Unity Assertions Reference

Background and Overview

Super Condensed Version

- An assertion establishes truth (i.e. boolean True) for a single condition. Upon boolean False, an assertion stops execution and reports the failure.
- Unity is mainly a rich collection of assertions and the support to gather up and easily execute those assertions.
- The structure of Unity allows you to easily separate test assertions from source code in, well, test code.
- Unity's assertions:
    - Come in many, many flavors to handle different C types and assertion cases.
    - Use context to provide detailed and helpful failure messages.
    - Document types, expected values, and basic behavior in your source code for free.

Unity Is Several Things But Mainly It's Assertions

One way to think of Unity is simply as a rich collection of assertions you can use to establish whether your source code behaves the way you think it does. Unity provides a framework to easily organize and execute those assertions in test code separate from your source code.

What's an Assertion?

At their core, assertions are an establishment of truth—boolean truth. Was this thing equal to that thing? Does that code doohickey have such-and-such property or not? You get the idea. Assertions are executable code (to appreciate the big picture on this read up on the difference between [link:Dynamic Verification and Static Analysis]). A failing assertion stops execution and reports an error through some appropriate I/O channel (e.g. stdout, GUI, file, blinky light).

Fundamentally, for dynamic verification all you need is a single assertion mechanism. In fact, that's what the `assert()` macro in C's standard library is for. So why not just use it? Well, we can do far better in the reporting department. C's `assert()` is pretty dumb as-is and is particularly poor for handling common data types like arrays, structs, etc. And, without some other support, it's far too tempting to litter source code with C's `assert()`'s. It's generally much cleaner, manageable, and more useful to separate test and source code in the way Unity facilitates.

Unity's Assertions: Helpful Messages *and* Free Source Code Documentation

Asserting a simple truth condition is valuable, but using the context of the assertion is even more valuable. For instance, if you know you're comparing bit flags and not just integers, then why not use that context to give explicit, readable, bit-level feedback when an assertion fails?

That's what Unity's collection of assertions do—capture context to give you helpful, meaningful assertion failure messages. In fact, the assertions themselves also serve as executable documentation about types and values in your source code. So long as your tests remain current with your source and all those tests pass, you have a detailed, up-to-date view of the intent and mechanisms in your source code. And due to a wondrous mystery, well-tested code usually tends to be well designed code.

Assertion Conventions and Configurations

Naming and Parameter Conventions

The convention of assertion parameters generally follows this order:
`TEST_ASSERT_X( {modifiers}, {expected}, actual, {size/count} )`

The very simplest assertion possible uses only a single "actual" parameter (e.g. a simple null check).

"Actual" is the value being tested and unlike the other parameters in an assertion construction is the only parameter present in all assertion variants.
"Modifiers" are masks, ranges, bit flag specifiers, floating point deltas.
"Expected" is your expected value (duh) to compare to an "actual" value; it's marked as an optional parameter because some assertions only need a single "actual" parameter (e.g. null check).
"Size/count" refers to string lengths, number of array elements, etc.

Many of Unity's assertions are apparent duplications in that the same data type is handled by several assertions. The differences among these are in how failure messages are presented. For instance, a `_HEX` variant of an assertion prints the expected and actual values of that assertion formatted as hexadecimal.

TEST_ASSERT_X_MESSAGE Variants

All assertions are complemented with a variant that includes a simple string message as a final parameter. The string you specify is appended to an assertion failure message in Unity output.

For brevity, the assertion variants with a message parameter are not listed below. Just tack on `_MESSAGE` as the final component to any assertion name in the reference list below and add a string as the final parameter.

Example:
`TEST_ASSERT_X( {modifiers}, {expected}, actual, {size/count} )`

becomes message-ified like thus…

```
TEST_ASSERT_X_MESSAGE( {modifiers}, {expected}, actual, {size/count},
message )
```

TEST_ASSERT_X_ARRAY Variants

Unity provides a collection of assertions for arrays containing a variety of types. These are documented in the Array section below. These are almost on par with the `_MESSAGE` variants of Unity's Asserts in that for pretty much any Unity type assertion you can tack on `_ARRAY` and run assertions on an entire block of memory.

```
TEST_ASSERT_EQUAL_TYPEX_ARRAY( expected, actual, {size/count} )
```

"Expected" is an array itself.
"Size/count" is one or two parameters necessary to establish the number of array elements and perhaps the length of elements within the array.

