Migrating test assets from C++test 6.x

This topic explains how to migrate your C++test 6.x test assets to C++test 10.x.

In this section:

- Text Configurations
- "Source Test" Test Suites
- Native Tests
- Stubs
- Command Line (cpptestcli) Invocations

Text Configurations

To import Test Configuration from C++test 6.x:

1. (Recommended) Ensure that any user-defined/custom rules from the C++test 6.x Test Configuration are accessible in C++test 10.x (i.e., by sharing them via Team Server).
2. Open the Test Configurations dialog by choosing **Parasoft> Test Configurations**.
3. Right-click the **User-defined** category, choose **Import C++test 6.x Test Configuration** from the shortcut menu, then use the file chooser to select the appropriate C++test 6.x Test Configuration (.tcfg) file.

C++test will create a new user-defined Test Configuration with the same name as the C++test 6.x Test Configuration. The following data will be imported from the C++test 6.x Test Configuration:

- Coding standards state (enabled or disabled)
- Enabled coding standard rules. C++test 6.x built-in rules will be mapped to the current version's set of built-in rules, which use different identifiers, messages, and severity levels. We recommend that any custom rules use "-" as a separator; for example: `<rule_category>-<rule_unique_id>` (`myrule-123, user-456,` etc.) This way, C++test 10.x will be able to automatically create a rule tree structure based on the rule categories.
- Unit testing generation state (enabled or disabled)
- Max. count of generated test cases
- Display test case results as post conditions
- Automatically insert assertion templates
- Memory allocation for test object (stack or heap)
- Use heuristics for input values
- Use objects of derived classes
- Initialize global variables
- Unit testing execution state (enabled or disabled)
- Timeout for a single test case
- Test only files created or modified since the cut-off date

All other parameters in the new configuration will be set to their default values.

If C++test 10.x cannot access any rules that were enabled in the C++test 6.x Test Configuration, warning messages will display to alert you to the problem.

If any rules are added to the UNKNOWN category, you can use rule mapping to change the category. See Modifying Rule Categories, IDs, Names, and Severity Levels for details.

Test Configurations stored on Team Server need to be imported into the User folder and then reuploaded to Team Server in C++test 10.x format.

"Source Test" Test Suites

There are two ways to import "source test" test suites from C++test 6.x: import all test suites into a single directory, or import the test suites for each C++test 6.x Test Unit into a separate subdirectory.

Before you import test suites, note that:

- Symbols data must be available for the "source" 6.x project ("Read symbols" action).
- The C++test 10.x project must contain the same (physical) source files as the 6.x project. The "logical" project layout can be different (i.e. you can use linked directories, etc.).
- "Included" type test suites created for multi-file units at the test unit or global symbols level will not be imported

To import test suites from a C++test 6.x project:

1. Choose **New> Other** from the shortcut menu.A wizard will open.
2. Select **Import C++test 6.x test suites**, then click **Next**.
3. In the **C++test 6.x project file** field, enter the location of the C++test 6.x project file. This file should have a .cpf extension.

4. In the **Imported test suites location** field, choose the drop-down menu item for the import strategy you want to use (import all test suites into a single directory, or import the test suites for each C++test 6.x Test Unit into a separate subdirectory), then customize the entered text to specify the location where you want the test suites stored. When specifying the location, you can use the following variables:

- `${src_file_name}` - Name of the context file (a "context file" is a source file that describes the compilation unit in which the given tested function is defined).
- `${src_file_base_name}` - Name of context file without extension.
- `${src_file_ext}` - Extension of context file.
- `${src_file_loc_rel}` - Context file project relative location.
- `${src_file_uid}` - Context file unique identifier.
- `${unit_name}` - Imported test suite C++test 6.x test unit name.

5. Modify the following options if desired:

- **Automatically rename duplicated test suite names:** When enabled, C++test will automatically rename an imported test suite if its name conflicts with the name of another test suite in the target project.
- **Automatically rename duplicated test suite files:** When enabled, C++test will automatically rename an imported test suite file if it conflicts with another test suite file in the target project.

