# APPENDIX 1

# JCUT Tutorial

## 1.1 JCUT Introduction

This section presents several subsections which explain what *JCUT* is, its main objectives, a high level overview on how it works, and some of the most outstanding features.

### 1.1.1 What is it?

*JCUT* is a simple command line tool to perform unit testing for the C programming language. This tool uses a compiler framework: LLVM and Clang application programming interfaces (APIs). This means that some interesting features implemented by this tool are parsing, compiling and executing of specific functions in C code at runtime. *JCUT* stands for Just in time C Unit Testing.

### 1.1.2 Objectives

The main objectives behind the design and implementation of this tool are:

- Become a useful resource for educational purposes for C novice programmers and universities.
- Minimize the amount of test code you need to write for your software written in C.
- Simplify the test code you write. Writing a test for a function should be as simple as if you were calling that function with its parameters somewhere else in your program.

- Minimize the compilation times for your tests and in turn for your program. The *JCUT* tool will actually compile only the functions you are testing, leaving the rest untouched.

- Encourage Test Driven Development (TDD) and unit testing among C programmers.

### 1.1.3 How does it work?

*JCUT* operates in mainly in two modes:

1. **Interpreter mode.** Enables a basic command prompt in which a user can run functions defined in a C source file using the *JCUT language*[1]. See section 1.2.8 on how to use the interpreter mode.

2. **Batch mode.** The same lines a user would write in interpreter mode can be all be written in a test file so *JCUT* can process all the tests.

Regardless of the mode the user chooses *JCUT* takes as input two important things to operate: C source code and *test definitions* using the *JCUT language*. It is worth mentioning some points for each of these two inputs.

1. *C source code*: *JCUT* makes use of the Clang APIs. Clang is a front-end for the C language family (C, C++, Objective-C and Objective-C++) which makes use of the LLVM compiler. This means that *JCUT* will analyze and process the C source code a user gives it pretty much like a compiler reporting any error if it exists. If the C source code given to *JCUT* uses a 3rd party library see section 2.3.

2. *Test definition*: A test definition is nothing but a function call passing it the arguments as any other function call in C. This can be entered either through the interpreter or in a test file.

   A test file is a plain text file with a *jcl* extension. In order for this tool to understand what the user wants to test and expects from a test the user needs to use the *JCUT*

[1] This language is in fact a simple, easy to use and learn language. If the user is familiar with the C programming language syntax then they have already learned most of the *JCUT language*.

*Language*. This language syntax and features will be introduced throughout this tutorial.

Everything written in a test file can be entered in the interpreter and viceversa.

Upon receiving these two inputs the tool will analyze the C code using Clang APIs and will generate code for only those functions defined in the test file or interpreter, and then run them using the LLVM JIT Execution Engine. Then a small report will be provided by the tool about the tests ran.

In section 2.5 the user can read what are the current limitations of this tool.

### 1.1.4 JCUT output

In this section it is explained how the *JCUT* output looks like and what it means.

```
GROUP NAME | TEST NAME | FUNCTION CALLED | RESULT | ACTUAL RESULT | EXPECTED RESULT |
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 xxxxxxxxx | xxxxxxxxx | xxxxxxxxxxxxxxx | xxxxxx | xxxxxxxxxxxxx | xxxxxxxxxxxxxxx |
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

TEST SUMMARY
Tests ran: 1
Tests PASSED: 1
Tests FAILED: 0
```

***Figure 1.1:*** JCUT output

The columns shown are explained from left to right:

**GROUP NAME** By default a group name is assigned to each test. Section 1.2.3 explains how to create groups of tests.

**TEST NAME** This name generated automatically by the tool based on the function under test.

**FUNCTION CALLED** This is the function under test and it will show the parameters used.

**RESULT** Whether the test has failed or passed.

**ACTUAL RESULT** If the function under test results a value it will be shown here whatever data type, otherwise it will show just void.

**EXPECTED RESULT** If any expected result was specified for a function this field will show what was the expected result. Section 1.2.1 shows how to specify an expected result.

At the very end you will see a small summary on the number of tests ran, passed and failed.

### 1.1.5 Determining whether a test passed or failed.

*JCUT* will check the return value of every function called and compare it against an expected value 1.2.1 to determine if the function passed or failed. However this method does not work for `void` functions. In such cases *JCUT* will assume a `void` function has *passed* when has completed its execution. If for some reason the `void` function fails to complete execution *JCUT* will mark it as *failed*.

### 1.1.6 Sandbox environment for executing tests

By default *JCUT* executes each of the tests in a separate process. The reason for this is to provide an isolated address space for each test so the user is free to play around with pointers. This means that if a function under tests dereferences a `NULL` pointer or access an invalid memory address the test itself will crash, but *JCUT* will detect this condition and report it to the user. This becomes handy when writing a new function and executing a huge amount of tests that need to be executed in an isolated environment.

This feature is implemented by `fork()`ing[2] the *JCUT* process to execute a single test, then the results are reported to the parent process and repeat these steps for each test. Using the system call `fork` adds an overhead to the execution of a test. Running inside the interpreter this overhead may not be noticeable, but when running a big amount of tests with a *test file* a difference can be noticed.

[2] `http://linux.die.net/man/2/fork`

The user can disable this feature by passing the *JCUT* option `-no-fork`. This will cause the test to be run in the same process as *JCUT*, thus avoiding the overhead of creating a new process. But if the test crashes in an unexpected way that will make *JCUT* crash. Section 2.2 explains more about passing options to *JCUT*.

## 1.2 JCUT Features

In this tutorial presents how can *JCUT* be used to test some C code. The way this tutorial presents the features is by introducing basic C functions on each section and then then some tests are introduced to test such functions.

For the sake of simplicity all the C code written will be located in a source file named *cfile.c*. The user has two options to run the tests: use the interpreter mode 1.2.8 or using a test file in batch mode. Using either is fairly simple. When using interpreter mode the user has to enter the following command line:

```
1   jcut cfile.c --
```

If the user wants to use a test file in batch mode a text file has to be created. This tutorial assumes that file is named *test.jcl*. The command line to run the tests in batch mode is:

```
1   jcut cfile.c -t test.jcl --
```

This tutorial will make use of the test file with batch mode.In this way the user will be able to have all tests written in a test file by the end of this tutorial and use it as a referenc.

NOTE: For the examples in which some code makes use of the C standard library (stdio.h, stdlib.h, etc.) the user will need to have properly installed such libraries and tell *JCUT* where they live. For windows users a small folder with the standard C standard library will be provided and the user can run the tool on that same folder.

Since *JCUT* is a command line tool for this tutorial the user will need two things open at all times:

- Command line window (also known as shell or terminal in the linux world). The user has to change to the directory where the folder jcut-tutorial is placed.
- Text editor (emacs, vi, notepad++, etc.)

The following sections describe all the features *JCUT* provides, and all the examples used are executed using a *test file*. If using the test file is too cumbersome the reader can use the *JCUT* interpreter, which is described in section 1.2.8.

