jcut tutorial

September 15, 2014

Contents

1 Introduction

1.1 What is it?

jcut is a simple command line tool to perform *unit testing for the C programming language* using the LLVM Just In Time (JIT) engine and some of the Clang APIs. This means that the tool will compile and execute only the functions you want to test. *jcut* stands for *Just in time C Unit Test*ing.

1.2 Objectives

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

- Minimize the amount of test code you need to write for your software written in C.
- Simplify the test code you write.
  - We believe that 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 compile 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.

While the main focus of our tool has always been oriented to unit testing and TDD we have found during the early stages of development and testing that this tool might be a good resource for educational purposes for C novice programmers and universities.

1.3 How does it work?

jcut takes as input 2 important things: your C source code and a test file.

- *C source code*: jcut makes use of the Clang APIs to analyze and process the source code, hence it will behave pretty much like a compiler. It will analyze your C code and report any error if it exists. Then it will analyze your test file to know what tests you want to test. If you use a 3rd party library see section 3.2 or 3.3.
- *Test file*: This is a plain text file with any extension. In order for this tool to understand what you want to test and expecting from a test you need to use the *jcut Language*. Don't worry about learning a new language, you will see in the rest of this tutorial that this language is really small, simple and based on a small subset of the C syntax. If you are comfortable with C's syntax then you're pretty much done learning this new language.

Upon receiving these 2 inputs the tool will analyze the C code using Clang APIs and will generate code for only those functions described in the test file and then run them using the LLVM JIT Execution Engine. Then a small report will be provided by the tool about the tests ran.

From a high level perspective the following diagram describes how jcut works.

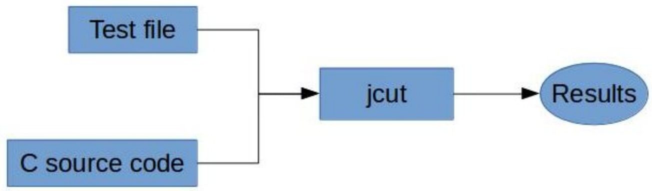

Figure 1: High level view of how jcut works

In section 4 you can check what are the current limitations of this tool.

1.4 *jcut* output

In this section we will briefly explain how the 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
```

The columns shown are explained from left to right:

- GROUP NAME: By default a group name is assigned to each test. As you will see in section 2.3 you can create your own groups.
- 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 2.1.3 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.

2 The tutorial

In this tutorial we will walk you through the usage of *jcut* to test some C code. We will start by writing some basic and simple C functions, and then some tests for it.

For the sake of simplicity all the C code we write will be located in a source file named *cfile.c* and our test file will be named *test.jcl*.

NOTE: For the examples in which we make use of the C standard library (stdio.h, stdlib.h, etc.) you 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 you can run the tool on that same folder.

Since *jcut* is a command line tool for this tutorial you will need two things open at all times:

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

2.1 The basics

2.1.1 The simplest test

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

```
// cfile.c
void my_function() {
        return;
}
```

Then create the test file *test_file.jcl* and type the function we want to call.

```
# test.jcl
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 you can place comments in your test file starting with the character #.

Open a command line, shell or terminal in your operating system and then to run the above test on the function we just wrote type the following command line:

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

Now let's break down each of the arguments in the command line:

- `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.
- `--` : Make sure you add a two dashes at the end of the command line. Why? *jcut* makes use of *clang* APIs, this makes it a *clang tool* and as such it can take a *compilation database*. The two dashes tells *clang APIs* we don't have a *compilation database* and those are all the arguments for *jcut* to work. For the sake of simplicity we won't talk about *compilation database* in this tutorial because we don't really need it, but we have to provide the two dashes for *jcut* to work happily.

After running this command line you 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 you were 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, it will compile and execute it using LLVM JIT Execution engine. Since this is a void function the tool assumes the test has passed if it completed its execution without interruption.

2.1.2 Passing arguments to a functions

Now we will see how to call a function takes some parameters and returns a value. Type the following function in your file *some_file.c*:

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

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

```
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.

2.1.3 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.

You can compare the actual result of a given function with an expected result with the following syntax in your *test file.*

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

This syntax may feel awkward for a C programmer, however it 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.

```
sum(2,2) != 4;
sum(2,2) == 4;
sum(2,2) >= 4;
sum(2,2) <= 4;
sum(2,2) > 4;
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.

