fact

Release 0.3

Lukas Kaltenbrunner, Simon Priller

Dec 10, 2021

CONTENTS

This document presents the fact test framework which allows to test programming exercises written in the programming language C. The test types supported by the framework are Compilation, Input-/Output, Code structure, Grey-box and OCLint. More details about each test type and it's features are given further below. The framework was developed for use with the automated programming analysis system Artemis. Note that the framework is based on the C test template from Artemis.

CHAPTER

ONE

TEST TYPES

- **Compilation** tests check if the submitted code compiles successfully with the defined compiler call.
- **Input-/Output** tests check if a program produces the expected output and exit code given a predefined input.
- **Code structure** tests determine if the code meets predefined structural requirements.
- **Grey-box** tests determine if predefined functions return the expected values and have the expected side effects. Grey box tests are written in `python`.
- **OCLint** tests determine if the analyzed code adheres to the enabled OCLint rules

CHAPTER

TWO

FEATURES

- Modularity: The framework is very modular. Meaning that any test can occur with multiple different configurations in addition to all other test types. (Even though it is recommended that it is always checked that the source file compiles).
- Expandability: It is fairly simple to add new test types. A new test just has to inherit from the `AbstractTest`.
- Simplicity: It was designed to enable easy test definition, configuration, and execution. Tests can be configured via a YAML-file.
- GNU-Make: The build process is based on GNU make.
- Artemis support: The framework can be used with Artemis.

CHAPTER

THREE

TEST DEFINITION FILE

All test cases of the various test types have to be registered in the test definition YAML file. For details on test definition files see *Test definition file*.

CHAPTER

FOUR

WORKFLOW FOR ARTEMIS

After a new `C` programming exercise was created in Artemis. The following steps are required to create the tests.

1. Write a task description for students
2. Write a `Makefile`
3. Write a template for the exercise
4. Write a solution for the exercise
5. Write/Configure test definitions
6. Register the test definitions
7. Create the `Tests.py` file (will be automated in future versions)
8. Optional, but strongly suggested: Check your solution

CHAPTER

FIVE

DOCUMENT STRUCTURE

5.1 Test definition file

The test definition file is structured as follows:

- `version` states the used version of the framework.
- `translation_unit` states the default name of the translation unit that is analyzed if not explicitly stated otherwise in an item of `tests`.
- `tests` is a list containing the registration of the various test cases. Each test case supports the following attributes:
  - `type` states the type of the test. Currently supported types are *compile*, *io*, *structural*, *grey_box* and *oclint*.
  - `name` states the name of the test case.
  - `requirements` is a list of test cases that need to pass before this test case can be run. Required prerequisites are listed by `name`.

Subsequently, an example configuration containing possible attributes for each test type is given. After the example, each parameter contained in `tests` is explained in detail.

Note: The following attributes for the type `compile`, `io` and `grey_box` are complete while the types `structural` and `oclint` have more attributes described in their respective documentation sections.

```yaml
version: "0.0"
translation_unit: main.c
tests:
  - type: compile
    name: Compile
    makefile_directory: dir         # Default: '../assignment/'
    makefile_name: Makefile         # Default: Makefile
    make_target: main               # Default: see description below
    make_timeout: 2                 # Default: 5
    sourcecode_directory: dir       # Default: null
  - type: structural
    name: TestCodeStructure
    translation_unit: main.c        # Default: see description below
    makefile_directory: dir         # Default: '../assignment/'
    exec_timeout: 2                 # Default: 5
    sourcecode_directory: dir       # Default: null
  - type: io
```

(continues on next page)

(continued from previous page)

```
18      name: IO
19      io_test_config: main_io.txt     # Default: see description below
20      c_file: main.c                  # Default: see description below
21      makefile_directory: dir         # Default: '../assignment/'
22      makefile_name: Makefile         # Default: Makefile
23      make_target: main               # Default: see description below
24      make_timeout: 2                 # Default: 5
25      exec_timeout: 2                 # Default: 5
26      sourcecode_directory: dir       # Default: null
27      requirements:
28        - Compile
29        - CodeStructure
30    - type: grey_box
31      name: GreyBox
32      makefile_directory: dir         # Default: '../assignment/'
33      makefile_name: Makefile         # Default: Makefile
34      make_target: main.so            # Default: see description below
35      library_name: main.so           # Default: see description below
36      make_timeout: 2                 # Default: 5
37      exec_timeout: 2                 # Default: 5
38      module_path: grey_box.py        # Default: grey_box.py
39      class: GreyBoxTest              # Default: GreyBoxTest
40      unit_test: true                 # Default: true
41      sourcecode_directory: dir       # Default: null
42    - type: oclint
43      name: Linter
44      translation_unit: TODO          # Default: TODO
45      sourcecode_directory: dir       # Default: null
46      exec_timeout: 2                 # Default: 5
```

Note:

- `requirements`, like shown in line 27-29 as an example in the above configuration file, can be added for each test type
- execution of the individual tests is sequential. Keep this in mind when configuring `requirements`.
- adding new test types/ multiples of the same test type can be done by simply adding another section of the appropriate type.
- not all test types have to be present

5.1.1 Test type details

`type: compile`

- `name`: Required attribute. This parameter represents the compile test case's name.
- `makefile_directory`: The default value is '*../assignment/*'. This parameter represents the path to the directory that contains the Makefile.
- `makefile_name`: The default value is *Makefile*. This parameter represents the Makefile's filename. The file must be located in `makefile_directory`.

- `make_target`: The default value is the file name (without file-extension) given in the top level's `translation_unit`. This parameter represents the name of the rule within `makefile_name` that is executed for this test case.
- `make_timeout`: The default value is *5*. This parameter represents the maximum number of seconds allotted for the complete execution of `make_target`.
- `sourcecode_directory`: The default value is *null*. If *null*, the source code is supposed to be in the same directory as the Makefile.

type: structural

- `name`: Required attribute. This parameter represents the structural test case's name.
- `translation_unit`: The default value is the top level's `translation_unit`. This parameter represents the translation unit analyzed in this test case.
- `makefile_directory`: The default value is *'../assignment/'*. This parameter is used as fallback if `sourcecode_directory` is not set.
- `exec_timeout`: The default value is *5*. This parameter represents the maximum number of seconds allotted for the execution of the structural test.
- `sourcecode_directory`: The default value is *null*. If *null*, the source code is supposed to be in the same directory as the Makefile.

Additional parameters that are relevant only for structural tests can be found in the dedicated *Code Structure* documentation.

type: io

- `name`: Required attribute. This parameter represents the IO-test case collection's name.
- `io_test_config`: The default value is the file name (without file-extension) given in the top level's `translation_unit` with the added suffix *_io.txt*. This parameter represents the IO-test case collection's file name. The file contains the definitions of the individual *Input-/Output* test cases.
- `c_file`: The default value is the top level's `translation_unit`. This parameter represents the c-file in which the *Substitution* or the *Replacement* has to be made. This parameter is only needed if the `io_test_config` contains *Substitution* and/or *Replacement* tests (two special cases of IO-tests).
- `makefile_directory`: The default value is *'../assignment/'*. This parameter represents the path to the directory that contains the Makefile and the compiled executable.
- `makefile_name`: The default value is *Makefile*. This parameter represents the Makefile's filename. The file must be located in `makefile_directory`. This parameter is only needed if the `io_test_config` contains *Substitution* and/or *Replacement* tests (two special cases of IO-tests).
- `make_target`: The default value is the file name (without file-extension) given in the top level's `translation_unit`. This parameter represents the name of the rule within `makefile_name` that is executed for this test case. This parameter is only needed if the `io_test_config` contains *Substitution* and/or *Replacement* tests (two special cases of IO-tests).
- `make_timeout`: The default value is *5*. This parameter represents the maximum number of seconds allotted for the complete execution of `make_target`. This parameter is only needed if the `io_test_config` contains *Substitution* and/or *Replacement* tests (two special cases of IO-tests).
- `exec_timeout`: The default value is *5*. This parameter represents the maximum number of seconds allotted for the complete execution of **each individual test case** defined in `io_test_config`.

- `sourcecode_directory`: The default value is *null*. If *null*, the source code is supposed to be in the same directory as the Makefile.

`type: grey_box`

- `name`: Required attribute. This parameter represents the GreyBox-test case collection's name.
- `makefile_directory`: The default value is *'../assignment/'*. This parameter represents the path to the directory that contains the Makefile.
- `makefile_name`: The default value is *Makefile*. This parameter represents the Makefile's filename. The file must be located in `makefile_directory`.
- `make_target`: The default value is the file name (without file-extension) given in the top level's `translation_unit` with the added suffix *.so*. This parameter represents the name of the makefile rule used to compile the student's code as a shared library.
- `library_name`: The default value is the file name (without file-extension) given in the top level's `translation_unit` with the added suffix *.so*. This parameter represents the name of the student submission's code compiled as a shared library.
- `make_timeout`: The default value is *5*. This parameter represents the maximum number of seconds allotted for the execution of the GreyBox-test.
- `exec_timeout`: The default value is *5*. This parameter represents the maximum number of seconds allotted for the complete execution of **all test cases** defined in GreyBox-test case collection.
- `module_path`: The default value is *grey_box.py*. This parameter represents the name of the python file containing the GreyBox-test case collection's implementation.
- `class`: The default value is *GreyBoxTest*. This parameter represents the name of the class that implements *GreyBoxTestRunner*.
- `unit_test`: The default value is *true*. If *true*, the error message is function-specific. Otherwise, the error messaging is slightly adjusted for a procedural test.
- `sourcecode_directory`: The default value is *null*. If *null*, the source code is supposed to be in the same directory as the Makefile.

`type: oclint`

- `name`: Required attribute. This parameter represents the oclint-test case's name.
- `translation_unit`: The default value is the top level's `translation_unit`. This parameter represents the translation unit analyzed in this test case.
- `sourcecode_directory`: The default value is *null*. If *null*, the source code is supposed to be in the same directory as the Makefile.
- `exec_timeout`: The default value is *5*. This parameter represents the maximum number of seconds allotted for the complete execution of **all test cases** defined in GreyBox-test case collection.

Additional parameters that are relevant only for oclint tests can be found in the dedicated *OCLint* documentation.

5.2 Compilation

A compilation test tries to execute a GNU-Make rule given in a Makefile. If errors occur during the execution (compilation) of the target, the test is marked as failed, and the error messages generated when executing the makefile rule are returned as feedback for the student. In case no errors occur, the student gets the feedback that the test has been run successfully.

The only limitations for compilation tests are set by what the chosen compiler offers. When using GCC as the compiler, among others, the following test cases would be possible. For the examples, it is supposed that a `ex.c`-file is the only necessary C-file. The snippets are to be interpreted as GNU Makefile with implicit rules.

5.2.1 Compilation

A simple check if the submitted code compiles at all:

```
CC=gcc
CFLAGS=-std=c11

ex: ex.c
```

5.2.2 Warnings

Check if the submitted code compiles without warnings:

```
CC=gcc
CFLAGS=-Wall -Werror -Wextra -Wpedantic -std=c11

ex: ex.c
```

5.3 Input-/Output

An Input-/Output test (IO-test) may consist of multiple test cases. For each test case, the framework runs the executable and provides it with the command-line arguments and `stdin`-input and checks if the obtained exit code, `stdout`-output and `stderr`-output match the expected values. If the output and error code does not match the expected values, the test is marked as failed, and the error messages generated by the framework are collected. Once all test cases have been run, the errors are returned as feedback for the student. In case no errors occur, the student gets the feedback that the IO-test has been run successfully.

