Goldilocks Documentation

Release 2.0.0

MousePaw Media

Jun 17, 2022

CONTENTS:

1 Goldilocks **3**
- 1.1 What is Goldilocks? . . . 3
- 1.2 Setting Up Tests . . . 3
- 1.3 Setting Up Suites . . . 7
- 1.4 Interfacing Functions . . . 9
- 1.5 Benchmarker Output . . . 10

2 GoldilocksShell **15**
- 2.1 What Is GoldilocksShell? . . . 15
- 2.2 Setting Up . . . 15
- 2.3 Interactive Mode . . . 16
- 2.4 Command Line Argument Mode . . . 18
- 2.5 A Complete Example . . . 20

3 Indices and tables **23**

Index **25**

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

GOLDILOCKS

1.1 What is Goldilocks?

Goldilocks is a complete testing and runtime-benchmark framework, based on MousePaw Media' LIT Standard. Although LIT is inherently different from "unit testing" and TDD, 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 benchmarks may be performed on many systems without the need for additional software.

The fatest way to run tests in Goldilocks is with the *GoldilocksShell*.

1.1.1 Including Goldilocks

To include Goldilocks, use the following:

```
#include "goldilocks/goldilocks.hpp"
```

1.2 Setting Up Tests

1.2.1 Structure

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

`get_title()`

Returns a string (of type `testdoc_t`) with the title of the test. This is a required function for any test.

Note: The title is separate from the ID (name) of the test used to register the test with the TestManager. You use the ID (name) to refer to the test; the title is displayed on the screen before running the test.

---

get_docs()

Returns a string (of type `testdoc_t`) with the documentation string for the test. This should describe what the test does. This is a required function for any test.

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.

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 should return a boolean indicating whether the tear-down was successful or not.

Note: Goldilocks currently ignores `prefail()`'s return.

---

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.

---

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. It returns a boolean indicating success.

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()` and `run()`. It is normally only if `run()` returns true, although it will be called at the end of benchmarking regardless of `run()`'s success.

This function should return a boolean indicating success. It has no fail handler itself, so `post()` should succeed in all reasonable circumstances.

Note: Goldilocks currently ignores `post()`'s return.

---

postmortem()

This is an optional teardown function which is usually called if a test fails (`run()` 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 should return a boolean indicating success. It has no fail handler itself, so `postmortem()` should succeed in all reasonable circumstances.

1.2.2 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 <iosqueak/channel.hpp>
#include <goldilocks/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."
```

(continues on next page)

(continued from previous page)

```
    }

    bool pre()
    {
        channel << cat_testing << "Do Pre Stuff" << IOCtrl::endl;
        return true;
    }
    bool prefail()
    {
        channel << cat_testing << "Do Prefail Stuff" << IOCtrl::endl;
        return true;
    }
    bool run()
    {
        channel << 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()
    {
        channel << cat_testing << "Do Janitorial Stuff" << IOCtrl::endl;
        return true;
    }
    bool postmortem()
    {
        channel << cat_testing << "Do Postmortem Stuff" << IOCtrl::endl;
        return true;
    }
    bool post()
    {
        channel << cat_testing << "Do Post Stuff" << IOCtrl::endl;
        return true;
    }
    ~TestFoo(){}
};
```

1.2.3 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);
```

1.2.4 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);

