CUnit

A Unit Testing Framework for C

Overview

CUnit is a lightweight system for writing, administering, and running unit tests in C. It provides C programmers a basic testing functionality with a flexible variety of user interfaces.

CUnit is built as a static library which is linked with the user's testing code. It uses a simple framework for building test structures, and provides a rich set of assertions for testing common data types. In addition, several different interfaces are provided for running tests and reporting results. These interfaces currently include:

Automated	Output to xml file	Non-interactive
Basic	Flexible programming interface	Non-interactive
Console	Console interface (ansi C)	Interactive
Curses	Graphical interface (Unix)	Interactive

CUnit Users Guide

Table of Contents

1. Introduction to Unit Testing with CUnit

1.1. Description

CUnit is a system for writing, administering, and running unit tests in C. It is built as a static library which is linked with the user's testing code.

CUnit uses a simple framework for building test structures, and provides a rich set of assertions for testing common data types. In addition, several different interfaces are provided for running tests and reporting results. These include automated interfaces for code-controlled testing and reporting, as well as interactive interfaces allowing the user to run tests and view results dynamically.

The data types and functions useful to the typical user are declared in the following header files:

Header File	Description
#include <CUnit/CUnit.h>	ASSERT macros for use in test cases, and includes other framework headers.
#include <CUnit/CUError.h>	Error handing functions and data types. Included automatically by CUnit.h.
#include <CUnit/TestDB.h>	Data type definitions and manipulation functions for the test registry, suites, and tests. Included automatically by CUnit.h.
#include <CUnit/TestRun.h>	Data type definitions and functions for running tests and retrieving results. Included automatically by CUnit.h.
#include <CUnit/Automated.h>	Automated interface with xml output.
#include <CUnit/Basic.h>	Basic interface with non-interactive output to stdout.
#include <CUnit/Console.h>	Interactive console interface.
#include <CUnit/CUCurses.h>	Interactive console interface (*nix).
#include <CUnit/Win.h>	Windows interface (not yet implemented).

1.2. Structure

CUnit is a combination of a platform-independent framework with various user interfaces. The core framework provides basic support for managing a test registry, suites, and test cases. The user interfaces facilitate interaction with the framework to run tests and view results.

CUnit is organized like a conventional unit testing framework:

```
                       Test Registry
                             |
                  ------------------------------
                  |                            |
              Suite '1'       . . . .      Suite 'N'
                  |                            |
            ---------------              ---------------
            |             |              |             |
        Test '11' ...  Test '1M'     Test 'N1'  ...  Test 'NM'
```

Individual test cases are packaged into suites, which are registered with the active test registry. Suites can have setup and teardown functions which are automatically called before and after running the suite's tests. All suites/tests in the registry may be run using a single function call, or selected suites or tests can be run.

1.3. General Usage

A typical sequence of steps for using the CUnit framework is:
Write functions for tests (and suite init/cleanup if necessary).
Initialize the test registry - CU_initialize_registry()
Add suites to the test registry - CU_add_suite()
Add tests to the suites - CU_add_test()
Run tests using an appropriate interface, e.g. CU_console_run_tests
Cleanup the test registry - CU_cleanup_registry

1.4. Changes to the CUnit API in Version 2

All public names in CUnit are now prefixed with 'CU_'. This helps minimize clashes with names in user code. Note that earlier versions of CUnit used different names without this prefix. The older API names are deprecated but still supported. To use the older names, user code must now be compiled with USE_DEPRECATED_CUNIT_NAMES defined.

The deprecated API functions are described in the appropriate sections of the documentation.

2. Writing CUnit Test Cases

2.1. Test Functions

A CUnit "test" is a C function having the signature:

void test_func(void)

There are no restrictions on the content of a test function, except that it should not modify the CUnit framework (e.g. add suites or tests, modify the test registry, or initiate a test run). A test function may call other functions (which also may not modify the framework). Registering a test will cause it's function to be run when the test is run.

An example test function for a routine that returns the maximum of 2 integers might look like:

```
int maxi(int i1, int i2)
{
  return (i1 > i2) ? i1 : i2;
}

void test_maxi(void)
{
  CU_ASSERT(maxi(0,2) == 2);
  CU_ASSERT(maxi(0,-2) == 0);
  CU_ASSERT(maxi(2,2) == 2);
}
```

2.2. CUnit Assertions

CUnit provides a set of assertions for testing logical conditions. The success or failure of these assertions is tracked by the framework, and can be viewed when a test run is complete.

Each assertion tests a single logical condition, and fails if the condition evaluates to FALSE. Upon failure, the test function continues unless the user chooses the 'xxx_FATAL' version of an assertion. In that case, the test function is aborted and returns immediately. FATAL versions of assertions should be used with caution! There is no opportunity for the test function to clean up after itself once a FATAL assertion fails. The normal suite cleanup function is not affected, however.

There are also special "assertions" for registering a pass or fail with the framework without performing a logical test. These are useful for testing flow of control or other conditions not requiring a logical test:

```
void test_longjmp(void)
{
    jmp_buf buf;
    int i;

    i = setjmp(buf);
    if (i == 0) {
        run_other_func();
        CU_PASS("run_other_func() succeeded.");
    }
    else
        CU_FAIL("run_other_func() issued longjmp.");
}
```

Other functions called by a registered test function may use the CUnit assertions freely. These assertions will be counted for the calling function. They may also use FATAL versions of assertions - failure will abort the original test function and its entire call chain.

The assertions defined by CUnit are:

```
#include <CUnit/CUnit.h>
```

CU_ASSERT(int expression)	
CU_ASSERT_FATAL(int expression)	
CU_TEST(int expression)	
CU_TEST_FATAL(int expression)	Assert that expression is TRUE (non-zero)
CU_ASSERT_TRUE(value)	
CU_ASSERT_TRUE_FATAL(value)	Assert that value is TRUE (non-zero)
CU_ASSERT_FALSE(value)	
CU_ASSERT_FALSE_FATAL(value)	Assert that value is FALSE (zero)
CU_ASSERT_EQUAL(actual, expected)	
CU_ASSERT_EQUAL_FATAL(actual, expected)	Assert that actual = = expected
CU_ASSERT_NOT_EQUAL(actual, expected))	
CU_ASSERT_NOT_EQUAL_FATAL(actual, expected)	Assert that actual != expected
CU_ASSERT_PTR_EQUAL(actual, expected)	
CU_ASSERT_PTR_EQUAL_FATAL(actual, expected)	Assert that pointers actual= = expected
CU_ASSERT_PTR_NOT_EQUAL(actual, expected)	
CU_ASSERT_PTR_NOT_EQUAL_FATAL(actual, expected)	Assert that pointers actual != expected
CU_ASSERT_PTR_NULL(value)	
CU_ASSERT_PTR_NULL_FATAL(value)	Assert that pointer value == NULL
CU_ASSERT_PTR_NOT_NULL(value)	
CU_ASSERT_PTR_NOT_NULL_FATAL(value)	Assert that pointer value != NULL
CU_ASSERT_STRING_EQUAL(actual, expected)	
CU_ASSERT_STRING_EQUAL_FATAL(actual, expected)	Assert that strings actual and expected are equivalent
CU_ASSERT_STRING_NOT_EQUAL(actual, expected)	
CU_ASSERT_STRING_NOT_EQUAL_FATAL(actual, expected)	Assert that strings actual and expected differ
CU_ASSERT_NSTRING_EQUAL(actual, expected, count)	
CU_ASSERT_NSTRING_EQUAL_FATAL(actual, expected, count)	Assert that 1st count chars of actual and expected are the same

CU_ASSERT_NSTRING_NOT_EQUAL(actual, expected, count)
CU_ASSERT_NSTRING_NOT_EQUAL_FATAL(actual, expected, count)   Assert that 1st count chars of actual and expected differ

CU_ASSERT_DOUBLE_EQUAL(actual, expected, granularity)
CU_ASSERT_DOUBLE_EQUAL_FATAL(actual, expected, granularity)   Assert that |actual - expected| <= |granularity|, Math library must be linked in for this assertion.