An IO-test is defined in a domain-specific language (DSL). The DSL has been designed to write test cases concisely. Details on the DSL and example IO-test definitions are given further below. Additionally, for future versions of the framework, we plan to add the possibility to define IO-tests directly in the main test definition.

A particular case of IO-tests are replacement/substitution tests. For these kinds of tests, the framework performs a substitution of parts of the student's code before recompiling and running an "ordinary" IO-test. These kinds of test cases can be useful to test code before input via command line parameters, or stdin processing are known to the students.

5.3.1 The error message

If an IO-test fails, the error message is divided into two sections: general overview and test case details. The details for each failed test case - depending on the adopted matching strategy (c.f. *start>* for details on matching strategies) - are:

- *Status*: A summary of what was OK and what was not.
- *Commandline parameters*: The command line parameters passed to the tested executable.
- *Input*: The input piped to the tested executables `stdin`.
- *Obtained output on stdout*: The obtained `stdout`-output produced by the analyzed executable if it was not as expected.
- *Expected output on stdout*: The expected `stdout`-output.
- *Hint stdout*: The hint indicating the necessary changes for the `stdout`-output (shown only if the analyzed `stdout`-output was not OK).
- *Obtained output on stderr*: The obtained `stderr`-output produced by the analyzed executable if it was not as expected.
- *Expected output on stderr*: The expected `stderr`-output.
- *Hint stderr*: The hint indicating the necessary changes for the `stderr`-output (shown only if the analyzed `stderr`-output was not OK).
- *Tested code*: Shows the code that is compiled and then tested after a *Replacement* and/or *Substitution* was performed.
- *Hint*: The hint to clarify what could have gone wrong, in case one was set in the IO-test case definition.

Note: If the output produced by the tested code contains characters outside of the ASCII range, the following error message is shown as part of the standard error messages Status section:

```
Your program generates output containing invalid characters that are outside of the ASCII
range (below 0 or above 127)!
```

Subsequently, two example error messages are given:

1. An example error message for a program that should determine if the value assigned to the int-variable `n` is divisible by seven or not and print `The number <n> is divisible by seven!` or `The number <n> is not divisible by seven!` respectively. `<n>` should be replaced with the value of the variable `n`. The full example, including test definition and accepted solution, can be found here (*Example 2*)
2. An example where a timeout error occurred.

More details about the error messages sections are given right after the examples.

Example 1:

```
Error: The output of your program does not match the expected output. In total 1 test␣
↪case failed:


                                    Test case
--------------------------------------------------------------------------------

Status
```

(continues on next page)

(continued from previous page)

```
======
Correct return code! Expected: '0' Obtained: '0'
Wrong output on stdout!
Wrong output on stderr!


Commandline parameters (none)
=============================


Input (none)
============


Substitutions
=============
- In line 4 of the tested code the value 11 assigned to n has been substituted with 5.

Obtained output on stdout
=========================
the number 11 is NOT divissible by sven


Expected output on stdout
=========================
The number 5 is not divisible by seven!

Hint stdout
===========
[-t][+T]he number [11=>should change automatically if n changes!] is [-NOT][+not] divi[-
↪s]sible by s[+e]ven[+!]

Obtained output on stderr
=========================
Error message to make the stderr paragraphs appear!


Expected output on stderr (none)
================================


Hint stderr
===========
[-Error message to make the stderr paragraphs appear!]

Tested code
===========
#include <stdlib.h>
#include <stdio.h>
int main(void) {
    int n = 5;
    if (n % 7) {
```

(continues on next page)

(continued from previous page)

```
        printf("the number 11 is NOT divissible by sven\n");
    } else {
        printf("the number %d is divissible by sven\n", n);
    }
    fprintf(stderr, "Error message to make the stderr paragraphs appear!\n");
    return EXIT_SUCCESS;
}



Hint (none)
===========
No hint available! Please, read the exercise description very carefully!
```

Example 2:

```
Error: The output of your program does not match the expected output. In total 1 test␣
↪case failed:


                                    Test case
-------------------------------------------------------------------------------

Status
======
Timeout: The execution of your program was canceled since it did not finish after 1␣
↪seconds!
This might indicate that there is some unexpected behavior (e.g., an endless loop) or␣
↪that
your program is very slow!


Commandline parameters (none)
=============================


Input (none)
============


Hint (none)
===========
No hint available! Please, read the exercise description very carefully!
```

Status

The *Status*-section of the test case details summarizes what went wrong and what was OK for the given test case. It is always comprised of three lines except if a timeout occurred and the test was aborted:

- The first line states whether the return code generated by the analyzed program was correct or not. It also displays the expected and obtained return codes.
- The second line states whether the output obtained on `stderr` was correct or not.
- The third line states whether the output obtained on `stdout` was correct or not.

At least one of the three lines must be a negative message. Alternatively - in case of a timeout - this section contains the following message:

```
Timeout: The execution of your program was canceled since it did not finish after 5␣
↪seconds!
This might indicate that there is some unexpected behavior (e.g., an endless loop) or␣
↪that
your program is very slow!
```

One of the two aforementioned cases is always true. Otherwise, the test case would not have failed.

Commandline parameters

The *Commandline parameters*-section of the test case details lists the command line arguments passed to the given test case's executable. In case no command line parameters were passed to the executable `(none)` is added to the section title for clarification.

Each passed command line parameter is wrapped by double quotes (") multiple command line parameters are divided by a whitespace character. If, for example, a program was given the two parameters `"p1"` and `"p2"` and the exit code, `stdout`-output or `stderr`-output was not as expected, the `Commandline parameters` section would contain `"p1" "p2"`

Note: This section is only shown if `show_input` is true. For more details about `show_input` and how it is used, see section *Start*.

Input

The *Input*-section of the test case details lists the values passed to the executables `stdin` stream. In case nothing is passed to the `stdin` stream `(none)` is added to the section title for clarification.

Each input value is provided in a separate line. Strings are wrapped in double quotes and can span over more than one line while still counting as one input value.

Note: This section is only shown if `show_input` is true. For more details about `show_input` and how it is used, see section *Start*.

Obtained output on stdout

The *Obtained output on stdout*-section of the test case details lists the output the executable produced on the `stdout` stream if it differs from the expected output. In case nothing was produced on the `stdout` stream and some kind of output was expected `(none)` is added to the section title for clarification.

Note: This section is shown by default if the produced `stdout`-output is not as expected. It can be turned off by setting the `show_output` argument. For more details about `show_output` and how it is used, see section *Start*. Please note that by setting `show_output=false` for *exact* matching *Obtained output on stderr* is also turned off.

Expected output on stdout

The *Expected output on stdout* section of the test case details lists the output the analyzed executable is expected to produce on the `stdout` stream. In case nothing is expected on the `stdout` stream `(none)` is added to the section title for clarification. This section is only available for *exact* matching.

Note: This section is not shown by default, but it can be turned on by setting the `show_expected` argument. For more details about `show_expected` and how it is used, see section *start> matching="exact"*.

Hint stdout

The *Hint stdout* section of the test case details shows the necessary changes for the `stdout`-output (shown only if the analyzed output was not OK). This section is only available for *exact* matching.

Note: This section is only shown if the analyzed output was not OK. Hints for `stdout` and `stderr` can be disabled, setting the `show_diff` argument. For more details about `show_diff` and how it is used, see section *start> matching="exact"*.

Obtained output on stderr

The *Obtained output on stderr*-section of the test case details lists the output the executable produced on the `stderr` stream if it differs from the expected output. In case nothing was produced on the `stderr` stream and some kind of output was expected `(none)` is added to the section title for clarification.

Note: This section is shown by default if the produced `stderr`-output is not as expected. It can be turned off by setting the `show_error` argument for *regex* and the `show_output` argument for *exact* matching. For more details about `show_error` and how it is used, see section *start> matching="regex"*. Please note that by setting `show_output=false` for *exact* matching *Obtained output on stdout* is also turned off.

Expected output on stderr

The *Expected output on stderr* section of the test case details lists the output the analyzed executable is expected to produce on the `stderr` stream. In case nothing is expected on the `stderr` stream `(none)` is added to the section title for clarification. This section is only available for *exact* matching.

Note: This section is not shown by default, but it can be turned on by setting the `show_expected` argument. For more details about `show_expected` and how it is used, see section *start> matching="exact"*.

Hint stderr

The *Hint stderr* section of the test case details shows the necessary changes for the `stderr`-output (shown only if the analyzed output was not OK). This section is only available for *exact* matching.

Note: This section is only shown if the analyzed output was not OK. Hints for `stdout` and `stderr` can be disabled, setting the `show_diff` argument. For more details about `show_diff` and how it is used, see section *start> matching="exact"*.

Tested code

The *Tested Code* section shows the code that is compiled and then tested after a *Replacement* and/or *Substitution* was performed.

Note: This section is only shown if a a *Replacement* and/or *Substitution* was performed. It can be disabled by setting the `show_substitution` argument. For more details about `show_substitution` and how it is used, see section *start> matching="exact"* and *start> matching="regex"* .

Hint

The *Hint* section of the test case details displays a hint defined for the given test case. If no hint was defined, `(none)` is added to the section title for clarification. Additionally, the message `No hint available! Please, read the exercise description very carefully!` is added to the hint section.

5.3.2 Domain-Specific Language (DSL)

This DSL is used to define IO-test cases for this framework. The supported commands are (in order of possible occurrence):

- *start> (required)* marks the start of a test case
- *r> (optional)* defines a replacement (replacement of a predefined character sequence with a character sequence defined in the test definition)
- *s> (optional)* defines a substitution (substitution of the initialization of a predefined variable with another value)
- *p> (optional)* command line parameters. Each parameter as a string.
- *Statement (can be used multiple times and in arbitrary order)*:

  - *i>* *(optional)* input: value that will be piped to the executables `stdin`. All defined input values are concatenated. A `\n` is added after each value.
  - *o>* *(optional)* output: confront the details for `matching="regex"` and `matching="exact"` in their respective sections.
  - *e>* *(optional)* error: confront the details for `matching="regex"` and `matching="exact"` in their respective sections.
  - *v>* *(optional)* variable: defines a variable that is valid for the rest of the test case definition.
- *end>* *(required)* marks the end of the test case.

More details on each instruction and its required parameters are given below.

Start `start>`

The keyword `start>` marks the beginning of an IO-test case. It supports a series of optional arguments depending on the chosen matching strategy. The matching strategy determines how output comparison works and how the feedback is generated. Possible values for `matching` are `"exact"` and `"regex"`.

`start> matching="exact"`

In case of `matching="exact"` for *o>* and *e>* elements exact matching is performed. In this case, a diff between the expected and obtained output can be shown.

