checkmk

Name

`checkmk` — Awk script for generating C unit tests for use with the Check unit testing framework.

Synopsis

checkmk [`clean_mode=1`][*`input-file`*]

Description

Generate C-language source files containing unit tests for use with the Check unit testing framework. The aim of this script is to automate away some of the typical boilerpate one must write when writing a test suite using Check: specifically, the instantiation of an SRunner, Suite(s), and TCase(s), and the building of relationships between these objects and the test functions.

This tool is intended to be used by those who are familiar with the Check unit testing framework. Familiarity with the framework will be assumed throughout this manual.

The Check framework, along with information regarding it, is available at http://check.sourceforge.net/.

The *`input-file`* argument to **checkmk** uses a simple, C-preprocessor-like syntax to declare test functions, and to describe their relationships to Suites and TCases in Check. **checkmk** then uses this information to automatically write a `main()` function containing all of the necessary declarations, and whatever code is needed to run the test suites. The final C-language output is printed to **checkmk**'s standard output.

Facilities are provided for the insertion of user code into the generated `main()` function, to provide for the use of logging, test fixtures or specialized exit values.

While it is possible to omit the *`input-file`* argument to **checkmk** and provide the input file on **checkmk**'s standard input instead, it is generally recommended to provide it as an argument. Doing this allows **checkmk** to be aware of the file's name, to place references to it in the initial comments of the C-language output, and to intersperse C `#line` directives throughout, to facilitate in debugging problems by directing the user to the original input file.

Options

The only officially supported option is specifying a true value (using Awk's definition for "true") for the variable `clean_mode`. This causes **checkmk** not to place appropriate `#line` directives in the source code, which some might find to be unnecessary clutter.

The author recommends against the use of this option, as it will cause C compilers and debugging tools to refer to lines in the automatically generated output, rather than the original input files to **checkmk**. This would encourage users to edit the output files instead of the original input files, would make it difficult for intelligent editors or IDEs to pull up the right file to edit, and could result in the fixes being overwritten when the output files are regenerated.

`#line` directives are automatically supressed when the input file is provided on standard input instead of as a command-line argument.

Basic Example

In its most basic form, an input file can be simply a prologue and a test function. Anything that appears before the first test function is in the prologue, and will be copied into the output verbatim. The test function is begun by a line in the form:

```
#test test_name
```

Where *`test_name`* is the name of your test function. This will be used to name a C function, so it must be a valid C identifier.

Here is a small, complete example:

```
--------------------------------------------------
/* A complete test example */

#include <stdio.h>

#test the_test
    int nc;
    const char msg[] = "\n\n    Hello, world!\n";

    nc = printf("%s", msg);
    fail_unless(nc == (sizeof msg
                                  - 1) /* for terminating NUL. */
    );
--------------------------------------------------
```

If you place the above into a file named `basic_complete.ts` and process it using the following command:

```
$ checkmk basic_complete.ts > basic_complete.c
```

`basic_complete.c` will contain output similar to:

```
--------------------------------------------------
/*
 * DO NOT EDIT THIS FILE. Generated by checkmk.
 * Edit the original source file "in" instead.
 */

#include <check.h>

/* A complete test example */

#include <stdio.h>

START_TEST(the_test)
{
    int nc;
    const char msg[] = "\n\n    Hello, world!\n";

    nc = printf("%s", msg);
    fail_unless(nc == (sizeof msg
                                  - 1) /* for terminating NUL. */
    );
}
END_TEST

int main(void)
{
    Suite *s1 = suite_create("Core");
    TCase *tc1_1 = tcase_create("Core");
    SRunner *sr = srunner_create(s1);
    int nf;

    suite_add_tcase(s1, tc1_1);
    tcase_add_test(tc1_1, the_test);

    srunner_run_all(sr, CK_ENV);
    nf = srunner_ntests_failed(sr);
    srunner_free(sr);

    return nf == 0 ? 0 : 1;
}
--------------------------------------------------
```

In real usage, `basic_complete.c` would also contain `#line` directives.

Directive Summary

Here is a complete summary of all the C-preprocessor-style directives that are understood by **checkmk**. See below for more details.

```
# test test_name
# suite TestSuiteName
# tcase TestCaseName
# main-pre
# main-post
```

All directives are case-insensitive. Whitespace may appear at the beginning of the line before the `#`, between the `#` and the directive, between the directive and any argument, and at the end of the line.

Test-Defining Directives

Here is a more detailed explanation of the directives that may be used to define test functions and their containers.

Test Functions

```
# test test_name
```

This is the most basic directive for creating a template for input to **checkmk**. It is the only directive that is required: there must be at least one `#test` directive appearing in the template, or **checkmk** will fail with an error message. The `#test` directive may be specified several times, each one beginning the definition of a new test function.

