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

Using data sources in stub

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

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),, 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 thesolutionlevel).

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

Using a NULL reference instead of the NULL string

To use the NULL reference instead of the NULL string, select **$** from the **Special Value Prefix** field, then use the $ prefix before entering NULL in the CSV file; for example:
$NULL

Adding Existing Data Sources at the Solution Level

To create a managed data source at the solution level:

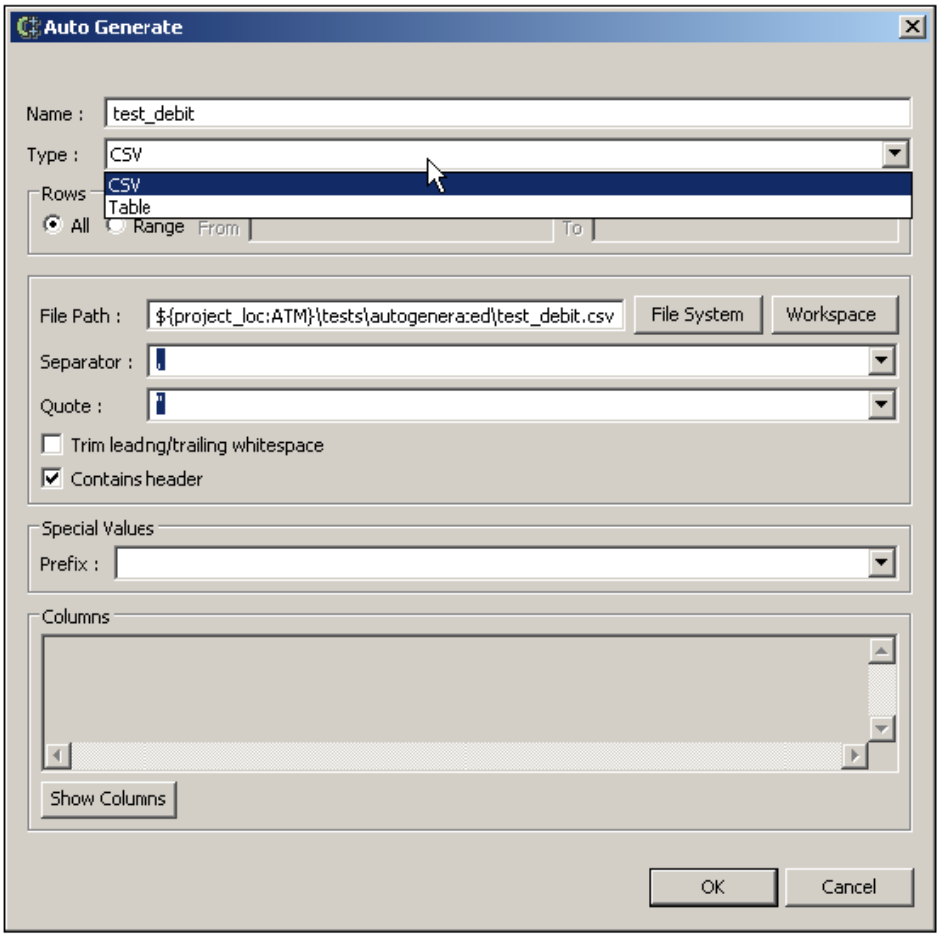
1. Right-click the **Global Data Sources** node in the Test Case Explorer view, then choose **Add New> Data Source**.

