Headlock Documentation

Release 0.5.0

Robert Hölzl

Feb 06, 2020

Contents:

`headlock` is a python/C bridge with automatic binding generator and focus on testing C code from python. For a more complete feature summary refer to *About*.

- Current Version: 0.5.0
- PyPI: https://pypi.org/project/headlock
- Git Repository + Issuetracker: https://github.com/mrh1997/headlock
- Documentation: https://headlock.readthedocs.io

CHAPTER 1

About

Headlock is designed as an adapter for testing C code via tests written in python. When being combined i.e. with pytest it provides a very powerful and convenient way of writing (unit-/integration-) tests for C code.

This results in differents goals when being compared to other (excellent) C/Python bridges like ctypes, cffi, swig, cython, ...:

- All the extra steps usually required for C are done under the hood. For the user adding a C file to a project is as simple as adding a python file. These steps done by headlock include:
  - No need to create extra Makefiles/Buildscripts for unit testing a single C module
  - No need to run extra build steps before using the C code.
  - No need to rewrite the C moduels interface definition (the header file) in Python.
- Provide a simple, intuitive API for accessing C objects. The philosophy of this API is to be as orthogonal as possible and stick as near as possible to the corresponding C language operators/objects. Thus the effort for learning it should be kept low.
- As being specially designed for unittesting, headlock includes support for typical testing tasks:
  - mock the underlying C modules in Python without any line of extra wrapper code
  - Vary not only the variables modifyable during runtime for testing corner cases but also the preprocessor defines.
- Exceptions raised in python callbacks (or python mocks) are automaticially forwarded to the calling python code by skipping the C-code-under-test via setjmp() in case of an exception.
- **[PLANNED]** Run the C code in a separate address space to guarentee real test isolation. This will not only prevent a crashing test from crashing the whole test-runner, but especially avoids that a misbehaving Module Under Test leaves the test process in an undefined state. Otherwise in the worst case this could cause one of the following tests to return different results than when being run separately.
- **[PLANNED]** Test the same piece of C code compiled for 32bit and 64bit from within a single testrun. This means it doesn't matter what architecture your python testcode is running on. As the C code is running in a separate Process it can control both variants, 32bit and 64bit.
- Especially make it work with embedded systems, so that

  - **[PLANNED]** C code can be executed on destination hardware. This is primary useful for integration tests as it allows to detect architecture specific problems (for non x86 hardware) and timing issues. Furthermore it allows to communicating with external components instead of mocking them which might show problems that where hidden by the mocks.
  - development of (non-device-driver) embedded C code can be done (via unittest) on a PC without the need to struggle with embedded hardware.
- Integrates well with Testing tools (like unittest, pytest, ...)
- **[PLANNED]** Being ToolChain agnostic via a plugin infrastructure. This includes not only the compiler, but for example on embedded systems also the infrastructure to load the firwmare into a device or communicate with it.

Explicitly Non-Goals Are:

- High Performance (This does not mean that it is slow. But if speed conflicts with one of the goals of this project, there will be no compromises in favour of speed).
- Being self-contained (At least LLVM and a C-compiler will always be required to be installed).
- Support for Python < 3.6

CHAPTER 2

Getting Started

2.1 Requirements

The following prerequisites are required by headlock and thus have to be installed before using it. Currently it is only explicitly tested with the minimal version requirements. Higher versions should work nevertheless.

- Windows or Linux are required
- CPython 3 Version 3.6 or higher is required
- LLVM (Version 7.0.0 or higher).
- GCC is required (in case of Windows MinGW64 is needed). Later other C compilers shall *be supported*!

2.2 Installation

The easiest way to install headlock is from PyPI via pip. To Install the most up-to-date stable release (0.5.0) run:

```
pip install headlock
```

Alternatively one can install the latest development branch directly from the github repository via

```
pip install git+https://github.com/mrh1997/headlock.git
```

Attention: AVIRA and maybe also other virus scanners seems to delay loading DLLs compiled a moment ago by multiple seconds (see this link). As this feature is essential for headlock, you must add you project directory to the list of directories, that shall not be scanned by the realtime scanner. Otherwise the first instantiation of a testsetup will require 10-30 seconds per run.

2.3 Usage

The following sample demonstrates the basic features of headlock. That is automaticially compile and load a piece of C code (including mocking and prepocessor macros via command line) and calling it from python.

