

Cutter

C Unit Testing Tool

v. 0.55.0

<http://cutter.pz.org>

8 April 2006

Thanks to all the people who made Cutter possible and to those whose comments and suggestions made it a better product.. To send us your suggestions, please send an e-mail to: cutter [at] pz [dot] org.

Cutter and this manual are available at no cost at <http://cutter.pz.org>

© Copyright 2005-6, Pacific Data Works LLC. All rights reserved. This manual may be distributed at part of Cutter. No fee may be charged for the electronic version of this manual. Reproduction fees for generation of this manual in hard copy are permitted, so long as they do not exceed twice the costs of the raw materials.

Please send corrections and updates to this manual to: cutter [at] pz [dot] org

Table of Contents

INSTALLATION	4
UNINSTALLING CUTTER	4
RUNNING CUTTER	5
THE CUTTER FILE.....	6
BASIC CUTTER FILE	6
ADDITIONAL FEATURES.....	7
<i>Program setup</i>	8
<i>Handling stderr and stdout</i>	8
<i>Setup/teardown and creating mock data items</i>	9
COMPILING AND RUNNING THE GENERATED TESTS.....	11
LINKING	11
SAMPLE CUTTER FILE	12
CUTTER COMMAND REFERENCE.....	15
.EXPECT	15
.GLOBAL	15
.PARAMS	15
.REDERR	16
.REDOUT.....	16
.SETUP	16
.START.....	16
.STOP	17
.SU	17
.TD	17
.TEARDOWN	17
.TEST	17

Installation

To install Cutter, download the executable file from <http://cutter.pz.org> and place it in a directory of your choice. Then run the program from there. This installation procedure is the same for Windows and Linux. Cutter does not touch the registry in Windows, nor does it create files or directories, or in anyway modify system settings or environmental variables.

Uninstalling Cutter

To remove Cutter from your system, simply erase the cutter executable file. No other changes need be made.

Running Cutter

Cutter reads a Cutter file that specifies what functional and unit tests to run on a program and generates complete C code that, when compiled, runs those tests as specified. To run Cutter, you need a Cutter file (described in later sections). To have Cutter convert generated the tests as C code, use the following command line (italicized terms should be replaced by you):

```
cutter cutter-filename [-o output-C-filename] [-v]
```

Terms in square brackets are optional. If no output file is specified, output is sent to the screen/console.

The `-v` option (for verbose) prints diagnostic data about how Cutter parsed the Cutter file. This information can be useful if the results you obtain differ from what you expect.

If you forget these options, simply run Cutter with no arguments, and a help message will appear explaining the command line and the available option.

The Cutter File

This section explains how to write a Cutter file. It presents a trivial example and then explores more substantial scripts. As you'll see, however, even the most complex scripts are easy to understand and write.

Basic Cutter File

The following sample file shows a minimal Cutter file. Subsequent examples will explore more-advanced features.

```
01 ; minimal Cutter file
02
03 .start
04
05 .function int test_main( int i, char *pc )
06     .test
07         .expect EXIT_FAILURE
08         .params 1, NULL
09
10 .stop
```

Listing 1. A minimal Cutter file.

In this file, the developer has specified that code for a single test of one function, called `test_main()` should be generated. Taking the lines from the top:

Line 01: Comments are indicated by a `;` which must be the first character in the line. The entire line is a comment.

Line 03: The beginning of the specification of the test script. Anything occurring before the `.start` command is ignored. Cutter files can have only one `.start` command.

Line 05: The function to be tested. The `.function` record contains the complete signature. It must show return values and the individual argument types. In this example, the names of the arguments (`i` and `pc`) need not be specified.

Line 06: identifies the beginning of a test of the previously named function.

Line 07: What the expected return from the function is. If the function returns this value when the test is run, the test is considered a success; anything else and the test is considered to have failed.