//Compare TestFoo and TestBar on 100 repetitions.
testmanager.run_compare("TestFoo", "TestBar", 100);
```

1.3 Setting Up Suites

A Suite is a collection of tests. In a typical use of Goldilocks, all tests are organized into Suites.

In addition to allowing on-demand loading groups of tests, a Suite can be "batch run", where all of its tests are run in succession. When one test fails, the batch run halts and returns false.

1.3.1 Structure

Every Goldilocks suite is derived from the `TestSuite` abstract class. This only has two functions to overload, but both are required.

`get_title()`

Returns a string (of type `testsuitedoc_t`) with the title of the suite. This is the a required function for any test.

Note: The title is separate from the ID (name) of the test used to register the test with the TestManager. You use the ID (name) to refer to the test; the title is displayed on the screen before running the test.

load_tests()

This function specifies which tests belong to the suite.

`TestSuite` provides a function `register_test()` which properly registers each test with both the suite and the TestManager itself. For convenience, it follows the same format as `TestManager::register_test()`, with the exception of an optional boolean argument for specifying a test which belongs to the suite, but should not be part of the Suite's batch run.

One reason to exclude a test from the batch run for the Suite is if the test is a stress test that takes a long time to run.

We can also register the comparative tests as an optional fourth argument.

Below is an example of a Suite's `load_tests`.

```
void TestSuite_MagicThing::load_tests()
{
    /* Register this test with both the suite and the test manager.
     * Also register the comparative form. */
    register_test("t101", new MagicThing_Poof(), true, new OtherThing_Poof());

    register_test("t102", new MagicThing_Vanish());

    register_test("t103", new MagicThing_Levitate());

    register_test("t104", new MagicThing_Telepathy());

    /* This test will be loaded by the suite, but will be excluded
     * from the batch run. */
    register_test("t105", new MagicThing_SawInHalf(), true);
}
```

We have registered five tests with this suite, not counting the comparative form of the one. Upon loading the suite, all five tests will be loaded into the test manager. However, if we were to batch run this suite, only four of those tests (t101, t102, t103, and t104) would be run.

1.3.2 Registering a Suite

Registering a suite with Goldilocks is as easy as registering a test. Simply use its `register_suite()` function. Once a suite class has been defined, as above, it is registered with…

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

As with tests, Goldilocks owns the instance of `TestSuiteFoo`, and automatically handles its deletion at the proper time.

> **Warning:** Goldilocks requires exclusive ownership of each suite object registered to it, the same as it does tests.

1.3.3 Loading a Suite

One of the major advantages of using a suite is that you can load its tests on demand. This is especially useful if you have hundreds or thousands of tests.

```
//Load a particular suite.
testmanager.load_suite("TestSuiteFoo");
```

Of course, sometimes you don't want to have to load each suite manually. As a shortcut, you can just load all suites currently registered with the test manager by calling…

```
//Load a particular suite.
testmanager.load_suite();
```

1.3.4 Running a Suite

You can start a batch run of all the suite's tests using…

```
//Batch run all tests in a suite.
testmanager.run_suite("TestSuiteFoo");
```

1.4 Interfacing Functions

Goldilocks provides a number of convenience functions to aid in creating an interactive command-line interface for the system.

In most cases, you can probably just use the GoldilocksShell (see *goldilocksshell*).

1.4.1 Functions

list_suites()

You can display the names and titles of all the tests currently registered in the test manager using…

```
// List all registered suites with their names and titles.
testmanager.list_suites();

// List all registered suites with their name only (no title).
testmanager.list_suites(false);
```

list_tests()

You can display the names and titles of all the tests currently registered (loaded) in the test manager using…

```
// List all registered tests with their names and titles.
testmanager.list_tests();