Notes:

- The `_MESSAGE` variant convention still applies here to array assertions. The `_MESSAGE` variants of the `_ARRAY` assertions have names ending with `_ARRAY_MESSAGE`.
- Assertions for handling arrays of floating point values are grouped with float and double assertions (see immediately following section).

Configuration

Floating Point Support Is Optional

Support for floating point types is configurable. That is, by defining the appropriate preprocessor symbols, floats and doubles can be individually enabled or disabled in Unity code. This is useful for embedded targets with no floating point math support (i.e. Unity compiles free of errors for fixed point only platforms). See Unity documentation for specifics.

Maximum Data Type Width Is Configurable

Not all targets support 64 bit wide types or even 32 bit wide types. Define the appropriate preprocessor symbols and Unity will omit all operations from compilation that exceed the maximum width of your target. See Unity documentation for specifics.

The Assertions in All Their Blessed Glory

Basic Fail and Ignore

`TEST_FAIL()`

This fella is most often used in special conditions where your test code is performing logic beyond a simple assertion. That is, in practice, `TEST_FAIL()` will always be found inside a conditional code block.

Examples:

- Executing a state machine multiple times that increments a counter your test code then verifies as a final step.
- Triggering an exception and verifying it (as in Try / Catch / Throw — see the CException project).

`TEST_IGNORE()`

Marks a test case (i.e. function meant to contain test assertions) as ignored. Usually this is employed as a breadcrumb to come back and implement a test case. An ignored test case has effects if other assertions are in the enclosing test case (see Unity documentation for more).

Boolean

```
TEST_ASSERT (condition)
TEST_ASSERT_TRUE (condition)
TEST_ASSERT_FALSE (condition)
TEST_ASSERT_UNLESS (condition)
```

A simple wording variation on `TEST_ASSERT_FALSE`. The semantics of `TEST_ASSERT_UNLESS` aid readability in certain test constructions or conditional statements.

```
TEST_ASSERT_NULL (pointer)
TEST_ASSERT_NOT_NULL (pointer)
```

Signed and Unsigned Integers (of all sizes)

Large integer sizes can be disabled for build targets that do not support them. For example, if your target only supports up to 16 bit types, by defining the appropriate symbols Unity can be configured to omit 32 and 64 bit operations that would break compilation (see Unity documentation for more). Refer to Advanced Asserting later in this document for advice on dealing with other word sizes.

```
TEST_ASSERT_EQUAL_INT (expected, actual)
TEST_ASSERT_EQUAL_INT8 (expected, actual)
TEST_ASSERT_EQUAL_INT16 (expected, actual)
TEST_ASSERT_EQUAL_INT32 (expected, actual)
TEST_ASSERT_EQUAL_INT64 (expected, actual)
TEST_ASSERT_EQUAL (expected, actual)
TEST_ASSERT_NOT_EQUAL (expected, actual)
TEST_ASSERT_EQUAL_UINT (expected, actual)
TEST_ASSERT_EQUAL_UINT8 (expected, actual)
TEST_ASSERT_EQUAL_UINT16 (expected, actual)
TEST_ASSERT_EQUAL_UINT32 (expected, actual)
TEST_ASSERT_EQUAL_UINT64 (expected, actual)
```

Unsigned Integers (of all sizes) in Hexadecimal

All `_HEX` assertions are identical in function to unsigned integer assertions but produce failure messages with the `expected` and `actual` values formatted in hexadecimal. Unity output is big endian.

```
TEST_ASSERT_EQUAL_HEX (expected, actual)
TEST_ASSERT_EQUAL_HEX8 (expected, actual)
TEST_ASSERT_EQUAL_HEX16 (expected, actual)
TEST_ASSERT_EQUAL_HEX32 (expected, actual)
TEST_ASSERT_EQUAL_HEX64 (expected, actual)
```

Masked and Bit-level Assertions

Masked and bit-level assertions produce output formatted in hexadecimal. Unity output is big endian.

`TEST_ASSERT_BITS (mask, expected, actual)`
: Only compares the masked (i.e. high) bits of `expected` and `actual` parameters.

`TEST_ASSERT_BITS_HIGH (mask, actual)`
: Asserts the masked bits of the `actual` parameter are high.

`TEST_ASSERT_BITS_LOW (mask, actual)`
: Asserts the masked bits of the `actual` parameter are low.

`TEST_ASSERT_BIT_HIGH (bit, actual)`
: Asserts the specified bit of the `actual` parameter is high.

`TEST_ASSERT_BIT_LOW (bit, actual)`
: Asserts the specified bit of the `actual` parameter is low.