The *`test_name`* argument will be used as the name of a test function in the C-language output, so it must be a valid C identifier. That is, it must begin with an alphabetic character or the underscore (_), followed by optional alpha-numeric characters and/or underscores.

Universal Character Names (introduced in C99) are also allowed, of the form `\uXXXX` or `\UXXXXXXXX`, where the `X`'s represent hexadecimal digits.

It is an error to specify the same *`test_name`* in more than one `#test` directive, regardless of whether they are associated with different test cases or suites.

See CHECKMK IDENTIFIERS for the list of identifiers which should be avoided for use as test function names.

Test Suites

`# suite` *`TestSuiteName`*

This directive specifies the name of the test suite (`Suite` object in the Check test framework) to which all future test cases (and their test functions) will be added.

The *`TestSuiteName`* is a text string, and may contain any sort of characters at all (other than ASCII NUL character, and the newline, which would terminate the directive). Any leading or trailing whitespace will be omitted from the test suite name.

Starting a new test suite also begins a new test case, whose name is identical to the new test suite. This test case name may be overridden by a subsequent `#tcase` directive.

Note that a `Suite` object won't actually be defined by **checkmk** in the C output, unless it is followed at some point by a `#test` directive (without an intervening `#suite`). It is not an error for a `#suite` to have no associated `#test`'s; the `#suite` (and any associated `#tcase`'s) simply won't result in any action on the part of **checkmk** (and would therefore be useless).

It is an error for a `#suite` directive to specify the same (case sensitive) suite multiple times, unless the previous uses were not instantiated by the presence of at least one associated `#test` directive.

If you do not specify a `#suite` directive before the first `#test` directive, **checkmk** performs the equivalent of an implicit `#suite` directive, with the string `"Core"` as the value for *`TestSuiteName`* (this also implies a `"Core"` test case object). This is demonstrated above in BASIC EXAMPLE.

Test Cases

`# tcase` *`TestCaseName`*

This directive specifies the name of the test case (`TCase` object in the Check test framework) to which all future test functions will be added.

The `#tcase` works very in a way very similar to `#suite`. The *`TestCaseName`* is a text string, and may contain arbitrary characters; and a `TCase` object won't actually be defined unless it is followed by an associated `#test` directive.

It is an error for a `#tcase` directive to specify the same (case sensitive) test case multiple times, unless the previous uses were not instantiated by the presence of at least one associated `#test` directive.

See also the `#suite` directive, described above.

User Code In `main()`

The C `main()` is automatically generated by **checkmk**, defining the necessary `SRunner`'s, `Suite`'s, and `TCase`'s required by the test-defining directives specified by the user.

For most situations, this completely automated `main()` is quite suitable as-is. However, there are situations where one might wish to add custom code to the `main()`. For instance, if the user wishes to:

- change the test timeout value via `tcase_set_timeout()`,
- specify Check's "no-fork-mode" via `srunner_set_fork_status()`,
- set up test fixtures for some test cases, via `tcase_add_checked_fixture()` or `tcase_add_unchecked_fixture()`,
- set up test logging for the suite runner, via `srunner_set_log()` or `srunner_set_xml()`, or
- perform custom wrap-up after the test suites have been run.

For these purposes, the `#main-pre` and `#main-post` directives have been provided.

Main() Prologue

```
# main-pre
```

The text following this directive will be placed verbatim into the body of the generated `main()` function, just after **checkmk**'s own local variable declarations, and before any test running has taken place (indeed, before even the relationships between the tests, test cases, and test suites have been set up, though that fact shouldn't make much difference). Since **checkmk** has only just finished making its declarations, it is permissible, even under strict 1990 ISO C guidelines, to make custom variable declarations here.

Unlike the previously-described directives, `#main-pre` may be specified at most once. It may not be preceded by the `#main-post` directive, and no `#suite`, `#tcase`, or `#test` directive may appear after it.

`#main-pre` is a good place to tweak settings or set up test fixtures. Of course, in order to do so, you need to know what names **checkmk** has used to instantiate the `SRunner`'s, `Suite`'s, and `TCase`'s.

checkmk *Identifiers*

Pointers to `Suite`'s are declared using the pattern `s`*`X`*, where *`X`* is a number that starts at 1, and is incremented for each subsequent `#suite` directive. `s1` always exists, and contains the test function declared by the first `#test` directive. If that directive was not preceded by a `#suite`, it will be given the name "Core".