Example diff output:

- obtained output = `the number 11 is NOT divissible by sven`
- expected output = `The number 5 is not divisible by seven!`
- diff output = `[-t][+T]he number [11=>should change automatically if n changes!] is [-NOT][+not] divi[-s]sible by s[+e]ven[+!]`

The values in `[]` show the necessary changes to get to the expected output. Possible operators are:

- + add characters
- - remove characters
- -> replace characters
- => dedicated error hint/message

Additionally the following optional arguments are supported:

- `show_input` (default= *false*): If *true*, feedback for the students includes the command line argument(s) and the stdin-input provided to the executable. By default, command-line arguments and stdin-input are not shown in case of errors occur.
- `show_output` (default= *true*): If *false*, feedback for the students does not include the output their program provided on stdout and stderr. By default, if the stdout-output differs from the expected stdout-output, it is shown in the feedback. The same holds for stderr-output.
- `show_expected` (default= *false*): If *true*, feedback for the students contains the expected output as defined in the test definition. By default, the expected output on stdout and stderr is not shown in case errors occur.
- `show_diff` (default= *true*): If *false*, feedback for the students does not contain the diff output like it is shown in the third line of the example above. By default the diff to the expected output on stdout and/or stderr is shown in case an error occurs.

- `hint` (default= *No hint available! Please, read the exercise description very carefully!*): The string assigned to this argument is added to the error message provided to the student in the *Hint* section of the testfeedback. If no hint is defined the default message is printed.
- `ignore_cases` (default= *false*): If *true* the comparison of expected and obtained output for stdout and stderr is done in a case insensitive manner. By default the comparison is case sensitive.
- `rstrip` (default= *false*): If *true* trailing whitespace characters at the end of the concatenated output string are removed before comparison. By default the output string is not modified before comparison.
- `line_rstrip` (default= *false*): If *true* trailing blanks and tabs at the end of each line are ignored.
- `escape` (no default): The string assigned to this argument defines the character escape sequence that starts and ends code injection. Code can be injected in *i>*, *o>*, *e>* and *p>* statements
- `show_substitution` (default= *true*): If *false* the feedback does not contain the code after *Replacement* and/or *Substitution*. By default this section is only present if at least one *Replacement* or *Substitution* was performed.
- `printable_ascii` (default= *false*): If *true* the output can contain only printable ascii characters.

Examples

Minimal `start>` for exact:

```
start> matching="exact"
```

Maximal `start>` for exact with default values:

```
start> matching="exact" show_input=False show_output=True show_expected=False show_
↪diff=True
       ignore_cases=False rstrip=False line_rstrip=False show_substitution=True
       printable_ascii=False
       hint="No hint available! Please, read the exercise description very carefully!"
```

Note:

- `escape` has no default value and is thus not mentioned in the maximal example with default values (confront *Example 1 (escape)* for an example on how to use `escape`)
- This produces the same error message the minimal example produces.

`start> matching="regex"`

In case of `matching="regex"` for *o>* and *e>* elements regex matching is performed. Additionally the following optional arguments are supported:

- `show_input` (default= *false*): If *true*, feedback for the students includes the command line argument(s) and the stdin-input provided to the executable. By default, command-line arguments and stdin-input are not shown in case errors occur.
- `show_output` (default= *true*): If *false*, feedback for the students does not include the output their program provided on stdout. By default, if the stdout-output differs from the expected stdout-output, it is shown in the feedback.
- `show_error` (default= *true*): If *false*, feedback for the students does not include the output their program provided on stderr. By default, if the stderr-output differs from the expected stderr-output it is shown in the feedback.

- `hint` (default= *No hint available! Please, read the exercise description very carefully!*): The string assigned to this argument is added to the error message provided to the student in the *Hint* section of the testfeedback. If no hint is defined the default message is printed.
- `escape` (no default): The string assigned to this argument defines the character escape sequence that starts and ends code injection. Code can be injected in *i>*, *o>*, *e>* and *p>* statements
- `show_substitution` (default= *true*): If *false* the feedback does not contain the code after *Replacement* and/or *Substitution*. By defaultthis section is only present if at least one *Replacement* or *Substitution* was performed.
- `printable_ascii` (default= *false*): If *true* the output can contain only printable ascii characters.

Examples

Minimal `start>` for regex:

```
start> matching="regex"
```

Maximal `start>` for regex with default values:

```
start> matching="regex" show_input=False show_output=True show_error=True show_
↪substitution=True
       printable_ascii=False
       hint="No hint available! Please, read the exercise description very carefully!"
```

Note:

- `escape` has no default value and is thus not mentioned in the maximal example with default values (confront *Example 1 (escape)* for an example on how to use `escape`)
- This produces the same error message the minimal example produces.

Replacement `r>`

The keyword `r>` defines a replacement, which is the replacement of substrings matching a predefined regular expression with a predefined string in the submitted code file. It is possible to define 0 or more replacements per test case. Each replacement has the following required parameters:

1. `pattern`: all parts of the code file matching this python regex pattern are replaced with the second argument. Type: *STRING*
2. `replacement`: all character sequences matching the first argument are replaced by this character sequence. Type: *STRING*
3. `hint`: character sequence that is shown in the error message should replacement fail. Type: *STRING*
4. `num_matches`: number of expected matches in the code. If the `pattern` does not match the given number of times, the substitution is considered a failure, and an error message containing the hint is shown. Type: *INT*

Note: Before performing the replacement, comments in the code are removed.

Example

```
r> "\"Hello World\\n\"" "\"Hello People!\\n\"" "Please, read the exercise description␣
↪very carefully! Your code should contain the character sequence '\"Hello World\n\"'␣
↪exactly once!" 1
```

Note: The character " has to be escaped inside a string. The same goes for the character \

Substitution s>

The keyword `s>` defines a substitution, which is the substitution of a variable's initialization with a predefined value. It is possible to define 0 or more substitutions per test case. Each substitution has the following required parameters:

1. `variable`: all initializations of the variable matching this regex pattern are replaced with the second argument. Type: *STRING*
2. `value`: this character sequence replaces the value the variable was initialized with. Type: *STRING*
3. `hint`: character sequence that is shown in the error message should substitution fail. Type: *STRING*
4. `num_matches`: number of expected matches in the code. If the number if initializations of the given `variable` does not match the given number of times, the substitution is considered a failure, and an error message containing the hint is shown. Type: *INT*

Note: Before performing the substitution, comments in the code are removed.

Example

Task: Define a variable named `n` of type `int`. Initialize `n` with an integer value of your choice between 1 and 2000 (limits included) and print the following text where <n> should be the value you initialized `n` with.

```
The value of n is <n>!
```

Full IO-test case definition:

```
start> matching="exact" rstrip=True
s> "n" "100" "Please, read the exercise description very carefully! Your printf␣
↪statement should not be hard coded!" 1
o> "The value of n is 100!"
end> 0
```

- *Solution 1 (accepted):*

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int n = 12;
    printf("The value of n is %d!\n", n);
    return EXIT_SUCCESS;
}
```

- *Solution 2 (not accepted):*

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    printf("The value of n is 100!\n");
```

(continues on next page)

(continued from previous page)

```
    return EXIT_SUCCESS;
}
```

Resulting error message:

```
Error: The output of your program does not match the expected output. In total 1␣
↪test case failed:


Please, read the exercise description very carefully! Your printf statement should␣
↪not be hardcoded!
```

- *Solution 3 (possibly accepted):*

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int n = 1;
    printf("The value of n is 100!\n");
    return EXIT_SUCCESS;
}
```

Note: Depending on the flags used for compilation, this solution may be accepted even though the variable n was never used. By adding a second test case and changing the substitution value hardcoded outputs can always be detected.

Improved testcase definition:

```
start> matching="exact" rstrip=True
s> "n" "100" "Please, read the exercise description very carefully! Your printf␣
↪statement should not be hard coded!" 1
o> "The value of n is 100!"
end> 0

start> matching="exact" rstrip=True
s> "n" "110" "Please, read the exercise description very carefully! Your printf␣
↪statement should not be hardcoded!" 1
o> "The value of n is 110!"
end> 0
```

Resulting error message:

```
Error: The output of your program does not match the expected output. In total 1␣
↪test case failed:


                                    Test case
------------------------------------------------------------------------------
↪--

Status
```

(continues on next page)

(continued from previous page)

```
======
Correct return code! Expected: '0' Obtained: '0'
Wrong output on stdout!
Correct output on stderr!


Substitutions
=============
- In line 4 of the tested code the value 1 assigned to n has been substituted with␣
↪110.

Obtained output on stdout
=========================
The value of n is 100!


Hint stdout
===========
The value of n is 1[0->1]0!

Tested code
===========
#include <stdio.h>
#include <stdlib.h>
int main() {
    int n = 110;
    printf("The value of n is 100!\n");
    return EXIT_SUCCESS;
}



Hint (none)
===========
No hint available! Please, read the exercise description very carefully!
```

Command-line parameters p>

The keyword p> defines the command line parameters, that is the command-line arguments given to the executable. The parameters are a list of double-quoted strings separated by a whitespace character. Each string corresponds to a separate parameter. The absence of p> in an IO-test case definition means no command line parameters.

Example:

```
p> "content of argv[1]" "content of argv[2]"
```

Statement

Statements are the main elements of an IO-test case. The different kinds of statements can occur multiple times in the same test case definition. The order they can appear in is arbitrary. Please note that *o>* and *e>* statements differ slightly for `matching="regex"` and `matching="exact"`. More details are given in their in their respective sections.

Input `i>`

The keyword `i>` defines input that is piped to `stdin`. It expects an argument of type *STRING*. All `i>` strings of one test case are joined with newline characters and piped to the `stdin` stream of the tested executable. Additionally, python code can be injected in the input statement if an escape sequence has been defined beforehand as an argument of *start>* (`escape` is available for both `regex`, and `exact` matching).

Example 1

The segment

```
i> "1"
i> "1"
i> "1"
i> "1"
```

would result in `"1\n1\n1\n1\n"` being piped to the tested executable `stdin` stream. The same can be achieved by simply writing

```
i> "1\n1\n1\n1"
```

directly.

Example 2: The following test definition generates the current date in the format `YYYY-mm-dd` as input for the program.

```
start> matching="exact" show_input=True escape="$$"
i> "$$import time; print(time.strftime('%Y-%m-%d'))$$"
end> 0
```

Output `o>`

The keyword `o>` defines output that is expected on `stdout`. The `o>` sections slightly differ between `matching="regex"` and `matching="exact"` more details on the differences in the respective sections below.

`o>` for `matching="regex"`

Within the `matching="regex"` context `o>` expects one argument of type *STRING*. This string is interpreted as a python regex. All `o>` strings of one test case are joined, the `stdout` output of the tested executable is matched against the resulting regex.

Examples

The segment

```
o> "Hello World!?\n?"
```

would accept all executables that produce the string `Hello World` followed by an optional exclamation point character and an optional newline character on the `stdout` stream. The same can be achieved by writing

```
o> "Hello "
o> "World"
o> "!?\n?"
```

o> for matching="exact"

Within the matching="exact" context o> expects one argument of type *STRING*. Optionally, a second *STRING* can be provided. The first argument is the exact character sequence that is expected as the output on stdout. The second - optional - argument defines a dedicated error hint or message that is shown if the first argument does not match the output exactly. If the second argument is missing, the hint is automatically generated to showcase the necessary changes in the output.

Example

The segment

```
o> "The resulting number is: "
o> "5" "wrong computation!"
o> "!"
```

would only accept the character sequence The resulting number is: 5!. Let's suppose the output produced by the analyzed program was the resullting numbr is: 11!! then the diff feedback would be [-t][+T]he resu[-l]lting numb[+e]r is: [11=>wrong computation!]![-!]. Let's split the feedback up into three parts (the three parts of the output declaration):

- [-t][+T]he resu[-l]lting numb[+e]r is: for the first part all necessary changes are shown individually.
- [11=>wrong computation!] for the second part, only the error message and no diff is shown because the additional argument was given in the second line of the o> defined above.
- ![-!] for the third part, all necessary changes are shown individually.

Possible operators inside of [] are:

- + add characters
- - remove characters
- -> replace characters
- => dedicated error hint/message

Error e>

The keyword e> defines output that is expected on stderr. The e> sections slightly differ between matching="regex" and matching="exact". The e> keyword works basically the same as the *o>* keyword with stderr instead of stdout and e> instead of o>.

Variable v>

The keyword v> defines a variable definition for statements, i.e., from the definition point onwards, each character sequence defined in the first argument appears inside an argument of non-variable definition statements is replaced by the character sequence defined in the second argument. It expects the following arguments:

1. var_name: the character sequence serving as a variable, i.e., the character sequence that is to be replaced. Type: *STRING*
2. value: each occurrence of the first argument inside an argument of a non-variable definition statement is replaced by this character sequence. Type: *STRING*

Example

In the following example, | stands for zero to five blank or tab characters.