// List all registered tests with their name only (no title).
testmanager.list_tests(false);
```

If a test is loaded via a suite, it will not appear in this list until its suite has actually been loaded during that session.

i_load_suite()

Identical usage to `load_suite()`, except it prompts the user for confirmation before loading a suite.

i_run_benchmark()

Identical usage to `run_benchmark()`, except it prompts the user for confirmation before running the benchmark.

i_run_compare()

Identical usage to `run_compare()`, except it prompts the user for confirmation before running the compare.

i_run_suite()

Identical usage to `run_suite()`, except it prompts the user for confirmation before running the suite.

i_run_test()

Identical usage to `run_test()`, except it prompts the user for confirmation before running the test.

1.5 Benchmarker Output

The Goldilocks benchmarker outputs a *lot* of information. This section describes how to read it.

1.5.1 Pass Types

To account for the effects of cache warming, Goldilocks makes three passes, each with a specific behavior:

- **Mama Bear** attempts to simulate a "cold cache," increasing the likelihood of cache misses. This is done by running tests A and B alternatingly.
- **Papa Bear** attempts to simulate a "hot cache," decreasing the likelihood of cache misses. This is done by running all repetitions of test A before running all repetitions of test B.
- **Baby Bear** attempts to simulate average (or "just right") cache warming effects, such as what might be seen in typical program executions. This is done by running eight repetitions of each test alternatingly - 8 As, 8 Bs, 8 As, etc.

After running all three passes, the benchmarker results are displayed.

1.5.2 Result Groups

At the top of the results, we see the `BASELINE MEASUREMENTS`. These are based on measuring the actual measurement function of our benchmarker.

These results are important, as this is an indicator of fluctuations in results from external factors. If either of the RSD (Relative Standard Deviation) numbers are high (>10%), the results of the benchmarker may be thrown off.

Next, we see the individual results for each test beneath each pass type. The verdict is displayed below both sets of results, indicating which test was faster, and by how much. The verdict is ultimately the difference between means, but if that difference is less than the standard deviation, it will indicate that the tests are "roughly equal."

1.5.3 Statistical Data

Let's break down the statistical data in our results.

Most lines show two sets of values, separated with a `/` character. The *left* side is the **RAW** value, accounting for each measurement taken. The *right* side is the **ADJUSTED** value, which is the value after outlier measurements have been removed from the data.

The **MEAN ()** is the average number of CPU cycles for a single run of the test.

The **MIN-MAX(RANGE)** shows the lowest and highest measurement in the set, as well as the difference between the two (the range).

OUTLIERS shows how many values have been removed from the ADJUSTED set. Outliers are determined mathematically, and removing them allows us to account for external factors, such as other processes using the CPU during the benchmark.

SD () shows our standard deviation, which indicates how much fluctuation occurs between results. By itself, the standard deviation is not usually meaningful.

The **RSD**, or Relative Standard Deviation, is the percentage form of the standard deviation. This is perhaps the most important statistic! The lower the RSD, the more precise the benchmark results are. If the RSD is too high, it will actually be flagged as red.

The statistical data above can provide a useful indicator of the reliability of the benchmark results.

A high RSD may indicate that the results are "contaminated" by external factors. It is often helpful to run the comparative benchmark multiple times, and taking the pass with the lowest RSD.

However, higher RSDs may be the result of the tests themselves, as we'll see in the following example.

Other warning signs that the results may be contaminated or inaccurate include:

- The presence of outliers in BASELINE.
- RSDs > 10% in BASELINE.
- Red-flagged RSDs (> 25%) (unless the test has a technical reason to fluctuate in CPU cycle measurements between tests).
- Significantly different verdicts between passes.

The precision and accuracy of the results may be further validated by running the comparative benchmark multiple times, especially across computers, and directly comparing the RSDs and verdict outcomes. While actual CPU cycle measurements may vary greatly between similar systems, the relative outcomes should remain fairly consistent on most systems with the same processor architecture.

Statistical Data Example

Let's look at the comparison between the "shift" (insert at beginning) functionality of FlexArray and `std::vector`. You can run this yourself using the tester, with the command `benchmark P-tB1002`.

We always start by screening the baseline:

```
BASELINE MEASUREMENTS
    MEAN (): 64 / 65
    MIN-MAX(RANGE): 58-75(17) / 58-75(17)
    OUTLIERS: 0 LOW, 0 HIGH
    SD (): 5.47 / 5.38
    RSD: 8% / 8%
```

We have no outliers and very low RSDs, so our results probably aren't contaminated. Of course, benchmarking is unpredictable, and external factors may change during the benchmarking itself. However, we have no reason here to throw out the results.

Had we seen an RSD greater than 10% for either result, it would have been wise to discard these results and rerun the benchmark altogether.

Now let's look at the first pass, MAMA BEAR, which is designed to demonstrate the effects of cache misses:

```
MAMA BEAR: [FlexArray: Shift 1000 Integers to Front (FlexArray)]
    MEAN (): 414650 / 401451
    MIN-MAX(RANGE): 262280-739036(476756) / 262280-323876(61596)
    OUTLIERS: 0 LOW, 5 HIGH
    SD (): 106700.22 / 76270.09
    RSD: 25% / 18%

MAMA BEAR: [FlexArray: Shift 1000 Integers to Front (std::vector)]
    MEAN (): 904723 / 876586
    MIN-MAX(RANGE): 664354-1537966(873612) / 664354-714892(50538)
    OUTLIERS: 0 LOW, 5 HIGH
    SD (): 232960.59 / 169329.87
    RSD: 25% / 19%
```

Unsurprisingly, both results show some high outliers. The RSDs are roughly equal, however, so this is probably the result of those cache misses or other related factors.

> **Warning:** How the two tests are structured matters! We are very careful to ensure both tests have the same structure and implementation, so the only difference between the two is the functions or algorithms we are directly comparing.

Looking at the result:

```
MAMA BEAR: VERDICT
        RAW: [FlexArray: Shift 1000 Integers to Front (FlexArray)] faster by approx.␣
↪490073 cycles.
    ADJUSTED: [FlexArray: Shift 1000 Integers to Front (FlexArray)] faster by approx.␣
↪398864.90807662549195 cycles.
```

FlexArray wins that round.

Now let's look at PAPA BEAR, which attempts to demonstrate cache warming:

```
PAPA BEAR: TEST [FlexArray: Shift 1000 Integers to Front (FlexArray)]
    MEAN (): 321917 / 325168
    MIN-MAX(RANGE): 305608-310824(5216) / 305608-310824(5216)
    OUTLIERS: 0 LOW, 0 HIGH
    SD (): 28252.27 / 28548.56
    RSD: 8% / 8%

PAPA BEAR: TEST [FlexArray: Shift 1000 Integers to Front (std::vector)]
    MEAN (): 654278 / 659817
    MIN-MAX(RANGE): 608020-765749(157729) / 608020-685548(77528)
    OUTLIERS: 0 LOW, 1 HIGH
    SD (): 53785.7 / 53494.46
    RSD: 8% / 8%
```

Unlike MAMA BEAR, these results have much lower RSDs - in fact, they are equal to the BENCHMARK RSDs (the ideal scenario) - and only one outlier between the two. This further lends itself to our theory that the higher RSDs in MAMA BEAR are the result of cache misses.

FlexArray wins this as well, albeit by a somewhat narrower margin:

```
PAPA BEAR: VERDICT
        RAW: [FlexArray: Shift 1000 Integers to Front (FlexArray)] faster by approx.␣
↪332361 cycles.
    ADJUSTED: [FlexArray: Shift 1000 Integers to Front (FlexArray)] faster by approx.␣
↪306100.43052620673552 cycles.
```

Finally, we look at BABY BEAR, which is intended to be the most similar to typical use scenarios:

```
BABY BEAR: TEST [FlexArray: Shift 1000 Integers to Front (FlexArray)]
    MEAN (): 317852 / 321814
    MIN-MAX(RANGE): 247433-323226(75793) / 306612-323226(16614)
    OUTLIERS: 1 LOW, 0 HIGH
    SD (): 33872.37 / 33610.86
    RSD: 10% / 10%

BABY BEAR: TEST [FlexArray: Shift 1000 Integers to Front (std::vector)]
    MEAN (): 648568 / 652663
    MIN-MAX(RANGE): 537774-780641(242867) / 537774-755231(217457)
    OUTLIERS: 0 LOW, 2 HIGH
    SD (): 60925.17 / 58541.29
    RSD: 9% / 8%
```

Our RSDs are slightly higher than with PAPA BEAR, but we still see relatively few outliers (a total of 3).

The BABY BEAR verdict indicates that FlexArray is the fastest, even in this scenario:

```
BABY BEAR: VERDICT
        RAW: [FlexArray: Shift 1000 Integers to Front (FlexArray)] faster by approx.␣
↪330716 cycles.
    ADJUSTED: [FlexArray: Shift 1000 Integers to Front (FlexArray)] faster by approx.␣
↪297238.13385525450576 cycles.
```

CHAPTER

TWO

GOLDILOCKSSHELL

2.1 What Is GoldilocksShell?

What good are tests without a way to run them? While you can certainly write your own class for interacting with Golidlocks, we wanted to provide a quick and easy way to run your tests and suites interactively.

Assuming you have one or more TestSuites configured (see *Setting Up Suites*), you can set up a GoldilocksShell relatively quickly!

2.2 Setting Up

We import GoldilocksShell with…

```
#include "goldilocks/shell.hpp"
```

We start by defining a new `GoldilocksShell` object instaces somewhere in our code, preferably in our `main()` function.

```
GoldilocksShell* shell = new GoldilocksShell(">> ");
```

The part in quotes is the *prompt string*, which will appear at the start of every line where the user can type in the interactive terminal.

Now we need to register all of our Goldilocks `TestSuite` classes with the shell. Note that we don't need to create instances of these ourselves, but merely pass the class as a type. We also need to specify the name of the suite in quotes: this name will be what is used to identify it in the shell.

```
shell->register_suite<TestSuite_Brakes>("s-brakes");
shell->register_suite<TestSuite_Hologram>("s-hologram");
shell->register_suite<TestSuite_TimeRotor>("s-timerotor");
shell->register_suite<TestSuite_CloisterBell>("s-cloisterbell");
```

That is it! We are now ready to use GolidlocksShell.

2.3 Interactive Mode

As long as we're running in the command line, we can hand over control to the GoldilocksShell via a single line of code…

```
shell->interactive();
```

The GoldilocksShell will immediately launch and take over the terminal, using IOSqueak.

2.3.1 Commands

In Interactive Mode, you are given a complete shell for executing Golilocks tests and suites.

Note: This initial version of GoldilocksTester does not offer support for the conventional shell commands, including history or arrow-key navigation. We'll be adding these in a later version.

To get help at any point, run `help`. To quit, type `exit`.

In this guide, we'll be using the default GoldilocksShell prompt symbol, `:`. This may be different for your project, depending on how you configured GoldilocksShell.

Listing and Loading Suites and Tests

When you first start GoldilocksShell, no tests have been loaded into memory. However, all the suites you specified are *ready* to be loaded.

To see all the available suites, run `listsuites`:

```
: listsuites
s-brakes: TARDIS Brakes Tests
s-hologram: TARDIS Holographic Interface Tests
s-timerotor: TARDIS Time Rotor Tests
s-cloisterbell: TARDIS Cloister Bell Tests
```

Thus, before you can run a test or suite, you must first **load** the suite containing it:

```
: load s-brakes
TARDIS Brakes Tests
Suite loaded.
```

Now you can run the `list` command to see all the loaded tests:

```
: list
t-brakes-engage: TARDIS Brakes: Engage Brakes
t-brakes-warn: TARDIS Brakes: No Brakes Warning
t-brakes-disengage: TARDIS Brakes: Disengage Brakes
t-brakes-fail: TARDIS Brakes: Brake Failure Protocol
t-brakes-pressure: TARDIS Brakes: Brake Pressure Test
```

Note: If `list` does not show any tests, be sure you've loaded at least one suite first.

If you just want to load *all* suites, simply run the `load` command without any arguments. It will ask you to confirm your choice:

```
: load
Load ALL test suites? (y/N) y
TARDIS Brakes Tests loaded.
TARDIS Holographic Interface Tests loaded.
TARDIS Time Rotor Tests loaded.
TARDIS Cloister Bell Tests loaded.
```

You can find out more information about any test using the `about` command:

```
: about t-brakes-engage
TARDIS Brakes: Engage Brakes
Ensures the controls are capable of engaging the brakes.
```

Running Tests and Suites

It is possible to run any test using the `run` command. This command always asks you to confirm before continuing:

```
: run t-brakes-engage
Run test TARDIS Brakes: Engage Brakes [t-brakes-engage]? (y/N) y
===== [TARDIS Brakes: Engage Brakes] =====
Pass 1 of 1
TEST COMPLETE
```

Optionally, you can repeat a test multiple times by specifying the number of times to repeat it.

> : run t-brakes-engage 5 Run test TARDIS Brakes: Engage Brakes [t-brakes-engage]? (y/N) y ===== [TARDIS Brakes: Engage Brakes] ===== Pass 1 of 5 Pass 2 of 5 Pass 3 of 5 Pass 4 of 5 Pass 5 of 5 TEST COMPLETE

You can also run an entire suite in one step:

```
: run s-brakes
Run test suite TARDIS Brakes Tests [s-brakes]? (y/N) y
===== [TARDIS Brakes Tests] =====
===== [TARDIS Brakes: Engage Brakes] =====
Pass 1 of 1
TEST COMPLETE
===== [TARDIS Brakes: No Brakes Warning] =====
Pass 1 of 1
TEST COMPLETE
===== [TARDIS Brakes: Disengage Brakes] =====
Pass 1 of 1
TEST COMPLETE
===== [TARDIS Brakes: Brake Failure Protocol] =====
Pass 1 of 1
TEST COMPLETE
===== [TARDIS Brakes: Brake Pressure Test] =====
Pass 1 of 1
TEST COMPLETE

SUITE COMPLETE
```

Note: If you specify a repeat number for running a suite, it will be ignored.

Benchmarking

Golidlocks supports *comparative benchmarking*. There are two ways to run such a benchmark.

The first method requires a comparative test to be specified within a suite (see *load_tests()*). If you've done this, you can benchmark the test and its comparative, and output the complete benchmark stats:

```
: benchmark t-brakes-engage
Run comparative benchmark between TARDIS Brakes: Engage Brakes [t-brakes-engage] and␣
↪TARDIS Brakes: Handbrake? (y/N) at 100 repetitions? (y/N) y
======================
|     BENCHMARKER    |
======================
```

Upon completion it will display the complete benchmarker stats (see *Benchmarker Output*).

You can also specify the number of times to run the benchmarker (the default is 100):

```
: benchmark t-brakes-engage 1000
Run comparative benchmark between TARDIS Brakes: Engage Brakes [t-brakes-engage] and␣
↪TARDIS Brakes: Handbrake? (y/N) at 1000 repetitions? (y/N) y
======================
|     BENCHMARKER    |
======================
```

You can also run a comparative benchmark on any two tests using the `compare` function. It functions in much the same way, except that you specify *two* tests instead of one, and then the optional repetition count:

```
: compare t-brakes-engage t-brakes disengage 500
Run comparative benchmark between TARDIS Brakes: Engage Brakes [t-brakes-engage] and␣
↪TARDIS Brakes: Disengage [t-brakes-disengage]? (y/N) at 1000 repetitions? (y/N) y
======================
|     BENCHMARKER    |
======================
```

2.4 Command Line Argument Mode

2.4.1 Invocation

GoldilocksShell is also designed to handle the same input arguments as your typical `int main()`, which allows you to invoke the shell using command-line arguments.

This is especially useful for integrating Goldilocks into a Continuous Integration [CI] system, such as Jenkins. If the specified tests and suites are successful, the program will exit with code `0`; failures will cause the program to exit with code 1.

To use this feature, you must simply pass the argument count and argument array to the GoldilocksShell's `command()` function. It handles its own argument parsing.

```
int main(int argc, char* argv[])
{
    // ...setup code here...

