Goldilocks Documentation

Release 2.0.0

MousePaw Media

Oct 13, 2022

CONTENTS:

Goldilocks is a complete testing and runtime-benchmark framework, based on MousePaw Media's LIT (Live-In Testing) Standard. Although LIT is inherently different from "unit testing" and TDD (test-driven development), Goldilocks may be used for either approach. It may also be used in conjunction with other testing systems.

The core idea of Goldilocks is that tests ship in the final code, and can be loaded and executed within normal program execution via a custom interface. A major advantage of this system is that tests and benchmarks may be performed on many systems without the need for harnesses, debug flags, or additional software.

The fastest way to run tests in Goldilocks is with the *Shell*.

CHAPTER

ONE

TESTS

Every Goldilocks test is derived from the Test abstract class, which has multiple functions that may be overloaded.

1.1 Test()

This is the class's construtor. Derived class must call the Test constructor and pass the name and docstring as follows:

```
SomeTest()
: Test("test_name", "Some test documentation string here.")
{}
```

1.2 pre()

This is an optional function that sets up the test to be run. In cases where a test is run multiple consecutive times, it is only called once. Thus, it must be possible to call `pre()` once, and then successfully call `run()` any number of times.

The function must return true if setup was successful, and false otherwise, to make sure the appropriate actions are taken.

1.3 prefail()

This is an optional function that tears down the test after a failed call to `pre()`. It is the only function to be called in that situation, and it will not be called under any other circumstances. It has no fail handler itself, so `prefail()` must succeed in any reasonable circumstance.

The function returns nothing.

1.4 run()

This is a required function for any test. It contains all the code for the test run itself. After `pre()` is called once (optionally), `run()` must be able to handle any number of consecutive calls to itself.

There must always be a version of `run()` that accepts no arguments. However, it is not uncommon to overload `run()` to accept a scenario string (part of the LIT Standard) for generating a particular scenario, or prompting a random one to be generated.

The function should return true if the test succeeded, and false if it failed.

Important: `run()` (with no arguments) should be consistent in its success. Assuming pre() was successful, if the first consecutive call to `run()` is successful, all subsequent calls to run() must also be successful. This is vital to the benchmarker functions, as they can call a single test up to 10,000 times. One consideration, then, is that run() should only use one scenario in a single lifetime, unless explicitly instructed by its function arguments to do otherwise.

1.5 `janitor()`

This is called after each repeat of `run()` during benchmarking and comparative benchmarking. It is designed to perform cleanup in between `run()` calls, but not to perform first time setup (`pre()`) or end of testing (`post()`) cleanup. Must return a boolean indicating success.

Janitor is always called before each call to `run()` or `run_optimized()`, including before the first `run()` call.

1.6 `post()`

This is an optional function which is called at the end of a test's normal lifetime. It is the primary teardown function, generally responsible for cleaning up whatever was created in `pre()`, `janitor()` and `run()`. It is normally only called if `run()` returns true, although it will be called at the end of benchmarking regardless of `run()`'s success.

This function returns nothing. It has no fail handler itself, so `post()` should succeed in all reasonable circumstances.

1.7 `postmortem()`

This is an optional teardown function which is usually called if a test fails (`run()` or `run_optimized()` returns `false`). It is responsible for cleaning up whatever was created in `pre()` and `run()`, much like `post()` is, but again only for those scenarios where `run()` fails.

This function returns nothing. If not defined, calls `post()`. It has no fail handler itself, so `postmortem()` should succeed in all reasonable circumstances.

1.7.1 Creating a Test

Creating a test is as simple as creating a class that inherits from `Test (from goldilocks.hpp)`, which is a pure virtual base class.

Important: The constructor and destructor must obviously be defined, however, it is not recommended that they actually do anything - all setup and teardown tasks must be handled by the other functions in order to ensure proper functionality - a test instance is defined once when Goldilocks is set up, but it is highly likely to have multiple lifetimes.

Only bool `run()` must be defined in a test class. The rest of the functions are already defined (they do nothing other than return true), so you only need to define them if you require them to do something.