2.1.4 What values can I use as expected result?

As of now the values you may use as expected result are*:*

- Constant integers (only decimal notation is supported, hexadecimal and octal is in the "to do" list).
- Floating point constants.
- Characters denoted with the syntax 'X' where X is any valid ASCII character.
- C-like 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 in comments.

```
# It will take the ASCII value of 'a' (49)
sum(2,2) == 'a';

# If sum returns an unsigned value,
# -47 will be treated as unsigned value
sum(2,2) == -47;

# It will take the memory address of the
# string and casting it to whatever data
# type the function returns
sum(2,2) == "this is a string";
```

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'.

2.1.5 Summary

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

Today is your lucky day because I will provide those functions for you :)

```
int fact_1(int a) {
	if(1 == a) return a;
	return a * fact_1(a-1);
}

int fact_2(int a) {
	int i = 1;
	while(a) i *= a--;
	return i;
}
```

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*.

2.2 Managing program state

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. 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 you define the state of your program by using 2 keywords from the *jcut language:* `before` and `after`.

The syntax for these two keywords is as follow:

```
[before { ... }]
function_under_test();
[after { ... }]
```

You have to write the `before` keyword before the function you want to test and type the `after` keyword after the function you want to test. The curly braces are mandatory for both the `before` and `after` keywords. The squared brackets indicate that either statement are optional for your test.

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

2.2.1 Modifying global variables

Write the following code in your source file.

```
int state; /* Global variable */

int get_state() {
       return state;
}
```

Write the following test in your test file.

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

Run this test through the command line.

```
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:

```
before { state = 10; } # Assignment operator =
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.

2.2.2 Checking global variables values

Imagine the *get_ state()* function has a return type of *void* and there was no way for us 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.

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

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:

```
before { state == 0; } # Comparison operator ==
get_state();
```

Is 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 a state 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.

If you happen to use any comparison operator in the `after` or `before` statements *jcut* will 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 met 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?

You can use the same operators for comparisons as you would in section 2.1.3. For instance:

```
before { state != 0; }
get_state();

before { state >= 4; }
get_state();

get_state();
after { state <= 1; }

get_state();
after { state > 0; }

get_state();
after { state < 10; }
```

All these examples have valid comparison operators.

2.2.3 Calling functions *before* and *after* a function under test

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

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

Now we will modify the variable *state* by calling a function rather than doing an assignment. Here is the new test:

```
before { modify_state(); } # Function call, state == 5
get_state() == 5; # Passes
after { modify_state(); } # You can call a function call here too, state == 10
```

This test will succed 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 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.

2.2.4 Summary

In this section we just learned that we can use the `before` and `after` keywords to execute three different types of statements:

1. Variable assignment `variable = X` where X is any value as described in section 2.1.3 and `variable` is a global variable of any type. For pointer types see section 2.5.1. For struct types see section 2.4.1.
2. Function calls.
3. Variable comparison with any value as described in section 2.1.3.

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

```
[before {<variable assignment>* | <function call>* | <variable comparison>*}]
function_under_test();
[after { <variable assignment>* | <function call>* | <variable comparison>* }]
```

Note the little star to the right of each statement. That indicates that you can have as many statements as you want in the *before* and *after* statements or none of them in any order. The squared bracket indicate that both the before and after statements are optional for a function under test.

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:

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

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.

2.3.1 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:

```
# test_file.jcl
my_function();
sum(2,2);
sum(2,2) == 4;
sum(2,2) != 5;
fact_1(3) == 6; # provided in section 2.1 Summary
fact_2(3) == 6; # provided in section 2.1 Summary
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.

2.3.2 Nesting groups

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

```
# test_file.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.

2.3.3 Managing program states for a given group

Recall section 2.2 in which we learned how to modify the program state for a given test with the keywords `before` and `after`. We 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.

```
group E {
        before_all { state = 10; } # Executed before all the tests in this group

        get_state() == 10; # test 1

        # jcut does not track what's changed inside this function call
        before { modify_state(); }
        get_state() == 15; #test 2

        modify_state(); # test 3

        after_all { state == 20; } # Executed after all the tests in this group
}

group F {
        before_all { state = 5; }

        get_state() == 5;

        before { modify_state(); }
        get_state() == 10;
}

group G {

        get_state() == 0;

        after_all { modify_state(); }
}
```

All these are valid tests and they all pass. Try running writing these tests in your *test.jcl* file. If you have 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 2.2.3.

2.3.4 Summary

In this section we learned how to group tests by using the keyword `group`. We 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:

```
group [Name] {
        [before_all { <variable assignment>* | <function call>* | <variable comparison>* }]
        ...
        [after_all { <variable assignment>* | <function call>* | <variable comparison>* }]
}
```

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 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 also have an optional name. If you omit this name the tool will assign this group an automatically generated group name.

2.4 Structures as global variables

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

```
/* Example C code */
struct Pair {
        int a;
        int b;
};
...
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 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:

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

2.4.1 Struct initialization list

Let us write a function that prints the contents of a global struct. See section 3.2 on how to use the C standard library. Open your *cfile.c* and type the following.

