NAME

atf-c++, **ATF_ADD_TEST_CASE**, **ATF_CHECK_ERRNO**, **ATF_FAIL**, **ATF_INIT_TEST_CASES**, **ATF_PASS**, **ATF_REQUIRE**, **ATF_REQUIRE_EQ**, **ATF_REQUIRE_ERRNO**, **ATF_REQUIRE_IN**, **ATF_REQUIRE_MATCH**, **ATF_REQUIRE_NOT_IN**, **ATF_REQUIRE_THROW**, **ATF_REQUIRE_THROW_RE**, **ATF_SKIP**, **ATF_TEST_CASE**, **ATF_TEST_CASE_BODY**, **ATF_TEST_CASE_CLEANUP**, **ATF_TEST_CASE_HEAD**, **ATF_TEST_CASE_NAME**, **ATF_TEST_CASE_USE**, **ATF_TEST_CASE_WITH_CLEANUP**, **ATF_TEST_CASE_WITHOUT_HEAD**, **atf::utils::cat_file**, **atf::utils::compare_file**, **atf::utils::copy_file**, **atf::utils::create_file**, **atf::utils::file_exists**, **atf::utils::fork**, **atf::utils::grep_collection**, **atf::utils::grep_file**, **atf::utils::grep_string**, **atf::utils::redirect**, **atf::utils::wait** - C++ API to write ATF-based test programs

SYNOPSIS

#include <atf-c++.hpp>

ATF_ADD_TEST_CASE(*tcs*, *name*);

ATF_CHECK_ERRNO(*expected_errno*, *bool_expression*);

ATF_FAIL(*reason*);

ATF_INIT_TEST_CASES(*tcs*);

ATF_PASS();

ATF_REQUIRE(*expression*);

ATF_REQUIRE_EQ(*expected_expression*, *actual_expression*);

ATF_REQUIRE_ERRNO(*expected_errno*, *bool_expression*);

ATF_REQUIRE_IN(*element*, *collection*);

ATF_REQUIRE_MATCH(*regexp*, *string_expression*);

ATF_REQUIRE_NOT_IN(*element*, *collection*);

ATF_REQUIRE_THROW(*expected_exception*, *statement*);

ATF_REQUIRE_THROW_RE(*expected_exception*, *regexp*, *statement*);

ATF_SKIP(*reason*);

ATF_TEST_CASE(*name*);

ATF_TEST_CASE_BODY(*name*);

ATF_TEST_CASE_CLEANUP(*name*);

ATF_TEST_CASE_HEAD(*name*);

ATF_TEST_CASE_NAME(*name*);

ATF_TEST_CASE_USE(*name*);

ATF_TEST_CASE_WITH_CLEANUP(*name*);

ATF_TEST_CASE_WITHOUT_HEAD(*name*);

void
atf::utils::cat_file(*const std::string& path, const std::string& prefix*);

bool
atf::utils::compare_file(*const std::string& path, const std::string& contents*);

void
atf::utils::copy_file(*const std::string& source, const std::string& destination*);

void
atf::utils::create_file(*const std::string& path, const std::string& contents*);

void
atf::utils::file_exists(*const std::string& path*);

pid_t
atf::utils::fork(*void*);

bool
atf::utils::grep_collection(*const std::string& regexp, const Collection& collection*);

bool

atf::utils::grep_file(*const std::string& regexp, const std::string& path*);

bool
atf::utils::grep_string(*const std::string& regexp, const std::string& path*);

void
atf::utils::redirect(*const int fd, const std::string& path*);

void
atf::utils::wait(*const pid_t pid, const int expected_exit_status, const std::string& expected_stdout, const std::string& expected_stderr*);

