Testudo, an automatic test system for C++ code

(version 0.7)

Miguel González Cuadrado
<mgcuadrado@gmail.com>

24th January 2021

Contents

List of Figures

List of Tables

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 dense 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 (§5);
- execution of selected tests;
- easy definition of tests (§6);
- clear, complete, flexible test reports;
- straightforward syntax for declarations, actions, showing values, checks on values, predicate checks, and scopes (§6);
- support for exception checks (§6.2.9) and unexpected exceptions (§6.7);
- support for complex types (§11);
- easy support for new types (§12);
- fixtures (§9);
- repeating steps for many values, like theorem checking (§8);
- mock objects (§10).

"Testudo" means "turtle" in Esperanto and Latin. The name of the turtle in the logo (see the title of this document) 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 colours[1].

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.

[1] Colour-blind-friendly, by the way.

Macros

Testudo relies rather heavily on code generation with C++ macros. As explained in §2, you can customise the names of all the macros to suit your needs or preferences. In the examples given in this documents, macros are highlighted like this (here, the macro is "`PERFORM`"):

```
1 PERFORM(firefly.say("Then, it's war!"));
```

Testudo macros are coded in such a way that they are impervious to the common problem of macro arguments containing commas, like for instance in templated types, which would normally confuse C++ macros (an argument such as "`map<int, float>`" would be parsed as two arguments: "`map<int`" and "`float>`"). Testudo achieves this by one of two means:

- having a single potentially macro-fooling argument for a macro, and having it be the last one;
- requiring parentheses around potentially macro-fooling arguments (this will always be made clear in this document).

The `testudo` namespace

All (non-macro) names defined by Testudo that are intended for usage by users are defined in the "`testudo`" namespace. This namespace will left out in the examples, as if "`using namespace testudo`" had been declared.

Showcase: code, test, and report

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, 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 viii and ix), 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.

In the report, you'll see different colours for different elements:

- orange is for separation, test header cartouches, and line number info; in test headers, the full file and line number info is given; for individual steps, only the line number is given, unless the step isn't in the same file as the test header it's a part of, in which case the full file and line number info is written;
- step type codes, test punctuation, and test summaries are written in blue;
- OK, fail, and error tags have their own, obvious colour schemes;
- red strings of dashes make it easy to match non-OK tags to their steps.

```
#include <list>
#include <stdexcept>

// Hold holds a double value; you can query it with pop(),
// but then, it doesn't hold it anymore; when a Hold that is holding a
// value is destroyed, it adds it to a list that can be consulted with
// Hold::get_forgotten();
class Hold {
  bool holding{false};
  double held;
  static inline std::list<double> forgotten;
public:
  Hold()=default;
  Hold(double i) { hold(i); } // hold on creation
  ~Hold() { // FIXME: the list ends up in the wrong order; see failed test
    if (holding)
      forgotten.push_front(held);
  }
  void hold(double i) {
    if (holding)
      throw std::runtime_error("already holding a double");
    held=i;
    holding=true;
  }
  double pop() {
    if (not holding)
      throw std::runtime_error("not holding any double");
    holding=false;
    return held;
  }
  bool is_holding() const { return holding; }
  static auto get_forgotten() { return forgotten; }
  static void clear_forgotten() { forgotten.clear(); }
  // FIXME: not yet implemented; see failed test:
  static bool is_forgotten_empty() { return false; }
};

#ifndef NAUTOTEST // we can disable tests by defining this macro
#include <testudo/testudo_lc>
#include <testudo/testudo_ext.h> // we need the support for lists

namespace {
  using namespace std;
  // these test steps are artificially gathered into a single test to showcase
  // Testudo instructions and the generated report; it checks functionality for
  // the class HalfDouble; in real life, this should be broken into
  // smaller tests with focused concerns; the resulting report lays out a full
  // narrative (you can understand it by reading only the report, without the
  // source code) [cutting here so you can have the test in one page]
```

```
  // this defines a test named "use_instructions", titled "use
  // instructions"

  define_top_test("testudo",
                  (use_instructions, "use instructions")) {
    print_text("index:\n"
               "  1. holding functionality\n"
               "  2. exceptions\n"
               "  3. list of forgotten doubles");
    print_break(); //————————————————————————————
    declare(Hold hf);
    check(not hf.is_holding())_true;
    perform(hf.hold(3.14));
    check(hf.is_holding())_true;
    check(hf.pop())_approx(3.14);
    check(not hf.is_holding())_true;
    print_break(); //————————————————————————————
    print_text("hf is empty now");
    step_id(popping_empty);
    check_try(hf.pop())_catch();
    perform(hf.hold(2.72));
    step_id(adding_to_already_holding);
    check_try(hf.hold(7.))_catch();
    print_break(); //————————————————————————————
    print_text("the forgotten doubles list is still empty");
    check(Hold::is_forgotten_empty())_true;
    check(Hold::get_forgotten().size())_equal(0u);
    in_scope("scope 1") {
      declare(Hold hf1(1.1)); // hold-on-construction syntax
      declare(Hold hf2(2.2));
      in_scope() { // unnamed scope
        declare(Hold hf3(3.3)); // will add to the list on scope closing
      }
      check(Hold::get_forgotten())_tol(.5)_approx({3.});
      show_value(hf2.pop());
      print_text("hf2 now empty; it won't add to the list");
    }
    check(not Hold::is_forgotten_empty())_true;
    check(Hold::get_forgotten())
      _approx({3.3, 1.1});
    perform(Hold::clear_forgotten());
    check(Hold::get_forgotten().size())_equal(0u);
    print_text("the following will raise an error");
    perform(hf.hold(9.9));
    print_text("this won't show, because of the error");
  }
}
#endif
```

```
 ____________________________________________
| testudo_doc.ttd:53                         |
| {testudo.use_instructions} use instructions |
'____________________________________________'
  | index:
  |   1. holding functionality
  |   2. exceptions
  |   3. list of forgotten doubles
59 ------------------------------------------------------------
60 : Hold hf ;
61 % not hf.is_holding()                                 [ OK ]
62 # hf.hold(3.14) ;
63 % hf.is_holding()                                     [ OK ]
64 % hf.pop() // 3.14 +/- eps                            [ OK ]
65 % not hf.is_holding()                                 [ OK ]
66 ------------------------------------------------------------
67 " hf is empty now "
[ popping_empty ]
69 & hf.pop() > " not holding any double "                [ OK ]
70 # hf.hold(2.72) ;
[ adding_to_already_holding ]
72 & hf.hold(7.) > " already holding a double "           [ OK ]
73 ------------------------------------------------------------
74 " the forgotten doubles list is still empty "
75 % Hold::is_forgotten_empty() : false ---------------- [FAIL]
76 % Hold::get_forgotten().size() == 0u                  [ OK ]
77 # { " scope 1 "
  78 : Hold hf1(1.1) ;
  79 : Hold hf2(2.2) ;
  80 # {
    81 : Hold hf3(3.3) ;
  # }
  83 % Hold::get_forgotten() // {3.} +/- .5              [ OK ]
  84 ? hf2.pop() : 2.2
  85 " hf2 now empty; it won't add to the list "
# } " scope 1 "
87 % not Hold::is_forgotten_empty()                      [ OK ]
88 % Hold::get_forgotten() // {3.3, 1.1} +/- eps : list{\
    1.1, 3.3} // list{3.3, 1.1} ------------------------ [FAIL]
90 # Hold::clear_forgotten() ;
91 % Hold::get_forgotten().size() == 0u                  [ OK ]
92 " the following will raise an error "
93 # hf.hold(9.9) ;
> uncaught exception " already holding a double " ------ [ERR-]
{testudo.use_instructions} 2/12 fail, 1 err ---------- [ERR-]
```

