

testlib

A library to help write test code

Copyright © 2006 Smithsonian Astrophysical Observatory

Parts are © 1991 Texas Instruments Incorporated, with the following permission notice:

Permission is granted to any individual or institution to use, copy, modify, and distribute this software, provided that this complete copyright and permission notice is maintained, intact, in all copies and supporting documentation.

Texas Instruments Incorporated provides this software "as is" without express or implied warranty.

1 Introduction

`testlib` is a library of routines and C/C++ macros which make the creation of code for testing libraries and programs a little easier.

The routines are available both to C and C++ programs, with the same API. The internals are packaged a little differently (see [Chapter 4 \[Implementation\]](#), page 10).

The macros take an expression and its expected result, and print out a message describing the test and the result of the equivalency. They keep track of the number of passes and failures, and output a summary at the end of the session. The main benefit of using these macros is that you don't have to worry about all of the pretty printing required to get a readable test report.

2 Usage

The first macro invoked should be **START**, which initiates a testing session. The test program then calls the **TEST...** macros, and at the end of the session calls **SUMMARY**. Generally, the result of **FAILURE** should be used to return an appropriate exit value at the conclusion of the test program.

For example, here's a snippet of the test code for the test library itself:

```
START("Testing Test");

{
    int a = 3;
    int b = 2;
    int c = 1;
    TEST("TEST, PASSED", 1, 1);
    TEST("TEST, PASSED", a, a);
    TEST("TEST, **FAILED**", 1, 0);
    TEST("TEST, **FAILED**", a, b);

    TEST_RUN("TEST_RUN, PASSED", a = 4, a, 4 );
    TEST_RUN("TEST_RUN, **FAILED**", a = 3, a, 4 );
}

{
    double d = 22.2;
    double e = 22.3;

    TEST_RUN_FP_TOL( "TEST_RUN_FP_TOL, **FAILED**", double g = d + 200 *
DBL_EPSILON, g, d, 20 * DBL_EPSILON );
}

SUMMARY();
```

The test output includes the line number where the test was called, which in some cases is inappropriate. Each of the **TEST** macros exists in an additional form with a suffix of **__LINE**. These forms take an additional leading parameter, the line number to output. For example

```
TEST_LINE(33, "TEST, PASSED", 1, 1);
```

3 Macros

3.1 Administrative Macros

Macros to start and finish test suites.

3.1.1 START

start a testing session

Synopsis

```
#include <testlib.h>
```

```
START(title)
```

Parameters

title the title to output

Description

START is called at the beginning of a test session. It resets internal counters and outputs a header to the standard output stream which includes the provided title.

Example

```
START( "Txt class" );
```

3.1.2 SUMMARY

output a summary of the tests

Synopsis

```
#include <testlib.h>
```

```
SUMMARY( );
```

Description

SUMMARY is called at the end of a testing session to output the number of tests which have passed and failed.

Example

```
SUMMARY( );
```

3.1.3 FAILED

return the number of tests which have failed

Synopsis

```
#include <testlib.h>
```

```
int FAILED( );
```

Description

FAILED returns the number of tests which have failed. This is useful for constructing an exit value for the test program to alert other software (make, for instance) that there has been a failure.

Example

```
return FAILED( ) ? EXIT_FAILURE : EXIT_SUCCESS;
```

3.2 Test Macros

Macros to test things.

3.2.1 TEST

test an expression against a value

Synopsis

```
#include <testlib.h>
```

```
TEST(desc, expr, res )
```

```
TEST_LINE(line, desc, expr, res )
```

Parameters

line

the line number to output

desc

a short description of the test being performed

expr

an expression to evaluate

res

the expected result of the expression

Description

TEST is called to perform a test of an expression against an expected result. It essentially performs the test `expr == res` .

Example

```
TEST( "+= operator, length",
      hh.get_length(), ff.get_length() + gg.get_length() - 1 );
```

3.2.2 TEST_PASS

indicate that a test passed

Synopsis

```
#include <testlib.h>

TEST_PASS( desc )
TEST_PASS_LINE( line )
```

Parameters

<i>line</i>	the line number to output
<i>desc</i>	a short description of the test being performed

Description

TEST_PASS is called to indicate that a test passed. It is usually used when the determination of such is more complicated than allowed by the other `testlib` mechanisms.

Example

```
TEST_PASS( "+= operator, length");
```

3.2.3 TEST_FAIL

indicate that a test failed

Synopsis

```
#include <testlib.h>

TEST_FAIL( desc )
TEST_FAIL_LINE( line )
```

Parameters

<i>line</i>	the line number to output
<i>desc</i>	a short description of the test being performed

Description

TEST_FAIL is called to indicate that a test failed. It is usually used when the determination of such is more complicated than allowed by the other `testlib` mechanisms.

Example

```
TEST_FAIL( "+= operator, length");
```

3.2.4 TEST_FP

test a floating point expression against a value, with a tolerance range

Synopsis

```
#include <testlib.h>

TEST_FP(desc, expr, res )
TEST_FP_LINE(line, desc, expr, res )
```

Parameters

<i>line</i>	the line number to output
<i>desc</i>	a short description of the test being performed
<i>expr</i>	a floating point expression to evaluate
<i>res</i>	the expected result of the floating point expression

Description

TEST_FP is called to determine the equivalence of a floating point expression against an expected result within a tolerance range. If the numeric difference between the two values is less than the default tolerance of $100.0 * \text{DBL_EPSILON}$, they are considered equivalent. (DBL_EPSILON is defined in `/usr/include/float.h`). To specify a tolerance, see [Section 3.2.5 \[TEST_FP_TOL\]](#), page 6.