Integer Ranges (of all sizes)

These assertions verify that the `expected` parameter is within +/- `delta` (inclusive) of the `actual` parameter. For example, if the expected value is 10 and the delta is 3 then the assertion will fail for any value outside the range of 7–13.

```
TEST_ASSERT_INT_WITHIN (delta, expected, actual)
TEST_ASSERT_INT8_WITHIN (delta, expected, actual)
TEST_ASSERT_INT16_WITHIN (delta, expected, actual)
TEST_ASSERT_INT32_WITHIN (delta, expected, actual)
TEST_ASSERT_INT64_WITHIN (delta, expected, actual)
TEST_ASSERT_UINT_WITHIN (delta, expected, actual)
TEST_ASSERT_UINT8_WITHIN (delta, expected, actual)
TEST_ASSERT_UINT16_WITHIN (delta, expected, actual)
TEST_ASSERT_UINT32_WITHIN (delta, expected, actual)
TEST_ASSERT_UINT64_WITHIN (delta, expected, actual)
TEST_ASSERT_HEX_WITHIN (delta, expected, actual)
TEST_ASSERT_HEX8_WITHIN (delta, expected, actual)
TEST_ASSERT_HEX16_WITHIN (delta, expected, actual)
TEST_ASSERT_HEX32_WITHIN (delta, expected, actual)
TEST_ASSERT_HEX64_WITHIN (delta, expected, actual)
```

Structs and Strings

`TEST_ASSERT_EQUAL_PTR (expected, actual)`
Asserts that the pointers point to the same memory location.

`TEST_ASSERT_EQUAL_STRING (expected, actual)`
Asserts that the null terminated ('`\0`') strings are identical. If strings are of different lengths or any portion of the strings before their terminators differ, the assertion fails. Two NULL strings (i.e. zero length) are considered equivalent.

`TEST_ASSERT_EQUAL_MEMORY (expected, actual, len)`
Asserts that the contents of the memory specified by the `expected` and `actual` pointers is identical. The size of the memory blocks in bytes is specified by the `len` parameter.

Arrays

`expected` and `actual` parameters are both arrays. `num_elements` specifies the number of elements in the arrays to compare.

`_HEX` assertions produce failure messages with expected and actual array contents formatted in hexadecimal.

For array of strings comparison behavior, see comments for `TEST_ASSERT_EQUAL_STRING` in the preceding section.

Assertions fail upon the first element in the compared arrays found not to match. Failure messages specify the array index of the failed comparison.

```
TEST_ASSERT_EQUAL_INT_ARRAY (expected, actual, num_elements)
TEST_ASSERT_EQUAL_INT8_ARRAY (expected, actual, num_elements)
TEST_ASSERT_EQUAL_INT16_ARRAY (expected, actual, num_elements)
TEST_ASSERT_EQUAL_INT32_ARRAY (expected, actual, num_elements)
TEST_ASSERT_EQUAL_INT64_ARRAY (expected, actual, num_elements)
TEST_ASSERT_EQUAL_UINT_ARRAY (expected, actual, num_elements)
TEST_ASSERT_EQUAL_UINT8_ARRAY (expected, actual, num_elements)
TEST_ASSERT_EQUAL_UINT16_ARRAY (expected, actual, num_elements)
TEST_ASSERT_EQUAL_UINT32_ARRAY (expected, actual, num_elements)
TEST_ASSERT_EQUAL_UINT64_ARRAY (expected, actual, num_elements)
TEST_ASSERT_EQUAL_HEX_ARRAY (expected, actual, num_elements)
```

`TEST_ASSERT_EQUAL_HEX8_ARRAY (expected, actual, num_elements)`
`TEST_ASSERT_EQUAL_HEX16_ARRAY (expected, actual, num_elements)`
`TEST_ASSERT_EQUAL_HEX32_ARRAY (expected, actual, num_elements)`
`TEST_ASSERT_EQUAL_HEX64_ARRAY (expected, actual, num_elements)`
`TEST_ASSERT_EQUAL_PTR_ARRAY (expected, actual, num_elements)`
`TEST_ASSERT_EQUAL_STRING_ARRAY (expected, actual, num_elements)`
`TEST_ASSERT_EQUAL_MEMORY_ARRAY (expected, actual, len, num_elements)`

`len` is the memory in bytes to be compared at each array element.

Floating Point (If enabled)

`TEST_ASSERT_FLOAT_WITHIN (delta, expected, actual)`

Asserts that the `actual` value is within +/- `delta` of the `expected` value. The nature of floating point representation is such that exact evaluations of equality are not guaranteed.