Note - Automated Renaming

If C++test renames a test suite or a test suite file, the appropriate information is added to the imported test suite file; for example:

```
// Test suite was renamed by C++test - original name: TestSuite_XX
// File was renamed by C++test - original file: TestSuite_XX.cpp
```

Information about renaming is also available in the C++test console; for example:

```
Importing test suite "TestSuite_XX" ...
...warning, test suite "TestSuite_XX" already exists in the project. Test suite renamed to
"TestSuite_XX_1".
```

6. Click **Finish**.

As test cases are imported, progress information will be displayed in the Console view. Progress information includes information about the imported test suites and problems encountered during the import, as well as an import summary.

During the import process, C++test checks test cases for obsolete constructors using __CPTR_Construct_Argument. If any obsolete constructor invocations are found, they will be changed into default constructors invocations. The test case will contain appropriate information, and the original code will be commented out. A warning message will be displayed in the C++test Console. The imported test suite file should be reviewed. If needed, the test cases should be modified to use different constructors.

Native Tests

Licensing Note

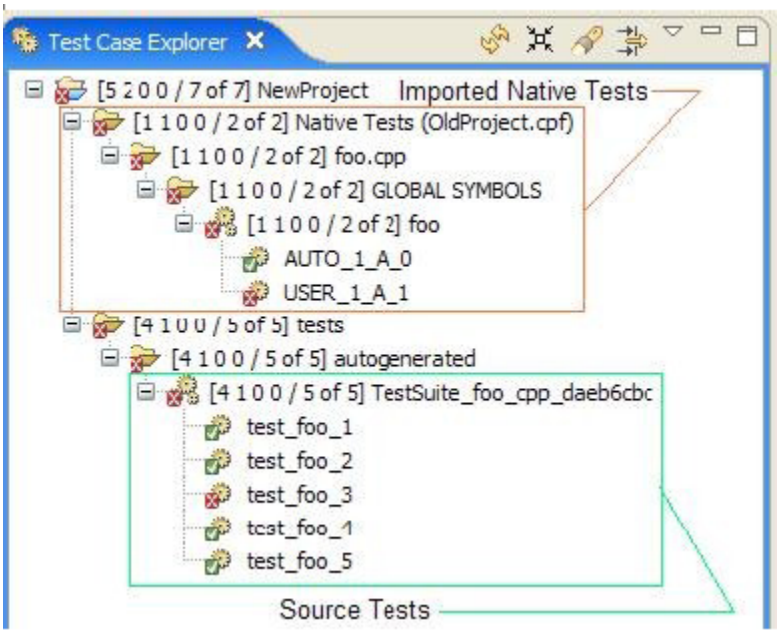
Importing C++test 2.3 / 6.x Native tests is available in C++test when the appropriate license feature is enabled. This license is required to enable all configuration items related to Native tests (for example, the **Add new> Native tests** action in the Test Case Explorer, options in the **Test Configurations** dialog, and Native tests project configuration in the C++test project properties).

To import native tests from C++test 6.x into C++test 10.x:

1. Create a new project that contains all of the source files that were originally tested by C++test 6.x. See Creating a Project for details.
2. Configure the project appropriately. See Setting Project and File Options for details.
3. Import native test cases as follows:
   a. Open the Test Case Explorer (Choose **Parasoft> Show View> Test Case Explorer**).
   b. Right-click the node for the project to which you want to add the tests, then choose **Add New> Native Tests** from the shortcut menu.
   c. In the file chooser that opens, specify the location of the C++test 6.x project file (.cpf file) that contains your native tests, then click **Open**.

C++test will automatically add the existing Native Tests to the Test Case Explorer view. They will be displayed according to the following hierarchy:

```
[NewProject]
[Native Tests (OldProject.cpf)]
  [File1.cpp]
    [Class1]
      [Function1]
        [TestCase1]
        [TestCase2]
```



Updating Tests

New test cases should be added using C++test 10.x functionality. If you need to modify a native test case, you can do so in C++test 6.x, then refresh the Test Case Explorer to synchronize.

Executing Tests

To run native tests in the current version of C++test:

1. If you have not already done so, create the appropriate Test Configuration as follows:
    a. Open the Test Configurations panel by choosing **Parasoft> Test Configurations**.
    b. Right-click the User-defined node and choose **Import**.
    c. Specify that you want to import the Test Configuration in `<INSTALL_DIR>/configs/NativeTests`.
    d. Set the `'CPPTEST_INSTALL_DIR'` environment variable to the C++test 2.3/6.x installation directory.
        - For example, use the export `CPPTEST_INSTALL_DIR=....` command in the bash shell or (on Windows) set up the system environment variable in **Control Panel> System> Advanced> Environment Variables**.
    e. In the Test Configuration's **Execution> General** tab, verify the command line parameters in **Command to run** and modify them if needed.