You can use variables and manifest constants (as in this example) and you can use relational operators (such as `>`, `<`, `>=`, `<=`, `!=`, `==` or `=`). If no operator is

specified, as in the example, it is understood to be `==`. Use of the `=` operator is the same as using `==` for this record.

Line 08: The `.params` record specifies the parameters to pass to the function during the test. In this example, the function will be passed a 1 and a `NULL`. A `.params` record must immediately follow an `.expect` record.

Line 10: indicated the end of the Cutter specification. Anything after the `.stop` record is ignored.

Cutter file generates a fair amount of setup code prior to generating the test code that derives strictly from this example. That test code is:

```
fprintf( pzScreen, "test 0001: %s: ",
        "int test_main( int i, char *pc ): " );
fflush( pzScreen );

if ( ( test_main ( 1, NULL ) ) == EXIT_FAILURE )
{
    fprintf( pzScreen, "passed\n" );
    fflush( pzScreen );
}
else
{
    fprintf( pzScreen, "FAILED\n" );
    fflush( pzScreen );
    pzCutterFailedTests++;
}
pzCutterTestsCompleted++;
```

Listing 2. Test-related C code generated by Cutter for Listing 1.

Presuming the test passes, the output from this test is:

```
test 0001: int test_main( int i, char *pc ): : passed
1 tests completed, of which 0 tests failed
```

Adding more tests to the Cutter script consists of adding three lines (a `.test`, `.expect`, and `.params` record) to a function. In this way, it is easy to generate hundreds of tests that pass different parameters and check for different return values. There is no limit on the number of functions that can be specified nor the number of tests per function. Functions and tests can repeat. Cutter will generate the code for them in the order in which they're specified in the Cutter file.

Additional Features

The following features enable Cutter to deliver the test results with a minimum of inconvenience:

Program setup

A `.global` record initiates a section of a file that contains global program data or directives. In this section, you would list special `#include` files or global variables that the tested functions rely on. The `.global` record and associated statements appear after the `.start` record and before the first `.function` record, as illustrated in Listing 3.

```
01 .start
02 .global
03     #include "portabletype.h"
04     char* test_args[] = { "tester", "sample.cut" };
05
06 .function int cutter_main( int argc, char *argv[] )
07     .redout pzRedirectStdout
08     .test
09         .expect EXIT_FAILURE
10         .params 1, NULL
11     .test
12         .expect != EXIT_FAILURE
13         .params 2, test_args
    . . .
```

Listing 3. A Cutter fragment showing a `.global` record in use

The `.global` record appears on line 2. Lines 3 and 4 are copied by Cutter directly into the generated C file, without any modification. Cutter stops copying lines when it encounters another Cutter command (in this case `.function` on line 6).

Notice that in Listing 3, the `.global` record created an array of two strings, which is later passed to `cutter_main()` as a parameter. This occurs in the second test on line 13 of the listing.

Handling stderr and stdout

When testing functions, especially when passing them invalid parameters to make sure they are handled gracefully, error messages are frequently written to `stdout` and `stderr`. To avoid these messages intermingling with the test results, Cutter redirects both streams to a nul device. In this way none of the error messages are seen. Most often this is the desired behavior; the tester simply wants to know whether the function completed correctly or returned an error code. And in the latter case, which error code.

However, there are times when it might be beneficial to capture the output to `stderr` and `stdout`. To do this in Cutter, you use two commands: `.rederr` (for redirect standard error) and `.redout` (for redirect stdout). These commands are placed after a `.function` record. The `.redout` and `.rederr` keywords are followed by the filename to which the output should be redirected, as shown in Listing 4.

```
01 .function int test_main( int argc, char *argv[] )
02     .redout c:\redirectFilename
03     .test
04         .expect EXIT_FAILURE
05         .params 1, NULL
```

Listing 4. Cutter file excerpt showing the use of `.redout`

If the filename contains spaces, enclose it in quotation marks as shown in Listing 5.

```
01 .function char *get_func_invocation_name( char *func_decl );
02     .rederr "pz Redir Stderr"
03     .test
04         .expect == NULL
05         .params "int func("
```

Listing 5. Cutter file excerpt showing the use of .rederr

Notice the filename with embedded spaces on line 2 of this listing.

