Test Support Kit

RANDALL MAAS

This memo describes the test support module:

- Overview of the test support module
- The programming interface to the test support module
- Detailed design of the test support procedures

1. OVERVIEW

The test support module is intended to support *test-related build configurations*. (These are intended to not be included in the Debug and Release builds. Those builds are for functioning product, albeit with different degrees of debugging related support. They should never, ever have any asserts.) The test build configuration is used to run test cases of the software procedures. It may be quite slow and usually does not run the main application.

2. SOFTWARE INTERFACE SPECIFICATION

This section describes how software would employ the test module. Each of the test asserts has very simple behaviour, the pattern is given below:

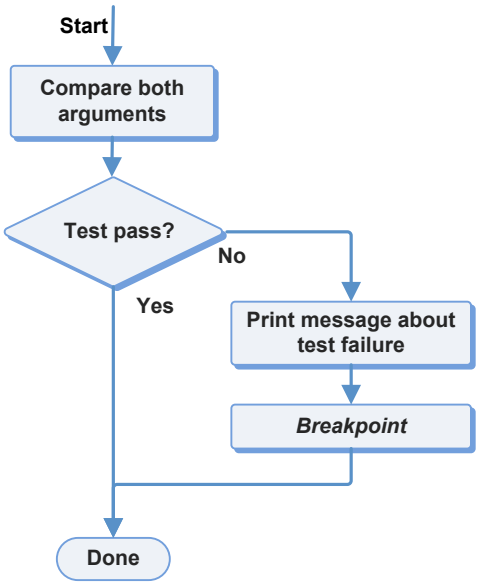

Figure 1: *Overview of the test*

Note: A message is displayed if the test failed, but not it if it passed. It was found that tests in loops (such as used when testing collections) produced too many messages.

2.1. EXAMPLES

Procedures that perform tests can (and should) be declared using the testCase() macro. The test procedures can be anonymous like so:

```
testCase()
{
  …
}

testCase()
{
  …
}
```

Or given a name, such as “accessTests1”:

```
testCase(accessTests1)
{
  …
}

testCase(accessTests2)
{
  …
}
```

The tests can be run by calling the test runner, which will call each of the tests declared with testCase():

```
testsRun();
```

Or, if preferable, by calling the named tests:

```
accessTests1();
```

2.2. DEFINES

The table below describes the macros provided to use the test module:

Table 1: *Test Support Defines*

Macro	Description
testCast(…)	Declares a test procedure to call
TstEquals(expr1,expr2,,...)	Asserts that two expressions have the same value (with tolerance for floating points) or are equal objects.
TstEquivalent(expr1,expr2,,...)	Asserts that two expressions have the exact same value or are the same object.
TstFail(...)	Generates a failure immediately.
TstFalse(expr,...)	Asserts that an expression is true. kinds
TstGreaterThan(expr1,expr2,,...)	Asserts that the value of first expression is greater than the other.
TstGreaterThanOrEqual(expr1,expr2,,...)	Asserts that the value of first expression is greater than or equal to the other.
TstLessThan(expr1,expr2,,...)	Asserts that the value of first expression is less than the other.
TstLessThanOrEqual(expr1,expr2,,...)	Asserts that the value of first expression is less than or equal to the other.
TstNull(expr,...)	Asserts that an expression is null.
TstNotEquals(expr1,expr2,,...)	Asserts that two expressions do not have the same value and not equal objects.
TstNotEquivalent(expr1,expr2,,...)	Asserts that two expressions do not have the exact same value or are not the same object.
TstNotNull(expr,...)	Asserts that an expression is not null.
TstTrue(expr,...)	Asserts that an expression is true.

An optional message for the test can be passed to be displayed if the test fails.

2.3. PROCEDURES: SYNOPSIS

The table below describes the test support modules procedure interface:

Table 2: *Test support procedures*

Procedure	Description
testsRun()	Runs the tests

2.4. PROCEDURE INTERFACE

void testsRun()

This procedure runs the tests.