The following demo C code has 3 implementations for incrementing a given integer by the macro `INCREMENT_OFFSET` (has to be set via compiler command line):

```c
int increment(int number)
{

    return number + INCREMENT_OFFSET;
}

void increment_inplace(int * number)
{
    *number += INCREMENT_OFFSET;
    return;
}

struct add_operands_t {
    int op1, op2;
};
extern int adder(struct add_operands_t * ops);

int increment_via_extfunc(int number)
{
    // adder_from_other_module() is not part of this file!
    struct add_operands_t ops = { number, INCREMENT_OFFSET);
    return external_adder(&ops);
}
```

Lets assume this code is stored in `test.c` and shall be tested from python. Then the following python file (in this case in the same directory) will call the `increment*()` functions and test if their result is correct:

```python
from headlock.testsetup import TestSetup, CModule

@CModule('test.c', INCREMENT_OFFSET=1)
class TSSample(TestSetup):
    pass

ts = TSSample()

# test increment():
assert ts.increment(10) == 11

# test increment_inplace()
int_var = ts.int(10)
ts.increment_inplace(int_var.adr)
assert int_var == 11

# test increment_via_extfunc()
ts.adder_mock = lambda ops: ops.op1 + ops.op2    # mock required func
assert ts.increment_via_extfunc(10) == 11

# this call is recommended (although it will be done implicitly otherwise)
ts.__unload__()
```

CHAPTER 3

User Manual

3.1 C/Python Proxy Objects

The basic building blocks in accessing C from python is a set of python classes that are used as proxies for C Data object. These python proxy objects allow to run the same actions on the C Data objects but from python. Especially they allow to convert the C Data Object to cross the language barrier by converting it to a corresponding python object and vice versa on request.

These python proxy objects are required in headlock for getting access to:

- global C variables
- values returned by C function
- C Data Object allocated manually via *C Type Objects*

By far the most important feature of proxy objects is the `.val` attribute. This attribute is readable and writable. It allows the C Data Object represented by the proxy to be converted to/from a corresponding python object. The type of this python object depends on the type of the C Data Object (see *C/Python Type Mapping*).

Note: To avoid specifying .val on every read access, the proxy object classes are implementing most python operators (where meaningful). Thus one can write for example `<proxy-obj> + 4 != <proxy-obj>` instead of `<proxy-obj>.val + 4 != <proxy-obj>.val`. Note that this is all about convenience. Internally the exactly same operations are run.

This is not the case when assigning. This means `ts.<global-var> = 3` instead of `ts.<global-var>.val = 3` is not possible!

3.1.1 C/Python Type Mapping

When C Data Objects shall cross the language barrier they have to be converted to/from a matching python object. Such a convertion can either done via the `.val` attribute or when creating an object by passing a python object as first

parameter into the constructors of *C Type Objects*. This chapter specifies the python objects that can be passed to/from python proxies in dependance of the underlying C Data Objects.

Please note that depending on the underlying C type most proxies accept multiple python types to be converted to C while they return always the same python type. The detailed mapping between C Data Objects and the python objects is listed in the following table. Please note that the first python type (marked as "default") is the type returned when retrieving the .val attribute.

C Type	Python Type of <proxy>.val attribute
• `(un)signed char` • `(un)signed int` • `(un)signed short` • `(un)signed long`	• `int` (default) • `ByteLike` of size 1 (translates to the corresponding ASCII code) • `str` of size 1 (translates to the corresponding unicode point)
`_Bool`	• `bool` (default)
Pointer (`*`)	• `int` (default): corresponds to the address of the C object • `Array proxy object`: When passing an array proxy to a pointer, the pointer is initialized with the address of the C array (corresponds to casting a array to a pointer in C). • `Iterable` (i.e. list or tuple): same behaviour as when assigning an iterable to an array (see Array). Please note that it is also possible to assign an iterable to a pointer to void (`void *`). In this case the elements of the array are assigned byte by byte to the memory referred by the pointer.
Function Pointer(`*`)	• `int` (default): corresponds to the address of the underlying function. • `callable`: A python function that shall be wrapped as C pointer
Array (`[]`)	• `Iterable` (`list` is default): Maps the elements of the iterator element by element to the proxies of the elements of the array. Thus `p.val = [x, y]` is identical to `p[0].val = x; p[1].val = y`
Structure (`struct`)	• `Mapping` (`dict` is default): Every entry in the Mapping is mapped to the member in the C struct with the same name. Thus `s.val = { a: x, b: y }` is identical to ``s.a.val = x; s.b.val = y``. When assigning a python mapping to .val and the python mapping does not contain all entries of the C struct, the missing entries are set to the null-value of the corresponding type. • `Iterable`: The elements if the iterater are mapped to the members of the struct in their definition order. I.e. lets assume the following definition: `struct { int a, b };`. In this case `s.val = (x, y)` is identical to `s.a.val = x; s.b.val = y`. When assigning a python iterable to .val and the C struct contains more members then the iterable provides, the remaining members are set to the null-value of the corresponding type.
...	...