CU_ASSERT_DOUBLE_NOT_EQUAL(actual, expected, granularity)
CU_ASSERT_DOUBLE_NOT_EQUAL_FATAL(actual, expected, granularity)   Assert that |actual - expected| > |granularity|, Math library must be linked in for this assertion.

CU_PASS(message)   Register a passing assertion with the specified message. No logical test is performed.

CU_FAIL(message)
CU_FAIL_FATAL(message)   Register a failed assertion with the specified message. No logical test is performed.

2.3. Depecated v1 Assertions

The following assertions are deprecated as of version 2. To use these assertions, user code must be compiled with USE_DEPRECATED_CUNIT_NAMES defined. Note that they behave the same as in version 1 (issue a 'return' statement upon failure).

#include <CUnit/CUnit.h>

Deprecated Name	Equivalent New Name
ASSERT	CU_ASSERT_FATAL
ASSERT_TRUE	CU_ASSERT_TRUE_FATAL
ASSERT_FALSE	CU_ASSERT_FALSE_FATAL
ASSERT_EQUAL	CU_ASSERT_EQUAL_FATAL
ASSERT_NOT_EQUAL	CU_ASSERT_NOT_EQUAL_FATAL
ASSERT_PTR_EQUAL	CU_ASSERT_PTR_EQUAL_FATAL
ASSERT_PTR_NOT_EQUAL	CU_ASSERT_PTR_NOT_EQUAL_FATAL
ASSERT_PTR_NULL	CU_ASSERT_PTR_NULL_FATAL
ASSERT_PTR_NOT_NULL	CU_ASSERT_PTR_NOT_NULL_FATAL
ASSERT_STRING_EQUAL	CU_ASSERT_STRING_EQUAL_FATAL
ASSERT_STRING_NOT_EQUAL	CU_ASSERT_STRING_NOT_EQUAL_FATAL
ASSERT_NSTRING_EQUAL	CU_ASSERT_NSTRING_EQUAL_FATAL
ASSERT_NSTRING_NOT_EQUAL	CU_ASSERT_NSTRING_NOT_EQUAL_FATAL
ASSERT_DOUBLE_EQUAL	CU_ASSERT_DOUBLE_EQUAL_FATAL
ASSERT_DOUBLE_NOT_EQUAL	CU_ASSERT_DOUBLE_NOT_EQUAL_FATAL

3. The Test Registry

3.1. Synopsis

#include <CUnit/TestDB.h> (included automatically by <CUnit/CUnit.h>)

```
typedef struct    CU_TestRegistry
typedef           CU_TestRegistry*    CU_pTestRegistry

CU_ErrorCode       CU_initialize_registry(void)

void               CU_cleanup_registry(void)

CU_pTestRegistry   CU_get_registry(void)

CU_pTestRegistry   CU_set_registry(CU_pTestRegistry    pTestRegistry)

CU_pTestRegistry   CU_create_new_registry(void)

void               CU_destroy_existing_registry(CU_pTestRegistry    * ppRegistry)
```

3.2. Internal Structure

The test registry is the repository for suites and associated tests. CUnit maintains an active test registry which is updated when the user adds a suite or test. The suites in this active registry are the ones run when the user chooses to run all tests.

The CUnit test registry is a data structure CU_TestRegistry declared in <CUnit/TestDB.h>. It includes fields for the total numbers of suites and tests stored in the registry, as well as a pointer to the head of the linked list of registered suites.

```
typedef struct CU_TestRegistry
{
  unsigned int uiNumberOfSuites;
  unsigned int uiNumberOfTests;
  CU_pSuite      pSuite;
} CU_TestRegistry;

typedef CU_TestRegistry* CU_pTestRegistry;
```

The user normally only needs to initialize the registry before use and clean up afterwards. However, other functions are provided to manipulate the registry when necessary.

3.3. Initialization

CU_ErrorCode CU_initialize_registry(void)

The active CUnit test registry must be initialized before use. The user should call CU_initialize_registry() before calling any other CUnit functions. Failure to do so will likely result in a crash.

An error status code is returned:

CUE_SUCCESS	initialization was successful.
CUE_NOMEMORY	memory allocation failed.

3.4. Cleanup

void CU_cleanup_registry(void)

When testing is complete, the user should call this function to clean up and release memory used by the framework. This should be the last CUnit function called (except for restoring the test registry using CU_initialize_registry() or CU_set_registry()).

Failure to call CU_cleanup_registry() will result in memory leaks. It may be called more than once without creating an error condition. Note that this function will destroy all suites (and associated tests) in the registry. Pointers to registered suites and tests should not be dereferenced after cleaning up the registry.

Calling CU_cleanup_registry() will only affect the internal CU_TestRegistry maintained by the CUnit framework. Destruction of any other test registries owned by the user are the responsibility of the user. This can be done explictly by calling CU_destroy_existing_registry(), or implicitly by making the registry active using CU_set_registry() and calling CU_cleanup_registry() again.

3.5. Other Registry Functions

Other registry functions are provided primarily for internal and testing purposes. However, general users may find use for them and should be aware of them.

These include:

CU_pTestRegistry CU_get_registry(void)

Returns a pointer to the active test registry. The registry is a variable of data type CU_TestRegistry. Direct manipulation of the internal test registry is not recommended - API functions should be used instead. The framework maintains ownership of the registry, so the returned pointer will be invalidated by a call to CU_cleanup_registry() or CU_initialize_registry().

CU_pTestRegistry CU_set_registry(CU_pTestRegistry pTestRegistry)

Replaces the active registry with the specified one. A pointer to the previous registry is returned. It is the caller's responsibility to destroy the old registry. This can be done explictly by calling CU_destroy_existing_registry() for the returned pointer. Alternatively, the registry can be made active using CU_set_registry() and destroyed implicitly when CU_cleanup_registry() is called. Care should be taken not to explicitly destroy a registry that is set as the active one. This can result in multiple frees of the same memory and a likely crash.

CU_pTestRegistry CU_create_new_registry(void)

Creates a new registry and returns a pointer to it. The new registry will not contain any suites or tests. It is the caller's responsibility to destroy the new registry by one of the mechanisms described previously.

void CU_destroy_existing_registry(CU_pTestRegistry* ppRegistry)

Destroys and frees all memory for the specified test registry, including any registered suites and tests. This function should not be called for a registry which is set as the active test registry (e.g. a CU_pTestRegistry pointer retrieved using CU_get_registry()). This will result in a multiple free of the same memory when CU_cleanup_registry() is called. Calling this function with NULL has no effect.

3.6. Deprecated v1 Data Types & Functions

The following data types and functions are deprecated as of version 2. To use these deprecated names, user code must be compiled with USE_DEPRECATED_CUNIT_NAMES defined.

#include <CUnit/TestDB.h> (included automatically by CUnit/CUnit.h>).

Deprecated Name	Equivalent New Name
_TestRegistry	CU_TestRegistry
_TestRegistry.uiNumberOfGroups	CU_TestRegistry.uiNumberOfSuites
PTestRegistry->uiNumberOfGroups	CU_pTestRegistry->uiNumberOfSuites
_TestRegistry.pGroup	CU_TestRegistry.pSuite
PTestRegistry->pGroup	CU_pTestRegistry->pSuite
PTestRegistry	CU_pTestRegistry
initialize_registry()	CU_initialize_registry()
cleanup_registry()	CU_cleanup_registry()
get_registry()	CU_get_registry()
set_registry()	CU_set_registry()

4. Managing Tests & Suites

In order for a test to be run by CUnit, it must be added to a test collection (suite) which is registered with the test registry.