.rederr and .redout can be specified anywhere after a .function record or a .test record. Once either stdout or stderr has been redirected to a disk file, they can no longer be disabled. All program output to these streams will be written to these files. Note that test results will appear on the console regardless of redirection of stdout or stderr.

Setup/teardown and creating mock data items

A guiding principle of writing unit tests is that each test should stand on its own. Hence, tests should not be ordered in such a way that the results of a previous test are used as the input to another test. Rather each test should set-up the data or conditions it needs and then clean up after itself, as needed, in a process called tear-down.

Cutter supports set-up and tear-down code, as shown in Listing 6:

```
01 .function ui32  get_linked_list_count( void );
02     .test
03     .setup
04         create_linked_list();
05         add_rec_to_linked_list( "test", +1 );
05     .expect = 1
06     .params
07     .teardown
08         delete_linked_list();
```

Listing 6. Cutter use of set-up and tear-down.

This function counts the number of items in a linked list and returns an unsigned 32-bit integer containing that count. Before the count can be obtained, the link list structure must be created and a node added to the list. This is done in lines 3 -5, via the use of a .setup record. Everything between .setup and the .expect record is code that is copied as is into the generated C file. After the test, the .teardown record lines 7-8 identified the code that cleans up after the test: it deletes the linked list structure. Like the .setup record, the code that appears between a .teardown record and the next Cutter dot record is copied as is to the generated C file.

All test can use a .setup and/or .teardown records. Sometimes, it is exceedingly unwieldy to duplicate setup and teardowns for every test, and it's easier to do the set-up once at the beginning of a function (that is, at the beginning of a series of tests). Cutter

allows for this: `.setup` records can be placed after `.function` records and before any `.test` record. Likewise, `.teardown` records can occur after all tests.

Note: `.setup` records can be specified using the short-hand `.su`. Likewise, `.teardown` records can use the short-hand `.td`.

Compiling and Running the Generated Tests

The generated Cutter file is a complete C program. When you compile it, the generated executable will run all the specified tests and report the results to the console, as they execute.

To compile the Cutter-generated tests, use the same switches as you compile the program source code. If errors are encountered, they generally arise from problems where you have specified invalid code in:

- 1) the `.global` record
- 2) the `.expect` record
- 3) the `.params` record (the most common)

These three record types copy code you provide directly to the generated file, so they are the most common sources of error. Cutter's own generated code has been tested thoroughly and compiles cleanly. So, if you discover an error in emitted code unrelated to your own specified code, please let us know at once—including sample code to reproduce the problem.

Linking

If your code tests a library or similar functions—not a part of `main()`—you should link to those modules and then run the tests. Because you will probably want to run Cutter frequently, it should be included in your build command scripts or have its own batch/command file for your project.

If you are testing functions in a file that contains an instance of `main()`, you will encounter a problem because the linker will find two instances of `main()`—one in your program and one in the test file. The way to solve this is to move `main()` from your program into its file and put all the other functions in their own source file. Here is how Cutter's own source code does this¹:

```
// calls cutter_main(). Done this way so that cutter_main can be
// unit tested.

extern int cutter_main( int argc, char *argv[] );

int main( int argc, char *argv[] )
{
    return cutter_main( argc, argv );
}
```

Then link only to the old file (the one in which `main()` originally appeared, but from which it has been removed). This should allow the link to occur without any further problems.