### 1.2.1 The basics

#### The simplest test

In the text editor create a file named *cfile.c* then write the following function in it.

```
1  /** cfile.c **/
2  void my_function() {
3    return;
4  }
```

Then create the test file *test.jcl* and type the function under test.

```
1  # test.jcl
2  my_function();
```

Note the semi-colon ';' at the end of the function, follows the same syntax as C for calling a function. Also note that the user can place comments in your test file starting with the character #. Open a command line, shell or terminal and then to run the above test by typing the following command line:

```
jcut cfile.c -t test.jcl --
```

Even though this command is simple it is worth giving a more detailed explanation of what is happening.

**jcut** This is the *JCUT* binary. It might be a relative or absolute location depending on where you are located in your file system.

**cfile.c** This your C source code being under test.

**-t** This is an argument which tells jcut the following argument is the test file containing all the test definitions.

**test.jcl** This is the actual test file.

**- -** These two dashes at the end of the command line are mandatory. Why? *JCUT* makes use of clang APIs, this makes it a clang tool and as such it can take a compilation database in a .json file. The two dashes tells clang APIs such .json file is not available and that the compilation database should be built from the command line arguments after the two dashes `--`. For the sake of simplicity this tutorial won't talk about compilation databases because they aren't really needed. Just remember to provide the two dashes for *JCUT* to work happily.

**NOTE**: If the C file provided `#includes` a header file, the user may want to provide the path where that header file is located with the command line `-I path/to/headers`. See section 2.3 for more information about it.

After running this command the reader will see some output reporting the results of executing the function defined in the test file. Section 1.4 describes what the output means.

Running a test with *JCUT* is as simple as typing the function in the test file as if one was calling a function in C code. What the tool will do is taking the given test function `my_function()`, it will search for its definition in the input C files, compile it and execute it using the Clang APIs and the LLVM JIT Execution engine. Since this is a void function the tool assumes the test has passed if it completed its execution without interruption.

### Passing arguments to a functions

Testing a function that takes some parameters and returns a value is pretty ease. Type the following function in the file *cfile.c*:

```
1  int sum(int a, int b) {
2    return a+b;
3  }
```

Then write down a test for it which simply calls it with any parameter you want in *test.jcl*.

```
1       sum(2,2);
```

Execute the tests with the command

```
jcut cfile.c -t test.jcl --
```

If the function has no syntax errors *JCUT* will run it and report the status of it as passed.

`Exercise` Write a function which performs a multiplication between two integers and test it with different arguments in your test file.

### Expected results

A function is one of the most important elements in which a program is distributed and the execution of any program depends on the return values of such functions.

*JCUT* allows comparing the actual result of a given function with an expected result with the following syntax:

```
1       sum(2,2) == 4;
```

This syntax tells the tool that the return value from calling the function `sum()` should be compared with the value **4**. Execute the test and see what reports tells.

```
jcut cfile.c -t test.jcl --
```

JCUT supports the following operators for comparing the return value of a function with any value you give it.

```
1       sum(2,2) != 4;
2       sum(2,2) == 4;
3       sum(2,2) >= 4;
4       sum(2,2) <= 4;
5       sum(2,2) > 4;
6       sum(2,2) < 4;
```

The comparison operators behave and follow the same rules from a C program.

`Exercise` Write several tests for the `sum()` function that make use of these operators.

### What values can be used as expected result?

As of now the values that can be used as expected result are:

- Constant integers (only decimal notation is supported).
- Floating point constants.
- Constant characters denoted with the syntax 'X' where X is any valid ASCII character.
- C constant strings: "this is a string". Remember that when taking a string constant for any comparison you are taking that string address. *JCUT* will do the same thing.

How *JCUT* will treat each value is shown the following code snippets with comments.

```
1   # It will take the ASCII value of 'a' (97)
2   sum(2,2) == 'a';
3
4   # If sum returns an unsigned value,1
5   # -47 will be treated as unsigned value
6   sum(2,2) == -47;
7
8   # It will take the memory address of the
9   # string and casting it to whatever data
10  # type the function returns
11  sum(2,2) == "fortunate";
```

In all cases the tool will cast the expected value to whatever data type the function under test returns and then perform the comparison. The expected value is ignored for functions that return `'void'`.

### Summary

The syntax for testing a function in *JCUT* is summarized this way:

*test-function* :

*function-call expected-result*$_{opt}$ ;

The *opt* subscript indicates that *expected-result* is an optional part after doing a*function-call*.

To wrap up and use all the things learned in this section try writing a function that calculates the factorial of a number using both a recursive and iterative solution.

The following functions can be used as a reference for writing some tests.

```
1   int fact_1(int a) {
2       if(1 == a) return a;
3           return a * fact_1(a-1);
4   }
5
6   int fact_2(int a) {
7       int i = 1;
8       while(a) i *= a--;
9       return i;
10  }
```

Try writing some tests for these two functions in the test file. Also try writing your own versions of this function using recursion and test it using *JCUT*.

### 1.2.2 Managing program state

A common concept across many unit testing frameworks is that of *test fixture*, which is a piece of code which is often repeated in a test. This *test fixture* allows a test to perform some *setup* and *cleanup*, also known as *teardown*, in order to set the program state with the right configuration for a test to be run. This setup and teardown are executed *before* and *after* the function under test respectively.

Any non-trivial program will contain its own state represented as a set of data structures which are modified by a series of function calls. In C programming, the state of a program is often times represented either by global variables of primitive data types, global `structs` or a series of function calls which in turn access the variables handling the program state.

Sometimes a given function behavior depends not only on the parameter values it receives but also on the current value of one or several global variables or structures, or even calling other functions before in a specific order.

*JCUT* lets one handle such situations by using *test-fixtures* during *setup* and *teardown* of a test. This is achieved with two keywords: `before` and `after`.

The syntax for these two keywords is as follow:

*test-definition* :

*test-setup*$_{opt}$ *test-function* *test-teardown*$_{opt}$

*test-setup* :

`before {` *test-fixture* `}`

*test-function* :

*function-call* *expected-result*$_{opt}$ `;`

*test-teardown* :

`after {` *test-fixture* `}`

*test-fixture* :

*function-call* `;` *test-fixture*$_{opt}$

*variable-assignment* `;` *test-fixture*$_{opt}$

*expected-expression* `;` *test-fixture*$_{opt}$

Both the *test-setup* and *test-teardown* are optional for a function under test. A *test-setup* is defined by using the `before` keyword before the function under test followed by some curly braces and a *test-teardown* is defined by using the `after` keyword after the function under test followed by some curly braces. A *test-fixture* can be formed of one or more *function-call*, global *variable-assignment* or an *expected-expression*.

The important thing to note here is that the user can specify three types of statements inside the curly braces that will be executed before and after the function under test respectively in the order they appear. The 3 different of statements inside the curly braces are described with the following examples.

