



Testudo, an automatic test system for C++ code

(version 0.3)

Miguel González Cuadrado
<mgcuadrado@gmail.com>

21st October 2020

Contents

Intro	iii
1 Test output formats	1
2 Test definition and test instruction styles	1
3 How to use Testudo	3
4 Tests and test hierarchies	3
4.1 Non-top test nodes	4
4.2 Top test nodes	4
5 Test steps	5
5.1 Declarations and actions	5
5.2 Checks	6
5.3 Adding information to the report	11
5.4 Identifying steps	13
5.5 Printing fixed text and separations	13
5.6 Fake declarations and actions	14
5.7 Unexpected exceptions	15
5.8 Fatal errors	15
6 Test-aware functions	16

7	Fixtures	17
7.1	Definition	17
7.2	Usage	18
7.3	Fixture members and their initialisation	18
8	With-data loops	19
9	Mock objects	20
9.1	Mock-method macros	21
9.2	How to schedule mock-method behaviour	24
9.3	How to check mock-method logs	25
9.4	How to check mock-method ledgers	26
10	Testudo support for STL types	28
11	Adding Testudo support for your types	28
11.1	Validity	29
11.2	Textual representation	29
11.3	Equality	29
11.4	Difference between two values	29
A	Testudo installation	31
B	Editor configuration	31
C	Using your own test definition and test instruction names	31

List of Figures

List of Tables

I	Macro names in the default styles	2
----------	--	----------

This manual is in its early stages. I'm going first for completeness, not for clarity. My plans: finish covering the functionality of Testudo, convert into a web-friendly format, turn it into a guide rather than a reference.

Intro

Automatic testing: unit testing, integrated testing, test-driven development... You can do all of these with Testudo.

Brief rundown of Testudo's features:

- tree-like organisation of tests (§4);
- execution of selected tests;
- easy definition of tests (§5);
- clear, complete, flexible test reports;
- straightforward syntax for declarations, actions, showing values, checks on values, predicate checks, and scopes (§5);
- support for exception checks (§5.2.7) and unexpected exceptions (§5.7);
- support for complex types (§10);
- easy support for new types (§11);
- fixtures (§7);
- repeating steps for many values, like theorem checking (§8);
- mock objects (§9).

"Testudo" means "turtle" in Esperanto and Latin. The name of the turtle in the logo is Testarudo, which means "stubborn" in Spanish. Testarudo is indefatigable, and will tenaciously flag you errors, until you correct them.

Testudo runs a suite of tests specified by the user, and produces an XML printout of the results. You can convert the XML file into a variety of formats, for immediate perusal, result tracking, statistics, publication, et cetera. The blue question mark and the green [OK] and red [FAIL] flags in the logo are a reference to the way instructions, passes, and fails are displayed when a test result is converted to a text report with colours¹.

A *test suite* in Testudo is composed of an ordered series of *tests*, each test consisting of a sequence of *steps*. Tests are organised as a tree, with test nodes, and proper tests on some test nodes.

When a test suite is run, all tests are performed in order, according to their position in the test tree, and within each test, the test steps are executed in order. Some test steps are *checks*, which can succeed or fail. Check successes and failures are tallied and summarised at the end of their test. Accumulated tallies are also kept for each test node, which combine the tallies of the sub-tree rooted at each test node.

As a quick, self-paced introduction to Testudo test step syntax, i've laid out hereafter the source code for a class, the source code for a test for the functionality,

¹ Colour-blind-friendly, by the way.

and the resulting test report. If my \LaTeX trickery doesn't fail me, and you have printed this on physical paper, you'll have the source code for the test and the test report on opposite pages (pages vi and vii), so you can't better understand how one relates to the other.

The code is meant for this showcase, not for production, and shouldn't be construed as an example of good coding: you'll see questionably design and coding practices; you'll see bugs and unimplemented functionality so they can be pointed out in the report as test fails; you'll see weird vertical spacing to achieve syncing between the two printed pages; and you'll see too many concerns crammed into a single test.

```

1 #include <list>
2 #include <stdexcept>
3
4 // Hold holds a double value; you can query it with pop(),
5 // but then, it doesn't hold it anymore; when a Hold that is holding a
6 // value is destroyed, it adds it to a list that can be consulted with
7 // Hold::get_forgotten();
8 class Hold {
9     bool holding{false};
10    double held;
11    static inline std::list<double> forgotten;
12 public:
13     Hold()=default;
14     Hold(double i) { hold(i); } // hold on creation
15     ~Hold() { // FIXME: the list ends up in the wrong order; see failed test
16         if (holding)
17             forgotten.push_front(held);
18     }
19     void hold(double i) {
20         if (holding)
21             throw std::runtime_error("already holding a double");
22         held=i;
23         holding=true;
24     }
25     double pop() {
26         if (not holding)
27             throw std::runtime_error("not holding any double");
28         holding=false;
29         return held;
30     }
31     bool is_holding() const { return holding; }
32     static auto get_forgotten() { return forgotten; }
33     static void clear_forgotten() { forgotten.clear(); }
34     // FIXME: not yet implemented; see failed test:
35     static bool is_forgotten_empty() { return false; }
36 };
37
38 #ifndef NAUTOTEST // we can disable tests by defining this macro
39 #include "testudo_lc.h"
40 #include "testudo_ext.h" // we need the support for lists
41
42 namespace {
43     using namespace std;
44     // these test steps are artificially gathered into a single test to showcase
45     // Testudo instructions and the generated report; it checks functionality for
46     // the class HalfDouble; in real life, this should be broken into
47     // smaller tests with focused concerns; the resulting report lays out a full
48     // narrative (you can understand it by reading only the report, without the
49     // source code) [cutting here so you can have the test in one page]

```

```

50 // this defines a test named "use_instructions", titled "use
51 // instructions"
52 define_top_test("testudo", use_instructions,
53                 "use instructions") {
54     print_multiline_text("index:\n"
55                          "  1. holding functionality\n"
56                          "  2. exceptions\n"
57                          "  3. list of forgotten doubles");
58     print_break(); // -----
59     declare(Hold hf);
60     check(not hf.is_holding())_true;
61     perform(hf.hold(3.14));
62     check(hf.is_holding())_true;
63     check(hf.pop())_approx(3.14);
64     check(not hf.is_holding())_true;
65     print_break(); // -----
66     print_text("hf is empty now");
67     step_id(popping_empty);
68     check_try(hf.pop())_catch();
69     perform(hf.hold(2.72));
70     step_id(adding_to_already_holding);
71     check_try(hf.hold(7.))_catch();
72     print_break(); // -----
73     print_text("the forgotten doubles list is still empty");
74     check(Hold::is_forgotten_empty())_true;
75     check(Hold::get_forgotten().size())_equal(0u);
76     show_scope("scope 1") {
77         declare(Hold hf1(1.1)); // hold-on-construction syntax
78         declare(Hold hf2(2.2));
79         show_scope() { // unnamed scope
80             declare(Hold hf3(3.3)); // will add to the list on scope closing
81         }
82         check(Hold::get_forgotten())_tol(.5)_approx({3.});
83         show_value(hf2.pop());
84         print_text("hf2 now empty; it won't add to the list");
85     }
86     check(not Hold::is_forgotten_empty())_true;
87     check(Hold::get_forgotten()
88           _approx({3.3, 1.1}));
89     perform(Hold::clear_forgotten());
90     check(Hold::get_forgotten().size())_equal(0u);
91     print_text("the following will raise an error");
92     perform(hf.hold(9.9));
93     print_text("this won't show, because of the error");
94 }
95 }
96 #endif