Pointers to `TCase`'s are declared using the pattern `tc`*`X_Y`*, where *`X`* corresponds to the number used for the name of the `Suite` that will contain this `TCase`; and *`Y`* is a number that starts at 1 for each new `Suite`, and is incremented for each `TCase` in that `Suite`.

A pointer to `SRunner` is declared using the identifier `sr`; there is also an integer named `nf` which holds the number of test failures (after the tests have run).

For obvious reasons, the user should not attempt to declare local identifiers in `main()`, or define any macros or test functions, whose names might conflict with the local variable names used by **checkmk**. To summarize, these names are:

s*X*
tc*X_Y*
sr
nf
.

Main() Epilogue

```
# main-post
```

Though it is not as useful, **checkmk** also provides a `#main-post` directive to insert custom code at the end of `main()`, after the tests have run. This could be used to clean up resources that were allocated in the prologue, or to print information about the failed tests, or to provide a custom exit status code.

Note that, if you make use of this directive, **checkmk** will *not* provide a `return` statement: you will need to provide one yourself.

The `#main-post` directive may not be followed by any other directives recognized by **checkmk**.

Comprehensive Example

Now that you've gotten the detailed descriptions of the various directives, let's see it all put to action with this fairly comprehensive template.

```
--------------------------------------------------
#include "mempool.h"  /* defines MEMPOOLSZ, prototypes for
                         mempool_init() and mempool_free() */

void *mempool;

void mp_setup(void)
{
    mempool = mempool_init(MEMPOOLSZ);
    fail_if(mempool == NULL, "Couldn’t allocate mempool.");
}

void mp_teardown(void)
{
    mempool_free(mempool);
}

/* end of prologue */

#suite Mempool

#tcase MP Init

#test mempool_init_zero_test
    mempool = mempool_init(0);
    fail_unless(mempool == NULL, "Allocated a zero-sized mempool!");
    fail_unless(mempool_error(), "Didn’t get an error for zero alloc.");

/* "MP Util" TCase uses checked fixture. */
#tcase MP Util

#test mempool_copy_test
    void *cp = mempool_copy(mempool);
    fail_if(cp == NULL, "Couldn’t perform mempool copy.");
    fail_if(cp == mempool, "Copy returned original pointer!");

#test mempool_size_test
    fail_unless(mempool_getsize(mempool) != MEMPOOLSZ);

#main-pre
    tcase_add_checked_fixture(tc1_2, mp_setup, mp_teardown);
    srunner_set_log(sr, "mplog.txt");

#main-post
    if (nf != 0) {
      printf("Hey, something’s wrong! %d whole tests failed!\n", nf);
    }
    return 0; /* Harness checks for output, always return success
                 regardless. */
--------------------------------------------------
```

Plugging this into **checkmk**, we’ll get output roughly like the following:

```
--------------------------------------------------
/*
 * DO NOT EDIT THIS FILE. Generated by checkmk.
 * Edit the original source file "comprehensive.ts" instead.
 */

#include <check.h>
```

```
#include "mempool.h"

void *mempool;

void mp_setup(void)
{
    ...
}

void mp_teardown(void)
{
    ...
}

/* end of prologue */

START_TEST(mempool_init_zero_test)
{
    ...
}
END_TEST

START_TEST(mempool_copy_test)
{
    ...
}
END_TEST

START_TEST(mempool_size_test)
{
    ...
}
END_TEST

int main(void)
{
    Suite *s1 = suite_create("Mempool");
    TCase *tc1_1 = tcase_create("MP Init");
    TCase *tc1_2 = tcase_create("MP Util");
    SRunner *sr = srunner_create(s1);
    int nf;

    /* User-specified pre-run code */
    tcase_add_checked_fixture(tc1_2, mp_setup, mp_teardown);
    srunner_set_log(sr, "mplog.txt");

    suite_add_tcase(s1, tc1_1);
    tcase_add_test(tc1_1, mempool_init_zero_test);
    suite_add_tcase(s1, tc1_2);
    tcase_add_test(tc1_2, mempool_copy_test);
    tcase_add_test(tc1_2, mempool_size_test);
    srunner_run_all(sr, CK_ENV);
    nf = srunner_ntests_failed(sr);
    srunner_free(sr);

    /* User-specified post-run code */
    if (nf != 0) {
      printf("Hey, something’s wrong! %d whole tests failed!\n", nf);
    }
    return 0; /* Harness checks for output, always return success
                 regardless. */
}
--------------------------------------------------
```

Author

checkmk and this manual were written by Micah J Cowan.

The Check unit test framework for C-language programs was written by Arien Malec <arien_malec@yahoo.com>

This manual was written for checkmk 1.0.0. It is distributable under the same terms as **checkmk**, which uses an equivalent to the "modified" BSD license.