### Modifying global variables

Write the following code in the source file.

```
1  int state; /* Global variable */
2
3  int get_state() {
4      return state;
5  }
```

Write the following test in your test file.

```
1               get_state() == 0;
```

Run this test through the command line.

```
1               jcut cfile.c -t test.jcl --
```

Remember that the C standard specificies that global variables will be initialized to 0, thus our test succeeds and *JCUT* will report the actual result.

Now try writing the following test:

```
1               before { state = 10; } # Assignment operator =
2               get_state() == 0; # Comparison operator ==
```

Try running this test and you will see that it fails.

The reason why it fails is because *JCUT* modified the variable state and assigned a 10 to it right before executing the function `get_state()`. Thus the actual result from calling `get_state()` is 10. You can have as many variable assignment as you want in both the before and after statements.

### Checking global variables values

Imagine the `get_state()` function has a return type of `void` and there was no way to check which value the variable state has after calling the function under test. This problem can be solved by making a comparison using the after keyword.

```
1               before { state = 10; } # Assignment operator =
2               get_state(); # Imagine this function returns 'void'
3               after { state == 10; } # Comparison operator == 10
```

This test will succeed as *JCUT* will check that the variable state has a value of 10 after running the function under test. Note that when there is no expected value specified after function under test the return value is ignored regardless of its data type.

Sometimes you may want to check for a specific value of a given variable in order to run a test. For instance:

```
1               before { state == 0; }
2               get_state();
```

Is a valid test which checks the value of a variable before running a test. *JCUT* will notify you when a comparison in either a `before` or `after` statement fails, indicating which one and the test itself will fail. This is because determining if a test passed or failed depends on a state `before` and `after` after the execution of the function under test.

You can have as many comparisons as you want in both the before and after statements.

**Determining the result of a test with several comparisons.**

Using any comparison operator in the after or before statements will make *JCUT* use them along with the function return value to determine whether the test passed or failed.

This means that the result of a function is determined with the following conditions.

1. The function under test finishes its execution.
2. If a return value is expected it will be checked and has to be equals to the one indicated in the test file.
3. If several comparisons are provided they all will have to hold true (an AND operation is performed).

These three conditions have to hold true in order for a test to pass. If either fails the test will be reported as failed.

**What comparison operators are supported for making comparisons in the before and after statements?**

The same operators for comparisons used in section 2.1.3 can be used in the `before` and `after`. For instance:

```
1
2               before { state != 0; }
                get_state();
3
4               before { state >= 4; }
5               get_state();
6
7               get_state();
8               after { state <= 1; }
9
```

```
10              get_state();
11              after { state > 0; }
12
13              get_state();
14              after { state < 10; }
```

All these examples have valid comparison operators and are valid tests.

### Calling functions before and after a function under test

Now add the following function to your C source file some_file.c.

```
1               void modify_state() {
2                   state += 5;
3               }
```

Now try modifying the variable `state` by calling a function rather than doing an assignment. Here is the new test:

```
1                # Function call, state == 5
2               before { modify_state(); }
3               get_state() == 5; # Passes
4               # You can call a function call here
5               after { modify_state(); }
```

This test will succeed because the function `modify_state()` is called before the `get_state()` function modifying the global variable. In the after statement we also call the function that modifies the global variable. Calling the function `modify_state()` in the after statement for this toy example is useless, but you can think of calling a function that cleans up all the things done in the before statement or during the function under test.

Calling a function in the before or after statements can be done when:

- The function under test depends on other functions to be called either before or after it.
- When performing an elaborated setup for a function, i.e. opening a socket, allocate resources, etc.
- When performing a cleanup operation, usually the opposite from what a setup function would do, i.e. closing a socket, free resources, etc.

NOTE: There is one important thing to note here. *JCUT* will backup any global variable you modify in the after or before statements using the assignment operator = and then restore their original value right after the current test and before starting executing the next test. Thus leaving the program state in its original value before starting execution of the current. However, *JCUT* will not detect when a function called in the after or before statements has modified a global variable, thus the modified variables will remain with this new value for the next test that is run. If you call any setup function in the before statement, make sure you call a cleanup function in the after statement to leave the program state in an optimal state in which other tests can be run.

### Summary

This section explained how one can use the `before` and `after` keywords to execute three different types of statements:

***variable-assignment*** of the type `variable = X` where `X` is any value as described in section 1.2.2 and `variable` is a global variable of any type. For pointer types see section 1.2.5. For struct types see section 1.2.4.

***function-call*** like a regular C function call.

***expected-expression*** is a variable comparison with any value as described in section 1.2.1.

The syntax for the `before` and `after` keywords is as follows:

*test-definition* :

*test-setup*$_{opt}$ *test-function* *test-teardown*$_{opt}$

*test-setup* :

`before {` *test-fixture* `}`

*test-teardown* :

`after {` *test-fixture* `}`

*test-fixture* :

*function-call* `;` *test-fixture*$_{opt}$

*variable-assignment* ; *test-fixture*$_{opt}$

*expected-expression* ; *test-fixture*$_{opt}$

### 1.2.3 Grouping related tests

Any non-trivial program will have its set of tests, and often times the amount such tests will be big. A common practice to maintain a set of tests in the long term is to group the tests into related sets of tests that comply with a certain criteria.

*JCUT* lets you group any amount of tests by using the keyword group. The syntax is as follow:

```
1 group [Name] {
2 ...
3 }
```

You can optionally provide a name for a given group.

The reason to have groups is mainly to 2 things:

1. Provide a program state for a set of tests in order to avoid having redundant before and after statements for the same set of tests.

2. Provide a way to filter and customize the reports for each group of tests.

#### The default group

Taking all the examples we've done in the previous section you should have a test.jcl file that looks somewhat like this:

```
1 # test.jcl
2 my_function();
3 sum(2,2);
4 sum(2,2) == 4;
5 sum(2,2) != 5;
6 fact_1(3) == 6;
7 fact_2(3) == 6;
8 get_state() == 0;
```

By default *JCUT* adds all the tests in a test file into a default group which is named group_0. You will see in the test reports a 'group_0' for every test you run.

### Nesting groups

Let us take some of the tests from the test.jcl and group them into nested groups.

```
# test.jcl
my_function();
group A {

    sum(2,2);

    group B {
        sum(2,2) == 4;

        group C {
            sum(2,2) != 5;
        }
    }
}

group D {
    fact_1(3) == 6;
    fact_2(3) == 6;
}
get_state() == 0;
```

Inside a group you can have any number of tests or even any number of nested groups. There is no limit in how many groups can be nested. The tests and groups found within another group will be executed in the same order they appear in your test file.