```
/** some_file.c **/
#include <stdio.h>

struct Pair {
        int a;
        int b;
};

struct NestedPair {
        int a;
        int b;
        struct Pair nested;
};

struct NestedPair global_pair;

void print_global_pair(){
        printf("global_pair.a = %d\n", global_pair.a);
        printf("global_pair.b = %d\n", global_pair.b);
        printf("global_pair.subpair.a = %d\n", global_pair.nested.a);
        printf("global_pair.subpair.b = %d\n", global_pair.nested.b);
}
```

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

```
# test 1
print_global_pair();

# test 2
before {global_pair = {1}; }
print_global_pair();
```

```
# test 3
before {global_pair = {1,2}; }
print_global_pair();

# test 4
before {global_pair = {1,2, {5} }; }
print_global_pair();

# test 5
before {global_pair = {1,2, {5,6} }; }
print_global_pair();
```

These 5 tests show the basics of C struct initialization list syntax in *jcut*. The first test will print the contents of the global struct `global_pair`. On the following tests we just modify the contents of the struct `global_pair` one element at a time to appreciate how the struct initialization works, even for nested structs. Note that you cannot assign values starting from the last element in the struct, it hast to start from the first element.

2.4.2 Pointers to structs

The initialization of structs through a pointer is pretty much the same as we just saw with a little difference. However we need to explain how *jcut* works with pointers. We will explain that and also initializing structs with poitners in the next section.

2.4.3 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.

2.5 Pointers

As C programmer you already know 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 somwhere 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.

2.5.1 Automatic memory allocation

Whenever you have a function that takes a pointer or a global variable of any pointer type, you can tell *jcut* to allocate memory for it by using the following syntax:

```
[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 is it allocates `n*sizeof(<data type>)` bytes. Let us write an example. Write the following C code in your file *some_file.c*.

```
#include <stdio.h>

int * g_ptr = NULL;

void print_g_ptr(){
        if(g_ptr)
                printf("address: %p, contents: %d\n",g_ptr,*g_ptr);
        else
                printf("ERROR g_ptr is NULL pointer!\n");
}
```

Then write a simple test in your test file *test_file.jcl.*

```
print_g_ptr();
```

After running this test you should see that your test printed the error stating *g_ptr* is a null pointer. In order to have a working test we can tell *jcut* to allocate memory for the data type pointed to by the pointer. Let's do it by writting a new test:

```
# allocates enough memory for 1 data type g_ptr points to.
before { g_ptr = [1]; }
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 you could allocate enough memory for an array of integers by writing the following test.

```
# allocates enough memory for 1024 data type g_ptr points to.
before { g_ptr = [1024]; }
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.

You don't have to worry about *free*ing the memory allocated, *jcut* will do that for you automatically after running this test. If you want the memory allocated to last for the execution of several tests you may want put the expression `g_ptr = [n]` in a `before_all` statement. See section 2.3 for more information on `groups`.

2.5.2 Memory initialization

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

```
[n:x]
```

We just add a colon and then we pass 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:

```
# allocates enough memory for 1 data type g_ptr points to.
before { g_ptr = [1:5]; }
print_g_ptr();
```

After running the test you will see that the contents of the memory allocated is actually a *5*.

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

```
// Print the contents of the array pointed to by buf
void print_buffer(unsigned char *buf, unsigned size) {
        short step = 1;
        unsigned char *i = buf;
        unsigned char *end = buf + size;

        while(i < end) {
                printf("%x ", *i);
                if(step%10 == 0)
                    printf("\n");
                ++step;
                ++i;
        }
        printf("\n");
}
```

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

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

The function `print_buffer` will print `size` bytes of the array pointed to by `buf`. We just tested this function with *jcut* by allocating 20 elements of type `unsigned char` and passing that pointer to the function. Then we passed the size of the array which is of 20 `unsigned char`s (rememeber the size of an unsigned char is 1 byte). After running that function you will see printed 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.

2.5.3 Pointers to structures

If you had a global variable which is a pointer to a struct or a function that receives a pointer to a struct and you want to initialize that pointer the section 2.5.1 and section 2.5.2 applies the same to pointer to structs. If you'd like to initialize each field of the struct just as described in section 2.4.1 the only thing you need to change is the initialization value to a struct initialization list. The following example demonstrates this. Write this function in your file *cfile.c*.

```
// 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");
}
```

Write the following test:

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

You can see that instead of providing an integer value to initialize the memory allocated, we provide a struct initializer as described in section 2.4.1.

jcut currently does not support initializing a struct which is passed by *value.* For instance imaging the the print_ptr_pair function is declared as:

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

The following in a test file are invalid:

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

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

2.5.4 Summary

In this section we learned that we can tell *jcut* to allocate 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:{...}]`.