Example

```
TEST_FP( "float equiv", 33 * 2.2, 36 );
```

3.2.5 TEST_FP_TOL

test a floating point expression against a value, within a tolerance range

Synopsis

```
#include <testlib.h>

TEST_FP_TOL(desc, expr, res, tol )
TEST_FP_TOL_LINE(line, desc, expr, res, tol )
```

Parameters

line

	the line number to output
<i>desc</i>	a short description of the test being performed
<i>expr</i>	a floating point expression to evaluate
<i>res</i>	the expected result of the floating point expression
<i>tol</i>	the tolerance within which the <i>expr</i> and the <i>res</i> are equivalent.

Description

TEST_FP_TOL is called to determine the equivalence of a floating point expression against an expected result within a specified tolerance range. If the numeric difference between the two values is less than the specified tolerance, they are considered equivalent.

Example

```
TEST_FP_TOL( "float equiv", 33 * 2.2, 36, 2 );
```

3.2.6 TEST_RUN_FP

Execute a statement, then test a floating point expression against a value, within a tolerance range.

Synopsis

```
#include <testlib.h>

TEST_RUN_FP(desc, stmt, expr, res )
TEST_RUN_FP_LINE(line, desc, stmt, expr, res )
```

Parameters

<i>line</i>	the line number to output
<i>desc</i>	a short description of the test being performed
<i>stmt</i>	a statement to be executed, independent of the expression. the statement may be arbitrarily complex.
<i>expr</i>	a floating point expression to evaluate
<i>res</i>	the expected result of the floating point expression

Description

TEST_RUN_FP is called to determine the equivalence of a floating point expression against an expected result within a specified tolerance range. If the numeric difference between the two values is less than the default tolerance of $100.0 * \text{DBL_EPSILON}$, they are considered equivalent. (DBL_EPSILON is defined in `/usr/include/float.h`). To specify a tolerance, see [Section 3.2.7 \[TEST_RUN_FP_TOL\]](#), page 8.

Example

```
TEST_RUN_FP( "float equiv", foo(&x), 33 * x, 36 );
```

3.2.7 TEST_RUN_FP_TOL

Synopsis

```
#include <testlib.h>

TEST_RUN_FP_TOL(desc, stmt, expr, res, tol )
TEST_RUN_FP_TOL_LINE(line, desc, stmt, expr, res, tol )
```

Parameters

<i>line</i>	the line number to output
<i>desc</i>	a short description of the test being performed
<i>stmt</i>	a statement to be executed, independent of the expression. the statement may be arbitrarily complex.
<i>expr</i>	a floating point expression to evaluate
<i>res</i>	the expected result of the floating point expression
<i>tol</i>	the tolerance within which the <i>expr</i> and the <i>res</i> are equivalent.

Description

TEST_RUN_FP_TOL is called to determine the equivalence of a floating point expression against an expected result within a specified tolerance range. If the numeric difference between the two values is less than the specified tolerance, they are considered equivalent.

Example

```
TEST_RUN_FP_TOL( "float equiv", foo(&x), 33 * x, 36, 0.01 );
```

3.2.8 TESTSTRING

test two strings for equivalence

Synopsis

```
#include <testlib.h>

TESTSTRING(desc, str, res )
TESTSTRING_LINE(line, desc, str, res )
```

Parameters

<i>line</i>	the line number to output
<i>desc</i>	a short description of the test being performed
<i>str1</i>	the first string
<i>res</i>	the expected string

Description

TESTSTRING is called to perform a test of a string against an expected result.

Example

```
TESTSTRING( "string test", foo, "foo" );
```

3.2.9 TEST_RUN

Execute a statement, then test an expression against a value

Synopsis

```
#include <testlib.h>

TEST_RUN(desc, stmt, expr, res )
TEST_RUN_LINE(line, desc, stmt, expr, res )
```

Parameters

<i>line</i>	the line number to output
<i>desc</i>	a short description of the test being performed
<i>stmt</i>	a statement to be executed, independent of the expression. the statement may be arbitrarily complex.
<i>expr</i>	an expression to evaluate
<i>res</i>	the expected result of the expression

Description

TEST_RUN is called to determine the equivalence of an expression with an expected result. The supplied *stmt* is first executed.

Example

```
TEST_RUN( "equiv", foo(&x), 33 * x, 36 );
```

4 Implementation

`testlib` is implemented as a set of C/C++ preprocessor macros which invoke a few compiled routines. In order to cut down on name space pollution, the C++ routines are stuck in a structure called `Test`. For C, the routines are given the prefix `Test_`. The user, of course, need not know this, unless these conventions interact with user code.

Table of Contents

1	Introduction	1
2	Usage	2
3	Macros	3
3.1	Administrative Macros	3
3.1.1	START	3
3.1.2	SUMMARY	3
3.1.3	FAILED	3
3.2	Test Macros	4
3.2.1	TEST	4
3.2.2	TEST_PASS	5
3.2.3	TEST_FAIL	5
3.2.4	TEST_FP	6
3.2.5	TEST_FP_TOL	6
3.2.6	TEST_RUN_FP	7
3.2.7	TEST_RUN_FP_TOL	8
3.2.8	TESTSTRING	8
3.2.9	TEST_RUN	9
4	Implementation	10