¹ Except for licensing verbiage at the beginning of the file, this is the entire contents of `main.c`

Sample Cutter File (used for testing Cutter)

```
; sample Cutter file for the full Cutter program (v.0.55)
; v. 1.2
; 2006-03-10

.start
.global
    #include "c:\dev\cutter\src\portabletype.h"
    #include "c:\dev\cutter\src\cutter.h"
    char* test_args[] = { "cutter", "sample.cut" };
    struct line_list list;

;-----
;                               tests for parser.c
;-----

.function ui32 get_linked_list_count( void );
    .setup
        init_line_list();
    .test
        .expect = 0
        .params
        .teardown
            list.count = 0;

.function i32 add_line_to_list_tail( char* line_data, i32 count,
                                    i32 type );
    .test
        .expect = OK
        .params "Hello Dolly!", 1, FUNCTION
    .test
        .expect = ERR
        .params NULL, 2, GLOBAL

.function ui32 get_linked_list_count( void )
    .redout pzRedoutFullCutter.txt
    .test
        ;one illegal addition + one legal, so final count s/be = 1
        .expect = 1
        .params

    .test
        .su
            dump_linked_list();
        ; make sure that dumping the linked list does not
        ; change the count
        .expect = 1
        .params

;-----
;                               tests for generate.c
```

```

;-----
.function i32 generate( FILE *outfile, char* infile_name )
.test
    .expect ERR
    .params stdout, NULL

    .test
    .expect ERR
    .params NULL, "sample.cut"

.function i32 output_prolog( FILE *fout )
.test
    .expect ERR
    .params NULL

    .test
    ; needs a test record to avoid an error.
    .su
        add_line_to_list_tail( ".test", 3, TEST );
    .expect ERR
    .params stdout

;-----
;
;           tests for cutter.c
;-----

.function int cutter_main( int argc, char *argv[] )
    .rederr pzRederrFullCutter.txt
    .test
        .expect EXIT_FAILURE
        .params 1, NULL

    .test
    ; pass the name of an existing file
    .expect != EXIT_FAILURE
    .params 2, test_args

    .test
    ; pass the name of a non-existent file
    .su
        test_args[1] = "froodnoggle.cut";
    .expect EXIT_FAILURE
    .params 2, test_args

.function i32 handle_redirect ( char* rec )
    .test
    ; pass invalid record (s/b .rederr or .redout only)
    .expect ERR
    .params ".redear"

    .test
    ; pass correct .rederr/.redout keyword but leave off filename
    .expect ERR
    .params ".rederr\n"

```

```

.test
    ; test valid record with filename containing spaces
    .expect OK
    .params ".rederr \"this is a test\""

.test
    ; test valid record with filename containing spaces
    .expect OK
    .params ".rederr copacabana"

.test
    ; double check this works for .redout as well
    .expect OK
    .params ".redout \"this is a test\""

.function char *get_func_invocation_name( char *func_decl );
.test
    .expect == NULL
    .params "int func("

.test
    .expect = NULL
    .params "int func())"

.test
    .expect NULL
    .params "()"

.stop

```

(Note: A variant of this file is included with the Cutter source code.)

Cutter Command Reference

The following reference presents the records in alphabetic order and is the definitive explanation of the syntax and operations of the commands. All Cutter commands begin with a period and are lower-case. Cutter is case-sensitive. Leading spaces and tabs are ignored, so indentation can be done using whatever style is preferred.

.expect

Specifies the expected return value that the test expects in order to pass. The relationship of the actual return value to this expected value can include logical and comparative operators (including all those permitted in C: `<`, `>`, `<=`, `>=`, `!`, `!=`, and `==`). Cutter uses can omit the logical relationship, in which case Cutter assumes it is `==`. Cutter also allows the shorthand of `=` to show `==`. So, the following three `.expect` records are all equivalent:

```
.expect NULL
.expect = NULL
.expect == NULL
```

With the exception of the modification of the `=` sign (as shown immediately above), Cutter simply copies the `expect` statement directly into the generated code. Hence, if you have `#included` a header file that includes manifest constants and wish to refer to those, you can do so. (In fact, `NULL` in the above example is an instance of this.) Nothing other than the expected return value should appear in this record.