4.1. Synopsis

#include <CUnit/TestDB.h> (included automatically by <CUnit/CUnit.h>)

```
typedef struct     CU_Suite
Typedef   CU_Suite*     CU_pSuite

typedef struct     CU_Test
typedef     CU_Test*    CU_pTest

typedef   void   (*CU_TestFunc)(void)
Typedef   int     (*CU_InitializeFunc)(void)
typedef   int     (*CU_CleanupFunc)(void)

CU_pSuite    CU_add_suite(const char* strName,
                          CU_InitializeFunc pInit,
                          CU_CleanupFunc pClean);

CU_pTest    CU_add_test(CU_pSuite pSuite,
                        const char* strName,
                        CU_TestFunc pTestFunc);

typedef struct    CU_TestInfo
typedef struct    CU_SuiteInfo

CU_ErrorCode    CU_register_suites(CU_SuiteInfo suite_info[]);
CU_ErrorCode    CU_register_nsuites(int suite_count, ...);
```

4.2. Adding Suites to the Registry

CU_pSuite CU_add_suite(const char* strName, CU_InitializeFunc pInit, CU_CleanupFunc pClean)

Creates a new test collection (suite) having the specified name, initialization function, and cleanup function. The new suite is registered with (and owned by) the test registry, so the registry must be initialized before adding any suites. The current implementation does not support the creation of suites independent of the test registry.

The suite's name must be unique among all suites in the registry. The initialization and cleanup functions are optional, and are passed as pointers to functions to be called before and after running the tests contained in the suite. This allows the suite to set up and tear down temporary fixtures to support running the tests. These functions take no arguments and should return zero if they are completed successfully (non-zero otherwise). If a suite does not require one or both of these functions, pass NULL to CU_add_suite().

A pointer to the new suite is returned, which is needed for adding tests to the suite. If an error occurs, NULL is returned and the framework error code is set to one of the following:

CUE_SUCCESS	suite creation was successful.
CUE_NOREGISTRY	the registry has not been initialized.
CUE_NO_SUITENAME	strName was NULL.
CUE_DUP_SUITE	the suite's name was not unique.
CUE_NOMEMORY	memory allocation failed.

4.3. Adding Tests to Suites

CU_pTest CU_add_test(CU_pSuite pSuite, const char* strName, CU_TestFunc pTestFunc)

Creates a new test having the specified name and test function, and registers it with the specified suite. The suite must already have been created using CU_add_suite(). The current implementation does not support the creation of tests independent of a registered suite.

The test's name must be unique among all tests added to a single suite. The test function cannot be NULL, and points to a function to be called when the test is run. Test functions have neither arguments nor return values.

A pointer to the new test is returned. If an error occurs during creation of the test, NULL is returned and the framework error code is set to one of the following:

CUE_SUCCESS	suite creation was successful.
CUE_NOSUITE	the specified suite was NULL or invalid.
CUE_NO_TESTNAME	strName was NULL.
CUE_NO_TEST	pTestFunc was NULL or invalid.
CUE_DUP_TEST	the test's name was not unique.
CUE_NOMEMORY	memory allocation failed.

4.4. Shortcut Methods for Managing Tests

#define CU_ADD_TEST(suite, test) (CU_add_test(suite, #test, (CU_TestFunc)test))

This macro automatically generates a unique test name based on the test function name, and adds it to the specified suite. The return value should be checked by the user to verify success.

CU_ErrorCode   CU_register_suites(CU_SuiteInfo suite_info[])
CU_ErrorCode   CU_register_nsuites(int suite_count, ...)

For large test structures with many tests and suites, managing test/suite associations and registration is tedious and error-prone. CUnit provides a special registration system to help manage suites and tests. It's primary benefit is to centralize the registration of suites and associated tests, and to minimize the amount of error checking code the user needs to write.

Test cases are first grouped into arrays of CU_TestInfo instances (defined in <CUnit/TestDB.h>):

```
CU_TestInfo test_array1[] = {
  { "testname1", test_func1 },
  { "testname2", test_func2 },
  { "testname3", test_func3 },
  CU_TEST_INFO_NULL,
};
```

Each array element contains the (unique) name and test function for a single test case. The array must end with an element holding NULL values, which the macro CU_TEST_INFO_NULL conveniently defines. The test cases included in a single CU_TestInfo array form the set of tests that will be registered with a single test suite.

Suite information is then defined in one or more arrays of CU_SuiteInfo instances (defined in <CUnit/TestDB.h>):

```
CU_SuiteInfo suites[] = {
  { "suitename1", suite1_init-func, suite1_cleanup_func, test_array1 },
  { "suitename2", suite2_init-func, suite2_cleanup_func, test_array2 },
  CU_SUITE_INFO_NULL,
};
```

Each of these array elements contain the (unique) name, suite initialization function, suite cleanup function, and CU_TestInfo array for a single suite. As usual, NULL may be used for the initialization or cleanup function if the given suite does not need it. The array must end with an all-NULL element, for which the macro CU_SUITE_INFO_NULL may be used.

All suites defined in a CU_SuiteInfo array can then be registered in a single statement:

```
CU_ErrorCode error = CU_register_suites(suites);
```

If an error occurs during the registration of any suite or test, an error code is returned. The error codes are the same as those returned by normal suite registration and test addition operations. The function CU_register_nsuites() is provided for situations in which the user wishes to register multiple CU_SuiteInfo arrays in a single statement:

```
CU_ErrorCode error = CU_register_nsuites(2, suites1, suites2);
```

This function accepts a variable number of CU_SuiteInfo arrays. The first argument indicates the actual number ot arrays being passed.

4.5. Deprecated v1 Data Types & Functions

The following data types and functions are deprecated as of version 2. To use these deprecated names, user code must be compiled with USE_DEPRECATED_CUNIT_NAMES defined.

#include <CUnit/TestDB.h> (included automatically by CUnit/CUnit.h>).

Deprecated Name	Equivalent New Name
TestFunc	CU_TestFunc
InitializeFunc	CU_InitializeFunc
CleanupFunc	CU_CleanupFunc
_TestCase	CU_Test
PTestCase	CU_pTest
_TestGroup	CU_Suite
PTestGroup	CU_pSuite
add_test_group()	CU_add_suite()
add_test_case()	CU_add_test()
ADD_TEST_TO_GROUP()	CU_ADD_TEST()
test_case_t	CU_TestInfo
test_group_t	CU_SuiteInfo
test_suite_t	no equivalent - use CU_SuiteInfo
TEST_CASE_NULL	CU_TEST_INFO_NULL
TEST_GROUP_NULL	CU_SUITE_INFO_NULL
test_group_register	CU_register_suites()
test_suite_register	no equivalent - use CU_register_suites()