```
v> "|" "[ \t]{0,5}"

o> "|a: |5\n"
o> "|b: |2\n"
o> "|c: |3\n"
o> "|other: |10[\n]*"
```

the same statements without variable substitution would be

```
o> "[ \t]{0,5}a: [ \t]{0,5}5\n"
o> "[ \t]{0,5}b: [ \t]{0,5}2\n"
o> "[ \t]{0,5}c: [ \t]{0,5}3\n"
o> "[ \t]{0,5}other: [ \t]{0,5}10[\n]*"
```

Note: Variables in statements have been introduced to make statements more comfortable to read and less cluttered with regular expressions. The variable names have to be chosen carefully because every occurrence of the var_name character sequence is replaced by value!

Delimiter end>

The keyword end> marks the end of an IO-test case. Optionally, a list of one or more *INT*-values separated by , can be given. These arguments represent the accepted exit-codes for the program. If no *INT*-values are given all exit-codes are accepted.

Examples

```
end>
```

```
end> 0
```

```
end> 1
```

```
end> 1,2,3
```

5.3.3 Examples

Example 1 `escape` **:**

IO-Test definition:

```
start> matching="exact" escape="$$"
o> "Date: $$import time; print(time.strftime('%Y-%m-%d'), end='')$$"
end> 0
```

Solution 1 (accepted):

```c
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

int main(void) {
    time_t t = time(NULL);
    struct tm* tm;
    char date[11];
    tm = localtime(&t);

    strftime(date, sizeof(date), "%Y-%m-%d", tm);
    printf("Date: %s", date);
    return EXIT_SUCCESS;
}
```

Solution 2 (not accepted):

```c
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

int main(void) {
    time_t t = time(NULL);
    struct tm* tm;
    char date[11];
    tm = localtime(&t);

    strftime(date, sizeof(date), "%Y-%m-%d", tm);
    printf("%s\n", date);
    return EXIT_SUCCESS;
}
```

Error feedback for solution 2:

```
Error: The output of your program does not match the expected output. In total 1 test␣
↪case failed:


                                          Test case
------------------------------------------------------------------------------------------
↪------
```

(continues on next page)

(continued from previous page)

```
Status
======
Correct return code! Expected: '0' Obtained: '0'
Wrong output on stdout!
Correct output on stderr!


Obtained output on stdout
=========================
2021-11-12


Hint stdout
===========
[+Date: ]2021-11-12[-
]

Hint (none)
===========
No hint available! Please, read the exercise description very carefully!
```

Example 2:

IO-Test definition:

```
start> matching="exact" show_input=true show_expected=true rstrip=true
s> "n" "5" "Please, read the exercise description very carefully! Your printf statement␣
↪should not be hard coded!" 1
o> "The number "
o> "5" "should change automatically if n changes!"
o> " is not divisible by seven!"
end> 0
```

Solution 1 (accepted):

```c
#include <stdlib.h>
#include <stdio.h>

int main(void) {
    int n = 11;
    if (n % 7) {
        printf("The number %d is not divisible by seven!\n", n);
    } else {
        printf("The number %d is divisible by seven!\n", n);
    }
    return EXIT_SUCCESS;
}
```

Solution 2 (not accepted):

```c
#include <stdlib.h>
#include <stdio.h>
```

(continues on next page)

(continued from previous page)

```
int main(void) {
    int n = 11;
    if (n % 7) {
        printf("the number 11 is NOT divissible by sven\n");
    } else {
        printf("the number %d is divissible by sven\n", n);
    }
    fprintf(stderr, "Error message to make the stderr paragraphs appear!\n");
    return EXIT_SUCCESS;
}
```

Error feedback for solution 2:

```
Error: The output of your program does not match the expected output. In total 1 test␣
↪case failed:


                                      Test case
------------------------------------------------------------------------------------------

Status
======
Correct return code! Expected: '0' Obtained: '0'
Wrong output on stdout!
Wrong output on stderr!


Commandline parameters (none)
=============================


Input (none)
============


Substitutions
=============
- In line 4 of the tested code the value 11 assigned to n has been substituted with 5.

Obtained output on stdout
=========================
the number 11 is NOT divissible by sven


Expected output on stdout
=========================
The number 5 is not divisible by seven!

Hint stdout
===========
[-t][+T]he number [11=>should change automatically if n changes!] is [-NOT][+not] divi[-
↪s]sible by s[+e]ven[+!]
```

(continues on next page)

(continued from previous page)

```
Obtained output on stderr
=========================
Error message to make the stderr paragraphs appear!


Expected output on stderr (none)
================================


Hint stderr
===========
[-Error message to make the stderr paragraphs appear!]

Tested code
===========
#include <stdlib.h>
#include <stdio.h>
int main(void) {
    int n = 5;
    if (n % 7) {
        printf("the number 11 is NOT divissible by sven\n");
    } else {
        printf("the number %d is divissible by sven\n", n);
    }
    fprintf(stderr, "Error message to make the stderr paragraphs appear!\n");
    return EXIT_SUCCESS;
}



Hint (none)
===========
No hint available! Please, read the exercise description very carefully!
```

5.4 Code Structure

A code structure test analyzes the structure of a translation unit's source code and enforces the predefined rules. Each translation unit requires its own set of rules. If the analyzed code violates at least one rule, the test is failed, and feedback on what rules have been violated is given. Otherwise, feedback that the code structure test has been passed successfully is given.

There are three different types of rules:

1. *General rules:* These rules can be defined on every level.
2. *Global rules:* These rules can only be defined at a global level.
3. *Function rules:* These rules can only be defined at the function level.

Note: If a rule is defined globally, each function is expected to abide by it, but rules on the function level supersede

global definitions within their bodies.

In the following sections, the rules, their meaning, and default values are explained in detail. The remainder of this section lists the different rules for each level.

General rules:

- *input*: allow/disallow/expect `stdin` input.
- *output*: allow/disallow/expect `stdout` output.
- *insecure*: allow/disallow/expect `stderr` output.
- *recursion*: allow/disallow/expect recursion.
- *disallowed_keywords*: List of disallowed keywords.
- *expected_keywords*: List of expected keywords.
- *allowed_function_calls*: List of allowed function calls.
- *disallowed_function_calls*: List of disallowed function calls.
- *expected_function_calls*: List of expected function calls.
- *expected_variable_declarations*: List of expected variable definitions.

Rules exclusive to the global definition:

- *required_functions*: List of function prototypes defining required functions.
- *disallowed_includes*: List of C-library names that are not allowed.
- *global_variables*: allow/disallow global variable declarations.
- *compile_args*: compiler arguments for clang AST-generation
- *functions*: Key that indicates the start of the list function rule segments.

Additional rules for function level are:

- *function*: function name

A key mechanic of code structure tests is that all keyword and function usages and whether recursive functions are used are propagated to the calling function.

5.4.1 General Rules

These rules can be used on both the global and function level. If a rule is defined on the global level, functions must abide by it as well, but functions can supersede rules defined on the global level either partially or fully. The following sections describe all general rules.

input

This rule indicates whether input from `stdin` is *allowed*, *disallowed* or *expected*. This is achieved by allowing/disallowing or expecting the usage of at least one of the following functions:

- `fgetc`
- `fgets`
- `fgetwc`
- `fread`
- `fscanf`
- `fscanf_s`
- `fwscanf`
- `fwscanf_s`
- `getc`
- `getc_unlocked`
- `getchar`
- `getchar_unlocked`
- `getdelim`
- `getline`
- `gets`
- `gets_s`
- `getw`
- `getwc`
- `getwchar`
- `getwline`
- `pread`
- `read`
- `readv`
- `scanf`
- `scanf_s`
- `sscanf`
- `sscanf_s`
- `swscanf`
- `swscanf_s`
- `vfscanf`
- `vfscanf_s`
- `vfwscanf`
- `vfwscanf_s`

- `vscanf`
- `vscanf_s`
- `vsscanf`
- `vsscanf_s`
- `vswscanf`
- `vswscanf_s`
- `vwscanf`
- `vwscanf_s`
- `wscanf`
- `wscanf_s`

If defined globally, this rule can be superseded on function level either partially, by defining *allowed_function_calls*, or *expected_function_calls*, or as a whole by redefining `input`.

Note: Please note that *expected* is not yet implemented! Currently, *expected* behaves just like *allowed*.

Possible values:

- `true`: usage of at least one of the functions is expected (this is not implemented yet! currently `true` behaves just like `null`)
- `null`: usage of the functions is allowed but not mandatory.
- `false`: usage of the functions is disallowed

Example:

```
input: false
```

Default value:

If it is not present on the function level, the global settings remain unchanged. If the keyword `input` is omitted on the global level, input is allowed.

`output`

This rule indicates whether output on `stdout` is *allowed*, *disallowed* or *expected*. This is achieved by allowing/disallowing or expecting the usage of at least one of the following functions:

- `ferror`
- `fprintf`
- `fprintf_s`
- `fpurge`
- `fputc`
- `fputs`
- `fputwc`
- `fputws`

- `fputws_l`
- `fread`
- `freopen`
- `fropen`
- `fwprintf`
- `fwrite`
- `perror`
- `printf`
- `printf_s`
- `putc`
- `putchar`
- `puts`
- `putw`
- `putwc`
- `putwchar`
- `pwrite`
- `vasprintf`
- `vfprintf`
- `vfprintf_s`
- `vfwprintf`
- `vfwprintf_s`
- `vprintf`
- `vprintf_s`
- `vwprintf`
- `vwprintf_s`
- `wprintf`
- `write`
- `writev`

If defined globally, this rule can be superseded on function level either partially, by defining *allowed_function_calls* or *expected_function_calls*, or as a whole by redefining `output`.

Note: Please note that *expected* is not yet implemented! Currently, *expected* behaves just like *allowed*.

Possible values:

- `true`: usage of at least one of the functions is expected (this is not implemented yet! currently `true` behaves just like `null`)
- `null`: usage of the functions is allowed but not mandatory.

- `false`: usage of the functions is disallowed

Example:

```
output: null
```

Default value:

If it is not present on the function level, the global settings remain unchanged. If the keyword `output` is omitted on the global level output is allowed.

insecure

This rule indicates whether the usage of insecure functions is *allowed*, *disallowed* or *expected*. Insecure functions are functions that start or kill processes or threads as well as the `exec`-function family. This is achieved by allowing/disallowing or expecting the usage of at least one of the following functions:

- `fork`
- `system`
- `kill`
- `killpg`
- `execl`
- `execle`
- `execlp`
- `execv`
- `execvp`
- `execvP`
- `execve`
- `execvpe`
- `fexecve`
- `vfork`
- `clone`
- `tkill`
- `tgkill`
- `execveat`
- `pthread_create`
- `thrd_create`

If defined globally, this rule can be superseded on function level either partially, by defining *allowed_function_calls* or *expected_function_calls*, or as a whole by redefining `insecure`.

Note: Please note that *expected* is not yet implemented! Currently, *expected* behaves just like *allowed*.

Possible values:

- `true`: usage of at least one of the functions is expected (this is not implemented yet! currently `true` behaves just like `null`)
- `null`: usage of the functions is allowed but not mandatory.
- `false`: usage of the functions is disallowed

Example:

```
insecure: false
```

Default value:

If it is not present on the function level, the global settings remain unchanged. If the keyword `insecure` is omitted on the global level, insecure functions are is allowed.

recursion

This rule indicates whether recursion is *allowed*, *disallowed*, or *expected*. If defined globally, this rule can be superseded on function level by redefining `recursion`.

Possible values:

- `true`: recursion is expected. If used globally, this means that every function has to be recursive. If used on the function level, the current function has to be recursive.
- `null`: recursion is is allowed but not mandatory.
- `false`: recursion is disallowed.

Example:

```
recursion: true
```

Default value:

If it is not present on the function level, the global settings remain unchanged. If the keyword `recursion` is omitted on the global level, recursion is allowed.

disallowed_keywords

The usage of all keywords listed here is disallowed. If defined globally, this rule can be superseded on the function level by adding globally disallowed keywords to *expected_keywords*.

Possible values:

A list of c-keywords that is not allowed

Example:

```
disallowed_keywords:
  - for
  - while
  - do
```

Default value:

If it is not present on the function level, the global settings remain unchanged. If `disallowed_keywords` is omitted on the global level, all keywords are allowed.

expected_keywords

The usage of all keywords listed here is expected. If defined globally, this rule can be superseded on the function level by adding globally expected keywords to *disallowed_keywords*.

Possible values:

A list of c-keywords that is expected

Example:

```
expected_keywords:
  - if
  - unsigned
```

Default value:

If it is not present on the function level, the global settings remain unchanged. If `expected_keywords` is omitted on the global level, all keywords are allowed.

allowed_function_calls

The usage of all function calls of external functions listed here is allowed. If defined globally, this rule can be superseded on the function level by adding globally allowed function calls to *disallowed_function_calls* or *expected_function_calls*.

Possible values:

A list of c-library function calls that is allowed.

Example:

```
allowed_function_calls:
  - printf
  - fprintf
```

Default value:

If it is not present on the function level, the global settings remain unchanged. If `allowed_function_calls` is omitted on the global level, all function calls are allowed.

disallowed_function_calls

The usage of all function calls of external functions listed here is disallowed. If defined globally, this rule can be superseded on the function level by adding globally disallowed function calls to *allowed_function_calls* or *expected_function_calls*.

Possible values:

A list of C-library function calls that are not allowed.

```
disallowed_function_calls:
  - pow
```

Default value:

If it is not present on the function level, the global settings remain unchanged. If `disallowed_function_calls` is omitted on the global level, all function calls are allowed.

expected_function_calls

The usage of all function calls of external functions listed here is expected. If defined globally, this rule can be superseded on the function level by adding globally expected function calls to *allowed_function_calls* or *disallowed_function_calls*.

Possible values:

A list of C-library function calls that are expected.

```
expected_function_calls:
  - qsort
```

Default value:

If it is not present on the function level, the global settings remain unchanged. If `expected_function_calls` is omitted on the global level, all function calls are allowed.

expected_variable_declarations

The variables defined in this list are expected to be defined with the same type in the analyzed code. If defined globally, this means that each function body must contain this variable definition. If defined on function level, only the function body in question must contain the variable definitions. This rule can not be superseded on the function level.

Note: We plan on adding function parameters to the analyzed code for variable declarations. This is not yet implemented.

Possible values:

The list of variable declarations that are expected.

Example:

```
expected_variable_declarations:
  - 'long int n;'
  - 'long long value;'
```

Default value:

If `expected_variable_declarations` is omitted there are no expected variable declarations.

5.4.2 Global Rules

These rules can only be used on the global level, and it is not possible to influence them on the function level. The following sections describe all global rules.

required_functions

The function prototypes defined in this list must be present in the analyzed code.

Possible values:

The list of c function prototypes that must be present

Example 1:

```
required_functions:
  - "void foo(int);"
  - "int bar(void);"
```

Example 2:

```
required_functions:
  - "#include \"types.h\" \n void print_person(person_t *person);"
  - "#include <stdlib.h> \nvoid my_sort(char array[], size_t length);"
```

Note: As shown in Example 2, non standard types and types defined with typedef can be loaded via includes. **Attention:** The `types.h` file must reside in the same location as the test configuration fie and the `Tests.py` file.

Alternatively the same required functions can be written as:

```
required_functions:
  - |
      #include "types.h"
      void print_person(person_t *person);
  - |
      #include <stdlib.h>
      void my_sort(char array[], size_t length);"
```

Default value:

If `required_functions` is omitted, there are no functions whose implementation is required.

disallowed_includes

The usage of C-library includes listed here is not allowed in the analyzed code.

Possible values:

The list of c-library includes that are disallowed

Example:

```
disallowed_includes:
  - 'stdio.h'
```

Default value:

If `disallowed_includes` is omitted, all c-library includes are allowed.

global_variables

Indicates whether global variables can be used in the analyzed translation unit or not. If `global_variables` is true, the usage of global variables is allowed. If `global_variables` is false, the usage of global variables is not allowed.

Example:

```
global_variables: false
```

Default value:

If `global_variables` is omitted, the usage of global variables is not allowed.

compile_args

This keyword can be used to define an array containing additional command line arguments that should be passed to clang when it is parsing the syntax tree. For example they can be used to specify include paths or warnings.

Example:

```
compile_args:
  - "-I/path/to/include"
  - "-Wall"
```

functions

This key indicates the start of the function rules list. It is required if rules for at least one function are defined. Otherwise, it is disallowed.

5.4.3 Function Rules

These rules can only be used on function level. The following sections describe all function rules.

Note: If function `f1` calls function `f2` then function `f2` must also abide to all rules for function `f1` because all checks are propagated backward along the call graph.

---

function

This key indicates the name of the function to which the following rules are applied. This key can only be used in the *functions* list.

Possible values:

The name of the function

Example:

```
functions:
  - function: print_something
    output: true
  - function: main
    output: true
```

Default value:

The `function` key can not be omitted if rules for a given function are to be added.

5.4.4 Example code structure test definitions:

In this section, a few examples for code structure test definitions are given.

Note: For security reasons, `insecure: false` is included in each of the following test cases. Omit `insecure: false` at your own risk.

Example 1 (incorrect): Output via printf is only allowed in function `print_hello_world` but `print_hello_world` is called in the function `main`

```
version: "0.0"
translation_unit: hello_world.c
tests:
  - type: structural
    name: Structural
    insecure: false
    output: false
    required_functions:
      - "void print_hello_world(void);"
    functions:
      - function: print_hello_world
        expected_function_calls:
          - printf
      - function: main
        expected_function_calls:
          - print_hello_world
```

Lets suppose the function `print_hello_world` calls the `printf` function. The `printf`-call triggers an infringement of the global rule `output: false` because the printf call is propagated backward along the call-graph from `print_hello_world` to `main`.

Example 1 (correct): Output is only allowed in functions `print_hello_world` and `main`

```
version: "0.0"
translation_unit: hello_world.c
tests:
  - type: structural
    name: Structural
    insecure: false
    output: false
    required_functions:
      - "void print_hello_world(void);"
    functions:
      - function: print_hello_world
        expected_function_calls:
          - printf
      - function: main
        expected_function_calls:
```

(continues on next page)

(continued from previous page)

```
16          - print_hello_world
17          - printf
```

Lets suppose the function `print_hello_world` calls the `printf` function. In this case the `printf`-call does not trigger an infringement of the global rule `output: false` because the rule has been overridden for all functions along the call-graph, in this specific case `print_hello_world` and `main`.

5.5 Grey Box

A grey box test analyzes the return type and side effects of functions. Output on `stdout` and `stderr` can currently not be monitored. The only functionality for output monitoring is implemented in *Input-/Output* tests. This is done by building a shared library file and calling the C-function directly from within the python code.

Compilation of the shared library file is done by the framework. The grey box test definition is an implementation of the abstract class *GreyBoxTestRunner*

Note: If the execution of the tested function ends unexpectedly a dedicated message is added to the test feedback. This message can be customized by overriding the function *exit_failure_message*.

The default message for the error signal 8 or 11 is:

- Error occurred during execution! Exit code: 8 This exit code might indicate that a fatal arithmetic error occurred (e.g., division by zero).
- Error occurred during execution! Exit code: 11 This exit code might indicate that your program tries to read and write outside the memory that is allocated for it.

For other exit codes, by default, only "Error occurred during execution! Exit code: " followed by the respective exit code is printed. Furthermore, the string produced by *function_call_details* is appended in any case.

Note: If during the execution of the tested function a timeout occurs a dedicated message to indicate the timeout is produced.

The message starts with the following general description:

Timeout: The execution of your program was canceled since it did not finish after `x` seconds! This might indicate that there is some unexpected behavior (e.g., an endless loop) or that your program is very slow! Last executed test:

`x` corresponds to the value configured in the *Test definition file*. Furthermore, the string produced by *function_call_details* is appended.

Note: For convenience sake - to save the trouble of checking the exported symbols of the shared library - we recommend adding all functions that will be tested within the grey box test as *required_functions* for the structural test and make the structural test a requirement of the grey box test (see *Test definition file* for requirements).

5.5.1 Example - Add

In the following, a task description of a simple assignment is given. Additionally, a potential solution and the workflow to create a grey box test for this task are provided.

Note: For this example, the following file structure is assumed.