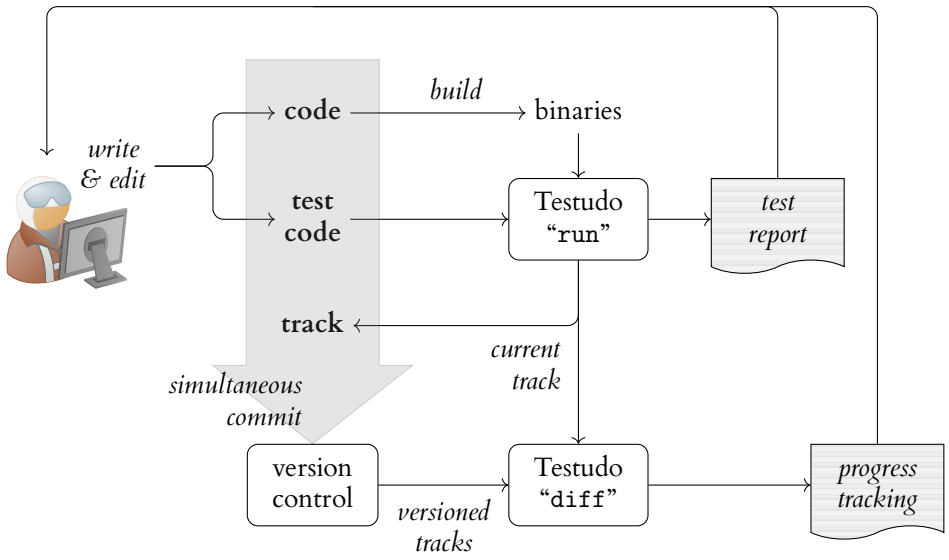

Figure 1: Development workflow with Testudo

Development workflow with Testudo

Testudo can be used with any workflow, with any testing philosophy. You can use it

- for test-driven development;
- to translate requirements into tests;
- to validate specifications;
- to continuously refactor code;
- to successfully refactor *legacy* code;
- to avoid reappearing bugs by turning them into tests;
- to simply ensure your code is as bug-free as possible.

This is how you can use Testudo to improve your development workflow:

- code Testudo tests for your code, either before the code itself (as in TDD and extreme programming), at the same time as the code, or after the code (for refactoring, for instance);
- use Testudo to run the tests often; integrate test running into your development cycle and discipline; for instance, you can institude a rule that every commit must leave the code with no failed checks or errors; or, if you have turned your requirements into tests, your rule can be about never introducing regressions;
- use Testudo to check the progress of your development, by committing the report status with every code commit; thus, you can know what the test result changes are for your working directory compared to the last committed revision, or compare any two revisions.

Figure 1 shows how Testudo is integrated into a workflow:

- you write and edit code, and tests for your code;

- you build your binaries using your build system;
- you use Testudo to run the tests on your binaries;
- based on the test reports you get, you make changes to your code and tests to ensure quality;
- Testudo also generates *tracks*; tracks are files that Testudo uses to track the test result progress between different versions of your code;
- when you're happy with your code and changes, you commit them; it's important that you commit your code, your tests, and your test tracks *simultaneously*; they act as a single, coherent entity;
- at any moment, you can use Testudo to report the difference between two tracks; the difference can be between the latest committed track and your current track ("how am i doing with this change?"), or between two committed tracks ("what was the progress between these two versions?"); the progress tracking also informs your changes to your code and tests.

1 Installation

In the "`testudo`" directory, do

- install Testudo with

```
make install
```

- uninstall Testudo (asking confirmation for each deletion) with

```
make uninstall
```

- check installation parameters with

```
make cat_install_params
```

By default, Testudo installs itself to "`$HOME/local`", where "`$HOME`" is the user's home directory. You can change this by defining an environment variable "`PREFIX`", or by passing "`make`" such a variable. The following commands are equivalent, and they will both install Testudo to "`/usr/local`":

- `PREFIX=/usr/local make install`
- `make PREFIX=/usr/local install`

2 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 invoked by running "`testudo 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 "`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 anything[2].

The "`color_text_with_lines`" format is similar to "`color_text`", but it also prints source code line information for each test step.

3 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:

[2] This is so because the "`xml`" format first builds the whole XML object for the printout before printing it.

- "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:

```
declare(int a=7); // declare a variable
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:

```
DECLARE(int a = 7); // declare a variable
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.

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.

4 How to use Testudo

4.1 The makefile template

You can use the file "Makefile.template" as a simple building system (by copying it into your project directory as "Makefile") or just for your Testudo tests. You just have to edit your copy and set the variables "SHAREDLIBNAME" variable and, optionally, "EXENAME".