5. Running Tests

5.1. Synopsis

```
#include <CUnit/Automated.h>
    void                CU_automated_run_tests(void)
    CU_ErrorCode        CU_list_tests_to_file(void)
    void                CU_set_output_filename(const char* szFilenameRoot)

#include <CUnit/Basic.h>
    typedef enum        CU_BasicRunMode
    CU_ErrorCode        CU_basic_run_tests(void)
    CU_ErrorCode        CU_basic_run_suite(CU_pSuite pSuite)
    CU_ErrorCode        CU_basic_run_test(CU_pSuite pSuite, CU_pTest pTest)
    void                CU_basic_set_mode(CU_BasicRunMode mode)
    CU_BasicRunMode     CU_basic_get_mode(void)
    void                CU_basic_show_failures(CU_pFailureRecord pFailure)

#include <CUnit/Console.h>
    void    CU_console_run_tests(void)

#include <CUnit/CUCurses.h>
    void    CU_curses_run_tests(void)

#include <CUnit/TestRun.h> (included automatically by <CUnit/CUnit.h>)
    unsigned int    CU_get_number_of_suites_run(void)
    unsigned int    CU_get_number_of_suites_failed(void)
    unsigned int    CU_get_number_of_tests_run(void)
    unsigned int    CU_get_number_of_tests_failed(void)
    unsigned int    CU_get_number_of_asserts(void)
    unsigned int    CU_get_number_of_successes(void)
    unsigned int    CU_get_number_of_failures(void)

    typedef    struct    CU_RunSummary
    typedef    CU_Runsummary*    CU_pRunSummary
    const      CU_pRunSummary    CU_get_run_summary(void)

    typedef struct    CU_FailureRecord
    typedef    CU_FailureRecord*     CU_pFailureRecord
    const      CU_pFailureRecord     CU_get_failure_list(void)
    unsigned int     CU_get_number_of_failure_records(void)
```

5.2. Running Tests in CUnit

CUnit supports running all tests in all registered suites, but individual tests or suites can also be run. During each run, the framework keeps track of the number of suites, tests, and assertions run, passed, and failed. Note that the results are cleared each time a test run is initiated (even if it fails).

While CUnit provides primitive functions for running suites and tests, most users will want to use one of the simplified user interfaces. These interfaces handle the details of interaction with the framework and provide output of test details and results for the user.

The following interfaces are included in the CUnit library:

Interface	Platform	Description
Automated	all	non-interactive with output to xml files
Basic	all	non-interactive with optional output to stdout
Console	all	interactive console mode under user control
Curses	Linux/Unix	interactive curses mode under user control

If these interfaces are not sufficient, clients can also use the primitive framework API defined in <CUnit/TestRun.h>. See the source code for the various interfaces for examples of how to interact with the primitive API directly.

5.3. Automated Mode

The automated interface is non-interactive. Clients initiate a test run, and the results are output to an XML file. A listing of the registered tests and suites can also be reported to an XML file.

The following functions comprise the automated interface API:

void   CU_automated_run_tests(void)

Runs all tests in all registered suites. Test results are output to a file named ROOT-Results.xml. The filename ROOT can be set using CU_set_output_filename(), or else the default CUnitAutomated-Results.xml is used. Note that if a distict filename ROOT is not set before each run, the results file will be overwritten.

The results file is supported by both a document type definition file (CUnit-Run.dtd) and XSL stylesheet (CUnit-Run.xsl). These are provided in the Share subdirectory of the source and installation trees.

CU_ErrorCode   CU_list_tests_to_file(void)

Lists the registered suites and associated tests to file. The listing file is named ROOT-Listing.xml. The filename ROOT can be set using CU_set_output_filename(), or else the default CUnitAutomated is used. Note that if a distict filename ROOT is not set before each run, the listing file will be overwritten.

The listing file is supported by both a document type definition file (CUnit-List.dtd) and XSL stylesheet

(CUnit-List.xsl). These are provided in the Share subdirectory of the source and installation trees.

Note also that a listing file is not generated automatically by CU_automated_run_tests(). Client code must explicitly request a listing when one is desired.

Void CU_set_output_filename(const char* szFilenameRoot)

Sets the output filenames for the results and listing files. szFilenameRoot is used to construct the filenames by appending -Results.xml and -Listing.xml, respectively.

5.4. Basic Mode

The basic interface is also non-interactive, with results output to stdout. This interface supports running individual suites or tests, and allows client code to control the type of output displayed during each run. This interface provides the most flexibility to clients desiring simplified access to the CUnit API.

The following public functions are provided:

CU_ErrorCode CU_basic_run_tests(void)

Runs all tests in all registered suites. Returns the 1st error code occurring during the test run. The type of output is controlled by the current run mode, which can be set using CU_basic_set_mode().

CU_ErrorCode CU_basic_run_suite(CU_pSuite pSuite)

Runs all tests in single specified suite. Returns the 1st error code occurring during the test run. The type of output is controlled by the current run mode, which can be set using CU_basic_set_mode().

CU_ErrorCode CU_basic_run_test(CU_pSuite pSuite, CU_pTest pTest)

Runs a single test in a specified suite. Returns the 1st error code occurring during the test run. The type of output is controlled by the current run mode, which can be set using CU_basic_set_mode().

Void CU_basic_set_mode(CU_BasicRunMode mode)

Sets the basic run mode, which controls the output during test runs. Choices are:

CU_BRM_NORMAL	Failures and run summary are printed.
CU_BRM_SILENT	No output is printed except error messages.
CU_BRM_VERBOSE	Maximum output of run details.

CU_BasicRunMode CU_basic_get_mode(void)

Retrieves the current basic run mode code.

```
void CU_basic_show_failures(CU_pFailureRecord pFailure)
```

Prints a summary of all failures to stdout. Does not depend on the run mode.

5.5. Interactive Console Mode

The console interface is interactive. All the client needs to do is initiate the console session, and the user controls the test run interactively. This include selection & running of registered suites and tests, and viewing test results. To start a console session, use

```
void CU_console_run_tests(void)
```

5.6. Interactive Curses Mode

The curses interface is interactive. All the client needs to do is initiate the curses session, and the user controls the test run interactively. This include selection & running of registered suites and tests, and viewing test results. Use of this interface requires linking the ncurses library into the application. To start a curses session, use

```
void CU_curses_run_tests(void)
```

5.7. Getting Test Results

The interfaces present results of test runs, but client code may sometimes need to access the results directly. These results include various run counts, as well as a linked list of failure records holding the failure details. Note that test results are overwritten each time a new test run is started, or when the registry is initialized or cleaned up.

Functions for accessing the test results are:

```
unsigned int    CU_get_number_of_suites_run(void)
unsigned int    CU_get_number_of_suites_failed(void)
unsigned int    CU_get_number_of_tests_run(void)
unsigned int    CU_get_number_of_tests_failed(void)
unsigned int    CU_get_number_of_asserts(void)
unsigned int    CU_get_number_of_successes(void)
unsigned int    CU_get_number_of_failures(void)
```

These functions report the number of suites, tests, and assertions that ran or failed during the last run. A suite is considered failed if it's initialization or cleanup function returned non-NULL. A test fails if any of its assertions fail. The last 3 functions all refer to individual assertions.

To retrieve the total number of registered suites and tests, use CU_get_registry()->uiNumberOfSuites and

CU_get_registry()−>uiNumberOfTests, respectively.

const CU_pRunSummary CU_get_run_summary(void)

Retrieves all test result counts at once. The return value is a pointer to a saved structure containing the counts. This data type is defined in <CUnit/TestRun.h> (included automatically by <CUnit/CUnit.h>):

```
typedef struct CU_RunSummary
{
   unsigned int nSuitesRun;
   unsigned int nSuitesFailed;
   unsigned int nTestsRun;
   unsigned int nTestsFailed;
   unsigned int nAsserts;
   unsigned int nAssertsFailed;
   unsigned int nFailureRecords;
} CU_RunSummary;


Typedef   CU_Runsummary*   CU_pRunSummary;
```

The structure variable associated with the returned pointer is owned by the framework, so the user should not free or otherwise change it. Note that the pointer may be invalidated once another test run is initiated.

const CU_pFailureRecord CU_get_failure_list(void)

Retrieves a linked list recording any failures occurring during the last test run (NULL for no failures). The data type of the return value is declared in <CUnit/TestRun.h> (included automatically by <CUnit/CUnit.h>). Each failure record contains information about the location and nature of the failure:

```
typedef struct   CU_FailureRecord
{
   unsigned int   uiLineNumber;
   char*             strFileName;
   char*             strCondition;
   CU_pTest          pTest;
   CU_pSuite         pSuite;

   Struct   CU_FailureRecord*   pNext;
   struct   CU_FailureRecord*   pPrev;

} CU_FailureRecord;

typedef   CU_FailureRecord*   CU_pFailureRecord;
```

The structure variable associated with the returned pointer is owned by the framework, so the user should not free or otherwise change it. Note that the pointer may be invalidated once another test run is initiated.

unsigned int CU_get_number_of_failure_records(void)

Retrieves the number of CU_FailureRecords in the linked list of failures returned by CU_get_failure_list(). Note that this can be more than the number of failed assertions, since suite initialization and cleanup failures are included.