`TEST_ASSERT_EQUAL_FLOAT (expected, actual)`

Asserts that the `actual` value is “close enough to be considered equal” to the `expected` value. If you are curious about the details, refer to the Advanced Asserting section for more details on this. Omitting a user-specified delta in a floating point assertion is both a shorthand convenience and a requirement of code generation conventions for CMock.

`TEST_ASSERT_EQUAL_FLOAT_ARRAY (expected, actual, num_elements)`

See Array assertion section for details. Note that individual array element float comparisons are executed using `TEST_ASSERT_EQUAL_FLOAT`. That is, user specified delta comparison values requires a custom-implemented floating point array assertion.

`TEST_ASSERT_FLOAT_IS_INF (actual)`

Asserts that `actual` parameter is equivalent to positive infinity floating point representation.

`TEST_ASSERT_FLOAT_IS_NEG_INF (actual)`

Asserts that `actual` parameter is equivalent to negative infinity floating point representation.

`TEST_ASSERT_FLOAT_IS_NAN (actual)`

Asserts that `actual` parameter is a Not A Number floating point representation.

`TEST_ASSERT_FLOAT_IS_DETERMINATE (actual)`

Asserts that `actual` parameter is a floating point representation usable for mathematical operations. That is, the `actual` parameter is neither positive infinity nor negative infinity nor Not A Number floating point representations.

`TEST_ASSERT_FLOAT_IS_NOT_INF (actual)`

Asserts that `actual` parameter is a value other than positive infinity floating point representation.

`TEST_ASSERT_FLOAT_IS_NOT_NEG_INF (actual)`

Asserts that `actual` parameter is a value other than negative infinity floating point representation.

`TEST_ASSERT_FLOAT_IS_NOT_NAN (actual)`

Asserts that `actual` parameter is a value other than Not A Number floating point representation.

`TEST_ASSERT_FLOAT_IS_NOT_DETERMINATE (actual)`

Asserts that `actual` parameter is not usable for mathematical operations. That is, the `actual` parameter is either positive infinity or negative infinity or Not A Number floating point representations.

Double (If enabled)

`TEST_ASSERT_DOUBLE_WITHIN (delta, expected, actual)`

Asserts that the `actual` value is within +/- `delta` of the `expected` value. The nature of floating point representation is such that exact evaluations of equality are not guaranteed.

`TEST_ASSERT_EQUAL_DOUBLE (expected, actual)`

Asserts that the `actual` value is “close enough to be considered equal” to the `expected` value. If you are curious about the details, refer to the Advanced Asserting section for more details. Omitting a user-specified delta in a floating point assertion is both a shorthand convenience and a requirement of code generation conventions for CMock.

`TEST_ASSERT_EQUAL_DOUBLE_ARRAY (expected, actual, num_elements)`

See Array assertion section for details. Note that individual array element double comparisons are executed using `TEST_ASSERT_EQUAL_DOUBLE`. That is, user

specified delta comparison values requires a custom-implemented double array assertion.

`TEST_ASSERT_DOUBLE_IS_INF (actual)`
: Asserts that `actual` parameter is equivalent to positive infinity floating point representation.

`TEST_ASSERT_DOUBLE_IS_NEG_INF (actual)`
: Asserts that `actual` parameter is equivalent to negative infinity floating point representation.

`TEST_ASSERT_DOUBLE_IS_NAN (actual)`
: Asserts that `actual` parameter is a Not A Number floating point representation.

`TEST_ASSERT_DOUBLE_IS_DETERMINATE (actual)`
: Asserts that `actual` parameter is a floating point representation usable for mathematical operations. That is, the `actual` parameter is neither positive infinity nor negative infinity nor Not A Number floating point representations.

`TEST_ASSERT_DOUBLE_IS_NOT_INF (actual)`
: Asserts that `actual` parameter is a value other than positive infinity floating point representation.

`TEST_ASSERT_DOUBLE_IS_NOT_NEG_INF (actual)`
: Asserts that `actual` parameter is a value other than negative infinity floating point representation.

`TEST_ASSERT_DOUBLE_IS_NOT_NAN (actual)`
: Asserts that `actual` parameter is a value other than Not A Number floating point representation.

`TEST_ASSERT_DOUBLE_IS_NOT_DETERMINATE (actual)`
: Asserts that `actual` parameter is not usable for mathematical operations. That is, the `actual` parameter is either positive infinity or negative infinity or Not A Number floating point representations.