The following example exhibits a properly-defined, though overly simplistic, test. In reality, we could have skipped `pre()`, `prefail()`, `janitor()`, `postmortem()`, and `post()`, but they are defined to demonstrate their behavior.

```
#include <iochannel.hpp>
#include <goldilocks.hpp>

class TestFoo : public Test
{
public:
    TestFoo(){}

    testdoc_t get_title()
    {
        return "Example Test";
    }

    testdoc_t get_docs()
    {
        return "This is the docstring for our example test."
    }

    bool pre()
    {
        ioc << cat_testing << "Do Pre Stuff" << IOCtrl::endl;
        return true;
    }
    bool prefail()
    {
        ioc << cat_testing << "Do Prefail Stuff" << IOCtrl::endl;
        return true;
    }
    bool run()
    {
        ioc << cat_testing << "Do Test Stuff" << IOCtrl::endl;
        char str[5000] = {'\0'};
        for(int a=0;a<5000;a++)
        {
            str[a] = 'A';
        }
        return true;
    }
    bool janitor()
    {
        ioc << cat_testing << "Do Janitorial Stuff" << IOCtrl::endl;
        return true;
    }
    bool postmortem()
    {
        ioc << cat_testing << "Do Postmortem Stuff" << IOCtrl::endl;
        return true;
    }
    bool post()
    {
        ioc << cat_testing << "Do Post Stuff" << IOCtrl::endl;
        return true;
    }
```

(continues on next page)

(continued from previous page)

```
    ~TestFoo(){}
};
```

1.7.2 Registering a Test

Registering a test with Goldilocks is a trivial task, thanks to its `register_test()` function. Once a test class has been defined, as above, simply register it via…

```
//Assuming testmanager is our instance of the Goldilocks test manager.
testmanager.register_test("TestFoo", new TestFoo);
```

Goldilocks will now actually own the instance of `TestFoo`, and automatically handle its deletion at the proper time.

Warning: Goldilocks actually requires exclusive ownership of each test object registered to it - thus, you should always pass the new declaration as the second argument. If you create the object first, and then pass the pointer, you run a high risk of a segfault or other undefined behavior.

The test can now be called by name using Goldilocks' various functions. (See below.)

You can also optionally register a comparative test for benchmarking, which will be run against the main test in the benchmarker.

```
//Assuming testmanager is our instance of the Goldilocks test manager.
testmanager.register_test("TestFoo", new TestFoo, new TestBar);
```

1.7.3 Running a Test

Once a test is registered with Goldilocks, running it is quite easy.

```
//Run the test once.
testmanager.run_test("TestFoo");

//Benchmark TestFoo on 100 repetitions.
testmanager.run_benchmark("TestFoo", 100);
```

CHAPTER TWO

EXPECT

An **Expect** is a single expression whose evaluation is monitored and recorded. Tests are generally composed of one or more Expectations.

The structure of an Expect is as follows:

```
Expect<That, Should=Should::Pass>(value, expected_value);
```

Some expectations have additional arguments, but the template parameters are the same throughout.

2.1 Macros

An Expect is executed in a Test via one of three macros: `REQUIRE`, `UNLESS`, or `CHECK`.

Typically, these macros would be used in the `run()` function of a Test, but they can also be used anywhere else in the Test where you need to verify an expectation before continuing.

2.1.1 `REQUIRE`

Requires an Expect to be met, else the function will fail.

```
REQUIRE(Expect<That::IsEqual>(40, 40));  // okay
REQUIRE(Expect<That::IsEqual>(2, 3));  // causes function to return false
```

2.1.2 `CHECK`

Evaluates an Expect, but never causes the function to fail.

```
CHECK(Expect<That::IsEqual>(40, 40));  // okay
CHECK(Expect<That::IsEqual>(2, 3));  // okay, although Expect failed
```

2.1.3 `UNLESS`

Requires an Expect to fail, else the function will fail. Unlike `Should::Fail`, this lets the Expect still fail as normal, but the function succeeds.