5.8. Deprecated v1 Data Types & Functions

The following data types and functions are deprecated as of version 2. To use these deprecated names, user code must be compiled with USE_DEPRECATED_CUNIT_NAMES defined.

Deprecated Name	Equivalent New Name
automated_run_tests()	CU_automated_run_tests() plus CU_list_tests_to_file()
set_output_filename()	CU_set_output_filename()
console_run_tests()	CU_console_run_tests()
curses_run_tests()	CU_curses_run_tests()

6. Error Handling

6.1. Synopsis

#include <CUnit/CUError.h> (included automatically by <CUnit/CUnit.h>)

```
typedef   enum      CU_ErrorCode
CU_ErrorCode      CU_get_error(void);
const char*       CU_get_error_msg(void);

Typedef   enum      CU_ErrorAction
void              CU_set_error_action(CU_ErrorAction action);
CU_ErrorAction    CU_get_error_action(void);
```

6.2. CUnit Error Handling

Most CUnit functions set an error code indicating the framework error status. Some functions return the code, while others just set the code and return some other value. Two functions are provided for examining the framework error status:

CU_ErrorCode CU_get_error(void)
const char* CU_get_error_msg(void)
The first returns the error code itself, while the second returns a message describing the error status. The error code is an enum of type CU_ErrorCode defined in <CUnit/CUError.h>. The following error code values are defined:

Error Value	Description
CUE_SUCCESS	No error condition.
CUE_NOMEMORY	Memory allocation failed.
CUE_NOREGISTRY	Test registry not initialized.
CUE_REGISTRY_EXISTS	Attempt to CU_set_registry() without CU_cleanup_registry().
CUE_NOSUITE	A required CU_pSuite pointer was NULL.
CUE_NO_SUITENAME	Required CU_Suite name not provided.
CUE_SINIT_FAILED	Suite initialization failed.
CUE_SCLEAN_FAILED	Suite cleanup failed.
CUE_DUP_SUITE	Duplicate suite name not allowed.
CUE_NOTEST	A required CU_pTest pointer was NULL.
CUE_NO_TESTNAME	Required CU_Test name not provided.
CUE_DUP_TEST	Duplicate test case name not allowed.
CUE_TEST_NOT_IN_SUITE	Test is not registered in the specified suite.
CUE_FOPEN_FAILED	An error occurred opening a file.
CUE_FCLOSE_FAILED	An error occurred closing a file.
CUE_BAD_FILENAME	A bad filename was requested (NULL, empty, nonexistent, etc.).
CUE_WRITE_ERROR	An error occurred during a write to a file.

6.3. Behavior Upon Framework Errors

The default behavior when an error condition is encountered is for the error code to be set and execution continued. There may be times when clients prefer for a test run to stop on a framework error, or even for the test application to exit. This behavior can be set by the user, for which the following functions are provided:

```
void CU_set_error_action(CU_ErrorAction action)
CU_ErrorAction CU_get_error_action(void)
```

The error action code is an enum of type CU_ErrorAction defined in <CUnit/CUError.h>. The following error action codes are defined:

Error Value	Description
CUEA_IGNORE	Runs should be continued when an error condition occurs (default)
CUEA_FAIL	Runs should be stopped when an error condition occurs
CUEA_ABORT	The application should exit() when an error conditions occurs

6.4. Deprecated v1 Variables & Functions

The following variables and functions are deprecated as of version 2. To use these deprecated names, user code must be compiled with USE_DEPRECATED_CUNIT_NAMES defined.

Deprecated Name	Equivalent New Name
get_error()	CU_get_error_msg()
error_code	None. Use CU_get_error()

CUnit Data Structures

Here are the data structures with brief descriptions:

APPPAD	Window elements
APPWINDOWS	Pointers to curses interface windows
CU_FailureRecord	Data type for holding assertion failure information (linked list)
CU_RunSummary	Data type for holding statistics and assertion failures for a test run
CU_Suite	CUnit suite data type
CU_SuiteInfo	Suite parameters
CU_Test	CUnit test case data type
CU_TestInfo	Test case parameters
CU_TestRegistry	CUnit test registry data type
TE	
test_suite	Deprecated (version 1)

CUnit File List

Here is a list of all files with brief descriptions:Automated.c Automated test interface with xml result output (implementation)

Automated.h [code]	Automated testing interface with xml output (user interface)
Basic.c	Basic interface with output to stdout
Basic.h [code]	Basic interface with output to stdout
Console.c	Console test interface with interactive output (implementation)
Console.h [code]	Console interface with interactive output (user interface)
CUCurses.h [code]	Curses testing interface with interactive output (user interface)
CUError.c	Error handling functions (implementation)
CUError.h [code]	Error handling functions (user interface)
CUnit.h [code]	Basic CUnit include file for user and system code
Curses.c	Curses test interface with interactive output (implementation)
MyMem.c	Memory management & reporting functions (implementation)
MyMem.h [code]	Memory management functions (user interface)
test_cunit.c	CUnit internal testingfunctions (implementation)
test_cunit.h [code]	Interface for CUnit internal testing functions
TestDB.c	Management functions for tests, suites, and the test registry (implementation)
TestDB.h [code]	Management functions for tests, suites, and the test registry (user interface)
TestRun.c	Test run management functions (implementation)
TestRun.h [code]	Test run management functions (user interface)
Util.c	Utility functions (implementation)
Util.h [code]	Utility functions (user interface)

Screenshots are available for the following CUnit interfaces:

Basic interface
Automated interface
Console interactive interface
Curses interactive interface (only on systems supporting curses)