```
assignment/
    add.c
    Makefile
test/
    add.yml
    add_grey_box.py
    Tests.py
```

Task

Implement a function with the signature `int add(int a, int b)` that adds the two integers `a` and `b` and returns the result. Test the function `add` in `main`!

Solution

`add.c`:

```c
#include <stdio.h>
#include <stdlib.h>

int add(int a, int b) {
    return a + b;
}

int main(void) {
    int a = 5;
    int b = 6;
    printf("%d + %d = %d\n", a, b, add(5, 6));
    return EXIT_SUCCESS;
}
```

`Makefile`:

```
CC=gcc
CFLAGS=-Wall -Werror -Wextra -Wpedantic -std=c11

.PHONY: add.so clean

add: add.c

add.so: CFLAGS+=-fPIC -shared
add.so: add.c
    $(CC) $(CFLAGS) $^ -o $@
```

(continues on next page)

(continued from previous page)

```
clean:
    rm add add.so
```

Test

1. Add a configuration file `add.yml`

```
version: "0.0"
translation_unit: add.c
tests:
  - type: compile
    name: TestCompile
  - type: structural
    name: TestCodeStructure
    requirements:
      - TestCompile
    required_functions:
      - 'int add(int a, int b);'
  - type: grey_box
    name: TestGreyBox
    requirements:
      - TestCompile
      - TestCodeStructure
    module_path: add_grey_box.py
    class: GreyBoxTest
    unit_test: true
```

2. Add a grey_box test file `add_grey_box.py`

```
import ctypes
from typing import Optional

from fact.c_util import test_case, create_error_hint
from fact.test_cases import GreyBoxTestRunner


class GreyBoxTest(GreyBoxTestRunner):

    def __init__(self, library_path: str) -> None:
        super().__init__(library_path)

        self.add = self.library.add
        self.add.argtypes = [ctypes.c_int, ctypes.c_int]
        self.add.restype = ctypes.c_int

    def run(self) -> None:
        self.test_cases_add()

    def test_cases_add(self) -> None:
        self.test_add(0, 0, show_expected=True, show_hint=True)
```

(continues on next page)

(continued from previous page)

```
        self.test_add(1, -1, show_expected=True, show_hint=True)
        self.test_add(-1, 1, show_expected=True, show_hint=True)
        self.test_add(-1, -1, show_expected=True, show_hint=True)
        self.test_add(1, 5, show_expected=True, show_hint=True)
        self.test_add(-32767, 32767, show_expected=True, show_hint=True)
        self.test_add(23, 52)
        self.test_add(-23, -52)

    @test_case('add')
    def test_add(self, a, b, show_expected=False, show_hint=False) -> Optional[str]:
        p_res = a + b
        c_res = self.add(a, b)
        if c_res == p_res:
            return None
        hint = None
        if show_hint:
            hint = f"Did you try 'a = {a}' and 'b = {b}'?"
        return create_error_hint(c_res, p_res, show_expected, hint)
```

3. Add `Tests.py` file

```
from fact.tester import Tester

tester = Tester.from_config('add.yml')
tester.run()
tester.export_result()
```

4. Run `Tests.py` to execute the tests

5.6 OCLint

An oclint test analyzes the translation unit's source code and enforces the predefined rules. Each translation unit requires its own set of rules. If the analyzed code violates at least one rule, the test is failed, and feedback on what rules have been violated is given. Otherwise, feedback that the oclint test has been passed successfully is given. The tool OCLint is used to perform the analysis for this test case.

In addition to the configuration options presented in *Test definition file* the following attributes are available:

- *suppress_line*: allow/disallow comments to suppress the analysis of the commented line.
- *suppress_range*: allow/disallow annotations to suppress the analysis.
- *disable_rules*: use all rules except for the rules given in this array.
- *apply_rules*: only use the rules given in this array.

Note: For the oclint analysis the `NDEBUG` flag is always set during OCLint's the integrated compilation step in order to disable the analysis of `assert` statements.

5.6.1 Attribute details

Details on the additional configuration options for oclint tests are given in the following sections.

suppress_line

This rule indicates whether students are allowed to suppress OCLint warnings using the `//!OCLint` comment.

Possible values:

- `true`: the suppression of OCLint warnings using the `//!OCLint` comment is allowed.
- `false` *(default)*: the suppression of OCLint warnings using the `//!OCLint` comment is not allowed.

Example:

```
suppress_line: true
```

If suppression of warnings is allowed all warnings generated by a line of code can be suppressed as follows:

```
void a() {
    int unusedLocalVariable; //!OCLint
}
```

See !OCLint Comment in the OCLint documentation for further details and examples.

suppress_range

This rule indicates whether students are allowed to suppress OCLint rules using annotations.

Possible values:

- `true`: the suppression of OCLint rules using annotations is allowed.
- `false` *(default)*: the suppression of OCLint rules using annotations is not allowed.

Example:

```
suppress_range: true
```

If suppression of rules is allowed all rules in the annotated scope are suppressed as follows:

```
__attribute__((annotate("oclint:suppress")))
```

See Annotations in the OCLint documentation for further details and examples.

disable_rules

Array indicating the OCLint rules that should not be applied to the analyzed translation unit.

Possible values:

All rules defined in the Rule Index of the OCLint documentation. The rules have to be written as given in the referenced rule index.

Example:

```
disable_rules:
  - ShortVariableName
  - UselessParentheses
```

apply_rules

Array indicating the OCLint rules that be applied to the analyzed translation unit.

Possible values:

All rules defined in the Rule Index of the OCLint documentation. The rules have to be written as given in the referenced rule index.

Example:

```
apply_rules:
  - GotoStatement
  - EmptyElseBlock
  - LongVariableName
  - LongLine
```

5.6.2 Example code structure test definitions:

In this section, a few examples for oclint test definitions are given.

Example 1: select all applied rules by hand.

```
version: "0.0"
translation_unit: hello_world.c
tests:
  - type: oclint
    name: Linter
    apply_rules:
      - GotoStatement
      - EmptyElseBlock
      - LongVariableName
      - LongLine
```

Example 2: select all applied rules by hand and allow students to disable rules.

```
version: "0.0"
translation_unit: hello_world.c
tests:
  - type: oclint
    name: Linter
    suppress_line: true
    suppress_range: true
    apply_rules:
      - GotoStatement
      - EmptyElseBlock
      - LongVariableName
      - LongLine
```

Example 3: disable selected rules and allow students to disable rules with line comments only.

