```
    ____ _______ ___ _ _ __   __        __ ?
   |_   | _ |  _|   | | |  \ / |    .-./'_)  [ OK ]
     | || _|\_  \| || | | | | |  ,/___\/    [FAIL]
------|_||___|___/|_|-\__|__/|__/-----U-U---------------mgc--
```

Testudo,
an automatic test system
for C++ code

(version 0.02)

Miguel González Cuadrado
<mgcuadrado@gmail.com>

2nd September 2020

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 reference.

Intro

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

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

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

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 run 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 subtree rooted at each test node.

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

```
#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_lc.h"
#include "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_multiline_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);
    { show_scope("scope 1");
      declare(Hold hf1(1.1)); // hold-on-construction syntax
      declare(Hold hf2(2.2));
      { show_scope(); // unnamed scope
        declare(Hold hf3(3.3)); // will add to the list on scope closing
      }
      check(Hold::get_forgotten())_approx(list{3.})_tol(.5);
      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(list{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.use_instructions} use instructions |
`----------------------------------------------'
  | index:
  |   1. holding functionality
  |   2. exceptions
  |   3. list of forgotten doubles
---------------------------------------------------------------
: Hold hf ;
% not hf.is_holding()                                   [ OK ]
# hf.hold(3.14) ;
% hf.is_holding()                                       [ OK ]
% hf.pop() // 3.14 +/- eps                              [ OK ]
% not hf.is_holding()                                   [ OK ]
---------------------------------------------------------------
" hf is empty now "
[ popping_empty ]
& hf.pop() > " not holding any double "                 [ OK ]
# hf.hold(2.72) ;
[ adding_to_already_holding ]
& hf.hold(7.) > " already holding a double "            [ OK ]
---------------------------------------------------------------
" the forgotten doubles list is still empty "
% Hold::is_forgotten_empty() : false ------------------- [FAIL]
% Hold::get_forgotten().size() == 0u                    [ OK ]
# { begin scope " scope 1 "
: Hold hf1(1.1) ;
: Hold hf2(2.2) ;
# { begin scope " <unnamed> "
: Hold hf3(3.3) ;
# } end scope " <unnamed> "
% Hold::get_forgotten() // list{3.} +/- .5              [ OK ]
? hf2.pop() : 2.2
" hf2 now empty; it won't add to the list "
# } end scope " scope 1 "
% not Hold::is_forgotten_empty()                        [ OK ]
% Hold::get_forgotten() // list{3.3, 1.1} +/- eps : list\
    {1.1, 3.3} // list{3.3, 1.1} ----------------------- [FAIL]
# Hold::clear_forgotten() ;
% Hold::get_forgotten().size() == 0u                    [ OK ]
" the following will raise an error "
# hf.hold(9.9) ;
>  uncaught exception " already holding a double " ----- [ERR-]
{testudo.use_instructions} 2/12 fail, 1 err ------------ [ERR-]
```

1 Test output formats

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

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

The "`color_text`" format outputs the printout directly as a full text report with colours, virtually identical to the one "`xml_to_color`" produces. The difference is that this format produces its output synchronously, so even if a fatal error happens, you'll get the whole output until just before the error, whereas the "`xml`" won't output anything[1].

2 Test definition and test instruction styles

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

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

```
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
```

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

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

style, then in the "uc", open style.

3 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 subtree 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 hande, *defining* a test node or a test means giving it full contents, including at least a name and a title, but possibly also a test function or a priority.

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

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

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

3.1 Non-top test nodes

When you declare or define a test node whose parent has been defined in the current translation unit[2], 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 a child test called "`switch_on`", titled "switch

[2]I call these non-top test nodes in opposition to top test nodes; see §3.2. Another name could have been "child nodes", since they are children to parents that have been defined in the same translation unit.

on after creation", that checks a tricorder medical subunit is off upon creation of the tricorder, and switches on appropriately:

```
define_test(medical, switch_on, "switch on after") {
  declare(Tricorder t); // see §4 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, "switch on after")
{
  DECLARE(Tricorder t); // see §4 for the test steps syntax
  CHECK(not t.medical.is_on()) TRUE;
  PERFORM(t.medical.push_on_button());
  CHECK(t.medical.is_on()) TRUE;
}
```

3.2 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 called "`doors_closed_on_start`", child to a test node whole full name is "`outatime.delorean`":

```
define_top_test("outatime.delorean", // parent full name
                doors_closed_on_start, // 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_on_start, // 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;
}
```

4 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.