Note: Additionally to the python types listed in the table a proxy object can be mapped to a C Data Object of the the same type. This means that you can do `<int-proxy>.val = <other-int-proxy>` which is an confinience feature that translates internally to `<int-proxy>.val = <other-int-proxy>.val`.

Note: apart from `.val` there are object convertion attributes for special cases. Currently the following attributes are implemented:

- `.c_str` (arrarys and pointers): convert a zero-terminated C string to/from a `bytes` object
- `.unicode_str` (arrarys and pointers): convert a zero-terminated wide-string to/from a python `str` object. Depending on the type of the underlying object this string has to be utf8 (pointer to 8-bit items), utf16 (pointer to 16-bit elements) or utf32 (pointer to 32-bit elements) encoded.
- `.tuple` (structures): enforce the python object to be set/retrieved as a tuple object (using basicially the same mapping as when setting `.val` to a tuple).

3.1.2 C Type Objects

Every C type is represented by a corresponding python class which is bound to the execution environment of *The TestSetup*. It can be used to allocate and initialize a new C Data Object within this execution environment by calling it (i.e. `ts.short()`). The return values are *C/Python Proxy Objects* that are referring to the created C Data Object. When calling the C type class it is possible to pass a value (i.e. `ts.short(9)`). This python object will be converted to the corresponding C value according to *C/Python Type Mapping*, that in turn is used to initalize the created C Data Object.

Please note that when creating a proxy object manaually via the approach described above, headlock will handle full memory management. That means the created C Data Object has the same live cycle as the python proxy and will be released automaticially when the python object is released.

Note: This includes also memory buffers that are allocated implicitly when passing a iterable to a pointer.

Apart of the instatiation of C Data Objects the following attributes/operators are available in the python class to simulate C operators:

Python Attribute/Operator	C Operator	Description
`<type-proxy>.ptr`	`<type> * ...`	Creates a pointer type that points to a <type> object
`<type-proxy>.array(<size>)`	`<type> ... [<size>]`	Creates an array type that refers to a C array of <type> object
`<type-proxy>.alloc_array(<size>)` `<type-proxy>.alloc_array(<list>)`	`<type> ... [<size>]`	Creates an array object proxy that refers to a C array of <type> object. The created array object is either initialized with null_val's (if only the arrays size is specified), or it is initialized by the passed iterable. In case an int is specified this is a shortcut for `t = <type-proxy>.array(<size>); o = t()`. In case an iterable is specified this is a shortcut for `t = <type-proxy>.array(len(<init>)) o = t(<list>)`.
`<type-proxy>.alloc_ptr(<size>)` `<type-proxy>.alloc_ptr(<limit>)`	`<type> ... = malloc(size*3)`	This is analoguous to `.alloc_array`. The only difference is, that it does not return the array object itself, but a pointer to it. Furthermore it works on `void`, in which case the base element is one byte (thus `ts.void.alloc_ptr(3)` will return a pointer to a 3 byte buffer). The returned pointer object manages the arrays lifecycle, which means that the array will be released automaticially when the pointer is released. Usually this operator is used to allocate a memorybuffer from within python.
`<type proxy>.sizeof`	`sizeof(<type>)`	returns the size of the Type proxy in bytes

As described in chapter *The Testsetup* when creating a testsetup all custom C type objects representing the C types from the MUT will be added to the created testsetup class as attributes.

Furthermore the build in C types are always available in the testsetup. Where required spaces are replaced by underscores:

- `ts.char`
- `ts.signed_char`
- `ts.unsigned_char`
- `ts.short`
- `ts.signed_short`
- `ts.unsigned_short`
- `ts.int`
- `ts.signed_int`
- `ts.unsigned_int`
- `ts.long`
- `ts.signed_long`
- `ts.unsigned_long`
- `ts.void`

The .ptr and .array operator allow to create new types on the fly. If you want to create a C type "array of 10 pointers to int" you simply run `ts.int.ptr.array(10)`.