4.2 The test code

First, choose a macro style (see §3). 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

```
#include "testudo_uc"
```

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

```
#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.

Your C++ must be compiled into a shared object file (this is automatically done by "make", using GNU g++ flags "-fPIC -shared").

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

Table 1: Macro names in the default styles

4.3 The tool

4.3.1 Using "`make`"

If you're using the makefile template, as described in §4.1, you can use a set of "`make`" commands to perform Testudo-related tasks. The standard workflow assumes you have a saved report and a saved track that represent your "baseline"; differences and progress and measured with respect to the baseline. When using a version control system, you must add the saved report and track, and commit them with their matching version of the code and test code.

Here are the available "`make`" commands:

- `make show_report`: runs the tests and shows a colour text report;
- `make show_sync_report`: runs the tests and shows a colour text report, but bypassing XML generation; this is useful if your code is crashing (see §2);
- `make summary_report`: runs the tests and shows a test summary;
- `make view_report`: runs the tests and shows a colour text report using the "`less`" text viewer;
- `make diff_report`: shows the (plain Unix) "`diff`" between the baseline and the current report; this is useful if you want to see just the new tests you've coded, and how they affect the tallies;
- `make diff_report_bw`: same as the previous one, but with no colour codes; this is useful when you're using a terminal or terminal emulator that doesn't support colours (e.g., if you're using Emacs' "`compile`" command);
- `make save_report`: runs the tests and saves the current report as a baseline for "`make diff_test`" and similar commands;
- `make track_progress`: shows the tracked progress with respect to the baseline;
- `make track_progress_bw`: same as the previous one, but with no colour codes;
- `make save_track`: runs the tests and saves the current track as a baseline for "`make track_progress`";
- `make diff_track`: runs "`make diff_report`" unconditionally followed by "`make track_progress`";
- `make diff_track_bw`: same as the previous one, but with no colour codes;
- `make save_test`: runs "`make save_report`" and "`make save_track`".

In most cases, you can do with only

- `make diff_track`;
- `make diff_track_bw`;
- `make save_test`.

Using these "`make`" commands, a simple coding workflow is

1. code, change, solve, refactor . . .
2. `make diff_track`;
3. iterate (to step 1) until happy;
4. `make save_test`;
5. commit;
6. go to step 1.

4.3.2 Directly calling the executable

You can also run the Testudo executable directly, if you can't or don't want to use the makefile.

The main executable for Testudo is called "`testudo`". It needs the following arguments to execute tests:

- first, "`run`" (this is the subcommand; there are other subcommands; run "`testudo help`" to get a list);
- "`-f <format>`" to specify the report format, where "`<format>`" is the format name (the ones available by default are "`xml`", "`color_text`", and "`color_text_with_lines`"; see §2 for details);
- the shared object files containing the tests to perform; these files are dynamically loaded by "`testudo`" before the test execution is started.

You can instruct Testudo to start the test execution from a specific node by passing "`-s <test-root>`" to the executable, where "`<test-root>`" is the full name for the node. The execution will be restritced to the subtree rooted at the node, and all test names and test node names will be relative to it.

Additionally, you can restrict the test execution to a list of nodes (and the subtrees rooted at them), by passing "`-i <node-name>`" for each of them.

Testudo reports source code locations using the file name and the line number of the source code line. The file name is relative to the directory from which the compiler was invoked. If you want to ignore a common initial path part, pass it as "`-d <common-path>`" (for instance, "`-d simulation/framework`") and that path part will be replaced by "`...`" in the test report.

5 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). Testudo gives you means to control the order of execution of tests, even across translation units.

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 that 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.

When you define a test node or a test, you give its name and title in a comma-separated parenthesised group:

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

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

The name allows you to refer to the test node or test later in the same translation unit, as a parent to another test node or test. If you aren't going to refer to the test node or test (which is the most common case for tests[3]), you can choose to mention only the title, with no parentheses:

```
define_test(medical, "switch on after") {
  ...
}
```

```
DEFINE_TEST(medical, "switch on after")
{
  ...
}
```

5.1 Non-top test nodes

When you declare or define a test node whose parent has been defined in the current translation unit[4], 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":

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

```
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 an unnamed child test called titled "switch on after creation", that checks a tricorder medical sub-unit is off upon creation of the tricorder, and switches on appropriately:

[3] The most usual way of structuring test nodes and tests is to have tests as leaves, and test nodes as non-leaves.

[4] I call these non-top test nodes in opposition to top test nodes; see §5.2. Another name could have been "child nodes", since they are children to parents that have been defined in the same translation unit.

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

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

5.2 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.

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

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

```
DEFINE_TOP_TEST_NODE("outatime.delorean", // parent full name
                     (flux_capacitor, // name
                      "flux capacitor features"), // title
                     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 titled "doors closed after constructions", child to a test node whole full name is "`outatime.delorean`":

```
define_top_test("outatime.delorean", // parent full name
                "doors closed after construction", // title
                150) { // priority
  declare(Delorean d);
  check(not d.left_door.is_open())_true;
  check(not d.right_door.is_open())_true;
}
```

```
DEFINE_TOP_TEST("outatime.delorean", // parent full name
                "doors closed after construction", // title
                150) // priority
{
  DECLARE(Delorean d);
  CHECK(not d.left_door.is_open()) TRUE;
  CHECK(not d.right_door.is_open()) TRUE;
}
```

6 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.

6.1 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

```
int n=7;
```

- using-declarations

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

- using-directives

```
using namespace std;
```

Examples of actions are:

- variable assignments

```
n=8;
```

- function invocations

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

6.1.1 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.

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

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

6.1.2 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.

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

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

6.2 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.

6.2.1 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 §12.1). The effect cannot 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.

6.2.2 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.

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

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

See §6.2.7 for an alternative version that can show you what the involved values were for a failed "true" check.

6.2.3 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"[5].

[5] I've chosen the word "nay" rather than the word "not" to avid 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`".

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

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

See §6.2.8 for an alternative version that can show you what the involved values were for a failed "false" check.

6.2.4 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.