### Managing program states for a given group

Recall section 1.2.3 in which the user learned how to modify the program state for a given test with the keywords `before` and `after`. The user can also modify and check the program state for a group of tests using the `before_all` and `after_all` keywords. This is their syntax:

```
group [Name] {
    [before_all { ... }]
    ...
    [after_all { ... }]
}
```

These two keywords will modify the program state for the current group they belong to. Note that both statements are optional. The only restriction if they are used is that

the `before_all` statement has to be defined at the beginning of the group before all the tests and the `after_all` statement has to be defined at the end of the group after all the tests in that group.

For instance the following examples are valid tests.

```
1  group E {
2      # Executed before all the tests in this group
3      before_all { state = 10; }
4      get_state() == 10; # test 1
5
6      # jcut does not track what's changed inside this function call
7      before { modify_state(); }
8      get_state() == 15; #test 2
9
10     modify_state(); # test 3
11
12     # Executed after all the tests in this group
13     after_all { state == 20; }
14 }
15
16 group F {
17     before_all { state = 5; }
18
19     get_state() == 5;
20
21     before { modify_state(); }
22     get_state() == 10;
23 }
24
25 group G {
26     get_state() == 0;
27     after_all { modify_state(); }
28 }
```

All these are valid tests and they all pass.

`Exercise` Try writing these tests in the *test.jcl* file. If the user has any doubt regarding the results of these tests and when the global variables are modified please refer to the note at the end of section 1.2.2.

### Summary

In this section the user learned how to group tests by using the keyword `group`. The user also learned how to modify the program state for a specific group by using the keywords

`before_all` and `after_all`. The final syntax for the group keyword is as follows:

*test-group* :

*group-setup*$_{opt}$ *test-definition* *group-teardown*$_{opt}$ *test-group*$_{opt}$

`group` *identifier*$_{opt}$ `{` *test-group* `}`

*group-setup* :

`before_all {` *test-fixture* `}`

*group-teardown* :

`after_all {` *test-fixture* `}`

*test-fixture* :

*function-call* `;` *test-fixture*$_{opt}$

*variable-assignment* `;` *test-fixture*$_{opt}$

*expected-expression* `;` *test-fixture*$_{opt}$

Both the `before_all` and `after_all` statements are optional for a given group, and they can contain any number of variable assignments, function calls or expected expressions that do variable comparisons with an expected value. Note the order in which you can define the `before_all` and `after_all` statements, they need to be defined at the beginning and end of the group respectively. A group can contain any number of tests. A group can also have an optional name. If you omit this name the tool will assign this group an automatically generated group name.

### 1.2.4 Structures as global variables

When using a structure as a global variable to keep track of a program state the user can assign any value to it just like any other global variable. The only difference is that the user has to use C89 `struct` initialization list syntax to initialize the values of a `struct`. Here is an example of such syntax.

```
1  /* Example C code */
2  struct Pair {
3      int a;
4      int b;
5  };
6  ...
7  struct Pair my_pair = { 10, 20 }; // a==10, b==20
```

The important thing here is that you have to provide a valid initialization value for the variable types in the same order as they are declared. If you omit an initialization value it has to be for the variables declared at the end of the `struct`. Any variable which is not given an initiazilation value using this syntax will be initialized to 0. For instance:

```
1     struct Pair my_pair = {10}; // a==10, b==0
```

### Struct initialization list

The following function prints the contents of a global `struct`. See section 3.2 on how to use the C standard library. Open the file *cfile.c* and type the following.

```
1   /** some_file.c **/
2   #include <stdio.h>
3   struct Pair {
4     int a;
5     int b;
6   };
7
8   struct Pair global_pair;
9
10  void print_global_pair(){
11    printf("global_pair.a = %d\n", global_pair.a);
12    printf("global_pair.b = %d\n", global_pair.b);
13  }
```

Now write the following tests in the test file *test.jcl*.

```
1     # test 1
2     print_global_pair();
3
4     # test 2
5     before {global_pair = {1}; }
6     print_global_pair();
7
8     # test 3
9     before {global_pair = {1,2}; }
10    print_global_pair();
```

These tests show the basics of C struct initialization list syntax in jcut. The first test will print the contents of the variable `struct global_pair`. On the following tests the contents of the `struct global_pair` are modified one element at a time to appreciate

how the `struct` initialization works. Note that you cannot assign values starting from the last element in the `struct`, it has to start from the first element.

#### Pointers to structs

The initialization of structs through a pointer is pretty much the same as in the previous section except for a little difference. Before explaining such difference is important to explain how *JCUT* works with pointers. That will be explained in the following section.

#### Summary

In this section we learned how you can initialize structs with the struct initialization list syntax. We have to initialize the struct fields in the same order they were declared. If we omit a struct field has to be one that was las declared in the struct. We can also initialize structs that contain more structs.

### 1.2.5 Pointers

An avid C programmer already knows the importance of pointers and how many times the design of functions revolves around the usage of pointers. Many functions that take a pointer to any data type assume that the memory for that data type was already allocated somewhere else in the code. This poses a problem for testing functions that make use of pointers. *JCUT* solves this problem by providing a simple syntax to allocate memory for a given function.

#### Automatic memory allocation

Whenever there is a function that takes a pointer or a global variable of any pointer type, one can instruct *JCUT* to allocate memory for it by using the following syntax ni a test file or *JCUT* interpreter:

```
pointer_variable = [n]
```

Using the squared brackets and putting inside a non-negative integer value will make *JCUT* to allocate the necessary bytes for the data type the pointer points to. That means

it allocates `n*sizeof(<data type>)` bytes.

The following example illustrates such feature. Write the following C code in the file *cfile.c*.

```
1   #include <stdio.h>
2
3   int * g_ptr = NULL;
4
5   void print_g_ptr(){
6       if(g_ptr)
7           printf("address: %p, contents: %d\n",g_ptr,*g_ptr);
8       else
9           printf("ERROR g_ptr is NULL pointer!\n");
10  }
```

Then write a simple test in the test file *test.jcl*.

```
print_g_ptr();
```

After running this test the user should see that the test printed the error stating `g_ptr` is a null pointer. In order to have a working test the user can tell *JCUT* to allocate memory for the data type pointed to by the pointer. That can be done by writing a new test:

```
1   before { g_ptr = [1]; }
2   print_g_ptr();
```

*JCUT* will allocate enough memory for 1 integer because the data type `g_ptr` points to is an integer. If the pointer was a pointer to a char, short integer, or even a `struct` type, *JCUT* will allocate enough memory for that data type. This means that one could allocate enough memory for an array of integers by writing the following test.

```
1   before { g_ptr = [1024]; }
2   print_g_ptr();
```

Of course the function `print_g_ptr()` only accesses the first element in the array, but still *JCUT* allocates enough memory to fit 1024 integers. The user doesn't have to worry about freeing the memory allocated, *JCUT* will do that automatically after running this test. If the user wants the memory allocated to last for the execution of several tests the expression `g_ptr = [n]` should be put in a `before_all` statement. See section 1.2.3 for more information on groups.