Parameters:

none

Returns:

none

The procedures is only valid (and useful) on platforms that provide access to a section/segment to store references to each of the test procedures.

3. DETAILED DESIGN

This section describes the detailed above macros and procedures used within the test support module.

3.1. IMPLEMENTATION OF THE TEST MACROS

The test interface (described earlier) is written using macros to wrap an internal procedure. The interface was implemented this way for 2 reasons:

- To gather the location in the code that the test failed;
- To trigger the breakpoint at the correct location (i.e., in the code that called the test)

The macros are written in two levels, to gather debugging related information:

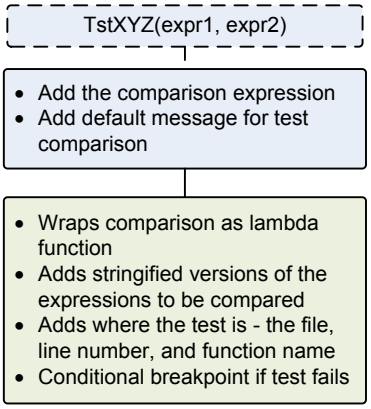

Figure 2: *Overview of the test interface macros*

Each test interface macro (usually) has a form similar to:

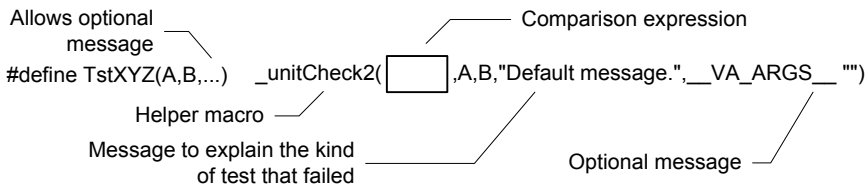

Figure 3: *Generic test assertion, using a helper*

The interface macro provides the test comparison expression, and default test message. Then it punts the work to a helper macro, which looks like:

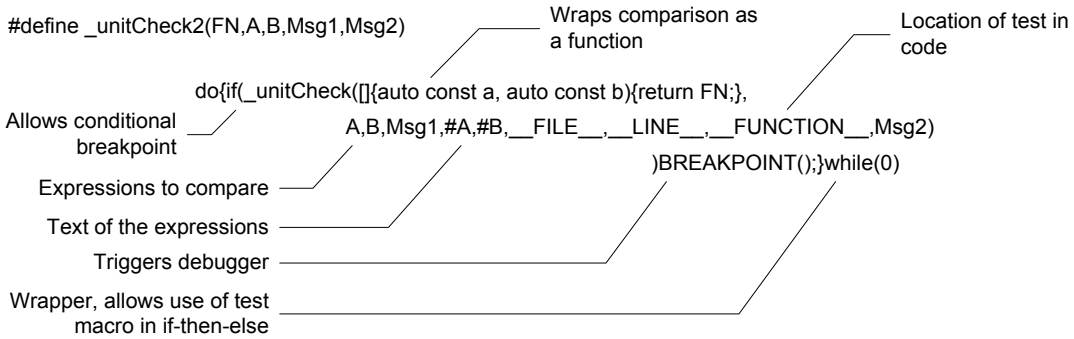

Figure 4: *Further details of the test assertion helper*

The helper macro wraps the comparison expression in C++11 lambda boiler plate, adds in the file, line, and function name of the test procedure, as well as the text form of the two expressions being compared.

3.2. IMPLEMENTATION OF THE TEST AUTO-RUNNER

The test auto-runner calls each of the test procedures declared by the testCase() macro. The macro defines pointers, in a special segment, to each test procedure. The test runner accesses the segment to get the list of pointers, and calls each.