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

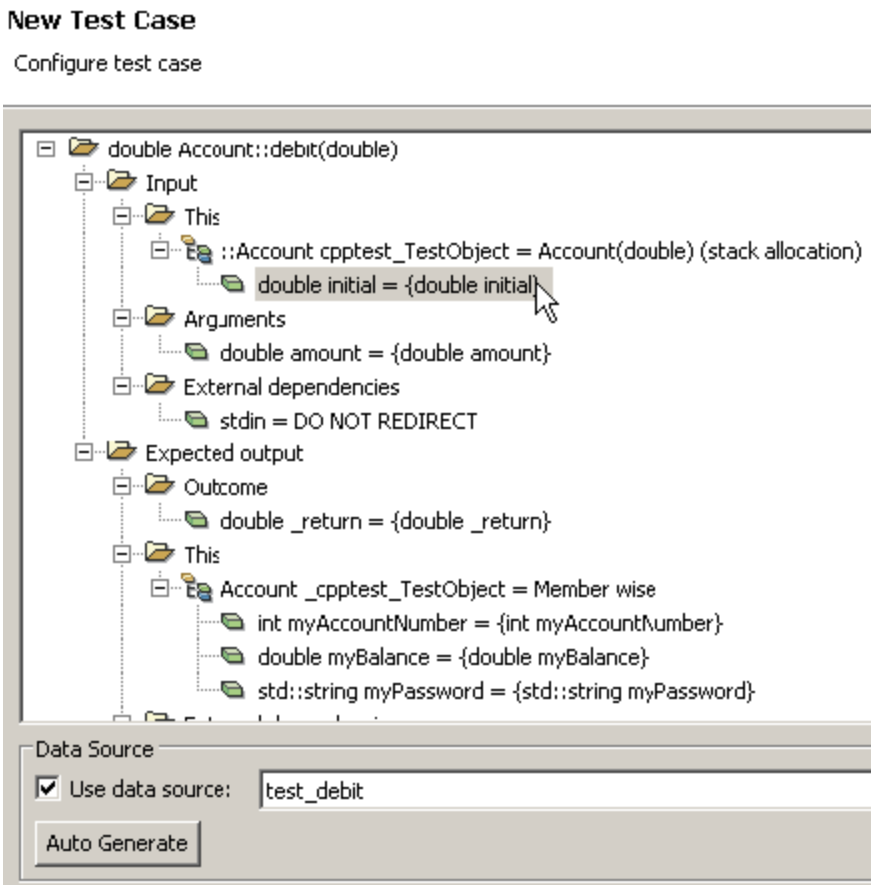
You can use data source values to parameterize existing (automatically-generated or user-defined) test cases, as well as to define test case values as you write test cases in the Test Case Wizard. Data source values can be taken from any data source that is defined in C++test as described above.

> **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 Caseusing 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).

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.
   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 a char value	`char CPPTEST_DS_GET_CHAR(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.
Return an enum value	`<scoped enum name> CPPTEST_DS_GET_ENUM (<scoped enum name>, const char* NAME)`	`<scoped enum name>` is a full name of enumeration including all namespace names. For example: INNER_NS::MyEnumeration, INNER_NS::MyClass::MyEnumeration
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.
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). 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
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

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

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).
enum value	`CPPTEST_DS_GET_ENUM()`	A simple enumerator name or a full name that includes all namespace names can be used. For example, if the MON enumerator is defined in ::INNER_NS::MyClass::DaysEnumeration, you can type either MON or ::INNER_NS::MyClass::MON. To use enumerators in Data Sources and Parameters step, enable Test Configuration's option: "Enable enum data autogeneration". You can find this option in: "Test Configuration > Execution tab > General tab > Instrumentation mode > Edit > Instrumentation features > Advanced options".

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\""!"

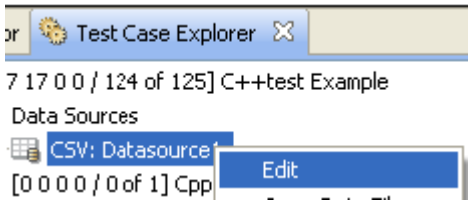
Excel spreadsheet (managed data source)	`CPPTEST_DS()`	No Excel-specific value preparation is needed. Note that the data needs to be formatted to be used for particular types as described above. To make sure that C++test processes values properly, you can use Text format for all cells in the spreadsheet. 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.
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\""!"
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.