### Memory initialization

*JCUT* provides an easy way to initialize all the memory allocated for a pointer by using the following syntax:

```
pointer_variable = [n:x]
```

Adding a colon and then passing an non-negative integer value `x` to indicate an initialization value to be stored in every byte of memory allocated for the given pointer. Rewrite the test from previous exercise to initialize the memory:

```
1  before { g_ptr = [1:5]; }
2  print_g_ptr();
```

After running the test the contents of the memory allocated is actually a 5.

`Excercise` To wrap up this exercise write down the following C function in the file *cfile.c*:

```
1  // Print the contents of the array pointed to by buf
2  void print_buffer(unsigned char *buf, unsigned size) {
3      short step = 1;
4      unsigned char *i = buf;
5      unsigned char *end = buf + size;
6      while(i < end) {
7          printf("%x ", *i);
8          if(step%10 == 0)
9              printf("\n");
10         ++step;
11         ++i;
12     }
13     printf("\n");
14 }
```

And write the following test in the file *test.jcl* and then run it:

```
1  print_buffer([20:6], 20);
```

The function `print_buffer` will print `size` bytes of the array pointed to by `buf`. This function was tested with *JCUT* by allocating 20 elements of type unsigned char and passing that pointer to the function. Then passing the size of the array which is of 20 unsigned chars (remember the size of an unsigned char is 1 byte). Running that function will print on screen the contents of the array which is 6.

If this function received a pointer to an integer `int *` *JCUT* would have allocated memory for 20 integers, that is 80 bytes.

### Pointers to structures

When using a global variable which is a pointer to a `struct` or a function that receives a pointer to a `struct` and the user wants to initialize the memory allocated for that pointer, both the section 1.2.4 still applies with a slight difference. The only thing needed to change is the initialization value to a `struct` initialization list. The following example demonstrates this. Write this function in the file *cfile.c*.

```
1  /** cfile.c **/
2  struct Pair {
3      int a;
4      int b;
5  };
6
7  struct NestedPair {
8      int a;
9      int b;
10     struct Pair nested;
11 };
12
13 void print_ptr_pair(struct NestedPair* ptr) {
14     if(ptr) {
15         printf("ptr.a = %d\n", ptr->a);
16         printf("ptr.b = %d\n", ptr->b);
17         printf("ptr.subpair.a = %d\n", ptr->nested.a);
18         printf("ptr.subpair.b = %d\n", ptr->nested.b);
19     } else
20         printf("ERROR ptr is NULL pointer!\n");
21 }
```

Write the following test:

```
1      print_ptr_pair([1:{1,2,{3,4}}]);
```

One can see that instead of providing an integer value to initialize the memory allocated, we provide a struct initializer as described in section 1.2.4. Providing an integer value rather the `struct` initialization list works too, but the difference is that every byte in the memory allocated will be initialized to that value.

*JCUT* currently **does not** support initializing a struct which is passed by value. For instance imagin the `print_ptr_pair()` function is declared as:

```
1  /* It's passed by value, no pointer used. */
2  void print_ptr_pair(struct NestedPair ptr);
```

The following tests file are invalid in a test file:

```
1  # Error when struct is passed by value.
2  print_ptr_pair([1:{1,2,{3,4}}]);
3
4  # Error when struct is passed by value.
5  print_ptr_pair({1,2,{3,4}});
```

The syntax presented in this section works only for functions taking a `struct` passed by a pointer or any global `struct` statically allocated or a global pointer to a `struct`.

#### Summary

This section presented how *JCUT* allocates enough memory for any pointer with the syntax `[n:x]` which makes it allocate `n*sizeof(<data type>)` bytes and initialize each byte of that allocated memory with the value x. If the pointer points to a `struct` we can allocate memory and initialize the values of the `struct` with the syntax `[n:{...}]`.

### 1.2.6 Mockup functions

#### Creating quick mockups

*JCUT* gives the ability to provide a fixed behavior for any function defined in a C source file. More precisely, it allows replacing a function's definition with a new definition that has a fixed return value for a specific test or a group of tests, and after running them *JCUT* will change back to the original definition letting the rest of tests use the original behavior.

Creating a mockup function for a test in *JCUT* is pretty easy, the user only needs to use the `mockup` statement and inside of it specifiy an existing function as if it were a function call, but with no arguments, then an assignment operator and then a value according to the return type of the function.

The following example illustrates such feature:

```
1 mockup { function() = 5; void_function() = void; }
```

The user specifies what functions a `mockup` will be created for by writting down the function as if it was a function with no arguments. Even though the function takes arguments there is no need to provide them because *JCUT* only needs to know that this is a function by reading the parenthesis.

After writing the function with the parenthesis the next step is to write down the assignment operator = (not to be confused with the comparison operator ==). After the assignment operator one of two things can written:

1. For any function returning any integer, float, double, or even any type of pointer, the user can specify a number which will be casted to whatever return type the function has. Note that for pointers this means the function will be returning the specified address, while it may sound dangerous, sometimes might be helpful to specify 0 (NULL) as return type for a function returning a pointer just to execute error checking code in the functions.

2. For functions which have a `void` return type the user has to type 'void' as in the example above. What this will do is just replacing the function body of the void function with an empty body function which does nothing. This might be sometimes helpful when is needed to test a series of function calls in a single test without having the side effects of a function which is always called.

### Where is my mockup called?

At this point the user might be wondering, what if I had a function when is executed it generates a big stack of function calls and the function I want a `mockup` for is called in many places? Will the `mockup` function be replaced in every place? The short answer is yes. Imagine you have function A, which calls function B and X (X being a helper function), function B calls function C and X too, then function C calls function D and X too, and so on. Also imagine this function X makes some high intensive computing which always returns 256.

Now you want to create a `mockup` function for function X that always returns 1 by using the `mockup` statement. When you do so jcut will replace the existing function definition of X with the one jcut creates when you specified the `mockup` statement. After running the current test, the original function definition of X will be restored to not interfere with other tests that wish to test the actual function body of X. The following diagram summarizes this example:

```
# Original call stack for function A
A
`|-X
 |-B
 `|-X
  |-C
  `|-X
   |-D
```

When using the `mockup` syntax in an imaginary *jcl* test file, you will have

```
mockup { X() = 1; }
A();
```

*JCUT* will replace the existing function definition (the one that makes high intensive computing to return 256) with a definition that always returns 1. The new call stack for your function A will be like this.

```
# Mockup call stack for function A
A
`|-X_MOCKUP
 |-B
 `|-X_MOCKUP
  |-C
  `|-X_MOCKUP
   |-D
```

After running the given test *JCUT* will restore the original call stack for function A.