> **Modifying New or Existing Test Configurations to Run Native Tests**
>
> To configure a Test Configuration to run Native test:
>
> - Check **Enable test execution** in the **Execution** tab.
> - Check **Execute legacy native test cases** in the **Execution> General** tab.
> - In the command to run field (immediately below **Execute legacy native test cases**), provide proper path to the C++test 2.3/6.x executable.

2. In the Test Case Explorer, select the tests you want to execute.
3. Run the Test Configuration that you created for executing Native tests.

Notes:

- Native tests will be executed in their original environment (with stubs, data sources, testobjects etc.)
- C++test 10.x will automatically launch C++test 6.x in the command line mode.
- C++test 10.x will automatically load test and line coverage results.
- The original C++test 6.x project, as well as the C++test 6.x installation, are required in order to execute native tests in their original environment.
- Command line execution is also supported (all native tests connected to a given C++test 10.x project will be executed).

Reviewing Results

The results of native test execution will be presented the same way as for source tests:

- For execution details, use the Test Case Explorer. This shows:

    - Passed/failed status will be color-coding
    - Execution statistics
- For details on detected problems, see the Quality Tasks view. This shows:
    - Exceptions with stack traces
    - Expected value mismatches
- For coverage details, use the Coverage view. This shows:
    - Line coverage for native tests

You can also:

- Open the code editor to analyze covered/not covered lines
- Create reports
- Publish results to Team Server for team sharing
- Publish results to DTP (see Connecting to DTP)

Limitations

- Native tests can be executed based on a file, test suite, or project selection. Individual selection of native test for execution is not available.
- C++test provides line coverage data for test execution. Other coverage metrics are not available.

Stubs

There are two ways to import "user" stubs from C++test 6.x: import all user stubs into a single file, or import the user stubs for each C++test 6.x Test Unit into a separate file. If you import into a single file, the stub methods are sorted so that duplicated stubs can be easily found. If you import into separate files, the file names correspond to the C++test 6.x Test Unit names.

To import user stubs from a C++test 6.x project:

1. Choose **New> Other** from the shortcut menu. A wizard will open.

2. Select **Import C++test 6.x user stubs**, then click **Next**.

3. In the **C++test 6.x project file** field, enter the location of the C++test 6.x project file. This file should have a .cpf extension.

4. In the **Imported stubs location** field, enter the location where you want the stubs stored. You can use the drop-down menu to configure this value to import all stubs into a single directory, import all stubs for the given C++test 10.x Text Context into separate subdirectories, or import all stubs for the given C++test 6.x Text Unit into separate subdirectories.

Note - Reproducing C++test 6.x behavior in C++test 10.3 using File Scope testing mode

The **Import all stubs for the given C++test 10.x Text Context into separate subdirectories** option might be especially useful when trying to reproduce C++test 6.x behavior in C++test 10.x using File Scope testing mode. When importing stubs using the predefined layout, all stubs for a particular C++test 10.x Test Context will be placed in a single directory.

To ensure that the executed tests use the appropriate set of stubs in the File Scope testing mode:

1. Create a custom Test Configuration based on Run Unit Tests (File Scope).
2. Modify the **Execution> Symbols> Use extra symbols from files found in** setting to use only context-specific stubs, e.g.: `${project_loc}/stubs/imported/${ctx_name}`

This way, only stubs imported into the 'foo.cpp' subdirectory will be used when C++test is testing foo.cpp file in a File Scope mode.

5. In the **Duplicated Stubs Import strategy** area, use the available controls to define a strategy for importing several stubs for a single function (for example, if a stub for a given function is defined in the context of several C++test 6.x Test Units). Available options are:

- **Import all duplicated stubs:** Tells C++test to import each stub for a given function without performing any special actions.
- **Comment out duplicated stubs:** Tells C++test to import only one of the stubs for a given function. All other stubs for this function will be commented out during the import process.
- **Merge duplicated stubs (when importing to the same location):** If this is enabled and C++test (using **Imported stubs location**) determines that several stubs for the same function are to be added to a single file, C++test will try to merge all stubs into a single stub definition.