```

```

1 | -----
2 | {testudo.use_instructions} use instructions |
3 | -----
4 |   index:
5 |     1. holding functionality
6 |     2. exceptions
7 |     3. list of forgotten doubles
8 | -----
9 | : Hold hf ;
10 | % not hf.is_holding() [ OK ]
11 | # hf.hold(3.14) ;
12 | % hf.is_holding() [ OK ]
13 | % hf.pop() // 3.14 +/- eps [ OK ]
14 | % not hf.is_holding() [ OK ]
15 | -----
16 | " hf is empty now "
17 | [ popping_empty ]
18 | & hf.pop() > " not holding any double " [ OK ]
19 | # hf.hold(2.72) ;
20 | [ adding_to_already_holding ]
21 | & hf.hold(7.) > " already holding a double " [ OK ]
22 | -----
23 | " the forgotten doubles list is still empty "
24 | % Hold::is_forgotten_empty() : false ----- [ FAIL ]
25 | % Hold::get_forgotten().size() == 0u [ OK ]
26 | # { " scope 1 "
27 |   : Hold hf1(1.1) ;
28 |   : Hold hf2(2.2) ;
29 |   # {
30 |     : Hold hf3(3.3) ;
31 |   }
32 |   % Hold::get_forgotten() // {3.} +/- .5 [ OK ]
33 |   ? hf2.pop() : 2.2
34 |   " hf2 now empty; it won't add to the list "
35 | # } " scope 1 "
36 | % not Hold::is_forgotten_empty() [ OK ]
37 | % Hold::get_forgotten() // {3.3, 1.1} +/- eps : list{1.1\
38 |   , 3.3} // list{3.3, 1.1} ----- [ FAIL ]
39 | # Hold::clear_forgotten() ;
40 | % Hold::get_forgotten().size() == 0u [ OK ]
41 | " the following will raise an error "
42 | # hf.hold(9.9) ;
43 | > uncaught exception " already holding a double " ----- [ ERR- ]
44 | {testudo.use_instructions} 2/12 fail, 1 err ----- [ ERR- ]

```


I Test output formats

When a test suite is run, Testudo outputs a printout detailing each test node, test, test step, and tallies. There are several formats for the printout. You must choose the format by passing to the executable the flag “-f” followed by the name of the format. Standard formats are “xml” and “color_text”, but you can add your own.

The “xml” format outputs the printout in XML format. This format records all details of the test suite, and is meant for consumption by an XML parser. The available parser is “xml_to_color”, which by default converts the printout to a full text report with colours. With the flag “-b”, the output is identical but with no colours. With “-s”, the output is only a summary, giving the check statistics for each test node and each test. The default version uses the XSLT file “testarudo.xslt” to interpret all possible elements and attributes in the XML printout.

The “color_text” format outputs the printout directly as a full text report with colours, virtually identical to the one “xml_to_color” produces. The difference is that this format produces its output synchronously, so even if a fatal error happens, you’ll get the whole output until just before the error, whereas the “xml” won’t output anything².

2 Test definition and test instruction styles

All test instructions described in this section are implemented as C++ macros. You can choose among different styles for the macro names, or even rename them altogether to your liking (see §C). Out-of-the-box, the available styles are:

- “lc” (lowercase), where all macro names are in lowercase, and continuing macros have a leading underscore, so that they can be stuck to the preceding expression nicely; this style is easy on the eyes, but may be too cluttered for some people; here’s an example:

```
1 declare(int a=7); // declare a variable
2 check(a+2)_equal(9); // check for equality
```

- “uc” (uppercase), where all macro names are in uppercase, and continuing macros are expected to be separated from the preceding expression by a space; this style shows clearly the parts of check instructions, but may be excessively macroish; here’s the same example as for “lc”, but in “uc” style:

```
1 DECLARE(int a = 7); // declare a variable
2 CHECK(a + 2) EQUAL(9); // check for equality
```

See Table 1 for a list of all macro syntax names, with their macro name in the “lc” and “uc” styles.

Whatever the style you choose, your editor may help you writing and reading test instructions, for instance by giving them a specific colour; see §B for details.

²This is so because the “xml” format first builds the whole XML object for the printout before printing it.

<i>macro syntax name</i>	<i>lc.tst style</i>	<i>uc.tst style</i>
<i>top-test-node</i>	top_test_node	TOP_TEST_NODE
<i>define-top-test-node</i>	define_top_test_node	DEFINE_TOP_TEST_NODE
<i>define-test-node</i>	define_test_node	DEFINE_TEST_NODE
<i>define-top-test</i>	define_top_test	DEFINE_TOP_TEST
<i>define-test</i>	define_test	DEFINE_TEST
<i>with-fixture</i>	with_fixture	WITH_FIXTURE
<i>visible-fixture</i>	visible_fixture	VISIBLE_FIXTURE
<i>with-data-define-top-test</i>	with_data_define_top_test	WITH_DATA_DEFINE_TOP_TEST
<i>with-data-define-test</i>	with_data_define_test	WITH_DATA_DEFINE_TEST
<i>step-id</i>	step_id	STEP_ID
<i>print-text</i>	print_text	PRINT_TEXT
<i>print-multiline-text</i>	print_multiline_text	PRINT_MULTILINE_TEXT
<i>print-break</i>	print_break	PRINT_BREAK
<i>declare</i>	declare	DECLARE
<i>perform</i>	perform	PERFORM
<i>fake-declare</i>	fake_declare	FAKE_DECLARE
<i>fake-perform</i>	fake_perform	FAKE_PERFORM
<i>fixture-member</i>	fixture_member	FIXTURE_MEMBER
<i>fixture-init</i>	fixture_init	FIXTURE_INIT
<i>with-data</i>	with_data	WITH_DATA
<i>check-try</i>	check_try	CHECK_TRY
<i>catch</i>	_catch	CATCH
<i>show-value</i>	show_value	SHOW_VALUE
<i>show-multiline-value</i>	show_multiline_value	SHOW_MULTILINE_VALUE
<i>show-scope</i>	show_scope	SHOW_SCOPE
<i>with-declare</i>	with_declare	WITH_DECLARE
<i>define-approx-epsilon</i>	define_approx_epsilon	DEFINE_APPROX_EPSILON
<i>set-approx-epsilon</i>	set_approx_epsilon	SET_APPROX_EPSILON
<i>show-approx-epsilon</i>	show_approx_epsilon	SHOW_APPROX_EPSILON
<i>check</i>	check	CHECK
<i>equal</i>	_equal	EQUAL
<i>not-equal</i>	_not_equal	NOT_EQUAL
<i>approx</i>	_approx	APPROX
<i>not-approx</i>	_not_approx	NOT_APPROX
<i>tol</i>	_tol	TOL
<i>verify</i>	_verify	VERIFY
<i>not-verify</i>	_not_verify	NOT_VERIFY
<i>true</i>	_true	TRUE
<i>false</i>	_false	FALSE
<i>predicate</i>	predicate	PREDICATE
<i>predicate-a</i>	predicate_a	PREDICATE_A
<i>predicate-c-a</i>	predicate_c_a	PREDICATE_C_A
<i>mock-method</i>	mock_method	MOCK_METHOD
<i>wrap-method</i>	wrap_method	WRAP_METHOD
<i>logged-args</i>	logged_args	LOGGED_ARGS
<i>logged-ret</i>	logged_ret	LOGGED_RET
<i>logged-ret-args</i>	logged_ret_args	LOGGED_RET_ARGS
<i>log-size</i>	log_size	LOG_SIZE
<i>schedule-ret</i>	schedule_ret	SCHEDULE_RET
<i>set-ret-default</i>	set_ret_default	SET_RET_DEFAULT
<i>get-call</i>	get_call	GET_CALL
<i>pop-call</i>	pop_call	POP_CALL
<i>call-ledger-report-to</i>	call_ledger_report_to	CALL_LEDGER_REPORT_TO

Table 1: Macro names in the default styles

In the following sections, matching “lc” and “uc” test instruction names are shown in the subsection titles, and all examples are given first in the “lc”, cluttered style, then in the “uc”, open style.

3 How to use Testudo

First, choose a macro style (see §2). Then, generate a header file for your style using the script “generate_style” (this is done automatically if you use “make”). Finally, include the generated header file in your C++ test source file. For instance, if you want to use the default upper-case macro style, do

```
1 #include "testudo_uc.h"
```

Additionally, if you require support for STL types, include “testudo_ext.h”:

```
1 #include "testudo_ext.h"
```

You can give your C++ test source file any extension, but if you give it the “.ttd” extension, “make” will detect it as a test file and compile it correctly.

4 Tests and test hierarchies

Tests are organised in a tree where each node, be it leaf or not, may or may not have an associated test. You can choose to execute the tests in a sub-tree rooted at any node.

In this context, *declaring* a test node means mentioning it by full name. If a test node with the appropriate full name exists already, the mention refers to it. Otherwise, a new test node is created, with no title, test function, or priority. On the other hand, *defining* a test node or a test means giving it full contents, including at least a name and a title, but possibly also a test function or a priority.

Test nodes and tests are declared and defined in any number of C++ translation units; each declaration or definition causes an action on the test tree (the creation or configuration of a node). We can’t control the order in which translation units are executed, but Testudo gives you means to control the order of execution of tests.