The testCase() macro does this as follows:[1]

1. The macro declares the procedure, with the passed name; if a name was not passed, it creates one from a test prefix and current line number. The procedure is declared static (if the test runner is supported) to prevent name collisions that are possible with the automatic name.
2. Defines a variable that is initialized to point to the procedure. The variable name is made by pasting a "_" onto the test procedure name. The variable is placed in the test segment, and made static to prevent name collisions.
3. Then procedure declaration is made so that the body following the testCase() macro will define the test procedure.

3.3. DEFINES

The table below describes the macros provided to use internally to the test support module:

Table 3: *Test Support Internal Defines*

Macro	Description
_ErrMsgHdr	The format for printing where in the code the test occurred. (The format varies slightly with IDE environment)
_ErrPrint (,...)	Macro to print a message to the debug console or standard-error.

3.4. PROCEDURES: SYNOPSIS

The table below describes the procedures used internally, within the test support module.

Table 4: *Internal Procedures*

Procedure	Description
_errorValue()	Prints the passed value (or object)
_unitCheck()	A helper to perform a test and report the results.
_unitReport ()	A helper to report on the test results.

[1] Ideally, we would have a define that just marked the tests, placing a pointer into the appropriate segment. However this was not sufficiently portable. Instead the testCase() macro is used to automatically name the test procedure(s).

3.5. PROCEDURE INTERFACE

void _errorValue (typename *a*)

This procedure is used to print the value in the appropriate format.

Parameters:

a The value or object to print

Returns:

none

The procedures to display value share the same name; C++ type selection is used to select correct one.

int _unitCheck (typename *fn*, typename *a*, typename *b*, char const* *errMsg*, const char* *a_str*, const char* *b_str*, const char**file*, int64_t *line*, char const* *funcName*, char const* *message*)

A helper to perform a test and report the results.

Parameters:

fn	The function used to compare the *a* and *b* values.
a	A value or object to compare.
b	A value or object to compare.
errMsg	The message to display describing what type of comparison failed.
a_str	The text of the expression used for the *a* value.
b_str	The text of the expression used for the *b* value.
file	The name of the file with the test case.
line	The line within the file where the test assertion was invoked.
funcName	The name of the function in which the test case was invoked.
message	The optional test-case supplied message to display on failure.

Returns:

0	the test passed
1	the test failed

int _unitCheck (typename *fn*, typename *value*, char const* *errMsg*, const char* *a_str*, const char* *b_str*, const char**file*, int64_t *line*, char const* *funcName*, char const* *message*)

A helper to perform a test and report the results.

Parameters:

fn	The function used to check the *value*.
value	A value or object to check.
errMsg	The message to display describing the type of comparison that failed.
a_str	The text of the expression used for the *value*.
b_str	The text of the expression used to check the *value* against.
file	The name of the file with the test case procedure.
line	The line within the file where the test assertion was invoked.
funcName	The name of the function in which the test case was invoked.
message	The optional test-case supplied message to display on failure.

Returns:

0	the test passed
1	the test failed

int _unitReport (typename *value*, char const* *errMsg*, const char* *a_str*, const char* *b_str*, const char**file*, int64_t *line*, char const* *funcName*, char const* *message*)

A helper to report on the test results.

Parameters:

value	A value or object to check.
errMsg	The message to display describing the type of comparison that failed.
a_str	The text of the expression used for the *value*.
b_str	The text of the expression used to check the *value* against.
file	The name of the file with the test case procedure.
line	The line within the file where the test assertion was invoked.
funcName	The name of the function in which the test case was invoked.
message	The optional test-case supplied message to display on failure.

Returns:

none

3.6. FILES

The table below describes the files employed in the test support module:

Table 5: *Test support files*

File	Description
osx-Debugger.c	A module to detect the presence of the debugger interacting with the application.
osx- testsRun.cpp	The implementation of the test auto-runner on OS X.
test.h	The header / declarations for the test support module.
win-testsRun.cpp	The implementation of the test auto-runner on Windows.