**Mockups for a single test.**

Now let us write some tests that can help us understand how the `mockup` statement behaves. Copy and paste the following code in your *cfile.c*:

```
1       #include <stdio.h>
2       int c;
3
4       void hello() {
5           printf("hello world\n");
6       }
7
8       void msg0() {
9           printf("I am in function %s\n", __func__);
10      }
11
12      void msg1() {
13          printf("I am in function %s\n", __func__);
14          msg0();
15      }
16
17      void msg2() {
18          printf("I am in function %s\n", __func__);
19          msg1();
20      }
21
22      int mult(int a, int b) {
23          return a*b;
24      }
25
26      int * get_ptr() {
27          return &c;
28      }
29
30      void print_ptr() {
31          printf("%p, %d\n", get_ptr(), (int)get_ptr());
32      }
```

Now in your *jcl* file add the following tests along with its comments so you can see what the test does.

```
1   # test 1
2   # Print the message 'hello world\n'
3   hello();
4
5   # test 2
6   # It won't print anything because hello function does nothing.
7   mockup { hello() = void; }
8   hello();
9
10  # test 3
11  # Print the address the function get_ptr() returns, 25555 in this case
```

```
12 mockup { get_ptr() = 25555; }
13 print_ptr();
14
15 # test 4
16 # Print the address the function get_ptr() returns, a real address.
17 print_ptr();
18
19 # test 5
20 # Print the following messages:
21 # I am in function msg2
22 # I am in function msg1
23 # I am in function msg0
24 msg2();
25
26 # test 6
27 # Print only 1 message
28 # I am in function msg2
29 mockup { msg1() = void; }
30 msg2();
```

You can see in test number 1 and 2 that we are testing a function which has a return type void. We created a mockup function for the `hello()` function with the syntax `hello() = void;` because `hello()` had void as return type. Tests number 3 and 4 call the function `print_ptr()` to print the address returned by the function `get_ptr()`. In test 3 we modify the address returned by the function `get_ptr()`, since we are just printing the address is safe to call this function without checking the returned address, but in a real world application you may want to improve the error checking in your code.

Test number 5 calls a function which prints a message and then calls another function that does the same thing. You will see 3 messages printed, but when we create a `mockup` function in test number you can see that only 1 message is printed.

### Can I use the mockup syntax with the before and after statements?

Yes you can. It is worth mentioning that any `mockup` created will take effect in any function called in the before and after statements. This means that the functions called in the before and after statement will have access to the `mockup` functions created.

The syntax combined for a single test is as follows

*test-definition* :

$$\textit{test-mockup}_{opt}\ \textit{test-setup}_{opt}\ \textit{test-function}\ \textit{test-teardown}_{opt}$$

*test-mockup* :

`mockup {` *mockup-fixture* `}`

*mockup-fixture* :

*mockup-function* `;` *mockup-fixture*$_{opt}$

*mockup-variable* `;` *mockup-fixture*$_{opt}$

*mockup-function* :

*function-call* `=` *constant*

`void`

*mockup-variable* :

*variable-assignment*

The mockup, before and after statements are all optional, but if you provide any of them they have to appear in the same order as shown in the previous syntax. 2.6.5

### Can I create a mockup that can be used for a specific group?

Yes you can. To do so you may want to use the keyword`mockup_all` and is defined at the beginning of a group. It is worth mentioning that `mockup` created for all the tests in the group will be valid for each of the tests in the group. Once the group is finished jcut will restore the original function definition.

The final syntax for the `mockup_all` keyword is as follows:

*test-group* :

*group-mockup*$_{opt}$ *group-setup*$_{opt}$ *test-definition* *group-teardown*$_{opt}$

`group` *identifier*$_{opt}$ `{` *test-group* `}`

*group-mockup* :

`mockup_all {` *mockup-fixture* `}`

The `mockup_all`, `before_all` and `after_all` statements are all optionall, but if you provide any of them they have to appear in the same order as shown in the previous syntax.

#### What happens when using mockup for a test inside a group that uses mockup_all?

Section 1.2.6 still holds true, and if you happen to create a mockup for the same function a mockup was created in the `mockup_all` statement, jcut will override the mockup created in the `mockup_all` statement and use the mockup created for the test, but once the test is finished *JCUT* will restore the `mockup` created in the `mockup_all` statement for the given group.

#### Summary

In this section we learned how you can use *JCUT* to create quick `mockup` functions to replace existing function definitions. This feature can be handy in several situations, for instance:

- You want to verify your error checking code in your functions by creating mockups that always return error codes.
- You want to force the execution flow of a function by creating several mockups for the functions called.
- Avoid calling time consuming functions that are not necessarily required for a test by creating an empty function.

`Exercise` Try writing some mockups for the functions from previous

### 1.2.7 Data Driven Testing

#### Using CSV files and data keyword

*JCUT* provides support for basic Data Driven Testing by following two simple steps. First you have to use a data placeholder, which is the character @, as input to each argument of a function under test or for the expected result. Second you need to provide the path for a CSV (Comma Separated Values) file by using the keyword data.

Let's look at an example which will show you how this feature works using the function sum from section 1.2.1:

```
1  /** cfile.c **/
2  int sum(int a, int b) {
3      return a+b;
4  }
```

And the test file would look like.

```
1  # test.jcl
2  sum(2,2);
```

When running *JCUT* it will test the function sum() with the given arguments. However there may be cases in which you really want to test a function using a big amount of parameters and writing that many tests is just too much work. You can have *JCUT* generate all those functions for you by using a data placeholder and a CSV file which contains all the parameters.

Create a CSV file called *data.csv* with the following contents.

```
1  1, 2
2  3, 6
3  7, 8
```

Now rewrite the test in your test.jcl file like this:

```
1  # test.jcl
2  data { "data.csv"; }
3  sum(@, @);
```

Note the new keyword data and the enclosing brackets, which contain a c-like string with the path to a CSV file and a semi colon ';'. Also note the data placeholders @.

There are some points to discuss about the CSV file and how the data placeholders are replaced:

- Each row in the CSV file corresponds to the parameters used for a single function call. *JCUT* will generate a function call for each row. You can have as many rows you'd like.

- Each data placeholder @ corresponds to each value separated by commas (column) and they are replaced from left to right. The number of data placeholders has to be equal or less to the number of columns in your CSV file. Each data placeholder

@ means "take the next column for the current row". For instance, if you had 5 columns, and you only have 1 @ *JCUT* will only take the first column for each row. 2 @'s indicate to take the first two, 3 @'s takes the first three and so on.

- All the rows in a CSV need to have the exact same amount of columns. It only takes one row to have a different number of columns and *JCUT* will stop.
- DO NOT PLACE COMMENTS IN THE CSV FILE, only comma separated values.