6. If you want C++test to modify the imported stub definition to ensure that the given stub will be used in the appropriate Test Context, enable **Insert code to connect stub definition with C++test 10.x Test Context**. For instance:

```
// [...]
if (CppTest_IsCurrentContext("examples/ATM/Source Files/Account.cxx")) {
// [...]
```

This option is especially useful with the **Merge duplicated stubs** setting. Here, C++test will create a single stub definition that will behave differently for each Test Context corresponding to the behavior of the C++test 6.x stubs. The default path of a stub will call the original definition of the stubbed function. For example:

```
double (::Account::CppTest_Stub_getBalance) ()
{
        if (CppTest_IsCurrentContext("/ATM/ATM/Account.cxx"))
        {
        //account
        }
        else if (CppTest_IsCurrentContext("/ATM/ATM/ATM.cxx"))
        {
        //atm
        }
        else
        {
                return getBalance();
        }
}
```

7. If you want C++test to add the specified text at the beginning of each imported stub file, enable **Insert custom code at beginning of each stub file**. For example, this can be used to add a common set of #include directives for required header files to all imported stub files.

8. If you want C++test to comment out duplicated stubs, enable **Comment out duplicated stubs**.

9. Click **Finish**.

As stubs are being imported, progress information will be displayed in the Console view. Progress information includes details about the stub files created, duplicate stubs identified, and conversion problems encountered, as well as an import summary.

C++test will automatically copy #include directives from the original source file (the file that the stub was created for in the C++test 6.x project) into the imported stub file.

Command Line (cpptestcli) Invocations

The following table shows the differences in command line usage between C++test 6.x and C++test 10.x. An x is used to indicate that no equivalent command is available.

C++test 6.x	C++test 10.x
`-Zicpf (file), --Zinput_cpf_project (file)`	`-data (WORKSPACE_DIR)`
`-Zocpf (file), --Zoutput_cpf_project (file)`	`x`
`-Zdsp (file), --Zdsp_project (file)`	`x`
`-Zvcproj (file), --Zvcproj_project (file)`	`x`
`-Zdc (name), --Zdsp_config (name)`	`x`
`-Zdc (name), --Zdsp_config (name)`	`x`
`-Zmcl (param), --Zmake_command_line (param)`	`x`
`-Ztf (name), --Ztest_file (name)`	`-resource (RESOURCE)`
`-Zeh (name), --Zexport_harness (name)`	`x`
`-Zrl (name), --Zread_logs (name)`	`x`
`-Ztc (name), --Ztest_config (name)`	`-config (CONFIG_URL)`
`-Zpc (name), --Zproject_config (name)`	`x`

-Zso (file), --Zsave_options (file)	x
-Zitc (file), --Zimport_test_cases (file)	x
-Zito (file), --Zimport_test_objects (file)	x
-Zis (file), --Zimport_suppressions (file)	x
-Zrf (file), --Zreport_file (file)	-report (REPORT_FILE)
-Zgh (name), --Zgenerate_html (name)	-report (REPORT_FILE)
-Zhrd (dir), --Zhtml_report_directory (dir)	-report (REPORT_FILE)
-Zgx, --Zgenerate_xml	-report (REPORT_FILE)
-Zxrd (dir), --Zxml_report_directory (dir)	-report (REPORT_FILE)
-Zpr, --Zpublish_results	-publish, -publishteamserver
-Zf, --Zforce	x
-Zow {on\off}, --Zoverwrite {on\off}	x
-Zoe [quiet], --Zonly_errors [quiet]	x
-Zq, --Zquiet	x
-Zvm(level), --Zverbosity_mode(level)	x
-Zgrs {on\off}, --Zgrs {on\off}	-localsettings (LOCALSETTINGS_FILE)
-Zga (name), --Zgrs_attribute (name)	-localsettings (LOCALSETTINGS_FILE)
-Zcs, --Zcompile_source	x
-Zrs, --Zreread_symbols	x
-Zlc (param), --Zlist_config (param)	x
-Zecf, --Zexpand_command_files	x
-Znt --Zno_tests	x
-h, --help	-help
-V, --version	-version
x	-nobuild
x	-showdetails
x	-appconsole stdout