



# Testudo, an automatic test system for C++ code

(version 0.02rc )

Miguel González Cuadrado  
<mgcuadrado@gmail.com>

22nd August 2020

## Contents

<b>Intro</b>	<b>ii</b>
<b>1 Test output formats</b>	<b>I</b>
<b>2 Test definition and test instruction styles</b>	<b>I</b>
<b>3 Tests and test hierarchies</b>	<b>2</b>
3.1 Non-top test nodes . . . . .	2
3.2 Top test nodes . . . . .	3
<b>4 Test steps</b>	<b>4</b>
4.1 Declarations and actions . . . . .	4
4.2 Checks . . . . .	4
4.3 Adding information to the report . . . . .	7
4.4 Printing fixed text and separations . . . . .	8
4.5 Fake declarations and actions . . . . .	9
4.6 Uncaught exceptions . . . . .	10
<b>5 Fixtures</b>	<b>10</b>
<b>6 Adding Testudo support for your types</b>	<b>II</b>
6.1 Textual representation . . . . .	II
6.2 Equality . . . . .	II
6.3 Difference between two values . . . . .	II

7	Testudo support for STL types	II
A	Editor configuration	II
B	Using your own test definition and test instruction names	II

## 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 L<sup>A</sup>T<sub>E</sub>X 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.

```

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

```

```

50 // this defines a test named "use_instructions", titled "use
51 // instructions"
52 define_top_test("testudo", use_instructions,
53                 "use instructions") {
54     print_multiline_text("index:\n"
55                          "  1. holding functionality\n"
56                          "  2. exceptions\n"
57                          "  3. list of forgotten doubles");
58     print_break(); // -----
59     declare(Hold hf);
60     check(not hf.is_holding())_true;
61     perform(hf.hold(3.14));
62     check(hf.is_holding())_true;
63     check(hf.pop())_approx(3.14);
64     check(not hf.is_holding())_true;
65     print_break(); // -----
66     print_text("hf is empty now");
67     check_try_catch(popping_empty, hf.pop());
68     // a couple of empty lines to get in sync with the report on the oposite
69     // page
70     perform(hf.hold(2.72));
71     check_try_catch(adding_to_already_holding, hf.hold(7.));
72     // ditto
73
74     print_break(); // -----
75     print_text("the forgotten doubles list is still empty");
76     check(Hold::is_forgotten_empty())_true;
77     check(Hold::get_forgotten().size())_equal(0u);
78     { show_scope("scope 1");
79       declare(Hold hf1(1.1)); // hold-on-construction syntax
80       declare(Hold hf2(2.2));
81       { show_scope(); // unnamed scope
82         declare(Hold hf3(3.3)); // will add to the list on scope closing
83       }
84       check(Hold::get_forgotten())_approx(list{3.})_tol(.5);
85       show_value(hf2.pop());
86       print_text("hf2 now empty; it won't add to the list");
87     }
88     check(not Hold::is_forgotten_empty())_true;
89     check(Hold::get_forgotten()
90           _approx(list{3.3, 1.1}));
91     perform(Hold::clear_forgotten());
92     check(Hold::get_forgotten().size())_equal(0u);
93     print_text("the following will raise an error");
94     perform(hf.hold(9.9));
95     print_text("this won't show, because of the error");
96 }
97 }
98 #endif

```

```

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

```



## I Test output formats

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

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

The “color\_text” format outputs the printout directly as a full text report with colours, virtually identical to the one “xml\_to\_color” produces. The difference is that this format produces its output synchronously, so even if an unrecoverable error happens, you’ll get the whole output until just before the error, whereas the “xml” won’t output anything<sup>1</sup>.

## 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:

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

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

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

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

---

<sup>1</sup>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 hand, *defining* a test node or a test means giving it full contents, including at least a name and a title, but possibly also a test function or a priority.

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

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

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

#### 3.1 Non-top test nodes

When you declare or define a test node whose parent has been defined in the current translation unit<sup>2</sup>, use the “define-test-node” or “define-test” syntaxes. As explained before, the execution order is the order in the code, so no priority is specified.