Nodes you define as siblings (i.e., with the same parent) in a given translation unit will be created in the order they are mentioned in the code, and will be run in that same order. For sibling nodes that aren’t defined in the same translation unit, you can control the order in which they are executed by giving each one a different priority (a non-negative number); nodes with lower priority come first. If two sibling nodes have the same priority, Testudo resorts to alphabetical ordering.

Test nodes have two kinds of names: the name and the full name. The “name” proper is a string with no periods or spaces in it, and represents the name the node has *relative to its parent*. A test node can’t have two children with the same name. The full name of a test node is obtained by chaining all the names of its ancestors in order, finishing with its own name, separated by periods. The *title* of a test node is an arbitrary string.

4.1 Non-top test nodes

When you declare or define a test node whose parent has been defined in the current translation unit³, use the “define-test-node” or “define-test” syntaxes. As explained before, the execution order is the order in the code, so no priority is specified.

So, for instance, if you have already defined a test node called “tricorder”, here’s how to define a child test node called “medical”, with the title “medical capabilities”:

```
1 define_test_node(tricorder, medical, "medical capabilities");
```

```
1 DEFINE_TEST_NODE(tricorder, medical, "medical capabilities");
```

For a test (a test node with a test function), the syntax includes the definition of the test function itself, as if it were a regular C++ function, only with its declarator part (the one where you specify the return type, the function name, and the parameters) replaced by a Testudo macro. If you have already defined a test node called “medical”, here’s how to define a child test called “switch_on”, titled “switch on after creation”, that checks a tricorder medical sub-unit is off upon creation of the tricorder, and switches on appropriately:

```
1 define_test(medical, switch_on, "switch on after") {  
2   declare(Tricorder t); // see §5 for the test steps syntax  
3   check(not t.medical.is_on()) _true;  
4   perform(t.medical.push_on_button());  
5   check(t.medical.is_on()) _true;  
6 }
```

```
1 DEFINE_TEST(medical, switch_on, "switch on after")  
2 {  
3   DECLARE(Tricorder t); // see §5 for the test steps syntax  
4   CHECK(not t.medical.is_on()) TRUE;  
5   PERFORM(t.medical.push_on_button());  
6   CHECK(t.medical.is_on()) TRUE;  
7 }
```

4.2 Top test nodes

Top test nodes are test nodes whose parent you haven’t defined in the same translation unit. You mention their parent by their full name, and Testudo makes sure the parent exists before the new child is defined. Test nodes created by mentioning their full name begin as *unconfigured* test nodes; that’s ok, and it won’t cause any harm, but it means that you’re not controlling their relative order to other test nodes (the order is still deterministic, though, since they get a default priority of 0), and they don’t have any title. You can *configure* an unconfigured test node by simply defining it, preferably at an appropriately higher-level translation unit, for clarity.

³I call these non-top test nodes in opposition to top test nodes; see §4.2. Another name could have been “child nodes”, since they are children to parents that have been defined in the same translation unit.

Here's how to define a top test node called "flux_capacitor", child to a test node whose full name is "outatime.delorean":

```
1 define_top_test_node("outatime.delorean", // parent full name
2                       flux_capacitor, // name
3                       "flux capacitor features", // title
4                       200); // priority
```

```
1 DEFINE_TOP_TEST_NODE("outatime.delorean", // parent full name
2                       flux_capacitor, // name
3                       "flux capacitor features", // title
4                       200); // priority
```

You can also define a top test (a top test node with a test function). So, here's how to define a test called "doors_closed_on_start", child to a test node whose full name is "outatime.delorean":

```
1 define_top_test("outatime.delorean", // parent full name
2                 doors_closed_on_start, // name
3                 "doors closed after construction", // title
4                 150) { // priority
5     declare(Delorean d);
6     check(not d.left_door.is_open()) _true;
7     check(not d.right_door.is_open()) _true;
8 }
```

```
1 DEFINE_TOP_TEST("outatime.delorean", // parent full name
2                 doors_closed_on_start, // name
3                 "doors closed after construction", // title
4                 150) // priority
5 {
6     DECLARE(Delorean d);
7     CHECK(not d.left_door.is_open()) TRUE;
8     CHECK(not d.right_door.is_open()) TRUE;
9 }
```

5 Test steps

You write a test function by declaring variables, performing actions, and checking their results. You must do these things in a particular way so they end up in the test report. This results in a test report that is easily readable and contains all information needed to understand the test. You can additionally print messages to aid the comprehension, or display separators to show a shift in the test focus.

5.1 Declarations and actions

Declarations and actions are about code that makes it verbatim through the macros into the final C++ code for the test program. The difference is that declarations introduce names that are valid until the end of the scope, and actions don't. An easy

way to distinguish them is to ask yourself if the instruction has the same effect if you surround it in curly braces; if it does, it's an action. Otherwise, it's a declaration.

Examples of declarations are:

- variable declarations

```
1 int n=7;
```

- using-declarations

```
1 using std::vector;
```

- using-directives

```
1 using namespace std;
```

Examples of actions are:

- variable assignments

```
1 n=8;
```

- function invocations

```
1 std::sort(v.begin(), v.end());
```

5.1.1 Declaration: **declare** – **DECLARE**

All declarations in a test must be enclosed in a “declare” instruction. They will be carried out as written, and written out to the report.

```
1 declare(using namespace std);
2 declare(pair<int, double> p={2, 3.5});
```

```
1 DECLARE(using namespace std);
2 DECLARE(pair<int, double> p = { 2, 3.5 });
```

5.1.2 Action: **perform** – **PERFORM**

All non-declaration instructions in a test must be enclosed in a “perform” instruction. They will be carried out as written, and written to the report.

```
1 perform(p.first+=10);
```

```
1 PERFORM(p.first += 10);
```

5.2 Checks

A check instruction is a verification made on the value of an expression. It's outcome is true or false. If true, it counts towards the tally of succeeded checks. If false, it counts towards the tally of failed checks.

5.2.1 Checked expression: `check` – `CHECK`

An expression-check instruction starts with a “check” instruction containing the value to check (usually an expression resulting from previous actions); it must be followed by at least one continuing macro, stating what the expected value of the expression is, and how the comparison is done.

A check where the argument to the “check” instruction is *invalid* always fails, no matter what the continuing macro says. Values are considered *valid* by default, but you can tailor the definition of validity for a type to suit your needs, by defining an “`is_valid()`” method or function (see §11.1). The effect cannot be achieved by checking for validity in the definition of “`operator==()`”, because if you did, negated checks (“false” macro, “not-equal” macro, et cetera) on invalid values would be successful.

5.2.2 Check the expression is true: `_true` – `TRUE`

In order to check the value of the expression for trueness, attach the “true” macro to the “check” instruction: the expression is converted to “bool”, and the test check is successful if and only if the resulting bool is true.

```
1 check(dispersion_rate<(1./accuracy))_true;  
1 CHECK(dispersion_rate < (1. / accuracy)) TRUE;
```

5.2.3 Check the expression is false: `_false` – `FALSE`

The opposite of the “true” macro is the “false” macro. With the “false” macro, the test output marks this check with the word “nay”⁴.

```
1 check(dispersion_rate<(1./accuracy))_false;  
1 CHECK(dispersion_rate < (1. / accuracy)) FALSE;
```

5.2.4 Check the expression is equal to a reference: `_equal` – `EQUAL`

In order to check whether the value of the expression is equal to a reference, attach the “equal” macro to the “check” instruction, giving it an argument stating the reference value. Testudo uses “`operator==()`” to compare the two values, and the test check is successful if and only if the result of the comparison is true.

```
1 check(captain_age)_equal(26+10);  
1 CHECK(captain_age) EQUAL(26 + 10);
```

The opposite of the “equal” macro is the “not-equal” macro:

`_not_equal` – `NOT_EQUAL`

⁴I’ve chosen the word “nay” rather than the word “not” to avoid confusion: imagine seeing a check for “`not a and b`”, which should be equivalent to “`(not a) and b`” rather than “`not (a and b)`”. No such confusion with “`nay a and b`”.