From these points we can tell about our previous example that *JCUT* will generate and run the following functions:

```
1  # jcut generated these functions
2  sum(1,2);
3  sum(3,6);
4  sum(7,8);
```

You can also use a data placeholder @ as an expected result.

```
1  # test.jcl
2  data { "data.csv"; }
3  sum(@, @) == @;
```

We need to add an extra column to our CSV file so *JCUT* can read the values and replace them as the expected result.

```
1  1, 2, 3
2  3, 6, 9
3  7, 8, 15
```

From this example *JCUT* will generate the following tests:

```
1  sum(1,2) == 3;
2  sum(3,6) == 9;
3  sum(7,8) == 15;
```

The following examples are all valid:

```
1  ################
2  # For each row in the CSV file take the first value only
3  data { "data.csv"; }
4  sum(@, 2) == 2;
5
```

```
data { "data.csv"; }
sum(2, @) == 2;

data { "data.csv"; }
sum(1, 2) == @;

################
# For each row in the CSV file take the **first 2** values
# from the data.csv file
data { "data.csv"; }
sum(@, @) == 2;

data { "data.csv"; }
sum(2, @) == @;

data { "data.csv"; }
sum(@, 2) == @;
```

Obviously whether the tests pass or fail depend on the values taken from the CSV file. Try running these examples and see what happens!

### What values can you place in the CSV file?

You can use the same values you would use in a regular test function in your jcl test file. For instance, recall the example from section 1.2.5.

```
/** cfile.c **/
void print_ptr_pair(struct NestedPair* ptr) {
    if(ptr) {
        printf("ptr.a = %d\n", ptr->a);
        printf("ptr.b = %d\n", ptr->b);
        printf("ptr.subpair.a = %d\n", ptr->nested.a);
        printf("ptr.subpair.b = %d\n", ptr->nested.b);
    } else
        printf("ERROR ptr is NULL pointer!\n");
}

# test.jcl
print_ptr_pair([1:{1,2,{3,4}}]);
```

You can use the following CSV file *data-structs.csv* in order to use data placeholders:

```
[1:{1,2,{3,4}}]
[1:{1,2,{3}}]
[1:{1,2}]
```

```
4       [1:{1}]
5       [1]
```

The test is written like this:

```
1       # test.jcl
2       data { "data-structs.csv"; }
3       print_ptr_pair(@);
```

*JCUT* will generate the following tests:

```
1       print_ptr_pair([1:{1,2,{3,4}}]);
2       print_ptr_pair([1:{1,2,{3}}]);
3       print_ptr_pair([1:{1,2}]);
4       print_ptr_pair([1:{1}]);
5       print_ptr_pair([1]);
```

It is highly recommended that all the values places in a CSV file are used only for a function under test because using it for different functions with different function prototypes may lead to errors. For instance, imagine a CSV file which contains data meant to be passed to a function that takes integers, but somehow you use that data to call a function that takes a pointer. This will have unexpected results, therefore it is recommended to have a CSV file for each function under test.

### Using data keyword with mockup, before and after statements

You can use the data keyword along with the before and after statements. The only restriction is the where you have to define it. You have to define it before the mockup statement. The following is a valid syntax:

*test-definition* :

*test-data*$_{opt}$ *test-mockup*$_{opt}$ *test-setup*$_{opt}$ *test-function* *test-teardown*$_{opt}$

*test-data* :

`data {` *string-constant* `;}`

*test-function* :

*function-call expected-result*$_{opt}$ `;`

*function-call* :

*function-name* `(` *function-argument* `)`

*function-argument* :

*constant*

*buffer-alloc*

*data-placeholder*

*constant* , *function-argument*

*buffer-alloc* , *function-argument*

*data-placeholder* , *function-argument*

*data-placeholder* :

`@`

There is an important note about using data placeholders @ and the keyword data with `before` and `after` statements. You cannot use data placeholders inside the `before` or `after` statements. Also you cannot use data place holders @ to create a `mockup` function. The following examples are **not** valid:

```
1  data { "some-data.csv"; }
2  mockup { func_1() = @; }
3  before { func_1(@); }
4  function_under_test(@, @) != @;
5  after { func_2(@); }
6
7  data { "some-data.csv"; }
8  mockup { func_1() = @; }
9  before { func_1(@); }
10 function_under_test(1, 1) != 1;
11 data { "some-data.csv"; }
12
13 mockup { func_1() = @; }
14 function_under_test(2, 2) != 2;
15 after { func_2(@); }
```

### Using `data` keyword and data placeholders @ with groups

You cannot use the `data` keyword and data placeholders with groups. There is no such `data_all`. The `data` keyword and data placeholders @ are meant to be used on a per test basis. This is due to simplicity of implementation and probing the acceptance of this feature among the users.

**Summary**

In this section you learned how to use a simple CSV file to define the data set for a given function. This feature can provide a great flexibility in cases in which you're interested to do some coverage testing for your functions and that requires writing a great amount of tests. You can easily generate your CSV file with a simple script or a spreadsheet application to generate a data set that follows a specific pattern to test your functions!.

As an exercise try rewriting some of the examples you've done so far to use a CSV file and data place holders. Or you could try the following:

```
1   /** cfile.c **/
2   #include <stdio.h>
3
4   void print(char *msg1, char *msg2) {
5       if(msg1)
6           printf("%s\n",msg1);
7       if(msg2)
8           printf("\t%s\n",msg2);
9       printf("END OF MESSAGE");
10  }
11
12  # test.jcl
13  data { "data-strings.csv"; }
14  print(@, @);
15
16  /** CSV File: data-strings.csv **/
17  "hello", " world"
18  "hello", 0
19  0, " world"
20  0, 0
```

### 1.2.8 JCUT interpreter

The tool comes with a very basic interpreter for the *JCUT* language. Using this interpreter is fairly simple. In order to enter the inter interpreter the user **has to omit the use of the test file with the flag "-t"** and just use the following command line:

```
jcut <c-source-files> -- <clang-llvm-options>
```

Whenever the flag "-t" is provided *JCUT* will detect it and run the tests in a test file rather than going into interpreter mode. This means that for running the interpreter the

user needs to enter a command line like this:

```
jcut cfile.c --
```

When getting into the *JCUT* interpreter the reader should see a message similar to this:

```
1  jcut Copyright (C) 2014  Adrian Ortega Garcia
2  This program comes with ABSOLUTELY NO WARRANTY;
3  This is free software, and you are welcome to redistribute it
4  under certain conditions; See the LICENSE.TXT file for details.
5
6  jcut interpreter!
7  For a complete list of available commands type /help
8
9  jcut $>
```

### How to execute a test from the interpreter?

Assuming the user entered the following command line:

```
jcut cfile.c --
```

And the function `"int sum(int a, int b);"` is defined in the file `"cfile.c"`, testing that function is as simple as typing the following on the interpreter prompt:

```
jcut $> sum(2,2);
```

By hitting the key `ENTER` the *JCUT* interpreter will execute that test. That will print the results as described in section 1.1.4.

### What expressions can be entered in the interpreter prompt?

Everything covered in this tutorial and written in a test file can be entered in the interpreter.

**IMPORTANT NOTE:** Some of the *JCUT* expressions may extend more than a single line, and typing such expressions in a single line would make the line hard to read and understand. The reader can instruct *JCUT* to begin a ***new line*** by pressing the keys `CTRL-J`. This will instruct *JCUT* to not execute the current line when the key `ENTER` is pressed and wait until the user has entered the complete expression. The *JCUT* prompt will become `"jcut ?>"` to indicate the user can enter as many lines as desired. Once the

user has entered the complete expression the way this expression can be processed by the interpreter is to press the `ENTER` key on an empty line so the *JCUT* prompt becomes again `"jcut $>"`. It is then when *JCUT* will execute the test.