2.6 Data Driven Testing

2.6.1 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 function under test as well as for the expected results. 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. Let's use the function sum from section 2.1.2

```
// cfile.c
int sum(int a, int b) {
        return a+b;
}
```

And our test file would look like.

```
# test.jcl
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 writting those 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. Let's create a CSV file called *data.csv* with the following contents.

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

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

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

Note the new keyword data and the enclosing brackets which contain a c-like string with the path to a CSV file. 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 corressponds 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 @* correponds 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 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:

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

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

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

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

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

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

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

The following examples are all valid:

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

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

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

################
# For each row in the CSV file take the **first 2** values only
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 take form the CSV file. Try running these examples and see what happens!

2.6.2 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 2.5.2,

```
// 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}]
[1:{1}]
[1]
```

The test is written like this:

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

jcut will generate the following tests:

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

It is highly recommended that all the values places in a CSV file are used only for a function because using it for different functions with different function prototypes may lead to errors.

2.6.3 Using `data` keyword with `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 `before` statement. The following is a **valid** syntax:

```
data { "some-data.csv"; }
before { ... }
function_under_test(@, @) != @;
after { ... }
```

There is an important note about using data placeholders `@`, the keyword `data` with `before` and `after` statements. You **cannot** use data placeholders inside the `before` or `after` statements. The following examples are **not** valid:

```
data { "some-data.csv"; }
before { func_1(@); }
function_under_test(@, @) != @;
after { func_2(@); }

data { "some-data.csv"; }
before { func_1(@); }
function_under_test(1, 1) != 1;

data { "some-data.csv"; }
function_under_test(2, 2) != 2;
after { func_2(@); }
```

2.6.4 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 probe the acceptance of this feature among the users.

2.6.5 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 writting 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 rewritting some of the examples you've done so far to use a CSV file and data place holders. Or you could try the following:

```
// cfile.c
#include <stdio.h>

void print(char *msg1, char *msg2) {
        if(msg1)
                printf("%s\n",msg1);
        if(msg2)
                printf("\t%s\n",msg2);
        printf("END OF MESSAGE");
}
```

```
# test.jcl
data { "data-strings.csv"; }
print(@, @);

// CSV File: data-strings.csv
"hola", " mundo"
"hola", 0
0, " mundo"
0, 0
```

2.7 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`.

Thank you!

3 Advanced topics

3.1 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:

```
int var = 0;
...
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 `group`, `before`, `after`, `before_all`, and `after_all` that reflect some common concepts, i.e. **test fixtures**, from the *Test Driven Development* methodology.

The whole *jcut language* syntax can be summarized like this (this is not a fully BNF notation):

```
[data { "path-to-data.csv"; } ]
[before {
        <variable assignment>*
        | <function call>*
        | <variable comparison>*
}]
<test function>
[after {
        <variable assignment>*
        | <function call>*
        | <variable comparison>*
}]

group [NAME] {
        [before_all {
                <variable assignment>*
                | <function call>*
                | <variable comparison>*
        }]
        ...
        [after_all {
                <variable assignment>*
                | <function call>*
                | <variable comparison>*
        }]
}
```

The square brackets indicate optional statements. The start next to some statements indicate they are optional. Groups are optional and they can contain nested groups or just tests.

3.2 Testing code which uses the C standard library

While writting tests you will find code that uses the C standard library stdio.h, stdlib.h, etc. and *jcut* may complain about not finding the header files for the libraries. All you need to do is to provide the command line flag

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

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.

For Windows users two small folders will be provided with the *jcut* binary include and lib. Assuming the binary is on the same folder level as the include and lib folder you can execute the tool with the following command:

```
jcut -I include some_file.c -t test_file.jcl
```

3.3 3rd party libraries

In case the C code you are testing makes use of a 3rd party library you will need to tell *jcut* where their header and source files are located by using the command lines as described in section 3.2

4 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
- Decimal notation only for integers in the test file.

5 Under development

The author is actively working to support the following features:

- Quick function mockup generation.
  - The tool generates a default function definition and uses it to replace whatever function the user desires for a specific test. This feature works with function declarations and function definitions.
- 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.
- Support for hexadecimal and octal notation for integer values.

If you'd like to see a specific feature don't hesitate to send your suggestion to `adrianog.sw@gmail.com`.

6 Appendices

A. jcut language keywords

- *before_all*. This statement lets you specifify 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.

- *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 won't 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:

  ==, !=, >=, <=, <, >