In practice, you're usually better off using `Should::Fail` in your Expect instead of this macro. This is only if you want the Expect itself to fail, but the function to pass as a result.

```
UNLESS(Expect<That::IsEqual>(true, false));  // okay
UNLESS(Expect<That::IsEqual>(2, 2);  // causes function to return false
```

2.2 That

The behavior of the Expect is primarily determined by the `That::` template parameter.

Most of these expectations depend on the indicated comparison operator being supported by all data types passed to the `` That::`. For example, if you use ``Expect<That::IsLess>(foo, bar) ``, then the types of `foo` and `bar` must be comparable via `foo < bar`.

2.2.1 IsTrue

`Expect<That::IsTrue>(op)`

Expects `op` to implicitly evaluate to `true`:

```
return (op == true);
```

If `op` is a non-null pointer, dereferences `op` and evaluates value. If `op` is `nullptr`, the Expect fails.

2.2.2 IsFalse

`Expect<That::IsFalse>(op)`

Expects `op` to implicitly evaluate to `false`:

```
return (op == false);
```

If `op` is a non-null pointer, dereferences `op` and evaluates value. If `op` is `nullptr`, the Expect fails.

2.2.3 IsEqual

`Expect<That::IsEqual>(left, right)`

Expects `left` and `right` to evaluate as equal:

```
return (left == right);
```

If either `left` or `right` are non-null pointers, they are dereferenced as appropriate, and the evaluation is run against the two values. If either is `nullptr`, the Expect fails.

2.2.4 IsNotEqual

`Expect<That::IsNotEqual>(left, right)`

Expects `left` and `right` to evaluate as *not* equal:

```
return (left != right);
```

If either `left` or `right` are non-null pointers, they are dereferenced as appropriate, and the evaluation is run against the two values. If either is `nullptr`, the Expect fails.

2.2.5 IsLess

`Expect<That::IsLess>(left, right)`

Expects `left` evaluates to less than `right`:

```
return (left < right);
```

If either `left` or `right` are non-null pointers, they are dereferenced as appropriate, and the evaluation is run against the two values. If either is `nullptr`, the Expect fails.

2.2.6 IsLessEqual

`Expect<That::IsLessEqual>(left, right)`

Expects `left` evaluates to less than or equal to `right`:

```
return (left <= right);
```

If either `left` or `right` are non-null pointers, they are dereferenced as appropriate, and the evaluation is run against the two values. If either is `nullptr`, the Expect fails.

2.2.7 IsGreater

`Expect<That::IsGreater>(left, right)`

Expects `left` evaluates to greater than `right`:

```
return (left > right);
```

If either `left` or `right` are non-null pointers, they are dereferenced as appropriate, and the evaluation is run against the two values. If either is `nullptr`, the Expect fails.

2.2.8 IsGreaterEqual

`Expect<That::IsGreaterEqual>(left, right)`

Expects `left` evaluates to greater than or equal to `right`:

```
return (left >= right);
```

If either `left` or `right` are non-null pointers, they are dereferenced as appropriate, and the evaluation is run against the two values. If either is `nullptr`, the Expect fails.

2.2.9 PtrIsNull

`Expect<That::PtrIsNull>(ptr)`

You must pass a pointer to this. Expects the pointer `ptr` to be `nullptr`:

```
return (ptr == nullptr);
```

2.2.10 PtrIsNotNull

`Expect<That::PtrIsNotNull>(ptr)`

You must pass a pointer to this. Expects the pointer `ptr` to *not* be `nullptr`:

```
return (ptr != nullptr);
```

2.2.11 PtrIsEqual

`Expect<That::PtrIsEqual>(left, right)`

You must pass pointers to this. Expects the pointers `left` and `right` to point to the same address in memory:

```
return (left == right);
```

This does not check that the pointers are non-null or valid, nor does it check that the pointers can be dereferenced.