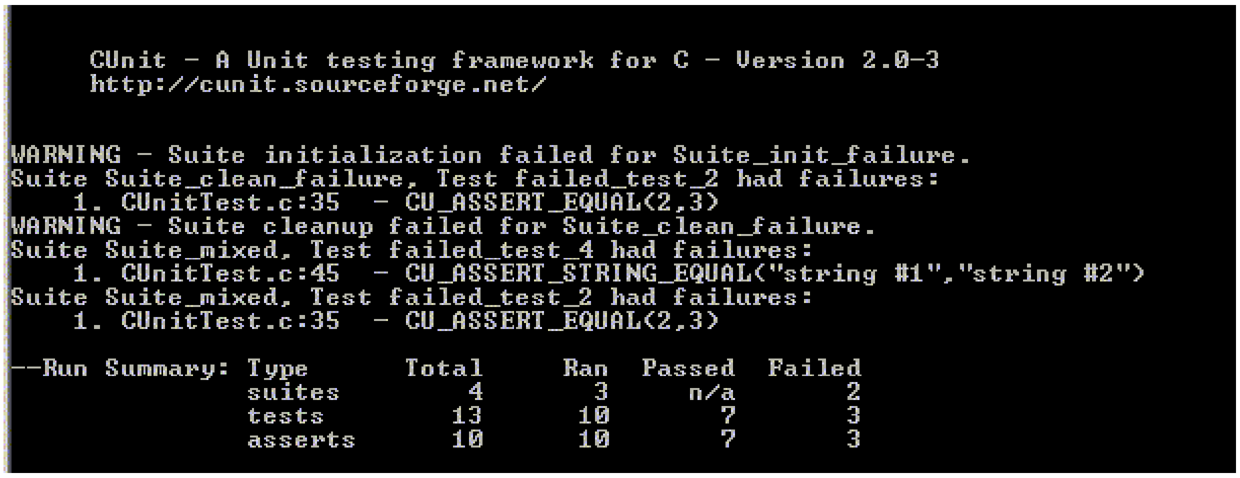
Here is the unit test code used to generate the screenshots:

```
#include "CUnit/Basic.h"
#include "CUnit/Console.h"
#include "CUnit/Automated.h"
#include "CUnit/CUCurses.h"      /* only on systems having curses */

int init_suite_success(void) { return 0; }
int init_suite_failure(void) { return -1; }
int clean_suite_success(void) { return 0; }
int clean_suite_failure(void) { return -1; }

void test_success1(void)
{
    CU_ASSERT(TRUE);
}

void test_success2(void)
{
    CU_ASSERT_NOT_EQUAL(2, -1);
}

void test_success3(void)
{
    CU_ASSERT_STRING_EQUAL("string #1", "string #1");
}

void test_success4(void)
{
    CU_ASSERT_STRING_NOT_EQUAL("string #1", "string #2");
}

void test_failure1(void)
{
    CU_ASSERT(FALSE);
}
```

```
void test_failure2(void)
{
    CU_ASSERT_EQUAL(2, 3);
}

void test_failure3(void)
{
    CU_ASSERT_STRING_NOT_EQUAL("string #1", "string #1");
}

void test_failure4(void)
{
    CU_ASSERT_STRING_EQUAL("string #1", "string #2");
}


int main()
{
    CU_pSuite pSuite = NULL;

    /* initialize the CUnit test registry */
    if (CUE_SUCCESS != CU_initialize_registry())
        return CU_get_error();

    /* add a suite to the registry */
    pSuite = CU_add_suite("Suite_success", init_suite_success, clean_suite_success);
    if (NULL == pSuite) {
        CU_cleanup_registry();
        return CU_get_error();
    }

    /* add the tests to the suite */
    if ((NULL == CU_add_test(pSuite, "successful_test_1", test_success1)) ||
         (NULL == CU_add_test(pSuite, "successful_test_2", test_success2)) ||
         (NULL == CU_add_test(pSuite, "successful_test_3", test_success3)))
    {
        CU_cleanup_registry();
        return CU_get_error();
    }

    /* add a suite to the registry */
    pSuite = CU_add_suite("Suite_init_failure", init_suite_failure, NULL);
    if (NULL == pSuite) {
        CU_cleanup_registry();
        return CU_get_error();
    }

    /* add the tests to the suite */
    if ((NULL == CU_add_test(pSuite, "successful_test_1", test_success1)) ||
         (NULL == CU_add_test(pSuite, "successful_test_2", test_success2)) ||
         (NULL == CU_add_test(pSuite, "successful_test_3", test_success3)))
    {
        CU_cleanup_registry();
        return CU_get_error();
    }

    /* add a suite to the registry */
    pSuite = CU_add_suite("Suite_clean_failure", NULL, clean_suite_failure);
    if (NULL == pSuite) {
        CU_cleanup_registry();
        return CU_get_error();
    }

    /* add the tests to the suite */
    if ((NULL == CU_add_test(pSuite, "successful_test_4", test_success1)) ||
         (NULL == CU_add_test(pSuite, "failed_test_2",     test_failure2)) ||
         (NULL == CU_add_test(pSuite, "successful_test_1", test_success1)))
    {
        CU_cleanup_registry();
        return CU_get_error();
    }

    /* add a suite to the registry */
    pSuite = CU_add_suite("Suite_mixed", NULL, NULL);
    if (NULL == pSuite) {
        CU_cleanup_registry();
        return CU_get_error();
    }

    /* add the tests to the suite */
    if ((NULL == CU_add_test(pSuite, "successful_test_2", test_success2)) ||
         (NULL == CU_add_test(pSuite, "failed_test_4",     test_failure4)) ||
         (NULL == CU_add_test(pSuite, "failed_test_2",     test_failure2)) ||
         (NULL == CU_add_test(pSuite, "successful_test_4", test_success4)))
    {
        CU_cleanup_registry();
        return CU_get_error();
```

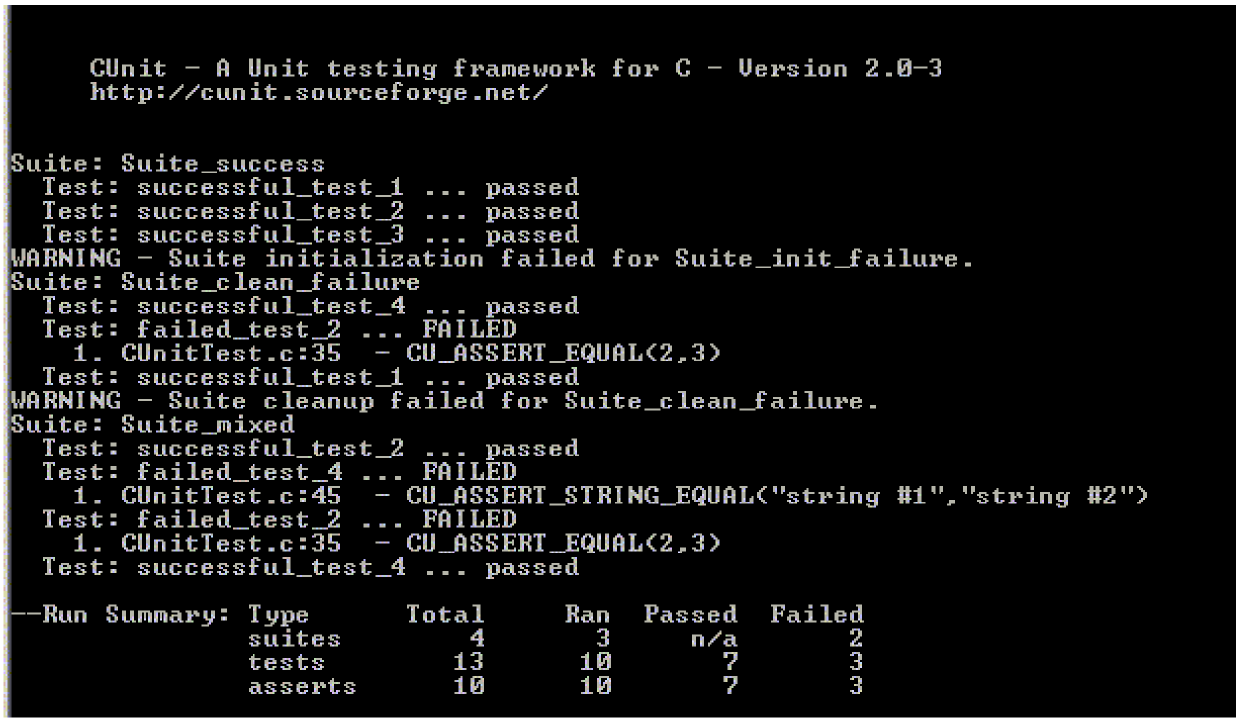
```
    }

    /* Run all tests using the basic interface */
    CU_basic_set_mode(CU_BRM_VERBOSE);
    CU_basic_run_tests();
    printf("\n");
    CU_basic_show_failures(CU_get_failure_list());
    printf("\n\n");

    /* Run all tests using the automated interface */
    CU_automated_run_tests();
    CU_list_tests_to_file();

    /* Run all tests using the console interface */
    CU_console_run_tests();

    /* Run all tests using the curses interface */
    /* (only on systems having curses) */
    CU_curses_run_tests();

    /* Clean up registry and return */
    CU_cleanup_registry();
    return CU_get_error();
}
```

The screenshots below were generated by this code using the CUnit Basic interface.