The expression in the “equal” and “not-equal” macros is prefixed with the type of the expression in the “check” instruction, so you can leave out the type in many cases that would otherwise require more verbosity. So, for instance, if “inventory” has type “map<string, int>”, the following two checks are equivalent:

```

1 check(inventory)
2   _equal(map<string, int>{{"apple", 2}, {"banana", 3}});
3 check(inventory)_equal({{"apple", 2}, {"banana", 3}});

1 CHECK(inventory)
2   EQUAL(map<string, int>{ { "apple", 2 }, { "banana", 3 } });
3 CHECK(inventory) EQUAL({ { "apple", 2 }, { "banana", 3 } });

```

5.2.5 Check the expression is near a reference: `_approx` – `APPROX`

For non-discrete types (floating-point, for instance), checking for equality isn’t useful, as tiny rounding errors would make such a test fail⁵. What you want instead is to check whether the value of the expression is near a reference. In order to do this, attach the “approx” macro to the “check” instruction, giving it an argument stating the reference value. Testudo uses “absdiff()” (see §11.4) to compute the absolute distance between the two values. The test check is successful if and only if that absolute distance is less than a certain tolerance.

```

1 check(computed_pi)_approx(2.*asin(1.));

1 CHECK(computed_pi) APPROX(2. * asin(1.));

```

The opposite of the “approx” macro is the “not-approx” macro:

`_not_approx` – `NOT_APPROX`

By default, the default tolerance used for nearness checks is taken from a variable named “approx_epsilon”, but we’ll call it “ ϵ ” hereafter. This variable is accessible in all tests. When it isn’t available in a given scope (such as in an auxiliary function used by a test), it must be created for the nearness checks to compile.

The default value for “approx_epsilon” is “1e-6” (one millionth), but it can be changed and inspected.

Similar to the “equal” macro, the expression in the “approx” and “not-approx” macros is prefixed with the type of the expression in the “check” instruction.

Define a value for ϵ : `define_approx_epsilon` – `DEFINE_APPROX_EPSILON`

In order to define ϵ (in a situation where it isn’t available), use the “define approx epsilon” macro with the initial value for ϵ .

```

1 define_approx_epsilon(1e-3); // one thousandth

1 DEFINE_APPROX_EPSILON(1e-3); // one thousandth

```

⁵In fact, when working with floating-point magnitudes, you should instruct your compiler to treat equality comparisons between floating-point values as errors.

Set the value of ε : **set_approx_epsilon** – **SET_APPROX_EPSILON**

When ε is accessible (in all tests, for instance), you can change its value with the “set approx epsilon” macro, giving it the new value. The new value will be used for all subsequent nearness checks, until it is changed again.

```
1 set_approx_epsilon(1e-3); // one thousandth
1 SET_APPROX_EPSILON(1e-3); // one thousandth
```

Show the value of ε : **show_approx_epsilon** – **SHOW_APPROX_EPSILON**

You can also show what the value of ε is in the test report, by using the “show approx epsilon” macro.

```
1 show_approx_epsilon();
1 SHOW_APPROX_EPSILON();
```

Set a specific tolerance for nearness: **_tol** – **TOL**

You can also choose to override the default tolerance value for a specific check, by attaching the “tol” macro with the tolerance value before the “approx” macro.

```
1 check(area)_tol(.1)_approx(3.5); // use one-tenth tolerance just this once
1 CHECK(area) TOL(.1) APPROX(3.5); // use one-tenth tolerance just this once
```

5.2.6 Predicate checks: **_verify** – **VERIFY**

The checked value can also be tested with a *predicate*. In Testudo, a predicate is a function object that accepts one argument and returns a bool. A predicate can be constructed using in one of three ways:

- from a lambda expression: **predicate** – **PREDICATE**

```
1 declare(auto is_negative=
2     predicate([](int x) { return x<0; }));
1 DECLARE(auto is_negative =
2     PREDICATE([](int x) { return x < 0; }));
```

- from a bool expression: **predicate_a** – **PREDICATE_A** (the expression must use the parameter name “a”)

```
1 declare(auto is_even=predicate_a((a%2)==0));
1 DECLARE(auto is_even = PREDICATE_A((a % 2) == 0));
```

- from a bool expression with capture: **predicate_c_a** – **PREDICATE_C_A** (like the previous one, but a first parenthesised⁶ argument gives the list of captures)

⁶Captures are usually surrounded with square brackets, but in this syntax, they must be surrounded by regular brackets.

```

1 declare(auto is_multiple_of=
2     [](auto n)
3     { return predicate_c_a((n), (a%n)==0); });

```

```

1 DECLARE(auto is_multiple_of =
2     [](auto n)
3     { return PREDICATE_C_A((n), (a % n) == 0); });

```

Using such predicate objects, the “check-verify” syntax checks whether the checked value verifies the predicate:

```

1 check(number_of_cards)_verify(is_even);
2 check(score)_verify(is_multiple_of(5));

```

```

1 CHECK(number_of_cards) VERIFY(is_even);
2 CHECK(score) VERIFY(is_multiple_of(5));

```

Predicates can be combined with the logical operators “not”, “and”, and “or”:

```

1 check(iterations)
2     _verify(not is_negative
3             and (is_even or is_multiple_of(5)));

```

```

1 CHECK(iterations)
2     VERIFY(not is_negative
3             and (is_even or is_multiple_of(5)));

```

The “check-verify” syntax, when it is the natural means of expression for a check, is superior to the “check-true” syntax because

- in case of failure, it shows the value of the checked value;
- it makes it possible to define and reuse concise predicates with good names, that improve clarity and raise the level of abstraction of the tests.

The opposite of the “verify” macro is the “not-verify” macro:

`_not_verify` – `NOT_VERIFY`

5.2.7 Exception checks: **`check_try _catch` – `CHECK_TRY _CATCH`**

Instead of checking the value of an expression, you can also check that evaluating an expression throws an exception. This is done with the “check-try-catch” instruction, passing it the expression that is expected to throw. Testudo will run the expression within a try-block; if an exception with the expected type (“std::exception” by default) is thrown, the exception is reported, and the test step is successful. If no exception is thrown, the test step is failed. If an exception with an unexpected type is thrown, the test step is failed, and an unexpected exception error is reported (see §5.7).

```

1 declare(list<int> numbers);
2 check_try(numbers.front())_catch();

```

```

1 DECLARE(list<int> numbers);
2 CHECK_TRY(numbers.front()) CATCH();

```

The “check-try-catch” instructions expects a exception with a type derived from “std::exception” by default. You can change the expected exception type by passing it as an argument to the “catch” part of the instruction:

```
1 declare(my_list<int> numbers);
2 check_try(numbers.front())_catch(my_list_exception);

1 DECLARE(my_list<int> numbers);
2 CHECK_TRY(numbers.front()) CATCH(my_list_exception);
```

5.3 Adding information to the report

Various pieces of information can be added to the report about the execution of the test, to help the human reader.

5.3.1 Showing values

You can show the value of an expression in the report. It doesn't add to the tally of tests, but it can add clarity about what's going on.

Show a plain value: `show_value` – `SHOW_VALUE`

The “show value” instruction shows the value of its argument inline.

```
1 show_value(helicopter.remaining_fuel());

1 SHOW_VALUE(helicopter.remaining_fuel());
```

Show a multiline value: `show_multiline_value` – `SHOW_MULTILINE_VALUE`

If the value to show may contain newlines, the format used by the “show multiline value” macro will be clearer: it will be suited for multiline values, and it will respect the newlines.

```
1 show_multiline_value(radio.communication_log());

1 SHOW_MULTILINE_VALUE(radio.communication_log());
```

5.3.2 Scopes: `show_scope` – `SHOW_SCOPE`

In some situations, such as when we want to check the effect of the destruction of an object that's gone out of scope, it can be useful to show where a scope begins and ends. This is done by using the “show scope” macro just before the opening brace of the scope, which writes a message to the report about the new scope. You don't have to add anything to the closing brace: Testudo will automatically write a message when the scope ends.