DESCRIPTION

ATF provides a C++ programming interface to implement test programs. C++-based test programs follow this template:

```
extern "C" {
... C-specific includes go here ...
}

... C++-specific includes go here ...

#include <atf-c++.hpp>

ATF_TEST_CASE(tc1);
ATF_TEST_CASE_HEAD(tc1)
{
   ... first test case's header ...
}
ATF_TEST_CASE_BODY(tc1)
{
   ... first test case's body ...
}

ATF_TEST_CASE_WITH_CLEANUP(tc2);
ATF_TEST_CASE_HEAD(tc2)
{
   ... second test case's header ...
}
ATF_TEST_CASE_BODY(tc2)
```

```
{
   ... second test case's body ...
}
ATF_TEST_CASE_CLEANUP(tc2)
{
   ... second test case's cleanup ...
}

ATF_TEST_CASE(tc3);
ATF_TEST_CASE_BODY(tc3)
{
   ... third test case's body ...
}

... additional test cases ...

ATF_INIT_TEST_CASES(tcs)
{
   ATF_ADD_TEST_CASE(tcs, tc1);
   ATF_ADD_TEST_CASE(tcs, tc2);
   ATF_ADD_TEST_CASE(tcs, tc3);
   ... add additional test cases ...
}
```

Definition of test cases

Test cases have an identifier and are composed of three different parts: the header, the body and an optional cleanup routine, all of which are described in atf-test-case(4). To define test cases, one can use the **ATF_TEST_CASE**(), **ATF_TEST_CASE_WITH_CLEANUP**() or the **ATF_TEST_CASE_WITHOUT_HEAD**() macros, which take a single parameter specifying the test case's name. **ATF_TEST_CASE**(), requires to define a head and a body for the test case, **ATF_TEST_CASE_WITH_CLEANUP**() requires to define a head, a body and a cleanup for the test case and **ATF_TEST_CASE_WITHOUT_HEAD**() requires only a body for the test case. It is important to note that these *do not* set the test case up for execution when the program is run. In order to do so, a later registration is needed through the **ATF_ADD_TEST_CASE**() macro detailed in *Program initialization*.

Later on, one must define the three parts of the body by means of three functions. Their headers are given by the **ATF_TEST_CASE_HEAD**(), **ATF_TEST_CASE_BODY**() and **ATF_TEST_CASE_CLEANUP**() macros, all of which take the test case's name. Following each of these, a block of code is expected, surrounded by the opening and closing brackets.

Additionally, the **ATF_TEST_CASE_NAME**() macro can be used to obtain the name of the class corresponding to a particular test case, as the name is internally managed by the library to prevent clashes with other user identifiers. Similarly, the **ATF_TEST_CASE_USE**() macro can be executed on a particular test case to mark it as "used" and thus prevent compiler warnings regarding unused symbols. Note that *you should never have to use these macros during regular operation.*

Program initialization

The library provides a way to easily define the test program's **main**() function. You should never define one on your own, but rely on the library to do it for you. This is done by using the **ATF_INIT_TEST_CASES**() macro, which is passed the name of the list that will hold the test cases. This name can be whatever you want as long as it is a valid variable value.

After the macro, you are supposed to provide the body of a function, which should only use the **ATF_ADD_TEST_CASE**() macro to register the test cases the test program will execute. The first parameter of this macro matches the name you provided in the former call.

Header definitions

The test case's header can define the meta-data by using the **set_md_var**() method, which takes two parameters: the first one specifies the meta-data variable to be set and the second one specifies its value. Both of them are strings.

Configuration variables

The test case has read-only access to the current configuration variables by means of the *bool* **has_config_var**() and the *std::string* **get_config_var**() methods, which can be called in any of the three parts of a test case.

Access to the source directory

It is possible to get the path to the test case's source directory from any of its three components by querying the 'srcdir' configuration variable.

Requiring programs

Aside from the *require.progs* meta-data variable available in the header only, one can also check for additional programs in the test case's body by using the **require_prog**() function, which takes the base name or full path of a single binary. Relative paths are forbidden. If it is not found, the test case will be automatically skipped.

Test case finalization

The test case finalizes either when the body reaches its end, at which point the test is assumed to have *passed*, or at any explicit call to **ATF_PASS**(), **ATF_FAIL**() or **ATF_SKIP**(). These three macros terminate the execution of the test case immediately. The cleanup routine will be processed afterwards

in a completely automated way, regardless of the test case's termination reason.