Advanced Asserting: Details On Tricky Assertions

This section helps you understand how to deal with some of the trickier assertion situations you may run into. It will give you a glimpse into some of the under-the-hood details of Unity's assertion mechanisms. If you're one of those people who likes to know what is going on in the background, read on. If not, feel free to ignore the rest of this document until you need it.

How do the EQUAL assertions work for FLOAT and DOUBLE?

As you may know, directly checking for equality between a pair of floats or a pair of doubles is sloppy at best and an outright no-no at worst. Floating point values can often be represented in multiple ways, particularly after a series of operations on a value. Initializing a variable to the value of 2.0 is likely to result in a floating point representation of 2×2^0, but a series of mathematical operations might result in a representation of 8×2^{-2} that also evaluates to a value of 2. At some point repeated operations cause equality checks to fail.

So Unity doesn't do direct floating point comparisons for equality. Instead, it checks if two floating point values are "really close." If you leave Unity running with defaults, "really close" means "within a significant bit or two." Under the hood, `TEST_ASSERT_EQUAL_FLOAT` is really `TEST_ASSERT_FLOAT_WITHIN` with the `delta` parameter calculated on the fly. For single precision, delta is the expected value multiplied by 0.00001, producing a very small proportional range around the expected value.

If you are expecting a value of 20,000.0 the delta is calculated to be 0.2. So any value between 19,999.8 and 20,000.2 will satisfy the equality check. This works out to be roughly a single bit of range for a single-precision number, and that's just about as tight a tolerance as you can reasonably get from a floating point value.

So what happens when it's zero? Zero—even more than other floating point values—can be represented many different ways. It doesn't matter if you have 0×2^0 or 0×2^{63}. It's still zero, right? Luckily, if you subtract these values from each other, they will always produce a difference of zero, which will still fall between 0 plus or minus a delta of 0. So it still works!

Double precision floating point numbers use a much smaller multiplier, again approximating a single bit of error.

If you don't like these ranges and you want to make your floating point equality assertions less strict, you can change these multipliers to whatever you like by defining UNITY_FLOAT_PRECISION and UNITY_DOUBLE_PRECISION. See Unity documentation for more.

How do we deal with targets with non-standard int sizes?

It's "fun" that C is a standard where something as fundamental as an integer varies by target. According to the C standard, an `int` is to be the target's natural register size, and it should be at least 16-bits and a multiple of a byte. It also guarantees an order of sizes:

```
char <= short <= int <= long <= long long
```

Most often, `int` is 32-bits. In many cases in the embedded world, `int` is 16-bits. There are rare microcontrollers out there that have 24-bit integers, and this remains perfectly standard C.

To make things even more interesting, there are compilers and targets out there that have a hard choice to make. What if their natural register size is 10-bits or 12-bits? Clearly they can't fulfill *both* the requirement to be at least 16-bits AND the requirement to match the natural register size. In these situations, they often choose the natural register size, leaving us with something like this:

```
char (8 bit) <= short (12 bit) <= int (12 bit) <= long (16 bit)
```

Um… yikes. It's obviously breaking a rule or two… but they had to break SOME rules, so they made a choice.

When the C99 standard rolled around, it introduced alternate standard-size types. It also introduced macros for pulling in MIN/MAX values for your integer types. It's glorious! Unfortunately, many embedded compilers can't be relied upon to use the C99 types (Sometimes because they have weird register sizes as described above. Sometimes because they don't feel like it?).

A goal of Unity from the beginning was to support every combination of microcontroller or microprocessor and C compiler. Over time, we've gotten really close to this. There are a few tricks that you should be aware of, though, if you're going to do this effectively on some of these more idiosyncratic targets.

First, when setting up Unity for a new target, you're going to want to pay special attention to the macros for automatically detecting types (where available) or manually configuring them yourself. You can get information on both of these in Unity's documentation.

What about the times where you suddenly need to deal with something odd, like a 24-bit `int`? The simplest solution is to use the next size up. If you have a 24-bit `int`, configure Unity to use 32-bit integers. If you have a 12-bit `int`, configure Unity to use 16 bits. There are two ways this is going to affect you:

1. When Unity displays errors for you, it's going to pad the upper unused bits with zeros.
2. You're going to have to be careful of assertions that perform signed operations, particularly `TEST_ASSERT_INT_WITHIN`. Such assertions might wrap your `int` in the wrong place, and you could experience false failures. You can always back down to a simple `TEST_ASSERT` and do the operations yourself.