Most of the time, with short scopes, you don't need to name the scope. This is done by using the “show scope” macro without any arguments. If the scope is longer, it may be clearer to name it, since the scope's begin and end messages will display its name. This is done by passing the scope name to the “show scope” macro.

```

1 declare(LoggedDestruction ld1("1"));
2 check(LoggedDestruction::n_destructions())_equal(0);
3 show_scope("outer scope") { // named scope
4     declare(LoggedDestruction ld2("2"));
5     show_scope() { // unnamed scope
6         declare(LoggedDestruction ld3("3"));
7     }
8     check(LoggedDestruction::n_destructions())_equal(1);
9 }
10 check(LoggedDestruction::n_destructions())_equal(2);

1 DECLARE(LoggedDestruction ld1("1"));
2 CHECK(LoggedDestruction::n_destructions()) EQUAL(0);
3 SHOW_SCOPE("outer scope") // named scope
4 {
5     DECLARE(LoggedDestruction ld2("2"));
6     SHOW_SCOPE() // unnamed scope
7     {
8         DECLARE(LoggedDestruction ld3("3"));
9     }
10    CHECK(LoggedDestruction::n_destructions()) EQUAL(1);
11 }
12 CHECK(LoggedDestruction::n_destructions()) EQUAL(2);

```

5.3.3 With-declare scopes: `with_declare` – `WITH_DECLARE`

Sometimes, you want to perform a sequence of steps using a temporary value that you want to (or can) compute only once. This is achieved by creating a scope with the “with-declare” macro. It works like the “show-scope” macro, but instead of a name, its argument is the declaration that will be in place in the macro. The declaration is identical to what you would give the “declare” macro. You can also use the “with-declare” macro with no braces if the scope contains only one statement.

```

1 with_declare(auto answer=client.request("get license")) {
2     check(answer.valid)_true;
3     check(answer.text)_equal("res://punk_license");
4 }
5 with_declare(auto answer=client.request("reset"))
6     check(answer.valid)_true;

1 WITH_DECLARE(auto answer=client.request("get license"))
2 {
3     CHECK(answer.valid) TRUE;
4     CHECK(answer.text) EQUAL("res://punk_license");
5 }
6 WITH_DECLARE(auto answer=client.request("reset"))
7     CHECK(answer.valid) TRUE;

```

If you need to use the “with-declare” macro with several variable declarations, you can always use a “structured binding declaration”:

```

1 with_declare(auto [action, occurrences]=tuple{"sin", 77*7}) {
2     check(action)_equal("sin");
3     check(occurrences)_equal(539);
4 }

1 WITH_DECLARE(auto [action, occurrences]=tuple{"sin", 77*7})
2 {
3     CHECK(action) EQUAL("sin");
4     CHECK(occurrences) EQUAL(539);
5 }

```

5.4 Identifying steps

You can identify certain steps, to make it easier to follow their evolution. It's particularly appropriate for checks⁷. Step IDs are relative to the test they're in, and their full name is prefixed with the full name of the current test, so they must include only the necessary information within the test scope. A step ID must be a valid C++ variable name.

The “step id” instruction prints a tag on the report that applies to the following step in the test.

```

1 define_test(medical, switch_on, "switch on after") {
2     declare(Tricorder t);
3     step_id(init_off); // in the tricorder medical test; no need to mention it
4     check(not t.medical.is_on())_true;
5 }

1 DEFINE_TEST(medical, switch_on, "switch on after") {
2     DECLARE(Tricorder t);
3     STEP_ID(init_off); // in the tricorder medical test; no need to mention it
4     CHECK(not t.medical.is_on()) TRUE;
5 }

```

5.5 Printing fixed text and separations

You can add fixed messages to the report, to aid the comprehension of the reader. They should be considered to play the same rôle as comments in source code.

5.5.1 Print inline text: `print_text` – `PRINT_TEXT`

The “print text” instruction displays its argument inline. The argument must be a string of any kind.

```

1 print_text("the speed hasn't been updated yet");

1 PRINT_TEXT("the speed hasn't been updated yet");

```

⁷—FIXME—I will add a feature to add a section to the full reports and summary reports with the results of the identified steps; this will make it possible to make id'd-step-only diffs to follow their evolution.—

5.5.2 Print multiline text:

print_multiline_text – **PRINT_MULTILINE_TEXT**

If the message contains newlines, use the “print multiline text” instruction instead: it’s suited for multiline text, and respects the newlines.

```
1 print_multiline_text("the order will be:\n"
2                       " 1. insert all elements\n"
3                       " 2. sort the container");

1 PRINT_MULTILINE_TEXT("the order will be:\n"
2                       " 1. insert all elements\n"
3                       " 2. sort the container");
```

5.5.3 Print a break: **print_break** – **PRINT_BREAK**

The “print break” instruction just prints a break, to show a change of focus in the test report.

```
1 print_break();

1 PRINT_BREAK();
```

5.6 Fake declarations and actions

Sometimes, you want to record a declaration on an action that won’t be carried out at all, as if it had. This can be the case, for instance, when there’s an instruction that makes sense for most compilation settings, but there’s a certain combination of compilation options where it doesn’t; in that case, for that compilation, you’ll want to record a fake instruction, and then silently carry out explicitly a replacement instruction, with no test instruction macro, so that test reports are the same across compilation settings.

5.6.1 Fake declaration: **fake_declare** – **FAKE_DECLARE**

You can report a fake declaration by enclosing an instruction in a “fake-declare instruction. The instruction will be written to the report, exactly as if it had been carried out, except it won’t have.

```
1 #ifdef DEBUGGING
2 declare(LoggedInt n_cases); // optimised to int in production
3 #else
4 fake_declare(LoggedInt n_cases);
5 int n_cases; // replacement declaration (naked)
6 #endif

1 #ifdef DEBUGGING
2 DECLARE(LoggedInt n_cases); // optimised to int in production
3 #else
4 FAKE_DECLARE(LoggedInt n_cases);
5 int n_cases; // replacement declaration (naked)
6 #endif
```

5.6.2 Fake action: `fake_perform` – `FAKE_PERFORM`

You can report a fake action by enclosing an instruction in a “fake-perform” instruction. The instruction will be written to the report, exactly as if it had been carried out, except it won’t have.

```
1 #ifdef DEBUGGING
2 perform(terrible_pointer.report()); // won't work in production
3 #else
4 fake_perform(terrible_p.report());
5 log << "terrible_p reported" << endl; // replacement action (naked)
6 #endif
```

```
1 #ifdef DEBUGGING
2 PERFORM(terrible_pointer.report()); // won't work in production
3 #else
4 FAKE_PERFORM(terrible_p.report());
5 log << "terrible_p reported" << endl; // replacement action (naked)
6 #endif
```

5.7 Unexpected exceptions

If an unexpected exception (i.e., an exception not in a “check-try-catch” instruction; see §5.2.7, or one in a “check-try-catch” instruction that isn’t caught because it isn’t the right type) is thrown in the course of a test, that particular test ends immediately, a description of the exception is written to the report, with a conspicuously coloured (where available) [ERR-] flag, and the test is marked as having one error. Then, the execution of the rest of the tests resumes. Other tests are not affected by the exception, and are executed as normal.

Errors aren’t the same as failed checks. They get their own tally. Errors aren’t an expected situation, even in a failed test that you may be using to do TDD. Therefore, test summaries mention the number of errors only when there is at least one error. A test that has at least one error isn’t marked with the [FAIL] flag, but rather with [ERR-].

5.8 Fatal errors

Fatal errors (i.e., errors that cause immediate termination of the executable, rather than throwing an exception) are something Testudo can’t do anything about. They will terminate the whole test suite execution. There’s a couple of things you can do to investigate what’s going on. First, you can run the tests using a synchronous format like “color_text” (see §1), to see where the execution stops (which test and which test step). You can then go investigate and fix the error, or run the tests excluding that particular test to know what the current situation is, barring the failing test. You can also run the test with a debugger, but in that case, be aware that Testudo’s macros add management code (classes, methods, functions, variables, statements...) to your instructions to achieve the intended results, and

their workings may be confusing, so try to pay no attention to that man behind the curtain.

6 Test-aware functions

You may want to perform the same test several times, with only minor variations, such as a type or a value⁸. This can be done with *test-aware* functions (or methods, or classes). Test-awareness involves:

- getting an object of type “`testudo::test_management_t`”;
- making it available as a variable called “`test_management`” in the scope where test macros are used.

The “`test_management_t`” is always available as “`test_management`” in a test definition.

Here’s an example where we test emptiness of two container-like classes:

```
1 template <typename Container>
2 void test_container_emptiness(
3     testudo::test_management_t test_management,
4     Container &container) {
5     check(container.empty())_true;
6 }
7
8 class Cauldron { ... };
9
10 define_test(container, emptiness_c, "Cauldron emptiness") {
11     declare(Cauldron container);
12     test_container_emptiness(test_management, container);
13 }
14
15 class Marmite { ... };
16
17 define_test(container, emptiness_m, "Marmite emptiness") {
18     declare(Marmite container);
19     test_container_emptiness(test_management, container);
20 }
```

```
1 template <typename Container>
2 void test_container_emptiness(
3     testudo::test_management_t test_management,
4     Container &container)
5 {
6     CHECK(container.empty()) TRUE;
7 }
8
9 class Cauldron { ... };
10
11 define_test(container, emptiness_c, "Cauldron emptiness")
12 {
```

⁸If what changes is a value, consider first whether “with-data” loops (§8) fit your need.


```

13 DECLARE(Cauldron container);
14 test_container_emptyiness(test_management, container);
15 }
16
17 class Marmite { ... };
18
19 define_test(container, emptyiness_m, "Marmite emptiness")
20 {
21     DECLARE(Marmite container);
22     test_container_emptyiness(test_management, container);
23 }

```

7 Fixtures

Test fixtures gather common functionality needed by several tests, most commonly test setup and test teardown.