2.2.12 PtrIsNotEqual

`Expect<That::PtrIsNotEqual>(left, right)`

You must pass pointers to this. Expects the pointers `left` and `right` to *not* point to the same address in memory:

```
return (left != right);
```

This does not check that the pointers are non-null or valid, nor does it check that the pointers can be dereferenced.

2.2.13 FuncReturns

`Expect<That::FuncReturns>(target, name_hint, func, args...)`

Passes the arguments `args...` to the function `func`, and expects the returned value to match `target`.

```
return (func(args...) == target);
```

The argument `name_hint` is a string. It is used only for displaying the name of the function in the test report.

2.2.14 FuncThrows

`Expect<That::FuncThrows>(target, name_hint, func, args...)`

Passes the arguments `args...` to the function `func`, and expects the returned value to throw the exception `target`.

```
try {
    func(args...);
} catch (const T& e) { // T is type of target
    return true;
} catch (...) {
    return false;
}
return false;
```

If the function is not supposed to throw anything, you can pass the value `Nothing()` to `target`.

The argument `name_hint` is used only for displaying the name of the function in the test report.

2.2.15 IsApproxEqual

`Expect<That::IsApproxEqual>(value, target, margin)`

Expects value to be approximately equal to target, within the margin.

```
return ((value - target) < 0 ? ((value - target) * (-1.0)) < margin : (value - target) <␣
↪margin));
```

If value is a non-null pointer, dereferences value and evaluates. If value is nullptr, returns false.

```
return (value != nullptr && (*value - target) < 0 ? ((*value - target) * (-1.0)) <␣
↪margin : (*value - target) < margin);
```

2.2.16 IsApproxNotEqual

`Expect<That::IsApproxNotEqual>(value, target, margin)`

Expects value to not be approximately equal to target, within the margin.

```
return ((value - target) < 0 ? ((value - target) * (-1.0)) > margin : (value - target) >␣
↪margin);
```

If value is a non-null pointer, dereferences value and evaluates. If value is nullptr, returns false.

```
return (value != nullptr && (*value - target) < 0 ? ((*value - target) * (-1.0)) >␣
↪margin : (*value - target) > margin));
```

2.2.17 IsInRange

`Expect<That::IsInRange>(value, lower, upper)`

Expects value to be in the inclusive range defined by lower and upper.

```
return (value >= lower && value <= upper);
```

If value is a non-null pointer, dereferences value and evaluates. If op is nullptr, returns false.

```
return (value != nullptr && *value >= lower && *value <= upper);
```

2.2.18 IsNotInRange

`Expect<That::IsNotInRange>(value, lower, upper)`

Expects value to be outside the inclusive range defined by lower and upper.

```
return (value < lower || value > upper);
```

If value is a non-null pointer, dereferences value and evaluates. If op is nullptr, returns false.

```
return (value != nullptr && (*value < lower || *value > upper));
```

2.3 Should

The `Should` template parameter determines how the outcome of the evaluation is interpreted.

`Should::Pass` means the evaluation should succeed for the expectation to be met. This is the default.

`Should::Fail` means the evaluation is NOT supposed to succeed for the expectation to be met. This is useful for guarding against false positives.

```
Minute min(60);
Hour hour(1);
Hour bad_hour(2);

// Check min == hour
REQUIRE(Expect<That::IsEqual>(min, hour));

// Check min != hour
REQUIRE(Expect<That::IsNotEqual>(min, bad_hour));

// Check min == bad_hour FAILS
REQUIRE(Expect<That::IsEqual, Should::Fail>(min, bad_hour));

// If the last Expect was OK, that would mean `min == bad_hour`, throwing
// the reliability of `min == hour` into doubt.
```

`Should::Pass_Silent` and `Should::Fail_Silent` are the same as their non-silent counterparts, but if the expectation is met, no output is produced. Instead, output is only produced if the expectation is NOT met.

CHAPTER

THREE

SUITES

CHAPTER

FOUR

MANAGER

CHAPTER

FIVE

BENCHMARKER

CHAPTER SIX

SHELL

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

INDEX