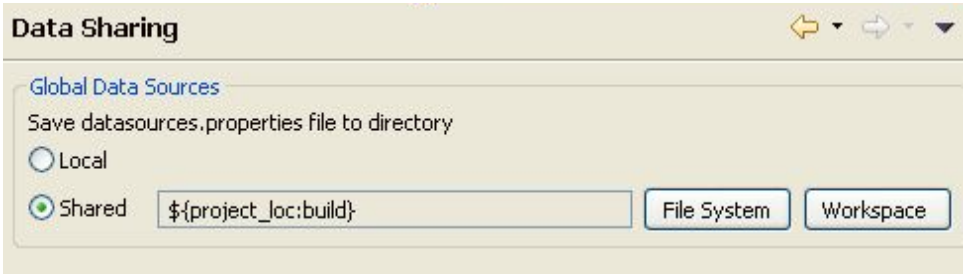
Sharing Data Sources

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 thesolutionlevel—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:

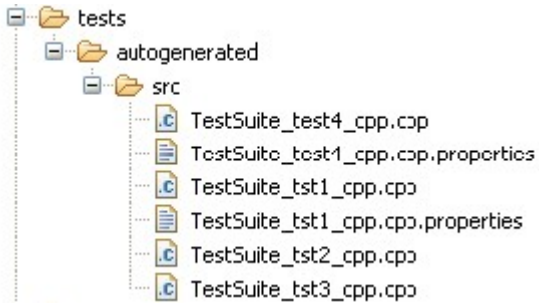
1. Choose **Parasoft> Preferences**. The Preferences dialog will open.
2. In the left pane, choose **Team> Data Sharing**.
3. Specify where you want the datasources.properties file saved.



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.

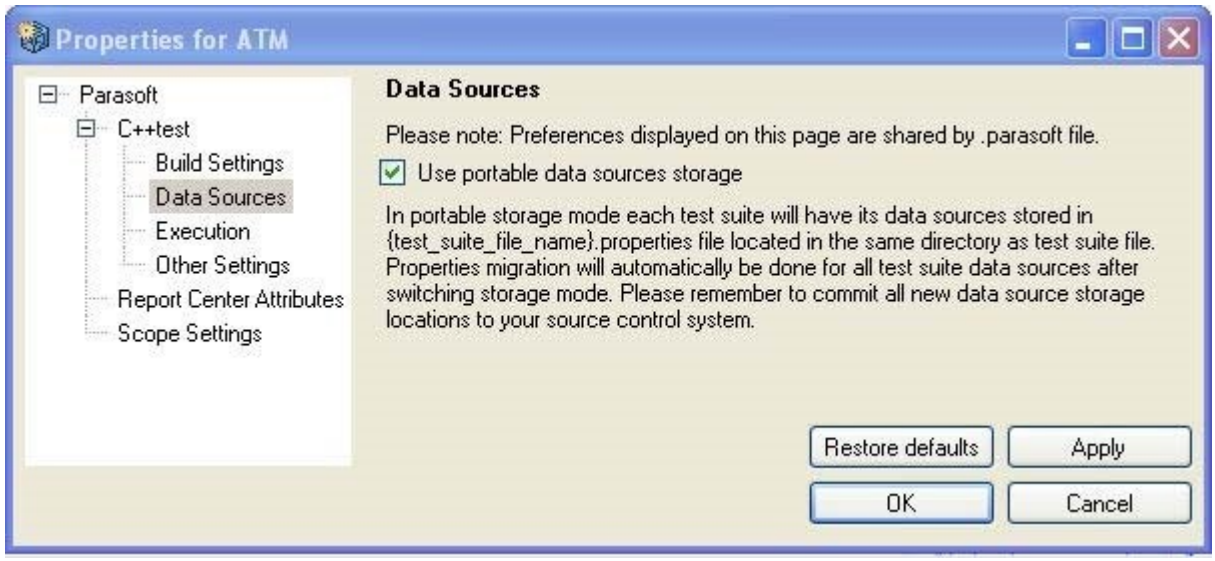


Portable data source mode applies only to data sources created at test suite level—it does not apply to data sources created at the project orsolutionlevel.

Changing Data Source Modes

To change data source settings:

1. Right-click the project node and choose **Parasoft>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, dataSourceCon-fig)` 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.

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;
}
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 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 dou-ble-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 **C++test> Wizards> Create New Test Suite**. 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.getBal-ance(), 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.