - Failure to choose this option when testing a single file that references functions defined in other sources from the tested project could result in unresolved symbols, which could cause linker errors.
    - Do not disable this option unless the source(s) under test do not reference symbols from additional files or if stubs are being provided for all such symbols.
- **Use extra symbols from files found in:** Determines where C++test looks for safe stubs, user-defined stubs, and factory function definitions to be used when executing the current test. You can specify multiple locations (for instance, if you have different stubs for different levels of testing or for host vs. target); just enter a semicolon before each additional location.
- **Ignore object/library files:** Specifies libraries and objects to ignore (e.g., to facilitate testing a file in isolation). Use a semicolon-separated list of patterns of command line options. Only options from the linker command line are ignored. Standard and compiler libraries or included with pragmas are not filtered.

Auto-generated stubs

- **Auto-generated stubs output location:** Specifies where automatically-generated stubs are saved. See Understanding and Customizing Automated Stub Generation for details on automatically-generated stubs.
- **Dynamic Stubs Configuration**: specifies the mode(s) of dynamic stubs configuration:
  - **Enable Stub Callbacks**: enables the Stub Callbacks mechanism for dynamic stubs configuration; enabled by default (see Using Stub Callbacks for details)
  - **Enable Stub API (deprecated)** - enables the stub API for for dynamic stubs configuration (see Using Stubs API (deprecated) for details)

Perform early check for potential linker problems

This option determines if C++test aborts unit testing execution if any missing symbol is detected during symbols/stubs analysis.If unresolved symbols are reported, see Resolving Linker Errors from Unresolved Symbols for tips on how to resolve them.

Runtime tab

- **Test executable run directory:** Determines the directory in which the test executable is created and should execute. If relative paths are used in test case sources or in original code, C++test will search for the referenced files in this directory.

- **Run tests in debugger:** Determines if C++test runs tests with a debugger. See Using a Debugger During Test Execution for instructions on how to run tests with a debugger.

- **Debug directly in Eclipse IDE using configuration:** Determines if C++test runs tests with a debugger directly in the Eclipse IDE (according to an Eclipse Debug Configuration you have configured). If you enable this option, you must also specify the Eclipse Debug Configuration you want to use.

- **Timeout for a single test case (in milliseconds):** Determines the maximum number of milliseconds that C++test will wait for a test case to execute.
- **Report unit test execution details:** Determines whether unit test execution details should be reported from executed test cases. This includes values reported from the CPPTEST_REPORT macros.

- Include tasks details: Determines whether reported tasks (e.g. exceptions, failed assertions, outcomes) should be included in test case execution details.
- Include passed assertions details: Determines whether checked and passed assertions should be included in test case execution details.

- **Report unverified test case outcomes:** Determines if the postconditions obtained during test execution are reported. If this option is disabled, then the **Automatically validate test case outcomes** option is not available.
  - Automatically validate test case outcomes: Determines if the postconditions obtained during test execution are automatically "verified" (e.g., converted to assertions). See Verifying Test Cases for Regression Testing for details on why and how to verify test cases.

- **Generate detailed test execution report**: Enables generating the Test Execution Details Report (see Understanding Reports for details)
- **Generate detailed coverage report for [coverage metric]**: Enables collecting coverage details for a selected coverage type (see Reviewing Coverage Information for information about coverage types).