ATF_PASS() does not take any parameters. **ATF_FAIL**() and **ATF_SKIP**() take a single string that describes why the test case failed or was skipped, respectively. It is very important to provide a clear error message in both cases so that the user can quickly know why the test did not pass.

Expectations

Everything explained in the previous section changes when the test case expectations are redefined by the programmer.

Each test case has an internal state called 'expect' that describes what the test case expectations are at any point in time. The value of this property can change during execution by any of:

expect_death(*reason*)

Expects the test case to exit prematurely regardless of the nature of the exit.

expect_exit(*exitcode*, *reason*)

Expects the test case to exit cleanly. If *exitcode* is not '-1', the runtime engine will validate that the exit code of the test case matches the one provided in this call. Otherwise, the exact value will be ignored.

expect_fail(*reason*)

Any failure (be it fatal or non-fatal) raised in this mode is recorded. However, such failures do not report the test case as failed; instead, the test case finalizes cleanly and is reported as 'expected failure'; this report includes the provided *reason* as part of it. If no error is raised while running in this mode, then the test case is reported as 'failed'.

This mode is useful to reproduce actual known bugs in tests. Whenever the developer fixes the bug later on, the test case will start reporting a failure, signaling the developer that the test case must be adjusted to the new conditions. In this situation, it is useful, for example, to set *reason* as the bug number for tracking purposes.

expect_pass()

This is the normal mode of execution. In this mode, any failure is reported as such to the user and the test case is marked as 'failed'.

expect_race(*reason*)

Any failure or timeout during the execution of the test case will be considered as if a race condition has been triggered and reported as such. If no problems arise, the test will continue execution as usual.

expect_signal(*signo*, *reason*)

Expects the test case to terminate due to the reception of a signal. If *signo* is not '-1', the runtime engine will validate that the signal that terminated the test case matches the one provided in this call. Otherwise, the exact value will be ignored.

expect_timeout(*reason*)

Expects the test case to execute for longer than its timeout.

Helper macros for common checks

The library provides several macros that are very handy in multiple situations. These basically check some condition after executing a given statement or processing a given expression and, if the condition is not met, they automatically call **ATF_FAIL**() with an appropriate error message.

ATF_REQUIRE() takes an expression and raises a failure if it evaluates to false.

ATF_REQUIRE_EQ() takes two expressions and raises a failure if the two do not evaluate to the same exact value. The common style is to put the expected value in the first parameter and the actual value in the second parameter.

ATF_REQUIRE_IN() takes an element and a collection and validates that the element is present in the collection.

ATF_REQUIRE_MATCH() takes a regular expression and a string and raises a failure if the regular expression does not match the string.

ATF_REQUIRE_NOT_IN() takes an element and a collection and validates that the element is not present in the collection.

ATF_REQUIRE_THROW() takes the name of an exception and a statement and raises a failure if the statement does not throw the specified exception. **ATF_REQUIRE_THROW_RE**() takes the name of an exception, a regular expression and a statement, and raises a failure if the statement does not throw the specified exception and if the message of the exception does not match the regular expression.

ATF_CHECK_ERRNO() and **ATF_REQUIRE_ERRNO**() take, first, the error code that the check is expecting to find in the *errno* variable and, second, a boolean expression that, if evaluates to true, means that a call failed and *errno* has to be checked against the first value.

Utility functions

The following functions are provided as part of the **atf-c++** API to simplify the creation of a variety of tests. In particular, these are useful to write tests for command-line interfaces.

void **atf::utils::cat_file**(*const std::string& path, const std::string& prefix*)

Prints the contents of *path* to the standard output, prefixing every line with the string in *prefix*.

bool **atf::utils::compare_file**(*const std::string& path, const std::string& contents*)

Returns true if the given *path* matches exactly the expected inlined *contents*.

void **atf::utils::copy_file**(*const std::string& source, const std::string& destination*)

Copies the file *source* to *destination*. The permissions of the file are preserved during the code.

void **atf::utils::create_file**(*const std::string& path, const std::string& contents*)