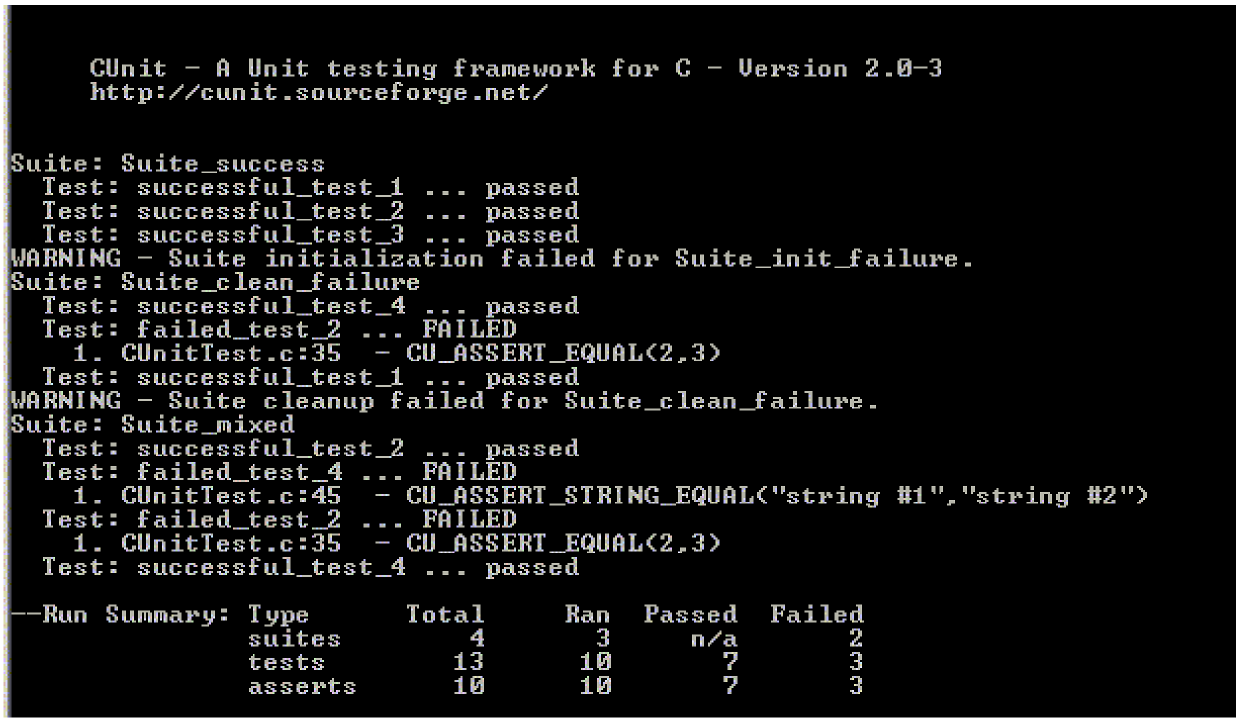
```
     CUnit - A Unit testing framework for C - Version 2.0-3
     http://cunit.sourceforge.net/


WARNING - Suite initialization failed for Suite_init_failure.
Suite Suite_clean_failure, Test failed_test_2 had failures:
    1. CUnitTest.c:35  - CU_ASSERT_EQUAL(2,3)
WARNING - Suite cleanup failed for Suite_clean_failure.
Suite Suite_mixed, Test failed_test_4 had failures:
    1. CUnitTest.c:45  - CU_ASSERT_STRING_EQUAL("string #1","string #2")
Suite Suite_mixed, Test failed_test_2 had failures:
    1. CUnitTest.c:35  - CU_ASSERT_EQUAL(2,3)

--Run Summary: Type      Total     Ran  Passed  Failed
               suites        4       3     n/a       2
               tests        13      10       7       3
               asserts      10      10       7       3
```

Console output at end of test run (normal mode)



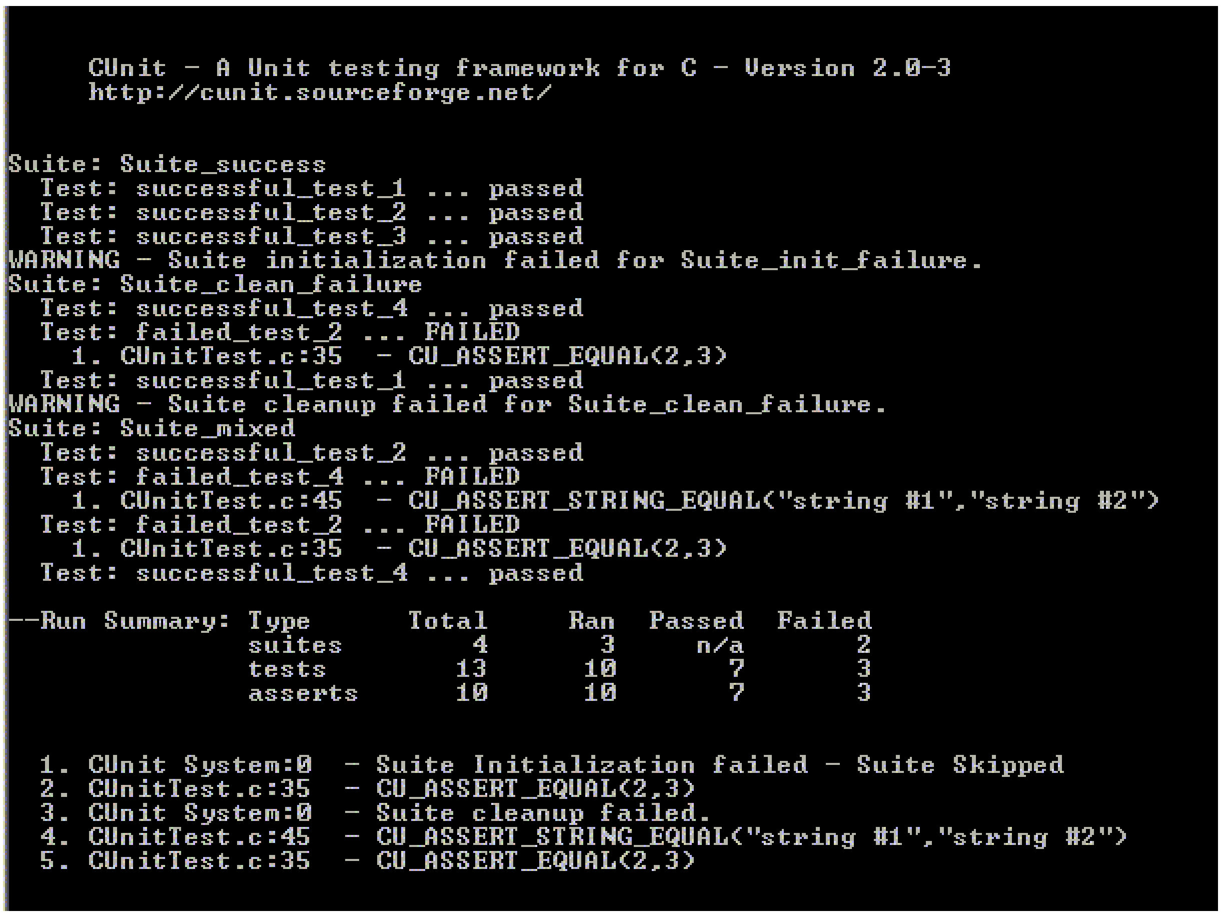
```
     CUnit - A Unit testing framework for C - Version 2.0-3
     http://cunit.sourceforge.net/


Suite: Suite_success
  Test: successful_test_1 ... passed
  Test: successful_test_2 ... passed
  Test: successful_test_3 ... passed
WARNING - Suite initialization failed for Suite_init_failure.
Suite: Suite_clean_failure
  Test: successful_test_4 ... passed
  Test: failed_test_2 ... FAILED
    1. CUnitTest.c:35  - CU_ASSERT_EQUAL(2,3)
  Test: successful_test_1 ... passed
WARNING - Suite cleanup failed for Suite_clean_failure.
Suite: Suite_mixed
  Test: successful_test_2 ... passed
  Test: failed_test_4 ... FAILED
    1. CUnitTest.c:45  - CU_ASSERT_STRING_EQUAL("string #1","string #2")
  Test: failed_test_2 ... FAILED
    1. CUnitTest.c:35  - CU_ASSERT_EQUAL(2,3)
  Test: successful_test_4 ... passed

--Run Summary: Type      Total     Ran  Passed  Failed
               suites        4       3     n/a       2
               tests        13      10       7       3
               asserts      10      10       7       3
```