```
version: "0.0"
translation_unit: hello_world.c
tests:
  - type: oclint
    name: Linter
    suppress_line: true
    disable_rules:
      - BitwiseOperatorInConditional
      - MissingBreakInSwitchStatement
      - UselessParentheses
      - HighCyclomaticComplexity
      - HighNPathComplexity
      - HighNcssMethod
      - ShortVariableName
      - EmptyIfStatement
      - MissingDefaultStatement
      - UselessParentheses
```

5.7 fact package

5.7.1 Submodules

5.7.2 fact.c_util module

Utility functions and decorators for grey box testing using python.

class fact.c_util.**CaptureStream**(*stream: TextIO*, *stream_name: str*)

Bases: object

Used to capture output on a stream. Please ensure that all data is written on the captured stream before capturing is stopped (e.g., flush the stream).

Note that this class is based on https://stackoverflow.com/a/29834357.

captured_text: str

get_data() → str

Returns the captured data

Returns The captured data or empty string, if no data was captured.

new_fileno: int

original_fileno: int

original_stream: TextIO

pipe_in: int

pipe_out: int

start()

Start capturing all data writen on the stream.

stop()

Stop capturing data.

stream_name: str

exception fact.c_util.**GreyBoxTimeoutError**(*function_call_details: str*, **args: object*)
Bases: TimeoutError

Raised when a timeout expired during a grey box test.

function_call_details: str

exception fact.c_util.**NonAsciiCharacter**(*output: str*, *stream: str*, **args: object*)
Bases: Exception

Raised when the captured output contains non ascii characters. All non printable ascii characters in output are replaced with the special character .

error_message_students(*function_name: str*) → str
Returns the default error message if a captured output contains non ascii characters.

Parameters **function_name** – Name of the output generating function

Returns The error message

output: str

stream: str

exception fact.c_util.**NonPrintableAsciiCharacter**(*output: str*, *stream: str*, **args: object*)
Bases: Exception

Raised when the captured output contains non printable ascii characters. All non printable ascii characters in output are replaced with the special character .

error_message_students(*function_name: str*) → str
Returns the default error message if a captured output contains non printable ascii characters.

Parameters **function_name** – Name of the output generating function

Returns The error message

output: str

stream: str

fact.c_util.**c_array_to_string**(*array: List[Any]*)
Returns a string representation of the values in the array. A brace-enclosed list is used to represent the values.

Parameters **array** – The array

Returns The string representation of the array

fact.c_util.**c_char_array_to_string**(*array: List[Any]*, *null_terminated: bool = False*) → str
Returns a string representation of the characters in the array. A brace-enclosed list is used to represent the characters. The characters are shown as single-byte integer character constants (e.g., `'a'`). Non printable ascii characters are replaced with `''`.

Parameters

- **array** – The array
- **null_terminated** – Should a null termination be added?

Returns The string representation of the array

fact.c_util.**c_pointer**(*pointer*) → str
Returns a hexadecimal string representation of a C-pointer.

Parameters **pointer** – The pointer

Returns The string representation

fact.c_util.**c_pointer_array_to_string**(*array: List[Any]*)
Returns a string representation of the pointers in the array. A brace-enclosed list is used to represent the values.

Parameters **array** – The array

Returns The string representation of the array

fact.c_util.**char_arr_c**(*length: int*)
Creates a ctypes char-array with the given length.

Parameters **length** – The length of the array

Returns The char-array

fact.c_util.**char_arr_c2p**(*array*) → List[Any]
Creates a list based on the values from a ctypes char-array.

Parameters **array** – The ctypes array comprising the elements which should be copied to the new list.

Returns The list

fact.c_util.**char_arr_p2c**(*array: str*)
Creates a ctypes char-array with the appropriate length. The provided string is used to fill the created array. Note that the created array is null terminated!

Parameters **array** – The string comprising the elements of the new array.

Returns The char-array

fact.c_util.**create_error_hint**(*actual=None*, *expected=None*, *show_expected=False*, *hint=None*) → str
Creates an error hint.

Parameters

- **actual** – The actual result
- **expected** – The expected result
- **show_expected** – Should the actual and expected results be shown?
- **hint** – An additional hint

Returns The error hint

fact.c_util.**int_arr_c**(*length*)
Creates a ctypes int-array with the given length.

Parameters **length** – The length of the array

Returns The int-array

fact.c_util.**int_arr_c2p**(*array*) → List[Any]
Creates a list based on the values from a ctypes array.

Parameters **array** – The ctypes array comprising the elements which should be copied to the new list.

Returns The list

fact.c_util.**int_arr_p2c**(*array: List[int]*)
Creates a ctypes int-array with the appropriate length. The provided array is used to fill the created array.

Parameters **array** – The array comprising the elements which should be copied to the new array.

Returns The int-array

fact.c_util.**int_pp_from_2d**(*array: numpy.ndarray*, *cols: int*)
Creates a ctypes multidimensional int-array based on the values from a numpy-array.

Parameters

- **array** – The data of the multidimensional array
- **cols** – Number of columns

Returns The pointer to the multidimensional array

fact.c_util.**test_case**(*name*)
Decorator for grey box testing. Every test case should use this decorator and specify the tested function or procedure name.

Parameters **name** – The name of the tested C function or procedure name

5.7.3 fact.io module

Parsing of the DSL for input-output tests

class fact.io.**IOParser**(*input_text: str*)
Bases: object

Parses an input-output test definition using the DSL

tests: List[*fact.io.IOTestConfig*]

class fact.io.**IOReplacement**(*parent*, *pattern: str*, *replace: str*, *hint: str*, *num_matches: int*)
Bases: object

Configuration of a variable substitution where the value of a variable is replaced.

hint: str

num_matches: int

pattern: str

replace: str

exception fact.io.**IOScriptExecutionError**
Bases: Exception

Raised when a script in a IO-Test does not terminate successfully.

class fact.io.**IOSubstitution**(*parent*, *variable: str*, *value: str*, *hint: str*, *num_matches: int*)
Bases: object

Configuration of a variable substitution where the value of a variable is replaced.

hint: str

num_matches: int

value: str

variable: str

class fact.io.**IOTestConfig**(*stdin: List[str]*, *args: List[str]*, *return_code: List[int]*, *replacements: List[*fact.io.IOReplacement*]*, *substitutions: List[*fact.io.IOSubstitution*]*, *settings:* fact.io.IOTestSettings)
Bases: abc.ABC

An abstract io test configuration that each io test configuration has to inherit from

arguments: List[str]

check_return_value(*obtained_return_value: int*) → bool
Returns whether the obtained return value matches any of the expected return codes

Parameters **obtained_return_value** – The obtained return_value

Returns True, return value matches any of the expected return codes

abstract check_stderr(*obtained: str*) → bool
Returns whether the obtained output matches the expected output on stderr

Parameters **obtained** – The obtained output on stderr

Returns True, if the obtained output matches the expected output

abstract check_stdout(*obtained: str*) → bool
Returns whether the obtained output matches the expected output on stdout

Parameters **obtained** – The obtained output on stdout

Returns True, if the obtained output matches the expected output

modifies_code() → bool
Returns whether the test configuration changes the source because of replacement or substitution.

Returns True, if the code is modified.

replacements: List[*fact.io.IOReplacement*]

return_code: List[int]

settings: *fact.io.IOTestSettings*

stdin: str

substitutions: List[*fact.io.IOSubstitution*]

abstract test_results(*output: Optional[subprocess.CompletedProcess]*, *substitution_description: List[str]*)
Returns the test result of an io-test.

Parameters

- **output** – The completed process
- **substitution_description** – Description of substitutions.

Returns IOTestResults

class fact.io.**IOTestConfigExact**(*stdin: List[str]*, *stdout: List[Tuple[str, str]]*, *stderr: List[Tuple[str, str]]*, *args: List[str]*, *return_code: List[int]*, *replacements: List[*fact.io.IOReplacement*]*, *substitutions: List[*fact.io.IOSubstitution*]*, *settings:* fact.io.IOTestSettingsExact)
Bases: *fact.io.IOTestConfig*

Configuration of an input-output test with regex matching

check_stderr(*obtained: str*) → bool
Returns whether the obtained output matches the expected output on stderr

Parameters **obtained** – The obtained output on stderr

Returns True, if the obtained output matches the expected output

check_stdout(*obtained: str*) → bool
Returns whether the obtained output matches the expected output on stdout

Parameters **obtained** – The obtained output on stdout

Returns True, if the obtained output matches the expected output

settings: *fact.io.IOTestSettingsExact*

stderr: List[Tuple[str, str]]

stderr_mod: str

stdout: List[Tuple[str, str]]

stdout_mod: str

test_results(*output: Optional[subprocess.CompletedProcess]*, *substitution_description: List[str]*)
Returns the test result of an io-test.

Parameters

- **output** – The completed process
- **substitution_description** – Description of substitutions.

Returns IOTestResults

class fact.io.**IOTestConfigRegex**(*stdin: List[str]*, *stdout: List[str]*, *stderr: List[str]*, *args: List[str]*, *return_code: List[int]*, *replacements: List[*fact.io.IOReplacement*]*, *substitutions: List[*fact.io.IOSubstitution*]*, *settings:* fact.io.IOTestSettingsRegex)

Bases: *fact.io.IOTestConfig*

Configuration of an input-output test with regex matching

check_stderr(*obtained: str*) → bool
Returns whether the obtained output matches the expected output on stderr

Parameters **obtained** – The obtained output on stderr

Returns True, if the obtained output matches the expected output

check_stdout(*obtained: str*) → bool
Returns whether the obtained output matches the expected output on stdout

Parameters **obtained** – The obtained output on stdout

Returns True, if the obtained output matches the expected output

settings: *fact.io.IOTestSettingsRegex*

stderr: str

stdout: str

test_results(*output: Optional[subprocess.CompletedProcess]*, *substitution_description: List[str]*)
Returns the test result of an io-test.

Parameters

- **output** – The completed process
- **substitution_description** – Description of substitutions.

Returns IOTestResults

class fact.io.**IOTestResults**(*test:* fact.io.IOTestConfig, *output: Optional[subprocess.CompletedProcess]*, *substitution_description: List[str]*)

Bases: abc.ABC

An abstract io test result that each io test result has to inherit from

ascii_msg(*output: str*, *stream: str*)
Returns the error message if the output contains non printable ascii characters. All non printable ascii characters in output are replaced with a special character ().

Parameters

- **output** – Erroneous output
- **stream** – Stream on which the erroneous output occurred

Returns The error message

abstract error_msg(*source_file: pathlib.Path*) → str
Returns the error message for the io test

Parameters **source_file** – The path to source_file

Returns Error message

is_successful() → bool
Checks whether exit code, stdout and stderr are as expected.

Returns True, if the test was successful

no_hint_msg = 'No hint available! Please, read the exercise description very carefully!'

substitution_description: List[str]

test: *fact.io.IOTestConfig*

test_return_value: bool

test_stderr: bool

test_stdout: bool

timeout_msg(*exec_timeout: int*)
Returns a timeout error message

Returns The error message

unicode_decode_msg(*output: str*)
Returns the error message for UnicodeDecodeError when reading the stdout and stderr output

Parameters **output** – Erroneous output

Returns The error message

class fact.io.**IOTestResultsExact**(*test:* fact.io.IOTestConfigExact, *output: Optional[subprocess.CompletedProcess]*, *substitution_description: List[str]*)
Bases: *fact.io.IOTestResults*

Test result of an io-test using exact matching.

error_msg(*source_file: pathlib.Path*) → str
Returns the error message for the io test

Parameters **source_file** – The path to source_file

Returns Error message

test: *fact.io.IOTestConfigExact*

class fact.io.**IOTestResultsRegex**(*test:* fact.io.IOTestConfigRegex, *output: Optional[subprocess.CompletedProcess]*, *substitution_description: List[str]*)

Bases: *fact.io.IOTestResults*

Test result of an io-test using regex matching.

error_msg(*source_file: pathlib.Path*) → str

Returns the error message for the io test

Parameters **source_file** – The path to source_file

Returns Error message

test: *fact.io.IOTestConfigRegex*

class fact.io.**IOTestSettings**

Bases: abc.ABC

An abstract io test setting that every io test setting has to inherit from.

escape_sequence: Optional[str] = None

hint: Optional[str] = None

A hint which may help students if they encounter an error in this test

printable_ascii: bool = False

The escape fraction. Code between two escape parts is executed. The code and the two escape parts are replaced with the resulting output on stdout.

show_input: bool = False

Should the test input be shown in the error message?

show_output: bool = True

Should the obtained output be shown?

show_substitution: bool = True

Should the substituted code be shown in the error message?

class fact.io.**IOTestSettingsExact**

Bases: *fact.io.IOTestSettings*

Settings of an io test with exact matching

ignore_cases: bool = False

Should the cases be ignored?

line_rstrip: bool = False

Should all whitespaces & tabs at the end of each line be ignored?

rstrip: bool = False

Should all whitespace characters at the end be ignored?

show_diff: bool = True

Should a diff of the expected and obtained outputs be shown?

show_expected: bool = False

Should the expected output be shown?

class fact.io.**IOTestSettingsRegex**

Bases: *fact.io.IOTestSettings*

Settings of an io test with regex matching

show_error: bool = True
Should the obtained output on stderr be shown?

fact.io.**cname**(*cls*) → str
Returns the class name

Parameters **cls** – Class

Returns The class name

5.7.4 fact.test_cases module

This module contains the different test cases which can be used for this C testing framework. Note that each test case must inherit from AbstractTest!

class fact.test_cases.**AbstractTest**(*test_name: str*, *sourcecode_runner:* fact.test_cases.SourcecodeRunner, *requirements: Optional[List[str]] = None*)
Bases: abc.ABC

A abstract test that every test has to inherit from.

requirements: List[str]

sourcecode_runner: *fact.test_cases.SourcecodeRunner*

start(*case*)
Starts the test run.

Parameters **case** – The test case where this test should get added to.

Returns None

start_msg() → str
Returns a startup message for this test

Returns The message

test_name: str