### Interpreter commands

As of now the interpreter supports the following commands:

/**pwd** Prints the current directory where *JCUT* is being run.

/**load** Loads all the specified source files separated by space. This command is meant to be used in cases where the user forgot to provide a source file in the initial command line.

/**unload** Unloads a specific source file from *JCUT* memory. When *JCUT* runs an action on the source file, it runs it on every source file. This may become handy when a source file is no longer needed for testing.

/**ls** Lists all the function prototypes contained in all the loaded source files. It's important to not confuse this "/ls" command with the popular linux command "ls".

/**help** Prints all the available commands.

/**exit** Exits the *JCUT* interpreter.

It's important to mention that this interpreter does not intent to compete or reimplement the behavior of the linux commands *ls, cd,* or *pwd.* The current implemented commands were chosen based on the tasks the author believed would be most useful when testing. However, the author welcomes feedback should any new command be added or delete existing ones.

### Interpreter command line capabilities

The *JCUT* interpreter command line provides basic support for the following functionality:

- History handling with the UP (CTRL-P) and DOWN (CTRL-N) arrow keys

- *JCUT* keyword completion with TAB key.
- Popular linux command line shortcuts:

  **CTRL-A** Move to the beginning of the line.

  **CTRL-E** Move to the end of the line.

  **CTRL-N** Move to the next command in the history.

  **CTRL-P** Move to the previous command in the history.

  **CTRL-F** Move cursor forward one character.

  **CTRL-B** Move cursor backwards one character.

  **CTRL-L** Clear screen.

#### Using the Interpreter under Windows

It is known that under Windows the arrow keys to navigate through command history does not work properly, it prints the value of the ASCII characters. The user can work around this issue by using their equivalent shortcuts as described in previous section, CTRL-P and CTRL-N.

### 1.2.9 End of tutorial

This concludes all the features available in jcut. We hope you have found our tool interesting and useful, feel free to keep using it or even try to break it. If you find any error or bug please send both C source file and test file to the author's email `adrianog.sw@gmail.com`. *JCUT* code lives at `https://github.com/elfus/jcut`.

Thank you!

# APPENDIX 2

# JCUT Advanced topics

## 2.1 Complex expressions in jcut language

jcut does not support complex expressions like the C programming language does. For instance the following expression is not supported:

```
1       int var = 0;
2       ...
3       var = fact(1) + fact(3);
```

The reason to not support complex expressions is that the author didn't want the jcut language to become another programming language on its own. The objectives behind the design of this language has always been to keep it small and simple, therefore a small portion of some statements syntax was borrowed from the C language in adition to the keywords `data`, `group`, `before`, `after`, `mockup`, `before_all`, `mockup_all` and `after_all` that reflect some common concepts, i.e. test fixtures, from the Test Driven Development methodology.

## 2.2 JCUT Command line options and Clang command line options

*JCUT* makes use of the Clang and LLVM APIs, as such you can provide specific options to Clang, LLVM or JCUT.

Throughout this tutorial the user may have noticed an extensive use of the *double dash* `"--"` as an option to *JCUT*. The two dashes `"--"` is separator for the *JCUT* specific options (left side) and the *clang-llvm* specific options (right side). The structure of a command line is as follows:

```
1    jcut <JCUT OPTIONS> -- <CLANG or LLVM OPTIONS>
```

## 2.3 Testing code which uses 3rd party libraries

In case the C code under test makes use of a 3rd party library it's necessary to tell the Clang APIs where the header files are located by using the `-I` flag. The command line flag is as follows:

```
-I path/to/include/files
```

It is important to note that this flag is needed to be provided after the two dashes provided to *JCUT* on the command line to indicate is an option for Clang. The resulting command line looks like this:

```
1    jcut cfile.c -t test.jcl -- -I path/to/include/files
```

## 2.4 Testing code which uses the C standard library

While writting the tests in this tutorial the reader will notice the code uses the C standard library `stdio.h, stdlib.h,`, etc. Along with this tutorial two small folders with a portable version of the C standard library, taken from the MinGW project, are provided. Assuming the *JCUT* binary is on the same folder level as the include folder you can execute the tool with the following command:

```
1    jcut cfile.c -t test.jcl -- -I include
```

For linux users this won't be much of a problem because *JCUT* will probably find the correct path to the standard library if it was properly installed using the proper package manager or installer.

## 2.5 Current Limitations

The *JCUT* tool has the following limitations:

- It can process only 1 C source file at a time, but this will change in the short time. This restricts testing of code spread into several C source files.
- Output customization. The end user can't customize the info and format of what is printed on screen.

## 2.6 Under development

The author is actively working to support the following features:

- Process several C source files in one command line. The user will be able to test code spread across several files.
- Output customization. The user will be able to customize the tool output with command line arguments or a file that describes how the output should be.

Any question or comment can be sent to `adrianog.sw@gmail.com`.

## 2.7 JCUT language keywords

**mockup_all** This statement lets you create an arbitrary function definition that always returns a fixed value or an empty function for void functions which can be used by all the tests in a given group.

**before_all** This statement lets you specify a set of functions to be called before all the tests in a given group. It also allows you specify the value for any global variable of any type. This statement ensures the global variables modified here will always hold the given state for each of the tests in that group. Thus, this statement won't keep track of the state of global variables modified in the test functions to be called. These statements will be executed only once before all the given tests in a group.

**after_all** Behaves pretty much the same as the `before_all` statement with the only difference that it will execute the statements after all the tests have been executed.

**data** This statement lets specify the path to a CSV file which contains the data for replacing data placeholders in a function call. *JCUT* will generate as many functions as there are rows in the CSV file.

**mockup** This statement lets you create an arbitrary function definition that always returns a fixed value or an empty function for void functions which can be used by a specific test.

**before** Behaves just like `before_all` except that all the statements executed will only affect the given tests. After executing the test, all the original values of global variables before assigning them the value stated in the `before` statement will be restored. Any modification done to the program state by the functions called in this statement won't be reverted.

**after** Behaves just like `after_all` except that all statements executed will only affect the given test. Any modification done to the program state by the functions called won't be reverted.

**group** The group keyword lets you group tests into a single logical related tests. This way the tool will let you specify what groups to execute only. A group can optionally contain more groups indefinitely. A group can optionally have name given by the user, if it's not provided the tool will generate a default name. This name will be used for error reporting purposes and easy tracking from the user. Any modification done to the program state by the functions called wonâĂŹt be reverted.

**comparison operators** The comparison operators are provided to compare the output of a given function or a comparison statement in a before or after statement and they behave just like in C. The operators available are:

```
1       ==, !=, >=, <=, <, >
```