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:

- About Stubs
- Viewing Stubs in the Stubs View
- Adding User-Defined Stubs to a Wizard-Generated Stub File
- Adding User-Defined Stubs to an Empty Stub File
- Generating Stubs for Symbols with Missing Definitions
- Understanding and Customizing Automatically-Generated Stubs
- Disabling Automatically-Generated Safe Stubs
- C++test API Functions for User-Defined Stubs
- Using Data Sources in Stubs
- Using Different Tests and/or Stubs for Different Contexts
- Using Stubs Driven By Test Cases
- Specifying Custom Compiler Options for User-Defined Stub Files

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.

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.

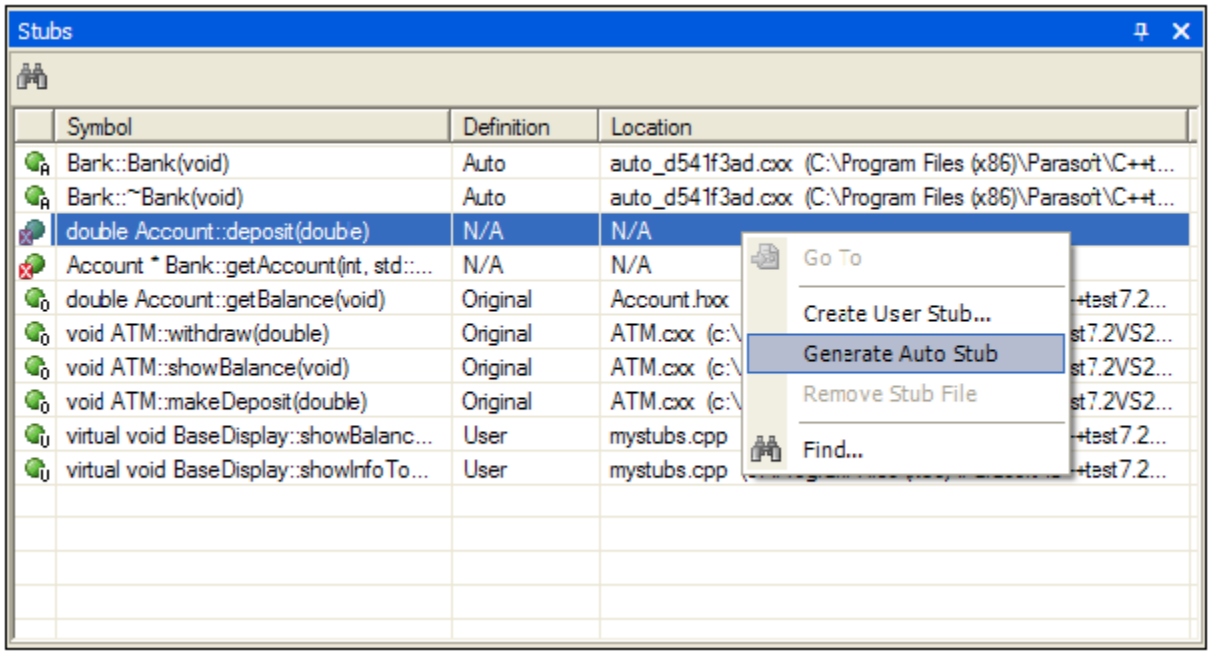
Accessing the Stubs View

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.

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,

      then choose **C++test> Wizard> Create New User Stub File**. A wizard will open.

      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 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 automat-ically 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.

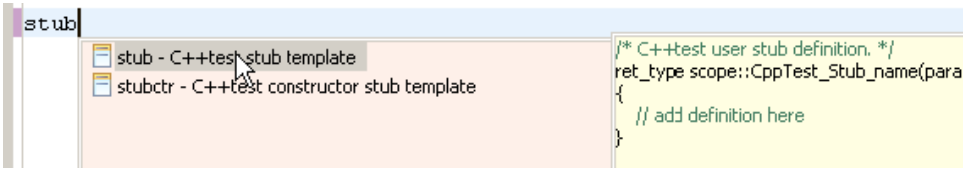
Adding User-Defined Stubs to an Empty Stub File

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,then choose **C++test> Wizard> Create New User Stub File**. A wizard will open.

      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).



   Create a stub template by right-clicking the code where you want it added, then choosing **Parasoft> C++test> Insert Snippet> [appropriate template_type]** (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.

- 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.

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
!=	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.

You can readily configure automatically-generated stubs either through the dynamic stubs configuration API (see Dynamic Stubs Configuration) or by using the Test Case Editor Stubs step (see Working with Steps).

In rare cases where the dynamic stubs configuration API or Test Case Editor is insufficient, you can completely replace the body of generated stubs with a custom logic implementation, such as the `CppTest_IsCurrentTestCase` stub function (see **"**C++test API Functions for User-Defined Stubs C++test API Functions for User-Defined Stubs).

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.
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.
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 Functions for Stubs Driven by Test Cases 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 Functions for Stubs Driven by Test Cases 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.

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.

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.