Console output at end of test run (verbose mode)



```
     CUnit - A Unit testing framework for C - Version 2.0-3
     http://cunit.sourceforge.net/


Suite: Suite_success
  Test: successful_test_1 ... passed
  Test: successful_test_2 ... passed
  Test: successful_test_3 ... passed
WARNING - Suite initialization failed for Suite_init_failure.
Suite: Suite_clean_failure
  Test: successful_test_4 ... passed
  Test: failed_test_2 ... FAILED
    1. CUnitTest.c:35  - CU_ASSERT_EQUAL(2,3)
  Test: successful_test_1 ... passed
WARNING - Suite cleanup failed for Suite_clean_failure.
Suite: Suite_mixed
  Test: successful_test_2 ... passed
  Test: failed_test_4 ... FAILED
    1. CUnitTest.c:45  - CU_ASSERT_STRING_EQUAL("string #1","string #2")
  Test: failed_test_2 ... FAILED
    1. CUnitTest.c:35  - CU_ASSERT_EQUAL(2,3)
  Test: successful_test_4 ... passed

--Run Summary: Type      Total     Ran  Passed  Failed
               suites        4       3     n/a       2
               tests        13      10       7       3
               asserts      10      10       7       3
```

Console output at end of test run with failures displayed (verbose mode)



```
     CUnit - A Unit testing framework for C - Version 2.0-3
     http://cunit.sourceforge.net/


Suite: Suite_success
  Test: successful_test_1 ... passed
  Test: successful_test_2 ... passed
  Test: successful_test_3 ... passed
WARNING - Suite initialization failed for Suite_init_failure.
Suite: Suite_clean_failure
  Test: successful_test_4 ... passed
  Test: failed_test_2 ... FAILED
    1. CUnitTest.c:35  - CU_ASSERT_EQUAL(2,3)
  Test: successful_test_1 ... passed
WARNING - Suite cleanup failed for Suite_clean_failure.
Suite: Suite_mixed
  Test: successful_test_2 ... passed
  Test: failed_test_4 ... FAILED
    1. CUnitTest.c:45  - CU_ASSERT_STRING_EQUAL("string #1","string #2")
  Test: failed_test_2 ... FAILED
    1. CUnitTest.c:35  - CU_ASSERT_EQUAL(2,3)
  Test: successful_test_4 ... passed

--Run Summary: Type      Total     Ran  Passed  Failed
               suites        4       3     n/a       2
               tests        13      10       7       3
               asserts      10      10       7       3


  1. CUnit System:0  - Suite Initialization failed - Suite Skipped
  2. CUnitTest.c:35  - CU_ASSERT_EQUAL(2,3)
  3. CUnit System:0  - Suite cleanup failed.
  4. CUnitTest.c:45  - CU_ASSERT_STRING_EQUAL("string #1","string #2")
  5. CUnitTest.c:35  - CU_ASSERT_EQUAL(2,3)
```

Example Code

```
/*
 *   Simple example of a CUnit unit test.
 *
 *   This program (crudely) demonstrates a very simple "black box"
 *   test of the standard library functions fprintf() and fread().
 *   It uses suite initialization and cleanup functions to open
 *   and close a common temporary file used by the test functions.
 *   The test functions then write to and read from the temporary
 *   file in the course of testing the library functions.
 *
 *   The 2 test functions are added to a single CUnit suite, and
 *   then run using the CUnit Basic interface.   The output of the
 *   program (on CUnit version 2.0-2) is:
 *
 *               CUnit : A Unit testing framework for C.
 *               http://cunit.sourceforge.net/
 *
 *          Suite: Suite_1
 *            Test: test of fprintf() ... passed
 *            Test: test of fread() ... passed
 *
 *          --Run Summary: Type      Total     Ran    Passed   Failed
 *                         suites        1       1      n/a        0
 *                         tests         2       2        2        0
 *                         asserts       5       5        5        0
 */

#include <stdio.h>
#include <string.h>
#include "CUnit/Basic.h"

/* Pointer to the file used by the tests. */
static FILE* temp_file = NULL;

/* The suite initialization function.
 * Opens the temporary file used by the tests.
 * Returns zero on success, non-zero otherwise.
 */
int init_suite1(void)
{
    if (NULL == (temp_file = fopen("temp.txt", "w+"))) {
        return -1;
    }
    else {
        return 0;
    }
}

/* The suite cleanup function.
 * Closes the temporary file used by the tests.
 * Returns zero on success, non-zero otherwise.
 */
int clean_suite1(void)
{
    if (0 != fclose(temp_file)) {
        return -1;
    }
    else {
        temp_file = NULL;
        return 0;
    }
}

/* Simple test of fprintf().
 * Writes test data to the temporary file and checks
 * whether the expected number of bytes were written.
 */
void testFPRINTF(void)
{
    int i1 = 10;

    if (NULL != temp_file) {
        CU_ASSERT(0 == fprintf(temp_file, ""));
        CU_ASSERT(2 == fprintf(temp_file, "Q\n"));
        CU_ASSERT(7 == fprintf(temp_file, "i1 = %d", i1));
    }
}

/* Simple test of fread().
 * Reads the data previously written by testFPRINTF()
 * and checks whether the expected characters are present.
 * Must be run after testFPRINTF().
 */
void testFREAD(void)
{
    unsigned char buffer[20];
    if (NULL != temp_file) {
        rewind(temp_file);
        CU_ASSERT(9 == fread(buffer, sizeof(unsigned char), 20, temp_file));
        CU_ASSERT(0 == strncmp(buffer, "Q\ni1 = 10", 9));
    }
}

/* The main() function for setting up and running the tests.
 * Returns a CUE_SUCCESS on successful running, another
 * CUnit error code on failure.
 */
int main()
{
    CU_pSuite pSuite = NULL;

    /* initialize the CUnit test registry */
    if (CUE_SUCCESS != CU_initialize_registry())
        return CU_get_error();

    /* add a suite to the registry */
    pSuite = CU_add_suite("Suite_1", init_suite1, clean_suite1);
    if (NULL == pSuite) {
        CU_cleanup_registry();
        return CU_get_error();
    }

    /* add the tests to the suite */
    /* NOTE - ORDER IS IMPORTANT - MUST TEST fread() AFTER fprintf() */
    if ((NULL == CU_add_test(pSuite, "test of fprintf()", testFPRINTF)) ||
        (NULL == CU_add_test(pSuite, "test of fread()", testFREAD)))
    {
        CU_cleanup_registry();
        return CU_get_error();
    }

    /* Run all tests using the CUnit Basic interface */
    CU_basic_set_mode(CU_BRM_VERBOSE);
    CU_basic_run_tests();
    CU_cleanup_registry();
    return CU_get_error();
}
```