Creates *file* with the text given in *contents*.

void **atf::utils::file_exists**(*const std::string& path*)

Checks if *path* exists.

pid_t **atf::utils::fork**(*void*)

Forks a process and redirects the standard output and standard error of the child to files for later validation with **atf::utils::wait**(). Fails the test case if the fork fails, so this does not return an error.

bool **atf::utils::grep_collection**(*const std::string& regexp, const Collection& collection*)

Searches for the regular expression *regexp* in any of the strings contained in the *collection*. This is a template that accepts any one-dimensional container of strings.

bool **atf::utils::grep_file**(*const std::string& regexp, const std::string& path*)

Searches for the regular expression *regexp* in the file *path*. The variable arguments are used to construct the regular expression.

bool **atf::utils::grep_string**(*const std::string& regexp, const std::string& str*)

Searches for the regular expression *regexp* in the string *str*.

void **atf::utils::redirect**(*const int fd, const std::string& path*)

Redirects the given file descriptor *fd* to the file *path*. This function exits the process in case of an error and does not properly mark the test case as failed. As a result, it should only be used in subprocesses of the test case; specially those spawned by **atf::utils::fork**().

void **atf::utils::wait**(*const pid_t pid, const int expected_exit_status, const std::string& expected_stdout, const std::string& expected_stderr*)

Waits and validates the result of a subprocess spawned with **atf::utils::wait**(). The validation involves checking that the subprocess exited cleanly and returned the code specified in *expected_exit_status* and that its standard output and standard error match the strings given in *expected_stdout* and *expected_stderr*.

If any of the *expected_stdout* or *expected_stderr* strings are prefixed with ‘save:’, then they specify the name of the file into which to store the stdout or stderr of the subprocess, and no comparison is performed.

ENVIRONMENT

The following variables are recognized by **atf-c++** but should not be overridden other than for testing purposes:

ATF_BUILD_CC	Path to the C compiler.
ATF_BUILD_CFLAGS	C compiler flags.
ATF_BUILD_CPP	Path to the C/C++ preprocessor.
ATF_BUILD_CPPFLAGS	C/C++ preprocessor flags.
ATF_BUILD_CXX	Path to the C++ compiler.
ATF_BUILD_CXXFLAGS	C++ compiler flags.

EXAMPLES

The following shows a complete test program with a single test case that validates the addition operator:

```
#include <atf-c++.hpp>

ATF_TEST_CASE(addition);
ATF_TEST_CASE_HEAD(addition)
{
   set_md_var("descr", "Sample tests for the addition operator");
}
ATF_TEST_CASE_BODY(addition)
{
   ATF_REQUIRE_EQ(0, 0 + 0);
```

```
    ATF_REQUIRE_EQ(1, 0 + 1);
    ATF_REQUIRE_EQ(1, 1 + 0);

    ATF_REQUIRE_EQ(2, 1 + 1);

    ATF_REQUIRE_EQ(300, 100 + 200);
}

ATF_TEST_CASE(open_failure);
ATF_TEST_CASE_HEAD(open_failure)
{
    set_md_var("descr", "Sample tests for the open function");
}
ATF_TEST_CASE_BODY(open_failure)
{
    ATF_REQUIRE_ERRNO(ENOENT, open("non-existent", O_RDONLY) == -1);
}

ATF_TEST_CASE(known_bug);
ATF_TEST_CASE_HEAD(known_bug)
{
    set_md_var("descr", "Reproduces a known bug");
}
ATF_TEST_CASE_BODY(known_bug)
{
    expect_fail("See bug number foo/bar");
    ATF_REQUIRE_EQ(3, 1 + 1);
    expect_pass();
    ATF_REQUIRE_EQ(3, 1 + 2);
}

ATF_INIT_TEST_CASES(tcs)
{
    ATF_ADD_TEST_CASE(tcs, addition);
    ATF_ADD_TEST_CASE(tcs, open_failure);
    ATF_ADD_TEST_CASE(tcs, known_bug);
}
```

SEE ALSO

atf-test-program(1), atf-test-case(4)