    // If we got command-line arguments...
    if(argc > 1)
    {
        return shell->command(argc, argv);
    }

    return 0;
}
```

2.4.2 Skipping Arguments

If you accept other arguments via command-line, you may ask GoldilocksShell to skip those. Just specify the number of arguments to skip in the third argument.

Important: GoldilocksShell already knows to skip the first argument, which is the program invocation. You only need to tell it how many *extra* arguments to skip.

For example…

```
// myprogram --goldilocks --run sometest
int main(int argc, char* argv[])
{
    // ...setup code here...

    // If we're supposed to invoke Goldilocks.
    if(argc > 1 && strcmp(argv[1], "--goldilocks") == 0)
    {
        // Asking GoldilocksShell to skip one argument...
        return shell->command(argc, argv, 1);
        // Now it will only process arguments starting from ``--run``...
    }

    return 0;
}
```

2.4.3 Usage

GoldilocksShell's command line interface accepts multiple arguments, which are used to load and run tests, suites, and benchmarks. Commands are always run from left to right, in order.

The basic commands are as follows:

- `--help` displays help.
- `--listsuites` lists all available suites.
- `--load suite` loads the suite `suite`.

- `--list` lists all loaded tests.
- `--run item` runs the test or suite `item`.
- `--benchmark item` benchmarks the test `item`.

Important: The command line does not include the `compare` function, nor the ability to specify the number of test repetitions.

Ordinarily, to run a test, you must first load the suite containing it. However, for the sake of convenience, if you don't explicitly load any tests in the command, it will just load all suites. Thus…

```
$ tester --run t-brakes-engage
```

…will just load all the suites before attempting to run the test `t-brakes-engage`.

If you want to only load a single suite, perhaps to see what tests it contains, just include the `--load` argument. (Remember, if you don't explicitly load any suites, all the suites will be loaded.)

```
$ tester --load s-brakes --list
```

Warning: Each command only accepts one argument! If you want to load multiple suites, you must precede each suite ID with the `--load` argument.

Arguments are run in order, from left to right, and the program doesn't exit until all of them are finished. This means you can run multiple tests in one command; success will only be reported (exit code `0`) if all the tests pass.

```
$ tester --load s-brakes --run t-brakes-engage --run t-brakes-disengage
```

The above command, after loading only the specified suite, will run the requested tests. If they *both* succeed, the program will exit reporting success (exit code `0`).

Warning: Each command only accepts one argument! If you want to load multiple suites, you must precede each suite ID with the `--load` argument.

We can also run benchmarks from the command line. `--benchmark` bases its success/fail condition on the Baby Bear comparison; success means either (a) the main test is faster than its comparative, or (b) the two tests are roughly identical in performance ("dead heat").

```
$ tester --load s-brakes --benchmark t-brakes-engage
```

2.5 A Complete Example

Let's tie all this together. Here's an example of a complete `int main()` function set up to use GoldilocksShell, as outlined in the previous sections.

```
int main(int argc, char* argv[])
{
    GoldilocksShell* shell = new GoldilocksShell(">> ");
```

(continues on next page)

(continued from previous page)

```
    shell->register_suite<TestSuite_Brakes>("s-brakes");
    shell->register_suite<TestSuite_Hologram>("s-hologram");
    shell->register_suite<TestSuite_TimeRotor>("s-timerotor");
    shell->register_suite<TestSuite_CloisterBell>("s-cloisterbell");

    // If we got command-line arguments...
    if(argc > 1)
    {
        return shell->command(argc, argv);
    }
    else
    {
        // Shift control to the interactive console.
        shell->interactive();
    }

    // Delete our GoldilocksShell.
    delete shell;
    shell = 0;

    return 0;
}
```

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

B

benchmark, *see* test

S

suite, 3
    get_title(), 7
    structure, 7

T

test, 3
    creating, 5
    get_docs(), 3
    get_title(), 3
    pre(), 4
    registering, 6
    running, 7
    structure, 3