.global

This record indicates the beginning of a series of one or more lines containing C code that should be placed as is into the generated program, prior to the definition of the first function. Typically, this code consists of `#include` directives and declarations of global variables.

The `.global` command record should contain nothing other than the `.global` keyword. All subsequent lines until the next Cutter record is encountered are copied verbatim to the generated file.

The `.global` record must appear before the first `.function` record. Listing 3 in this document gives an example of the `.global` record in use.

.params

This record specifies the parameters to pass to the function being tested. Cutter simply copies everything after the `.params` keyword and places between the parentheses of the function call. Listing 3 in this document provides several examples.

To call a function with no parameters, simply leave the rest of the `.params` record blank.

.rederr

In the code generated by Cutter, `stderr` (the standard error stream) is redirected to a null device. The result is messages written by the tests to `stderr` do not appear anywhere. This is frequently the desired behavior when testing. However, should you want to capture these messages, you can use this record to redirect `stderr` to a disk file. To do so, follow the `.rederr` keyword with a filename. If the filename contains embedded blanks, place it between quotation marks. Once `stderr` has been redirected to disk, it cannot be redirected again to a null device.

The `.rederr` record should occur immediately after a `.function` or `.redout` record. Any other placement generates an error.

.redout

In the code generated by Cutter, `stdout` (the standard out stream or the console) is redirected to a null device. The result is messages written by the tests to `stdout` do not appear anywhere. This is frequently the desired behavior when testing. However, should you want to capture these messages, you can use this record to redirect `stdout` to a disk file. To do so, follow the `.redout` keyword with a filename. If the filename contains embedded blanks, place it between quotation marks. Once `stdout` has been redirected to disk, it cannot be redirected again to a null device.

The `.redout` record should occur immediately after a `.function` or `.rederr` record. Any other placement generates an error.

.setup

This record enables the specification of code that must be executed before a test or before a series of tests. The lines between the `.setup` record and the next Cutter dot-record should be the required C setup code. It is passed through as is to the generated C file. This option enables the creation of mock data items as well. `.setup` records generally are placed after a `.test` record, but can be placed after `.function` records as well (and in both locations if needed.) Typically, the setup code is cleaned up after a test via a `.teardown` record, but this step is not required.

Note `.su` is a synonym for `.setup` and can be used anywhere in a Cutter script that `.setup` is legal.

.start

This record marks the beginning of the Cutter file that is used by Cutter to generate test code. Anything that precedes the `.start` record is ignored. The `.start` command record should not contain anything other than the `.start` command

Only comments and blank lines should precede a `.start` record. Do not rely on future versions of Cutter ignoring material before `.start`, so adhere to the comment convention of starting comment lines with a semi-colon (`;`)

.stop

This record tells Cutter that the last line in the Cutter file for generating code has been reached. Cutter ignores everything after the `.stop` record. The `.stop` command record should not contain anything other than the `.stop` command.

Only comments and blank lines should follow a `.stop` record. Do not rely on future versions of Cutter ignoring material after `.stop`, so adhere to the comment convention of starting comment lines with a semi-colon (`;`)

.su

A synonym for the `.setup` record. The terms can be used interchangeably in a Cutter script.

.td

A synonym for the `.teardown` record. The terms can be used interchangeably in a Cutter script.

.teardown

This record enables the specification of code that needs to be executed after a test or before a series of tests—typically cleaning up the results of the test and undoing any side-effects of the test code. The lines between the `.teardown` record and the next Cutter dot-record should be this clean-up C code. This code is passed through as is to the generated C file. `.teardown` records generally are placed after a `.params` record. They are never required by Cutter.

Note `.td` is a synonym for `.teardown` and can be used anywhere in a Cutter script that `.teardown` is legal.

.test

The `.test` record identifies the beginning of a test. It is subordinate to the last preceding `.function` record and identifies a test of that function. The keyword `.test` is the only word allowed in a `.test` record.