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

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

For a test (a test node with a test function), the syntax includes the definition of the test function itself, as if it were a regular C++ function, only with its declarator part (the one where you specify the return type, the function name, and the parameters) replaced by a Testudo macro. If you have already defined a test node called “medical”, here's how to define a child test called “switch\_on”, titled “switch

---

<sup>2</sup>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:

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

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

### 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”:

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

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

You can also define a top test (a top test node with a test function). So, here’s how to define a test called “doors\_closed\_on\_start”, child to a test node whose full name is “outatime.delorean”:

```
1 define_top_test("outatime.delorean", // parent full name
2                 doors_closed_on_start, // name
3                 "doors closed after construction", // title
4                 150) { // priority
5   declare(Delorean d);
```

```

6  check(not d.left_door.is_open())_true;
7  check(not d.right_door.is_open())_true;
8  }

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

```

## 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

#### 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.

```

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

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

```

#### 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.

```

1  perform(p.first+=10);

1  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.

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

```
1 check(captain_age)_equal(26+10);  
1 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<sup>3</sup>. 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.

```
1 check(computed_pi)_approx(2.*asin(1.));  
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 “ $\epsilon$ ” 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.

---

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

Define a value for  $\varepsilon$ : **define\_approx\_epsilon** – **DEFINE\_APPROX\_EPSILON**

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

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

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

Set the value of  $\varepsilon$ : **set\_approx\_epsilon** – **SET\_APPROX\_EPSILON**

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

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

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

Show the value of  $\varepsilon$ : **show\_approx\_epsilon** – **SHOW\_APPROX\_EPSILON**

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

```
1 show_approx_epsilon();
```

```
1 SHOW_APPROX_EPSILON();
```

#### 4.2.5 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.

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

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

#### 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 a descriptive name for a boolean variable, and the expression that is expected to throw. Testudo will create the boolean variable with an initial fals value, then run the expression within a try-block; if an exception is thrown, the boolean variable is made true, and the exception is reported. Finally, the boolean variable is checked for trueness, and enters thus the test tally.

```
1 declare(list<int> numbers);  
2 check_try_catch(dereference_null_ptr, numbers.front());
```

```
1 DECLARE(list<int> numbers);  
2 CHECK_TRY_CATCH(dereference_null_ptr, numbers.front());
```

## 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.

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

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

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

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

### 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.

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

```

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

```

## 4.4 Printing fixed text and separations

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

### 4.4.1 Print inline text: `print_text` – `PRINT_TEXT`

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

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

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

### 4.4.2 Print multiline text:

`print_multiline_text` – `PRINT_MULTILINE_TEXT`

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

```

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

```

```

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

```

### 4.4.3 Print a break: `print_break` – `PRINT_BREAK`

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

```
1 print_break();
```

```
1 PRINT_BREAK();
```

## 4.5 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.

### 4.5.1 Fake declaration: `fake_declare` – `FAKE_DECLARE`

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

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

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

### 4.5.2 Fake action: `fake_perform` – `FAKE_PERFORM`

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

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

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

## 4.6 Uncaught exceptions

If an uncaught exception (i.e., an exception not in a “check-try-catch” instruction; see §4.2.6) is raised in the course of a test, that particular test ends immediately, a description of the exception is written to the report, with a conspicuous flag, and the test is marked as having one error. Then, the test execution 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.

## 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:

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

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

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

```
1 define_test(delorean,
2             engine_starts_off, "engine is off at start",
3             with_fixture(OutATimeFixture)) {
4     check(not d->engine.is_running())_true;
5 }
```

```
1 DEFINE_TEST(delorean,
2             engine_starts_off, "engine is off at start",
3             WITH_FIXTURE(OutATimeFixture))
```



```
4 {  
5   CHECK(not d->engine.is_running()) TRUE;  
6 }
```

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.

## 6 Adding Testudo support for your types

### 6.1 Textual representation

### 6.2 Equality

### 6.3 Difference between two values

## 7 Testudo support for STL types

### A Editor configuration

### B Using your own test definition and test instruction names