7.1 Definition

This is how fixtures are implemented in Testudo. If you want to code a fixture, you have to code a class that derives from “testudo::Fixture”. Its constructor must accept an object of type “testudo::test_management_t, and pass it out to the constructor of “testudo::Fixture”. The constructor is the setup procedure; the destructor, if you code it, is the teardown procedure.

Here’s an example:

```

1 struct OutATimeFixture : testudo::Fixture {
2     OutATimeFixture(testudo::test_management_t test_management)
3         : Fixture(test_management)
4         { perform(d=new Delorean); }
5     ~OutATimeFixture()
6         { perform(delete d); }
7     Delorean *d; // dumb pointer, just so we can show a teardown procedure
8 };

```

```

1 struct OutATimeFixture : testudo::Fixture
2 {
3     OutATimeFixture(testudo::test_management_t test_management)
4         : Fixture(test_management)
5     {
6         PERFORM(d=new Delorean);
7     }
8     ~OutATimeFixture()
9     {
10         PERFORM(delete d);
11     }
12     Delorean *d; // dumb pointer, just so we can show a teardown procedure
13 };

```

7.2 Usage

In order to have a test use a fixture, you have to add the “with-fixture” macro or the “visible-fixture” macro just after the title in the definition (before other arguments if any); this works both with non-top and with top tests. Like this:

```
1 define_test(delorean,
2             engine_starts_off, "engine is off at start",
3             with_fixture(OutATimeFixture)) // with-fixture
4 {
5     check(not d->engine.is_running())_true;
6 }
7
8 define_test(delorean,
9             no_pu_at_start, "there's no Plutonium initially",
10            visible_fixture(OutATimeFixture)) // visible-fixture
11 {
12     check(d->pu())_approx(0.);
13 }
```

```
1 DEFINE_TEST(delorean,
2             engine_starts_off, "engine is off at start",
3             WITH_FIXTURE(OutATimeFixture)) // with-fixture
4 {
5     CHECK(not d->engine.is_running()) TRUE;
6 }
7
8 DEFINE_TEST(delorean,
9             no_pu_at_start, "there's no Plutonium initially",
10            VISIBLE_FIXTURE(OutATimeFixture)) // visible-fixture
11 {
12     CHECK(d->pu()) APPROX(0.);
13 }
```

If you use the “with-fixture” macro, Testudo macros used in the fixture implementation, whether in the constructor, the destructor, or any other method, aren’t logged to the test report. If you use the “visible-fixture” instead, Testudo macros in the fixture implementation are logged as usual, and the end of the constructor and the beginning of the destructor are logged.

You can code other methods in a fixture, and you can call them from test functions. In fact, the test function ends up being one of the methods of the fixture, so that’s why and how.

7.3 Fixture members and their initialisation

You should declare fixture attributes (member variables) and their initialisation with Testudo macros, so that they’re logged if the fixture is used with the “visible-fixture” macro. This is done by using the “fixture-member” macro for attribute

declarations⁹, and the “`fixture-init`” macro for attribute initialisations. The “`fixture-member`” macro encloses an attribute declaration of any kind: it can contain several declarations, or default values. The “`fixture-init`” macro accepts as its first argument the name of the attribute, and as its second argument its initial value. Here’s an example of usage:

```
1 struct NumbersFixture : testudo::Fixture {
2     NumbersFixture(testudo::test_management_t test_management)
3         : Fixture(test_management),
4           fixture_init(x, 1.), fixture_init(z, 3.14) { }
5     fixture_member(double x);
6     fixture_member(double y=-2.5, z);
7 };
```

```
1 struct NumbersFixture
2     : testudo::Fixture
3 {
4     NumbersFixture(testudo::test_management_t test_management)
5         : Fixture(test_management),
6           FIXTURE_INIT(x, 1.),
7           FIXTURE_INIT(z, 3.14) { }
8     FIXTURE_MEMBER(double x);
9     FIXTURE_MEMBER(double y=-2.5, z);
10 };
```

Other actions in the fixture implementation are enclosed in Testudo macros in the usual way (with the “`declare`” macro, the “`perform`” macro, and so on).

8 With-data loops: `with_data` – `WITH_DATA`

You may need to perform the same checks on many values. This happens, for instance, if you’re checking a certain property holds for all possible values of a type¹⁰; you would implement this by creating a large list of such values, and applying to them the same test steps. This can be easily done with the “`with-data`” loop syntax.

This syntax accepts two arguments: the name of the variable that’s going to be tested, and a container with all values to test. The container can be stored in advance in a variable, or built inline. The “`with-data`” macro is followed by the test to perform on the variable values. The test can be a single check instruction, or a brace-enclosed sequence of instructions and checks. The report output shows the variable name and the container, followed by the instructions and checks to be performed (only once), and one fail line per failed value; if all values pass the tests, an additional line is output with an OK flag for “all successful”.

You can chain “`with-data`” macros as you would chain “`for`” loops. Testudo is aware of chained “`with-data`” macros and will output parsimonious reports for them, grouping failed coordinated values into single-line fail reports, and outputting only one success line in case of success.

⁹The “`fixture-member`” macro has a limitation: there can only be one “`fixture-member`” macro usage per source code line. This isn’t a big deal, because that’s how you’ll use it anyway, unless you’re trying to have a badly packed source code, but i’m telling you just in case.

¹⁰My inner voice calls this feature “theorem checking”.

Here is an example:

```
1 declare(auto is_even=predicate_a((a%2)==0));
2 declare(list<int> even_numbers{2, 4, 8}); // even numbers
3 with_data(x, even_numbers)
4   check(x)_verify(is_even);
5 with_data(x, even_numbers) {
6   declare(int y=x+1);
7   check(y)_verify(not is_even);
8 }
9 with_data(x, even_numbers)
10   with_data(y, list<int>{2, 4, 9})
11     check(x%2)_equal(y%2); // will fail when y==9

1 DECLARE(auto is_even=PREDICATE_A((a % 2) == 0));
2 DECLARE(list<int> even_numbers{ 2, 4, 8 }); // even numbers
3 WITH_DATA(x, even_numbers)
4   CHECK(x) VERIFY(is_even);
5 WITH_DATA(x, even_numbers)
6 {
7   DECLARE(int y = x + 1);
8   CHECK(y) VERIFY(not is_even);
9 }
10 WITH_DATA(x, even_numbers)
11   WITH_DATA(y, list<int>{ 2, 4, 9 })
12     CHECK(x % 2) EQUAL(y % 2); // will fail when y==9
```

9 Mock objects

Testudo supports mock objects through its “Mock Turtle” module. Include header file “mock_turtle.h” to use it.

You can build your mock objects in any way you want: with simple method overriding, with virtual method overriding, or with no overriding and no derivation. The mock method macros define the methods precisely in accordance with your specification, so it’s up to you to decide.

Here’s Mock Turtle’s approach to mock testing:

- define mock classes using the mock-method macros “mock-method” and “wrap-method” (§9.1); this adds scheduling and logging machinery to the classes automatically;
- if necessary, set up across-method ledgers (§??);
- in a test definition, create instances of the mock classes;
- if necessary, set up across-instances ledgers (§??);
- schedule mock object behaviour as needed (§9.2);
- use mock objects as if they were the real thing;
- check logged behaviour against expected behaviour by accessing call logs and ledgers (§9.3, §9.4).

9.1 Mock-method macros

When you need to test a functionality, you may want to use mock implementations for some of the parts that are needed but aren't the focus of the test. Typically, these parts are functionality that is external to your code, or costly to run, or slow, or having dependencies on unavailable resources. The usual way to achieve this is in C++ by replacing the affected parts by objects that have the same interface as the problematic ones, but ad-hoc implementations tailored to the test.

How you implement a mock class depends on how the mocked class is defined, and you use it. It's usually a matter of replicating the interface or deriving from the mock class and overriding its virtual methods. The first approach would be appropriate, e.g., if the class is a template argument to function or another class. The second one is good for polymorphic designs.