```
check(captain_age)_equal(26+10);
```

```
CHECK(captain_age) EQUAL(26 + 10);
```

The opposite of the "equal" macro is the "not-equal" macro:

`_not_equal` – `NOT_EQUAL`

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 verboseness. So, for instance, if "`inventory`" has type "`map<string, int>`", the following two checks are equivalent:

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

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

6.2.5 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 fail[6]. 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 §12.4) 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.

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

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

The opposite of the "approx" macro is the "not-approx" macro:

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

`_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.

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

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

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.

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

```
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.

```
show_approx_epsilon();
```

```
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.

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

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

6.2.6 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`

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

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

- from a bool expression: `predicate_a` – `PREDICATE_A` (the expression must use the parameter name "a")

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

```
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 parenthesised[7] argument gives the list of captures)

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

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

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

```
check(number_of_cards)_verify(is_even);
check(score)_verify(is_multiple_of(5));
```

```
CHECK(number_of_cards) VERIFY(is_even);
CHECK(score) VERIFY(is_multiple_of(5));
```

Predicates can be combined with the logical operators "`not`", "`and`", and "`or`":

```
check(iterations)
  _verify(not is_negative
          and (is_even or is_multiple_of(5)));
```

```
CHECK(iterations)
  VERIFY(not is_negative
         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

[7] Captures are usually surrounded with square brackets, but in this syntax, they must me surrounded by regular brackets.

- 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`

6.2.7 Check the expression is true for certain values: `_true_for` – `TRUE_FOR`

Compared to all the information the previous checks give in case of a fail, the "true" (§6.2.3) macro is quite uninformative. For instance, when checking that a value is not greater than another value, if the check fails, that's all the information you'll get. It would be good to know what the involved values were. That is what the "true-for" macro does.

The "check-true-for" syntax works like "check-true", but you specify in parentheses a set of expressions whose values you'd like to see if the check fails.

```
check(a+b<c)_true_for(a, b, c);
```

```
CHECK(a+b<c) TRUE_FOR(a, b, c);
```

6.2.8 Check the expression is false for certain values: `_false_for` – `FALSE_FOR`

The opposite of the "true-for" macro is the "false-for" macro.

```
check(a+b<c)_false_for(a, b, c);
```

```
CHECK(a+b<c) FALSE_FOR(a, b, c);
```

6.2.9 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 §6.7).

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

```
DECLARE(list<int> numbers);
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:

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

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

6.3 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.

6.3.1 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. Values with embedded newlines are displayed in a suitable format.

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

The "show value" instruction shows the value of its argument inline.

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

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

6.3.2 Scopes: `in_scope` – `IN_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 "in-scope" macro just before the opening brace of the scope, which writes a line to the report about the new scope. You don't have to add anything to the closing brace: Testudo will automatically write a scope-closing line 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 "in scope" macro without any arguments. If the scope is longer, it may be clearer to name it, since the scope's begin and end lines will display its name. This is done by passing the scope name to the "in-scope" macro.

```
declare(LoggedDestruction ld1("1"));
check(LoggedDestruction::n_destructions())_equal(0);
in_scope("outer scope") { // named scope
  declare(LoggedDestruction ld2("2"));
  in_scope() { // unnamed scope
    declare(LoggedDestruction ld3("3"));
  }
  check(LoggedDestruction::n_destructions())_equal(1);
}
check(LoggedDestruction::n_destructions())_equal(2);
```

```
DECLARE(LoggedDestruction ld1("1"));
CHECK(LoggedDestruction::n_destructions()) EQUAL(0);
IN_SCOPE("outer scope") // named scope
{
  DECLARE(LoggedDestruction ld2("2"));
  IN_SCOPE() // unnamed scope
  {
    DECLARE(LoggedDestruction ld3("3"));
  }
  CHECK(LoggedDestruction::n_destructions()) EQUAL(1);
}
CHECK(LoggedDestruction::n_destructions()) EQUAL(2);
```

6.3.3 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 "in-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.

```
with_declare(auto answer=client.request("get license")) {
  check(answer.valid)_true;
  check(answer.text)_equal("res://punk_license");
}
with_declare(auto answer=client.request("reset"))
  check(answer.valid)_true;
```

```
WITH_DECLARE(auto answer=client.request("get license"))
{
  CHECK(answer.valid) TRUE;
  CHECK(answer.text) EQUAL("res://punk_license");
}
WITH_DECLARE(auto answer=client.request("reset"))
  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":

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

```
WITH_DECLARE(auto [action, occurrences]=tuple{"sin", 77*7})
{
  CHECK(action) EQUAL("sin");
  CHECK(occurrences) EQUAL(539);
}
```

6.4 Identifying steps

You can identify certain steps, to make it easier to follow their evolution. It's particularly appropriate for checks[8]. 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.

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

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

6.5 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 is source code.

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

The "print text" instruction displays its argument inline. The argument must be a string of any kind. Text with embedded newlines is displayed in a suitable format.

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

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

6.5.2 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.

```
print_break();
```

```
PRINT_BREAK();
```

[8]—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.—

6.6 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.

6.6.1 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.

```
#ifdef DEBUGGING
declare(LoggedInt n_cases); // optimised to int in production
#else
fake_declare(LoggedInt n_cases);
int n_cases; // replacement declaration (naked)
#endif
```

```
#ifdef DEBUGGING
DECLARE(LoggedInt n_cases); // optimised to int in production
#else
FAKE_DECLARE(LoggedInt n_cases);
int n_cases; // replacement declaration (naked)
#endif
```

6.6.2 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.

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

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

6.7 Unexpected exceptions

If an unexpected exception (i.e., an exception not in a "check-try-catch" instruction; see §6.2.9, 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-]`.

6.8 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 §2), 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.

7 Test-aware functions

You may want to perform the same test several times, with only minor variations, such as a type or a value[9]. This can be done with *test-aware* functions (or methods, or classes). Test-awareness involves:

- getting an object of type "`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:

```
template <typename Container>
void test_container_emptyness(
    test_management_t test_management,
    Container &container) {
  check(container.empty())_true;
```

[9] If what changes is a value, consider first whether "with-data" loops (§8) fit your need.