class fact.test_cases.**AbstractTimeoutTest**(*test_name: str*, *sourcecode_runner:* fact.test_cases.SourcecodeRunner, *requirements: Optional[List[str]] = None*)
Bases: *fact.test_cases.AbstractTest*

Abstract test for test cases that are not run via the subprocess module

requirements: List[str]

sourcecode_runner: *fact.test_cases.SourcecodeRunner*

test_name: str

class fact.test_cases.**CTest**(*makefile_directory*, *shared_lib_test: str*, *shared_lib_student*, *c_array_size: int = 8192*)
Bases: object

Loads a shared library which executes tests.

The execution, verification, and error reporting is left to the user. The shared library has to implement at least four function:

- *void *fact_init(char *lib_name)*: Initializes the test run and loads the shared library compiled from the student solution
- *int fact_tests(void *ptr)*: Tests the solution

- *int fact_errors(void *ptr, char *error, size_t max_error_len)*: Reports errors
- *int fact_free(void *ptr)*: Frees resources

c_array_size: int

makefile_directory: pathlib.Path

run_tests() → Optional[str]
Executes the shared library to test the solution.

Returns An error message, if an error occurs.

shared_lib_student: str

exception fact.test_cases.**ConfigurationError**
Bases: Exception

Raised when a test configuration is erroneous.

class fact.test_cases.**GreyBoxTestRunner**(*library_path: str*)
Bases: abc.ABC

Loads a shared library and executes tests implemented in the function run.

Note that each grey box test must inherit from GreyBoxTestRunner! The execution, verification, and error reporting is left to the user.

add_error(*function_name: str*, *hint: Optional[str] = None*) → None
Adds an error. Call this method to report an error in the grey box test

Parameters

- **function_name** – The name of the erroneous function
- **hint** – A hint that should be shown as part of the error message

Returns None

errors: Dict[str, List[str]]

exit_failure_message(*function_name: str*, *exitcode: int*, *args: List[any]*)
Adds an error message when the execution of the tested function does not terminate successfully.

Parameters

- **function_name** – The name of the tested function
- **exitcode** – The exitcode of the process executing the test function
- **args** – The arguments passed to the tested function

Returns None

abstract function_call_details(*function_name: str*, *args: List[any]*) → str
Returns a description of the function call for the tested function.

Parameters

- **function_name** – The name of the tested function
- **args** – The arguments passed to the tested function

Returns The description of the function call

library_path: str

abstract run() → None
Runs the grey box test. Implement your test in the overridden method.

Returns None

test_failed() → bool
Returns if a test failed.

Returns True, if any test failed.

class fact.test_cases.**Replacement**(*lineno: int*, *old_value: str*)
Bases: object

Represents a replacement of a literal assigned to a variable.

class fact.test_cases.**SourcecodeRunner**(*make_target: str*, *makefile_directory: Union[str, pathlib.Path]*, *makefile_name: str = 'Makefile'*, *sourcecode_directory: Optional[str] = None*, *make_timeout_sec: Optional[int] = None*, *exec_timeout_sec: Optional[int] = None*)
Bases: object

Runner for building and executing C programs.

build_executable(*target: Optional[str] = None*) → subprocess.CompletedProcess
Compiles the C program

Parameters **target** – Target name of the Makefile rule to be used

Returns Output

Raises **subprocess.CalledProcessError** – If the process exits with a non-zero exit code.

exec_timeout_sec: int

classmethod from_config(*test_config: Dict[str, Any]*, *default_make_target: Optional[str] = None*)
Configure a source code runner with a given yaml-file.

Parameters

- **test_config** – File name of the yaml-file containing the configuration
- **default_make_target** – The default target for the make file. Used as a fallback if no target is specified in the configuration.

make_target: str

make_timeout_sec: int

makefile_directory: Union[str, pathlib.Path]

makefile_name: str

run_executable(*arguments: List[str]*, *stdin: str*) → subprocess.CompletedProcess
Executes the C program with the provided arguments and input on stdin

Parameters

- **arguments** – Commandline arguments
- **stdin** – Input on stdin

Returns Output

Raises **subprocess.CalledProcessError** – If the process exits with a non-zero exit code.

sourcecode_directory: str

status_msg_timeout() → str
Returns a status message describing the make and execution timeout of this test runner.

Returns The status message

unexpected_error_msg_make(*target: Optional[str] = None*) → str
Returns an error message describing that an unexpected error occurred during the Makefile execution.

Parameters **target** – Target name of the Makefile rule to be used

Returns The error message

validate_test()
Checks whether the makefile_directory is present and is a directory.

Returns True, if the path is a directory

exception fact.test_cases.**SubstitutionException**(*errors*)
Bases: Exception

Raised when a substitution fails.

errors: List[str]

class fact.test_cases.**TestCodeStructure**(*test_name: str*, *translation_unit: str*, *config: Dict[str, Any]*, *sourcecode_runner:* fact.test_cases.SourcecodeRunner, *requirements: Optional[List[str]] = None*)
Bases: *fact.test_cases.AbstractTimeoutTest*

Test runner for structural tests. Analyzes the code structure of a program and checks whether the structure satisfies the requirement.

config: Dict[str, Any]

classmethod from_config(*test_config: Dict[str, Any]*)
Configure a test runner for structural tests with a given yaml-file.

Parameters **test_config** – File name of the yaml-file containing test configuration

translation_unit: str

class fact.test_cases.**TestCompile**(*test_name: str*, *sourcecode_runner:* fact.test_cases.SourcecodeRunner, *requirements: Optional[List[str]] = None*)
Bases: *fact.test_cases.AbstractTest*

Test runner for compilation tests. Compiles the C source code and check if the compilation was successful.

classmethod from_config(*test_config: Dict[str, Any]*)
Configure a test runner for compilation tests with a given yaml-file.

Parameters **test_config** – File name of the yaml-file containing test configuration

requirements: List[str]

sourcecode_runner: *fact.test_cases.SourcecodeRunner*

test_name: str

exception fact.test_cases.**TestFailedError**
Bases: Exception

Raised when a test failed.

class fact.test_cases.**TestGreyBox**(*test_name: str*, *library_name: str*, *test_case*, *sourcecode_runner:* fact.test_cases.SourcecodeRunner, *requirements: Optional[List[str]] = None*, *unit_test: bool = True*, *max_errors: int = 0*)
Bases: *fact.test_cases.AbstractTimeoutTest*

Test runner for grey-box tests. Executes tests defined in a GreyBoxTestRunner.

errors: Dict[str, List[str]]

classmethod from_config(*test_config: Dict[str, Any]*)
Configure a test runner for python grey-box tests with a given yaml-file.

Parameters **test_config** – File name of the yaml-file containing test configuration

library_name: str

max_errors: int

unit_test: bool

class fact.test_cases.**TestGreyBoxC**(*test_name: str*, *make_target_test: str*, *library_name_student: str*, *library_name_test: str*, *sourcecode_runner:* fact.test_cases.SourcecodeRunner, *requirements: Optional[List[str]] = None*, *max_error_len: int = 8192*)
Bases: *fact.test_cases.AbstractTimeoutTest*

Test runner for grey-box tests written in C.

classmethod from_config(*test_config: Dict[str, Any]*)
Configure a test runner for C/C++ grey-box tests with a given yaml-file.

Parameters **test_config** – File name of the yaml-file containing test configuration

library_name_student: str

library_name_test: str

make_target_test: str

max_error_len: int

class fact.test_cases.**TestIO**(*test_name: str*, *c_file: str*, *filename_io_test: str*, *sourcecode_runner:* fact.test_cases.SourcecodeRunner, *requirements: Optional[List[str]] = None*)
Bases: *fact.test_cases.AbstractTest*

Test runner for input-output tests. For IO-tests the following steps are performed:

1. If substitution is present, apply the substitution to the source code
2. Recompile if necessary
3. Run program with the defined commandline arguments and the input on stdin
4. Compare the exit code and match the regexes for stdout and stderr
5. Report results if an error occurred

c_file_path: pathlib.Path

execute_io_test(*test:* fact.io.IOTestConfig, *substitution_description: List[str]*) → Optional[str]
Executes an io test

Parameters

- **test** – Test case
- **substitution_description** – Description of substitutions.

Returns An error if the test case failed

filename_io_test: str

classmethod from_config(*test_config: Dict[str, Any]*)
Configure a test runner for input-output tests with a given yaml-file.

Parameters **test_config** – File name of the yaml-file containing test configuration

original_content: str

recompile()
Triggers a recompilation after execution

Returns None

remove_comments() → Optional[str]
Removes all comments in the source code.

Returns An error message, if an error occurs.

replace()
Manages insertion of the replacement and recovery of the original code.

Returns None

replacement_content: Optional[str]

valid_replacement(*test:* fact.io.IOTestConfig) → List[str]
Check if the number of replacements matches with the definition and applies the replacement if no error was found

Parameters **test** – Test case

Returns List of error messages

valid_substitution(*test:* fact.io.IOTestConfig) → List[str]
Check if the number of substitutions matches with the definition and applies the substitution if no error was found

Parameters **test** – Test case

Returns List of error messages

class fact.test_cases.**TestOclint**(*test_name: str*, *translation_unit: str*, *sourcecode_runner:* fact.test_cases.SourcecodeRunner, *requirements: Optional[List[str]] = None*, *suppress_line: bool = False*, *suppress_range: bool = False*, *disable_rules: Optional[List[str]] = None*, *apply_rules: Optional[List[str]] = None*)
Bases: *fact.test_cases.AbstractTest*

Test runner for static code analysis using OCLint.

apply_rules: List[str]

disable_rules: List[str]

classmethod from_config(*test_config: Dict[str, Any]*)
Configure a test runner for compilation tests with a given yaml-file.

Parameters **test_config** – File name of the yaml-file containing test configuration

suppress_line: bool

suppress_range: bool

translation_unit: str

class fact.test_cases.**TestStatus**(*value*)
Bases: enum.Enum

Models the four states - skipped, error, failure, and success - of a test.

ERROR = 'error'

FAILURE = 'failure'

SKIPPED = 'skipped'

SUCCESS = 'success'

fact.test_cases.**apply_substitution**(*variable: str*, *value: str*, *text: str*) → Tuple[str, List[*fact.test_cases.Replacement*]]

Searches in the string for matches. If a match was found, the original value of the variable initialization is substituted accordingly.

Can be used to substitute all scalar basic types (e.g., bool, char, short, int, long, float, double).

Parameters

- **variable** – variable name
- **value** – new value of the variable
- **text** – string searched for matches. If a match was found, the original value of the variable initialization is substituted accordingly

Returns String after substitution and list of replacements

5.7.5 fact.tester module

Infrastructure for test case registration and reporting.

exception fact.tester.**NotExecutedError**

Bases: Exception

Raised if the results are checked before they are computed.

class fact.tester.**TestCase**(*name: str*)

Bases: object

A single test case (e.g., test compilation or IO).

message: str

name: str

result: *fact.test_cases.TestStatus*

stderr: str

stdout: str

tester_output: str

time: datetime.timedelta

to_xml(*suite: xml.etree.ElementTree.Element*, *max_chars_per_output: int = 16384*) → None

Adds the results of the test to a XML test suite

Parameters

- **suite** – XML representation of the test suite
- **max_chars_per_output** – Maximal numbers of characters for stdout and stderr of the test output

Returns

class fact.tester.**TestSuite**(*name: str*)

Bases: object

Test suite comprising multiple test cases.

add_case(*case:* fact.tester.TestCase) → None
Adds a test case to the test suite.

Parameters **case** – Test case to add

Returns None

cases: **Dict[str, *fact.tester.TestCase*]**

errors: **int**

failures: **int**

get_test_cases() → Dict[str, *fact.tester.TestCase*]
Returns the test cases :return: Dict of str -> TestCase: Maps the test case name to a TestCase

name: **str**

skipped: **int**

successful: **int**

tests: **int**

time: **datetime.timedelta**

to_xml() → xml.etree.ElementTree.Element

Returns Results of the test suite as a XML element

class fact.tester.**Tester**(*name: str = 'FACT-0.0.5'*, *logging_level=10*)
Bases: object

Test runner used to add, execute tests and export the test results.

add_test(*test: Any*) → None
Adds a new test that will be run once "run()" is invoked.

Raises **NameError** – If the test_name of the provided test has already been added.

Parameters **test** – Test to add

Returns None

executed: **bool**

export_result(*output_path: str = '../test-reports/tests-results.xml'*) → None
Exports the test results into a JUnit format and stores it at the given output_path. The JUnit format is based on[1].

Parameters **output_path** – Path used to store the export

Returns None

classmethod **from_config**(*config_file: Union[str, pathlib.Path]*, *name: str = 'FACT-0.0.5'*, *logging_level=10*)
Configure a tester with a given yaml-file.

Parameters

- **config_file** – The yml-file
- **name** – The name of the tester
- **logging_level** – Debugging level used for printing on stdout

[1] https://github.com/junit-team/junit5/blob/master/platform-tests/src/test/resources/jenkins-junit.xsd

Returns The Tester

classmethod from_dict(*conf_dict: Dict[str, Any]*, *makefile_directory: Optional[str] = None*, *sourcecode_directory: Optional[str] = None*, *name: str = 'FACT-0.0.5'*, *logging_level=10*)

Configure a tester with a given dict.

Parameters

- **conf_dict** – Dict containing the test configuration
- **makefile_directory** – The directory in which the Makefile resides
- **sourcecode_directory** – The directory in which the translation unit resides
- **name** – The name of the tester
- **logging_level** – Debugging level used for printing on stdout

Returns The Tester

name: str

run() → None

Starts the tester and runs all tests added via "add_test(test: AbstractTest)".

successful() → Optional[bool]

Returns whether all tests were executed successfully.

Returns True, if all tests could be executed successfully. If the tests were not yet executed None is returned.

suite: *fact.tester.TestSuite*

tests: Dict[str, Any]

fact.tester.**validate_test_config**(*config_file*) → bool

Returns whether the test configuration is valid and all prerequisites are met (e.g., makefiles are present).

Returns True, if the test configuration is valid.

5.7.6 Module contents

Test framework for C-programs. The framework implements compilation tests, input-output tests, structural tests and grey box tests.

It is assumed that libclang from clang 11.0 is installed (https://apt.llvm.org/).

fact.**setup_libclang**()

Sets the appropriate libclang source path for the current environment.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

f

INDEX

G

H

I

L

M

N

U

V