Testudo mock-method macros will help you deal with the most usual cases for mock method implementations. There are two available macros to mock a method:

- the “mock-method” macro, that creates a dumb implementation, where return values for the method can be pre-scheduled, and with automatic logging of arguments and return value;
- the “wrap-method” macro, that allows you to provide a specific implementation, but still adds to it automatically the logging of arguments and return value.

Both macros allow you to specify how the method mocks its original version, whether by merely defining it, or by overriding a virtual one in the base class.

Before you use these macros, you must define the mock class. The only requirement for the mock class is that it publicly must derive from “`testudo::CallLedger`”. Here are a couple of examples, one for a mock class deriving from a virtual class, and one for a non-derived mock class:

```
1 class KettleBase {
2 public:
3     virtual ~KettleBase()=default;
4     virtual void fill(float volume)=0;
5     virtual float temperature() const=0;
6 };
7
8 class KettleMock
9     : public KettleBase,
10     public testudo::CallLedger {
11 public:
12     // method mocks and wraps overriding base methods...
13 };
14
15 class ContainerMock
16     : public testudo::CallLedger {
17 public:
18     // method mocks and wraps defining a cointainer interface...
19 };
```

9.1.1 Mocking a method: `mock_method` – `MOCK_METHOD`

In order to mock a method, use the “mock-method” macro where you would normally define the method, followed by a semicolon. This will automatically create an implementation for the method that logs all calls (argument values, return value, and order across methods and objects) to it (§9.3, §9.4), and allows you to set up a schedule for return values (§9.2).

The arguments to the “mock-method” macro are, in order (see below for an explanation of the usage of the word *parenthesised*),

- the *parenthesised* return type;
- the name of the method (or a its name and an alternative name; see below);
- a *parenthesised* list of *parenthesised* arguments;
- optional method specifiers like “const”, “override”, or “final”, separated by spaces; leave this argument out if you don’t need it.

The return type macro argument is *parenthesised*, which means it must be enclosed between “(” and “)”. This is done so the macros aren’t confused by templated types that may contain commas in them¹¹, like “`map<int, float>`”. If the return type is “void”, you still have to write it in parenthesis. So, examples of return type arguments to the “mock-method” macro are

- `(int)`
- `(void)`
- `(map<pair<int, string>, float>)`

Similarly, the list-of-arguments macro argument is a *parenthesised* list of *parenthesised* arguments. This means that, for the same reasons as for the return type argument, not only is the list enclosed in parentheses, but each argument is also enclosed in parentheses¹². For each argument, you have to specify its type, and can optionally include in the parentheses for the type an argument name. The argument name is discarded by the “mock-method” macro, but will be very important when wrapping (rather than mocking) methods with the “wrap-method” macro (see §9.1.2). Here are some examples of list-of-arguments arguments to the “mock-method”:

- `()`
- `((int))`
- `((int), (unsigned char))`
- `((int quantity))`
- `((string name), (int))`
- `((map<int, float> const &), (int id))`

The name-of-method macro argument is simply the name of the method, with no parentheses. Its associated scheduler and logger objects will be named after the method, and this will cause name clashes if you have overloaded methods in the same class. In that case, you can give your method an alternative name, meaning

¹¹I’ve seen Google Mock solves this issue in a similar way, but the parentheses are optional, meaning that if your type doesn’t contain any macro-fooling comma, you can leave them out. I prefer to make parenthesisation mandatory, because once you get used to it, you won’t be left wondering why your macro doesn’t work if you forget to use it for a type that requires it.

¹²I’m thorry thith hath become tho lithpy; there’th no other way with theepluthpluth macroth.

that, although its real name will be used for the method definition, everything else (the scheduler, the logger, and references to it in ledgers) will use the alternative name. To specify an alternative name, just replace the name-of-method argument with the name of the method and the alternative name separated by a comma, in parentheses. So, for instance, the following wouldn't work

```
1 class ImageMock
2   : public CallLedger {
3 public:
4   mock_method((void), set, ((float)));
5   mock_method((void), set, ((int)));
6 };
```

so we'll have to identify one of the "set()" methods (or both) with an alternative name:

```
1 class ImageMock
2   : public CallLedger {
3 public:
4   mock_method((void), set, ((float)));
5   mock_method((void), (set, set_int), ((int)));
6 };
```

Finally, the method-specifiers macro argument consists of what you'd put after the method signature if you were defining it directly. Things like "const" (for method constness), "override", or "final" go there, separated by spaces. If there's nothing to specify here, just leave out the macro argument (no need to add the comma after the previous argument). Here's an example with const and non-const overrides:

```
1 class KettleBase {
2 public:
3   virtual ~KettleBase()=default;
4   virtual void fill(float volume)=0;
5   virtual float temperature() const=0;
6 };
7
8 class KettleMock
9   : public KettleBase,
10     public testudo::CallLedger {
11 public:
12   mock_method((void), fill, ((float volume)), override);
13   mock_method((float, temperature, (), const override);
14 };
```

9.1.2 Wrapping a method: `wrap_method` – `WRAP_METHOD`

Wrapping a method is similar to mocking a method, but instead of having the method automatically generated, and its return values scheduled, you get to specify exactly what the method does and returns.

In order to wrap a method, use the "wrap-method" macro in the same way that you'd use the "macro-method", but instead of adding a semicolon after it, write

your method implementation in braces. The “wrap-method” is essentially taking the place of the signature of your method. You still have to add “const” before the method implementation if appropriate (but not “override” or “final”); this will amount, in most cases where you have “const” in the macro method-specifier argument, to repeat it just after the macro.

What the “wrap-method” macro affords you compared to writing your own method is automatic log and ledger handling (§9.3, §9.4), which is identical to that generated by the “mock-method” macro. Scheduling is not added to a wrapped macro, since you’ll be specifying what is returned.

Here’s an example with wrapped methods:

```

1 class Tally {
2 public:
3     virtual ~Tally()=default;
4     virtual void add_counter(int delta)=0;
5     virtual int total() const=0;
6 };
7
8 class TallyMock
9     : public Tally, public testudo::CallLedger {
10 public:
11     int counter=0;
12     wrap_method((void), add_counter, ((int delta))) {
13         counter+=delta;
14     }
15     wrap_method((int), total, (), const) const { // const twice
16         return counter;
17     }
18 };

```

9.2 How to schedule mock-method behaviour

By default, a mock-method generated by the “mock-method” macro always returns the default value of its default type. You can change this by setting different default value or by scheduling a set of values to return in sequence. You can also set it to return the result of evaluating a function with no arguments (given, for instance, as a lambda expression). The general behaviour of the scheduler when asked for a return value is this:

- while the queue of scheduled return values isn’t empty, return the next value, and pop it from the queue;
- if the queue of scheduled return values is empty, return the default return value, or the result of evaluating the default function.

You can change the default value in one of two ways:

- by assigning a value or function to the “mock-method” macro expression in the mock class definition:

```

1 class SoupMock
2     : public Soup, public CallLedger {
3 public:

```



```

4   ...
5   // by default, return true:
6   mock_method((bool), is_tasty, (), const)=true;
7   ...
8 };

```

- by using the “set-ret-default” macro as a method call on a mock object:

```

1 define_test(mock_turtle, victorian, "Victorian recipe") {
2     declare(auto soup_mock=make_shared<SoupMock>());
3     declare(bool is_tasty_now=true);
4     perform(soup_mock->set_ret_default(
5         is_tasty,
6         [&is_tasty_now]() { return is_tasty_now; });
7     ...
8 }

```

In order to schedule a sequence of values,

You can also set the default value or any of the scheduled return values to throw an exception, rather than returning a value:

9.3 How to check mock-method logs

Each mock method (mocked or wrapped) keeps a log of calls to it, with arguments and return values. After you’ve run code that uses your mock objects, you can perform checks on the logs, with the usual Testudo syntax.