```
}

class Cauldron { ... };

define_test(container, "Cauldron emptyness") {
  declare(Cauldron container);
  test_container_emptyness(test_management, container);
}

class Marmite { ... };

define_test(container, "Marmite emptyness") {
  declare(Marmite container);
  test_container_emptyness(test_management, container);
}
```

```
template <typename Container>
void test_container_emptyness(
    test_management_t test_management,
    Container &container)
{
  CHECK(container.empty()) TRUE;
}

class Cauldron { ... };

define_test(container, "Cauldron emptiness")
{
  DECLARE(Cauldron container);
  test_container_emptyness(test_management, container);
}

class Marmite { ... };

define_test(container, "Marmite emptiness")
{
  DECLARE(Marmite container);
  test_container_emptyness(test_management, container);
}
```

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 type[10]; 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.

[10] My inner voice calls this feature "theorem checking".

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. When inline, a valid container can be specified by a simple braced list of values. 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.

Here is an example:

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

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

If you want to use structured binding for the variable in the "with-data" macro, surround the variable names with parenthesis instead of square brackets:

```
declare(list<tuple<int, int, int>>
          list_of_sums{{1, 3, 4},
                       {3, 7, 11}, // will fail
                       {10, 15, 25}});
with_data((a, b, sum), list_of_sums)
  check(sum)_equal(a+b);
```

```
DECLARE(list<tuple<int, int, int>>
```

```
          list_of_sums{ { 1, 3, 4 },
                        { 3, 7, 11 }, // will fail
                        { 10, 15, 25 } });
WITH_DATA((a, b, sum), list_of_sums)
  CHECK(sum) EQUAL(a+b);
```

8.1 With-data loops with multiline containers: `with_multiline_data` – `WITH_MULTILINE_DATA`

When you define the container directly in the "with-data" macro, if it's long, the report will break it in lines of the appropriate length, as it would do for any text longer than one line. But in this case, this will probably obscure the structure of the container. In such cases, you can use the "with-multiline-data" macro instead. It will do its best to break the container in such a way that the report shows one element per line.

```
with_multiline_data((a, b, sum), list<tuple<int, int, int>>{
    {1, 3, 4},
    {3, 7, 11}, // will fail
    {10, 15, 25}})
  check(sum)_equal(a+b);
```

```
WITH_MULTILINE_DATA((a, b, sum), list<tuple<int, int, int>>{
    {1, 3, 4},
    {3, 7, 11}, // will fail
    {10, 15, 25}})
  CHECK(sum) EQUAL(a+b);
```

8.2 How to generate data for with-data loops

You can use the function "`generate_data()`" to generate data for with-data loops. The arguments for that function are the number of data to generate, and a function argument with no arguments that will be called repeatedly to generate the data. A common case is the generation of a large number of pseudorandom data for verification of properties of types or algorithms.

Imagine you've coded a 2D integer vector class "`VectorI2`"

```
class VectorI2 {
public:
  VectorI2(int x, int y);
  int x, y;
};
VectorI2 operator+(VectorI2 v, VectorI2 w); // sum
bool operator==(VectorI2 v, Vector I2w); // equality (§12.3)
string to_text(VectorI2 v); // text representation (§12.2)
```

and you want to verify on it the following theorem (commutativity of the sum):

$$\forall(v, w) \in \texttt{VectorI2} \times \texttt{VectorI2}, v + w = w + v \tag{1}$$

You can code a pseudorandom "`VectorI2`" generator

```
VectorI2 random_vector2i(int max_abs_x, int max_abs_y);
```

and use it like this to check the theorem on 100×100 pairs of pseudorandom values with a maximum absolute coordinate value of 20:

```
declare(auto generate_20_20=
        []() { return random_vector2i(20, 20); });
with_data(v, generate_data(100, generate_20_20))
  with_data(w, generate_data(100, generate_20_20))
    check(v+w)_equal(w+v);
```

```
DECLARE(auto generate_20_20=
        []() { return random_vector2i(20, 20); });
WITH_DATA(v, generate_data(100, generate_20_20))
  WITH_DATA(w, generate_data(100, generate_20_20))
    CHECK(v+w) EQUAL(w+v);
```

9 Fixtures

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

9.1 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 "`Fixture`". Its constructor must accept an object of type "`test_management_t`, and pass it out to the constructor of "`Fixture`". The constructor is the setup procedure; the destructor, if you code it, is the teardown procedure.

Here's an example:

```
struct OutATimeFixture : Fixture {
  OutATimeFixture(test_management_t test_management)
    : Fixture(test_management)
    { perform(d=new Delorean); }
  ~OutATimeFixture()
    { perform(delete d); }
  Delorean *d; // dumb pointer, just so we can show a teardown procedure
};
```

```
struct OutATimeFixture : Fixture
{
  OutATimeFixture(test_management_t test_management)
    : Fixture(test_management)
  {
    PERFORM(d=new Delorean);
  }
  ~OutATimeFixture()
  {
    PERFORM(delete d);
```

```
  }
  Delorean *d;  // dumb pointer, just so we can show a teardown procedure
};
```

9.2 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:

```
define_test(delorean,
            "engine is off at start",
            with_fixture(OutATimeFixture)) // with-fixture
{
  check(not d->engine.is_running())_true;
}

define_test(delorean,
            "there's no Plutonium initially",
            visible_fixture(OutATimeFixture)) // visible-fixture
{
  check(d->pu())_approx(0.);
}
```

```
DEFINE_TEST(delorean,
            "engine is off at start",
            WITH_FIXTURE(OutATimeFixture)) // with-fixture
{
  CHECK(not d->engine.is_running()) TRUE;
}

DEFINE_TEST(delorean,
            "there's no Plutonium initially",
            VISIBLE_FIXTURE(OutATimeFixture)) // visible-fixture
{
  CHECK(d->pu()) APPROX(0.);
}
```

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.

9.3 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 declarations[11], 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:

```
struct NumbersFixture : Fixture {
  NumbersFixture(test_management_t test_management)
    : Fixture(test_management),
      fixture_init(x, 1.), fixture_init(z, 3.14) { }
  fixture_member(double x);
  fixture_member(double y=-2.5, z);
};
```

```
struct NumbersFixture : Fixture
{
  NumbersFixture(test_management_t test_management)
    : Fixture(test_management),
      FIXTURE_INIT(x, 1.),
      FIXTURE_INIT(z, 3.14) { }
  FIXTURE_MEMBER(double x);
  FIXTURE_MEMBER(double y=-2.5, z);
};
```

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).

10 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" (§10.1); this adds scheduling and logging machinery to the classes automatically;
- if necessary, set up across-method ledgers (§10.4);
- in a test definition, create instances of the mock classes;

[11] The "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.

- if necessary, set up across-instances ledgers (§10.4.1);
- schedule mock object behaviour as needed (§10.2);
- use mock objects as if they were the real thing;
- check logged behaviour against expected behaviour by accessing call logs and ledgers (§10.3, §10.4.2).

10.1 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 is that it publicly must derive from "`MockClass<>`". If your mock class derives from a base class (say "`Base`"), you can make it derive from "`MockClass<Base>`" instead, which derives from both "`Base`" and "`MockClass<>`", and inherits "`Base`" constructors. Here are a couple of examples, one for a mock class deriving from a virtual class, and one for a non-derived mock class:

```
class KettleBase {
public:
  virtual ~KettleBase()=default;
  virtual void fill(float volume)=0;
  virtual float temperature() const=0;
};

class KettleMock
  : public MockClass<KettleBase> {
public:
  // method mocks and wraps overriding base methods…
};
```

```
13
14 class ContainerMock
15   : public MockClass<> {
16 public:
17   // method mocks and wraps defining a cointainer interface…
18 };
```

10.1.1 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 (§10.3, §10.4), and allows you to set up a schedule for return values (§10.2).

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 them[12], 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 parentheses[13]. 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 §10.1.2). Here are some examples of list-or-arguments arguments to the "mock-method":

- `()`
- `((int))`
- `((int), (unsigned char))`

[12] I'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.

[13] I'm thorry thith hath become tho lithpy; there'th no other way with theepluthpluth macroth.

- `((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 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

```
class ImageMock
  : public MockClass<> {
public:
  mock_method((void), set, ((float)));
  mock_method((void), set, ((int)));
};
```

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

```
class ImageMock
  : public MockClass<> {
public:
  mock_method((void), set, ((float)));
  mock_method((void), (set, set_int), ((int)));
};
```

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:

```
class KettleBase {
public:
  virtual ~KettleBase()=default;
  virtual void fill(float volume)=0;
  virtual float temperature() const=0;
};

class KettleMock
  : public MockClass<KettleBase>> {
public:
  mock_method((void), fill, ((float volume)), override);
  mock_method((float, temperature, (), const override);
};
```

10.1.2 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 (§10.3, §10.4), 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:

```
class Tally {
public:
  virtual ~Tally()=default;
  virtual void add_counter(int delta)=0;
  virtual int total() const=0;
};

class TallyMock
  : public MockClass<Tally> {
public:
  int counter=0;
  wrap_method((void), add_counter, ((int delta))) {
    counter+=delta;
  }
  wrap_method((int), total, (), const) const { // const twice
    return counter;
  }
};
```

10.2 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.

10.2.1 Set the default return value: set_ret_default – SET_RET_DEFAULT

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

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

```
class SoupMock
  : public MockClass<Soup> {
public:
  ...
  // by default, return true:
  mock_method((bool), is_tasty, (), const override)=true;
  ...
};
```

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

```
define_test(mock_turtle, "Victorian recipe") {
  declare(auto soup_mock=make_shared<SoupMock>());
  declare(bool is_tasty_now=true);
  perform(soup_mock->set_ret_default(
            is_tasty,
            [&is_tasty_now]() { return is_tasty_now; });
  ...
}
```

10.2.2 Schedule return values or exceptions: schedule_ret – SCHEDULE_RET

In order to schedule a sequence of return values for a method, you must use the "schedule-ret" macro as a method call on a mock object, giving it first the name of the method you want to schedule for, and then any number of return values to use in sequence:

```
class SoupMock
  : public MockClass<Soup> {
public:
  ...
  mock_method((int), (temperature, temp), (), const final);
  ...
};
...
define_test(..., "...") {
  declare(auto soup_mock=make_shared<SoupMock>());
  // return a sequence of temperatures on subsequent calls:
  perform(soup_mock->schedule_ret(temp,
                                  20, 40, 60, 80, 100));
  ...
}
```

You can also set the default value or any of the scheduled return values to throw an exception, rather than returning a value. This is achieved by replacing a return value in the "schedule-ret" macro with a call to the function "`throw_exception()`", giving it the exception to throw:

```
class MockNamable
  : public MockClass<> {
public:
  mock_method((void), create, ());
  mock_method((bool), set_name_is_good, ((string)));
};
...
define_test(..., "...") {
  declare(MockNamable namable);
  // return true for the first call, then throw an exception for the second call:
  perform(
    namable.schedule_ret(set_name_is_good,
      true,
      throw_exception(runtime_error("already named"))));
  ...
}
```

If the return type of the method is `void`, you can use the special value "`void_v`" to represent one successful return. This is necessary when you want to schedule an exception after a number of uneventful returns:

```
define_test(..., "...") {
  declare(MockNamable namable);
  // return normally from the first call, then throw an exception for the second call:
  perform(
    namable.schedule_ret(create,
      void_v,
      throw_exception(runtime_error("already created"))));
  ...
}
```

10.3 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.

10.3.1 Logged arguments: `logged_args` – `LOGGED_ARGS`

"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:

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

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

- check a specific argument of a specific call:

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

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

10.3.2 Logged return values: `logged_ret` – `LOGGED_RET`

Similarly, the "logged-ret" macro returns a vector of tuples containing the return values for all calls to a mock method[14].

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

10.3.3 Logged arguments and return values: `logged_ret_args` – `LOGGED_RET_ARGS`

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.

```
declare(auto soup_mock=make_shared<SoupMock>());
...
// five calls, with specific arguments and return values
check(soup_mock->logged_ret_args(add_ingr))
  _equal({{{4}, {"calf brains", 4}},
```

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

```
            {{7}, {"pork liver", 3}},
            {{8}, {"water", 1}},
            {{9}, {"water", 1}},
            {{10}, {"water", 1}}});
```

10.3.4 Number of calls: `log_size` – `LOG_SIZE`

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

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

Checks with mock-method logs lend themselves to "with-declare" scopes (§6.3.3), where the declaration is for instance the result of a "logged-ret-args" macro, and the contents are checks on that value.

10.3.5 Utilities for log checking

The following bool-returning functions can be helpful when checking things like "this method has always been called with the argument 9", or "all values returned by this method have been different":

- "`is_always(c, a)`" checks whether all elements of container "`c`" are "`a`";
- "`is_never(c, a)`" checks whether none of the elements of container "`c`" are "`a`";
- "`is_constant(c)`" checks whether all the elemennts of container "`c`" are the same;
- "`all_different(c)`" checks whether all the elements of container "`c`" are different.

10.4 How to check mock-method ledgers

In addition to per-mock-method logs (§10.3), Mock Turtle keeps track of the order in which different methods are invoked. This tracking is managed by the class "`MockClass<>`" (the one mock classes must derive from).

10.4.1 Checks across mock objects: `call_ledger_report_to` – `CALL_LEDGER_REPORT_TO`

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[15]. 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.

[15] "`CallLedger`" is just an alias for "`MockClass<>`"; this naming reflects better its usage as a general call ledger.

You will only need a separate "`CallLedger`" if you want to track calls across several objects. If you're only interested in tracking calls across several methods with the same object, you can do it by using the mock object itself as the call ledger.

Here's an example of cross-object call ledger setup:

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

10.4.2 Scanning the ledger: `get_call` – `GET_CALL` and `pop_call` – `POP_CALL`

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 "`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:

```
with_declare(auto call=it.get_call(*soup_mock, add_ingr))
  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:*

```
with_declare(auto call=it.pop_call(*soup_mock, add_ingr))
  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_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 (§6.3.3).

Here's an example with mock-method ledgers:

```
declare(auto soup_mock=make_shared<SoupMock>());
...
with_declare(auto it=iterate(soup_mock)) {
  with_declare(auto call=it.pop_call(*soup_mock, add_ingr)) {
    check(call.ret())_equal(4);
    check(call.args())_equal("calf brains", 4);
  }
  ...
  with_declare(auto call=it.pop_call(*soup_mock, temp))
    check(call.ret())_equal(20);
  ...
  with_declare(auto call=it.pop_call(*soup_mock, is_tasty))
    check(call.ret())_true;
  // check it was the last call:
  check(it.done())_true;
}
```

Here's an example involving the "call-ledger-report-to" macro:

```
declare(CallLedger cl);
declare(MockLevel lev1(12));
perform(call_ledger_report_to(lev1, &cl));
declare(MockLevel lev2(47));
perform(call_ledger_report_to(lev2, &cl));
perform(equalize(lev1, lev2)); // this is the function we're checking
check(lev1.readout())_equal(52);
check(lev2.readout())_equal(52);
declare(auto it=iterate(cl));
// fast forward to the first invocation to "large_up()":
perform(while (it.method_name() not_eq "large_up")
          it.next());
check(it.mock_name())_equal("lev1");
check(it.method_name())_equal("large_up");
perform(it.next());
check(it.mock_name())_equal("lev1");
check(it.method_name())_equal("readout");
```

11 Testudo support for STL containers

If you're going to use STL containers as subjects for your tests, meaning you're going to show their value, or use it for checks, just include the "`testudo_ext.h`" header:

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

This will make STL containers printable by Testudo. Additionally, although STL containers can be compared with the standard "`operator==()`" operator, so there's nothing to add in that regard, "`testudo_ext.h`" adds the following two features

- an STL container will be considered valid (§6.2.1) if and only if all its elements are;
- "approx" checks (§6.2.5) will use the Manhattan distance on the container elements.

Bottom line: if you include "`testudo_ext.h`", Testudo will work well on STL containers.

12 Adding Testudo support for your types

A type needs four features in order to be fully Testudo-supported:

- it *may* support the notion of validity; any check done on an invalid value is failed, irrespective of the value or the precise check made on it (§6.2.1); by default, all values of a type are considered valid unless you decide to code validity for the type, so you often won't need to deal with validity at all;
- it *must* have a textual representation; this is used when you show a value with the "show-value" macro (§6.3.1), but also when the value is shown for a failed check;
- it *can* support testing for equality, if you need it for your tests, in particular, if you use the "equal" macro (§6.2.4);
- it *can* support the notion of "absolute difference", if you need it for your tests, in particular, if you use the "approx" macro (§6.2.5).

The implementation of each one of these features for your types is described in the following sections. You don't need to include any style (§3) to define them, or the whole Testudo set of functionality. It's enough if you include the "`testudo_base.h`" header (and if your definitions don't use the basic implementations in Testudo, not even that will be needed):

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

In general, when you customize Testudo support for your type, you either code a general function for your type, like "`operator==()`" or redirection to output stream, or define a Testudo-specific function in the same namespace as your type. In the latter case, the function name will end with "`_testudo`" (see each section for the exact name). If you need to use any of those Testudo-specific functions (for example, when coding the function for your type, the definition may use the same function for an attribute contained within your type), you should use the equivalent function without the "`_testudo`", and in the "`testudo`" namespace.

So, for instance, if you're defining validity for type "`my_space::MyType<T>`", you have to define

```
namespace my_space {
  template <typename T>
  bool is_valid_testudo(MyType<T> ct &mt);
}
```

but if the definition depends on the validity of a contained attribute with type "`list<T>`", then you'll have to use "`testudo::is_valid()`":

```
namespace my_space {
  template <typename T>
  bool is_valid_testudo(MyType<T> ct &mt) {
    return testudo::is_valid(mt.list_of_elements);
  }
}
```

12.1 Validity

All values are considered valid by default by Testudo (§6.2.1). If you have a type that may have invalid values (i.e., values that always yield failed Testudo checks), you have code a "`bool is_valid()`" function in the type's namespace, that accepts as its sole argument an object of the type:

```
namespace my_space {
  class MyVector {
  public:
    ...
  };
  bool is_valid(MyVector const &mv) { ... }
}
```

If you want to refer to the validity of a value "v" of another type contained in your type, use the expression "`testudo::is_valid(v)`".

12.2 Textual representation

In order to tell Testudo how to produce a text representation for a value of your type, you can either

- code the "redirection to output stream" operator for your type:

```
namespace my_space {
  class MyVector {
  public:
    double x, y;
    ...
  };
  ostream &operator<<(ostream &os, MyVector const &mv) {
    return os << "(" << mv.x << " " << mv.y << ")";
  }
}
```

- or code a "`to_text_testudo()`" function in the type's namespace, that accepts as its sole argument an object of the type, and returns a string that represents it:

```
namespace my_space {
  class MyVector {
  public:
    double x, y;
    ...
  };
  string to_text_testudo(MyVector const &mv) {
    return "("+to_string(mv.x)+" "+to_string(mv.y)+")";
  }
}
```

If you want to use the textual representation of a value "`v`" of another type contained in your type, use the expression "`testudo::to_text(v)`".

12.3 Equality

Checks for equality between values (§6.2.4) are done by simply using the "`==`" operator on the values. So you just have to code the appropriate "`operator==()`" for your type to be supported by Testudo checks for equality:

```
namespace my_space {
  template <typename T>
  class MyPair {
  public:
    ...
    T first, second;
  };
  template <typename T>
  bool operator==(MyPair<T> const &mp1,
                  MyPair<T> const &mp2) {
    // see below for "testudo::are_equal()"
    return (testudo::are_equal(mp1.first, mp2.first)
              and testudo::are_equal(mp1.second, mp2.second));
  }
}
```

Similarly to the other cases, you can also choose to define the "`are_equal_testudo()`" function for your type, if you prefer Testudo to use a different implementation for equality, and you should use "`testudo::are_equal()`" to test for equality of other values:

```
namespace my_space {
  template <typename T>
  class MyPair { ... };
  template <typename T>
  bool are_equal_testudo(MyPair<T> const &mp1,
                         MyPair<T> const &mp2) {
    return (testudo::are_equal(mp1.first, mp2.first)
```

```
8               and testudo::are_equal(mp1.second, mp2.second));
9   }
10 }
```

12.4 Difference between two values

Checks for nearness between values (§6.2.5) are done by checking if the absolute difference between the values is below the tolerance. For simple scalar types like `float` and `double`, this absolute difference is simple the absolute value of the difference, but for other types that you want to use "approx" checks on, you have to define exactly how it's computed. This is done by defining a "`absdiff_testudo()`" function in the type's namespace, that accepts the two values to compare, and returns the absolute difference as a `double` value.

```
1  class MyVector {
2  public:
3    double x, y;
4    ...
5  };
6  double absdiff_testudo(MyVector const &mv1,
7                         MyVector const &mv2) {
8    double dx=mv1.x-mv2.y;
9    double dy=mv1.y-mv2.y;
10   return sqrt(dx*dx+dy*dy);
11 }
```

The absolute difference need not be exactly the norm of the difference. Any function that is zero for identical values and grows for values that are more and more apart, while giving an order of magnitude of the actual difference, is OK:

```
1  double absdiff_testudo(MyVector const &mv1,
2                         MyVector const &mv2) {
3    double dx=mv1.x-mv2.y;
4    double dy=mv1.y-mv2.y;
5    return abs(dx)+abs(dy); // this is also OK
6  }
```

You may have to use "`testudo::absdiff_testudo()`" on other values contained in your type:

```
1  template <typename T>
2  class MyPair {
3  public:
4    ...
5    T first, second;
6  };
7  template <typename T>
8  double absdiff_testudo(MyPair<T> const &mp1,
9                         MyPair<T> const &mp2) {
10   return (testudo::absdiff(mp1.first, mp2.first)
11           +testudo::absdiff(mp1.second, mp2.second));
12 }
```

A Testudo installation

Dependencies:

- a C++17 compiler (duh);
- `bash`, `sed`, `awk` (used by scripts);
- `m4` (for generation of "`mock_turtle_macro_n.gh`");
- `make` (but you can use your own alternative);
- LaTeX, `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). For Emacs, if you have a style file named "`mystyle.txt`", do "`make emacs_add_keywords_mystyle.txt`" and you'll get a text file named "`emacs_add_keywords_mystyle.txt`" containing the appropriate customization expression for your "`.emacs`" file.

C Using your own test macro names

You can define your own style for Testudo macros, so macro names suit your exact needs and taste. You just have to copy one of the two provided style files, "`lc.tst`" or "`cp.tst`", into a file of your own with the "`.tst`" extension, say "`mystyle.tst`", and customize the second half of each line, which is the macro name. Then, instead of including "`testudo_lc`" or "`testudo_uc`", include "`testudo_mystyle`".

Here's an example where we adapt the macro names to be in Esperanto. The "`eo.tst`" file is shown in Figure 2, and this is a test specified with that style:

```
1 difini_teston(esperanto_test, numbers, "numbers") {
2   deklari(auto const dictionary=vortaro);
3   certigi(dictionary.at("unu"))_egalas(1);
4 }
```

```
TOP_TEST_NODE          ĉefa_testo_nodo
DEFINE_TOP_TEST_NODE   difini_ĉefa_testo_nodon
DEFINE_TEST_NODE       difini_testo_nodon

DEFINE_TOP_TEST difini_ĉefa_teston
DEFINE_TEST     difini_teston
WITH_FIXTURE    kun_fiksaĵo
VISIBLE_FIXTURE videbla_fiksaĵo

STEP_ID              paŝo_id
PRINT_TEXT           printi_tekston
PRINT_BREAK          printi_paŭzon

DECLARE              deklari
PERFORM              fari
FAKE_DECLARE         false_deklari
FAKE_PERFORM         false_fari
FIXTURE_MEMBER       fiksaĵo_membro
FIXTURE_INIT         fixture_init
WITH_DATA            kun_datumoj
WITH_MULTILINE_DATA  kun_multliniaj_datumoj
CHECK_TRY            certigi_try
CATCH                _catch
SHOW_VALUE           montri_valoron
IN_SCOPE             montri_amplekson
WITH_DECLARE         kun_deklaro

DEFINE_APPROX_EPSILON difini_proksimo_epsilonon
SET_APPROX_EPSILON    asigni_proksimo_epsilonon
SHOW_APPROX_EPSILON   montri_proksimo_epsilonon

CHECK      certigi
TRUE       _vera
FALSE      _malvera
TRUE_FOR   _vera_por
FALSE_FOR  _malvera_por
EQUAL      _egalas
NOT_EQUAL  _ne_egalas
APPROX     _proksimas
NOT_APPROX _ne_proksimas
TOL        _tol
VERIFY     _verigas
NOT_VERIFY _ne_verigas

PREDICATE     predikato
PREDICATE_A   predikato_a
PREDICATE_C_A predikato_c_a

MOCK_METHOD     imita_metodo
WRAP_METHOD     volva_metodo
LOGGED_ARGS     inskribitaj_arg
LOGGED_RET      inskribita_ret
LOGGED_RET_ARGS inskribitaj_ret_arg
LOG_SIZE        inskriboj
SCHEDULE_RET    plani_ret
SET_RET_DEFAULT asigni_ret_defaŭlton
GET_CALL        preni_vokon
POP_CALL        eltiri_vokon
CALL_LEDGER_REPORT_TO voko_ĉeflibro_informas
```

Figure 2: Definition of an Esperanto style in file "`eo.tst`"