4.1 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());
```

4.1.1 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 });
```

4.1.2 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);
```

4.2 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.

4.2.1 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.

4.2.2 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;
```

4.2.3 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);
```

4.2.4 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[3]. 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 §6.3) 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.));
```

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.

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();
```

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

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 after the "approx" macro.

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

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

4.2.5 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 boolean. 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 boolean 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 boolean expression with capture: `predicate_c_a` – `PREDICATE_C_A` (like the previous one, but a first parenthesised[4] 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`":

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

```
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

- 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.

4.2.6 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 §4.7).

```
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);
```

4.3 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.

4.3.1 Showing values

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

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

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

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

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

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

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

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

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

4.3.2 Scopes: `show_scope` – `SHOW_SCOPE`

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

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

```
declare(LoggedDestruction ld1("1"));
check(LoggedDestruction::n_destructions())_equal(0);
{ show_scope("outer scope"); // named scope
  declare(LoggedDestruction ld2("2"));
  { show_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);
{
  SHOW_SCOPE("outer scope"); // named scope
  DECLARE(LoggedDestruction ld2("2"));
  {
    SHOW_SCOPE(); // unnamed scope
    DECLARE(LoggedDestruction ld3("3"));
  }
  CHECK(LoggedDestruction::n_destructions()) EQUAL(1);
}
CHECK(LoggedDestruction::n_destructions()) EQUAL(2);
```

4.4 Identifying steps

You can identify certain steps, to make it easier to follow their evolution. It's particularly appropriate for checks[5]. 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, "switch on after") {
  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, "switch on after") {
  DECLARE(Tricorder t);
  STEP_ID(init_off); // in the tricorder medical test; no need to mention it
  CHECK(not t.medical.is_on()) TRUE;
}
```

4.5 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.

4.5.1 Print inline text: print_text – PRINT_TEXT

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

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

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

4.5.2 Print multiline text: print_multiline_text – PRINT_MULTILINE_TEXT

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

[5]—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.—

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

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

4.5.3 Print a break: `print_break` – `PRINT_BREAK`

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

```
print_break();
```

```
PRINT_BREAK();
```

4.6 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.

4.6.1 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
```

4.6.2 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
```

4.7 Unexpected exceptions

If an unexpected exception (i.e., an exception not in a "check-try-catch" instruction; see §4.2.6, 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-]`.

4.8 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 is going in. First, you can run the tests using a synchronous format like "`color_text`" (see §1), to see where the execution stops (which test and which test step). You can then go investigate and fix the error, or run the tests excluding that particular test to know what the current situation is, barring the failing test.

5 Fixtures

This is how fixtures are implemented in Testudo. If you want to code a fixture, you have to code a class that derives from "`testudo::Fixture`". Its constructor must accept a set set of parameters (just use the "test-parameters" macro), and pass them out to the constructor of "`testudo::Fixture`" (use the "test-arguments" macro). The constructor is the setup procedure; the destructor, if you code it, is the teardown procedure.

Here's an example:

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

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

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

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

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

DEFINE_TEST(delorean,
            no_pu_at_start, "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.

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[6], 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 : testudo::Fixture {
  NumbersFixture(test_parameters)
    : Fixture(test_arguments),
      fixture_init(x, 1.), fixture_init(z, 3.14) { }
  fixture_member(double x);
  fixture_member(double y=-2.5, z);
};
```

```
struct NumbersFixture
    : testudo::Fixture
{
  NumbersFixture(TEST_PARAMETERS)
    : Fixture(TEST_ARGUMENTS),
      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).

6 Adding Testudo support for your types

[6]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.

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

6.1 Textual representation

Implement "`ostream << t`", or "`to_text(t)`".

6.2 Equality

Implement "`t==u`".

6.3 Difference between two values

Implement "`double absdiff(t, u)`".

7 Testudo support for STL types

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

A Editor configuration

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

B Using your own test definition and test instruction names

Write a test style file, similar to "`lc.tst`" or "`cp.tst`" (use one of them as a template, and adapt the second half of each line).