“Invoking” the “logged-args” macro on a mock object will return a vector of tuples containing the args for all calls to a mock method. The syntax for the call follows the syntax for a regular method, and has as its argument the name (or alternative name) of a mock method. The vector contains a tuple for each call, and each tuple contains all the arguments of the call. So, for instance, in the following example, we’re checking that the “add_ingr” method was called five times, and we’re checking the specific arguments for all calls:

```

1 declare(auto soup_mock=make_shared<SoupMock>());
2 ...
3 // five calls, with specific arguments
4 check(soup_mock->logged_args(add_ingr))
5     _equal({{"calf brains", 4},
6             {"pork liver", 3},
7             {"water", 1},
8             {"water", 1},
9             {"water", 1}});

```

Since the value returned by the “logged-args” is a plain STL vector, you can use it in any vector-like way to check results. You can, for instance:

- check its size (although a better way is to use the “log-size” macro; see below);
- check a specific call by its number:

```

1 // check third call
2 check(soup_mock->logged_args(add_ingr)[2])
3     _equal({"water", 1});

```

- check a specific argument of a specific call:

```
1 // check first argument of third call:
2 check(get<0>(soup_mock->logged_args(add_ingr)[2]))
3 _equal("water");
```

- or more complex things like checking totals of numerical arguments.

Similarly, the “logged-ret” macro returns a vector of tuples containing the return values for all calls to a mock method¹³.

```
1 declare(auto soup_mock=make_shared<SoupMock>());
2 ...
3 // a single call, that returned true
4 check(soup_mock->logged_ret(is_tasty))_equal({{true}});
```

The “logged-ret-args” macro combines the “logged-ret” and “logged-args” macros: it returns a vector of pairs, where each pair contains a tuple with the return value and a tuple with the arguments of a call.

```
1 declare(auto soup_mock=make_shared<SoupMock>());
2 ...
3 // five calls, with specific arguments and return values
4 check(soup_mock->logged_ret_args(add_ingr))
5 _equal({{{4}, {"calf brains", 4}},
6         {{7}, {"pork liver", 3}},
7         {{8}, {"water", 1}},
8         {{9}, {"water", 1}},
9         {{10}, {"water", 1}}});
```

Finally, the “log-size” macro returns the number of calls to a mock method:

```
1 declare(auto soup_mock=make_shared<SoupMock>());
2 ...
3 // a single call
4 check(soup_mock->log_size(is_tasty))_equal(1);
```

Checks with mock-method logs lend themselves to “with-declare” scopes (§5.3.3), where the declaration is for instance the result of a “logged-ret-args” macro, and the contents are checks on that value.

9.4 How to check mock-method ledgers

In addition to per-mock-method logs (§9.3), Mock Turtle keeps track of the order in which different methods are invoked. This is what the class “CallLedger” (the one mock classes must derive from) is about.

You can even keep track of the order in which different methods are invoked *on different mock objects*, by having mock objects report to a stand-alone “CallLedger” object. This is achieved by using the “call-ledger-report-to” macro, with a first argument representing the mock object (by a reference or a pointer), and a second argument which is a pointer to the stand-alone “CallLedger” object. For purposes of logging, this “CallLedger” object will identify the mock object by the first argument to the “call-ledger-report-to” macro. Here’s an example of setup:

¹³These are tuples rather than naked return values in order to accommodate methods that return void, which have empty tuples.

```

1 declare(CallLedger cl);
2 declare(MockLevel lev1(12));
3 perform(call_ledger_report_to(lev1, &cl)); // identified as "lev1"
4 declare(auto lev2=shared_ptr<MockLevel>(47));
5 perform(call_ledger_report_to(lev2, &cl)); // identified as "lev2"

```

After you’ve run code that uses your mock objects, you can check the call ledgers. Since a ledger logs calls to methods with different signatures, you can’t access a ledger in one call. You have to use a kind of iterator instead, that we’ll call the *call ledger iterator*. You obtain the call ledger iterator by passing the call ledger to the “`testudo::iterate()`” function. Initially, the iterator points to the first ledger entry. This is what you can do with a call ledger “`it`”:

- increment it, with “`it.next()`”;
- check if it’s pointing past the end of the ledger, with “`it.done()`”;
- check if it’s *not* pointing past the end, by converting it to bool;
- reset it to point to the first ledger entry, with “`it.reset()`”;
- get the name of the mock object of the call it points to, with “`it.mock_name()`”;
- get the name of the method of the call it points to, with “`it.method_name()`”;
- get an object containing a tuple with the return value and a tuple with the arguments of the call it points to; this is called a *call record*; you must know what the mock object and the method were for the call, and the operation consists of “invoking” the “get-call” macro on the iterator, passing it a reference to the expected mock object, and the name (or alternative name) of the expected method; if the expected mock object or the method are not the actual ones, the returned object is *invalid*, and all checks performed on it will fail; here’s an example of usage of the “get-call” macro:

```

1 with_declare(auto call=it.get_call(*soup_mock, add_ingr))
2   check(call)_equal({7}, {"water", 1});

```

- *pop* the call record it points to, by using the “pop-call” macro on the iterator, rather than the “get-call” macro; these macros have the same arguments and return the same value, but “pop-call” automatically increments the iterator *if the returned call record is valid*:

```

1 with_declare(auto call=it.pop_call(*soup_mock, add_ingr))
2   check(call)_equal({7}, {"water", 1});

```

You can perform the following operations on a call record “`cr`”:

- get the return value (if not void), with “`cr.ret()`”;
- get a tuple with the arguments, with “`cr.args()`”;
- get the method name, with “`cr.method_name`”;
- get the validity of the call record, with “`is_valid()`”.

You can also get a human-readable string listing of the recorded calls for a call ledger “`cl`”, with “`print_calls(cl.calls())`”; each line shows the names of the mock object and mock method, and a number stating the zero-based order the call has on the particular mock-method log; you can print this string with “`show_multiline_value(print_calls(cl.calls()))`”. This can be useful when designing or debugging a test.

Similarly to mock-method logs, checks with mock-method ledgers lend themselves to “with-declare” scopes (§5.3.3).

Here’s an example with mock-method ledgers:

```
1 declare(auto soup_mock=make_shared<SoupMock>());
2 ...
3 with_declare(auto it=iterate(soup_mock)) {
4     with_declare(auto call=it.pop_call(*soup_mock, add_ingr)) {
5         check(call.ret())_equal(4);
6         check(call.args())_equal("calf brains", 4);
7     }
8     ...
9     with_declare(auto call=it.pop_call(*soup_mock, temp))
10        check(call.ret())_equal(20);
11    ...
12    with_declare(auto call=it.pop_call(*soup_mock, is_tasty))
13        check(call.ret())_true;
14    // check it was the last call:
15    check(it.done())_true;
16 }
```

Here’s an example involving the “call-ledger-report-to” macro:

```
1 declare(CallLedger cl);
2 declare(MockLevel lev1(12));
3 perform(call_ledger_report_to(lev1, &cl));
4 declare(MockLevel lev2(47));
5 perform(call_ledger_report_to(lev2, &cl));
6 perform(equalize(lev1, lev2)); // this is the function we’re checking
7 check(lev1.readout())_equal(52);
8 check(lev2.readout())_equal(52);
9 declare(auto it=iterate(cl));
10 // fast forward to the first invocation to “large_up()”:
11 perform(while (it.method_name() not_eq "large_up")
12         it.next());
13 check(it.mock_name())_equal("lev1");
14 check(it.method_name())_equal("large_up");
15 perform(it.next());
16 check(it.mock_name())_equal("lev1");
17 check(it.method_name())_equal("readout");
```

10 Testudo support for STL types

```
1 #include "testudo_ext.h"
```

11 Adding Testudo support for your types

```
1 #include "testudo_base.h"
```

II.1 Validity

Implement “`t.is_valid()`” or “`is_valid(t)`”.

II.2 Textual representation

Implement “`ostream << t`” or “`to_text(t)`”.

II.3 Equality

Implement “`t==u`”.

II.4 Difference between two values

Implement “`double absdiff(t, u)`”.

A Testudo installation

Dependencies:

- a C++17 compiler (duh);
- UNIX, bash, sed, awk (used by scripts);
- m4 (for generation of “mock_turtle_macro_n.h”);
- make (but you can use your own alternative);
- xsltproc (for XML-to-coloured-text translation);
- L^AT_EX, rubber (to generate the PDF documentation).

Run:

- “make diff_test” to diff-check the XML-to-coloured-text output against the expected result;
- “make diff_tests” to diff-check the XML-to-coloured-text output against the expected result and against direct-to-coloured-text output;
- “make” to generate the PDF documentation.

B Editor configuration

You can configure your editor to colour Testudo keywords (as defined by the test style file you’re using).

C Using your own test definition and test instruction names

Write a test style file, similar to “lc.tst” or “cp.tst” (use one of them as a template, and adapt the second half of each line). Ĉu esparantaj ĉapelitaj literoj funkcias por makronomoj? Tio estus interesa uzo de stilpersonigo.