3.1.3 Proxy Operators

Apart of the .val attribute the following attributes/operators are available in the python proxy to simulate C operators:

Python Attribute/Operator	C Operator	Sample
`<proxy>.adr`	`& <var>`	Returns a pointer proxy to the given <var>
`<ptr-proxy>.ref`	`* <ptr>`	Resolves the pointer <ptr> and returns the proxy object referred by <ptr>
`<array/ptr-proxy>[<ndx>]`	`<array/ptr>[<ndx>]`	Returns a Proxy that corresponds to the <ndx>th array/pointer element
`<struct-proxy>.<membername>` `<struct-proxy>["<membername>"]`	`<struct>.<member>`	Returns a proxy to a member of a struct/union. If <membername> conflicts with a python buildin-name the "[]" operator can be used instead of the "." operator.
`<proxy>.sizeof` | `sizeof(<var>)` | returns the size of the Object | proxy in bytes		
`<ctype-proxy>(<proxy>)` | `<ctype>(<cobj>)` | casts a object to another type by | creating a new one.		

3.1.4 Direct Memory Access

A further way of accessing the underlying C data object of a proxy is doing direct memory access instead of the C representation of the data. For example when creating a C `int` on a 32bit architecture, 4 bytes of memory will be reserved. These 4 bytes can be accessed bytewise via the `.mem` attribute. The `mem` attribute returns a `CMemory` object, which can be used to read and write any part of the underlying memory as `bytes` object via the slice operator. I.e. `i.mem[:2]` returns the first two bytes of the underlying memory.

As assignment of a `ByteLike` object to `.mem` or comparison of a `ByteLike` object to `.mem` are very common actions, the following simplifications are allowed for convenience:

- `i.mem = b'test'` instead of `v = b'test'; i.mem[:len(v)] = v`
- `i.mem == b'test'` instead of `v = b'test'; i.mem[:len(v)] == v`

3.2 The Testsetup

One core concept of headlock is the *testsetup*. A *testsetup* covers:

- module(s) under test (MUT): one or multiple C file(s).
- custom C macro definitions, that were applied when compiling the C code
- python mocks that emulates missing C modules on which the MUT is relying
- an interface to the MUT's implementations (functions, global variables) as *C/Python Proxy Objects* and its interface (typedefs, struct/enum/union declarations) as *C Type Objects*. Of course also macro defintions are available as far is they do not use too much preprocessor magic.

- **[PLANNED]** an environment (after instantiation) where the modules under test are executed. This is usually a separate process to avoid that buggy C code is interfering with the python code or the other C modules. But an environment might even be another machine or embedded processor.

One may define any number of testsetups per python file. Even of the same C file but i.e. with different preprocessor settings or different mocks. Furthermore one may instantiate every testsetup multiple times (even in parallel for example to simulate a network **[PLANNED]**).

In headlock a testsetup is represented by a python class which is derived (direct or indirect) from `headlock.testsetup.TestSetup`. It is even possible to use `headlock.testsetup.TestSetup` directly as playground (where no C code is required, but only to to interact with the testsetups environment/address space):

```
from headlock.testsetup import TestSetup

ts = TestSetup()
ptr = ts.char.ptr(b'HELLO WELT\0')     # create buffer with HELLO WORLD
print(ptr.c_str)                       # print content of this buffer
```

The decorator `headlock.testsetup.CModule()` adds one or multiple Modules Under Tests (C-files) to a testsetup. Headlock implements this by deriving the decorated class into a class of the same name. This new class contains all functions, globals variables, types, defines of the C modules:

- All `typedef` (as well as the buildin C-types) are added as *C Type Objects*. These python classes can be used to retrieve information about the type. As soon as the C-code is loaded these proxies can also be used to instantiate C objects of the corresponding type.
- struct/union/enum custom types are handled the same way as typedefs. But as they are in a different namespace (like in C). This is why they are not directly attributes of the testsetup object but have to be accessed via `.enum.<name>`, `.struct.<name>` or `.union.<name>`).
- Preprocessor defines are translated to python variables / functions when possible. If the C code cannot be translated to python code (i.e. due to preprocessor magic) the testsetup will nevertheless be created. But when accessing the corresponding macro a `ValueError` will be thrown.
- Global variables and functions are available as *C/Python Proxy Objects* as soon as the C-code is loaded.

After class creation one has access to every preprocessor define. Via the *C Type Objects* of typedefs, global variables and functions one can even do introspection of C modules without instantiating the testsetup class!

When instantiating a testsetup object the first time it will automaticially compile and link all C modules referred in CModules. For further instantiations the built binary will be reused (until the python script is restarted):

```
from headlock.testsetup import TestSetup, CModule

@CModule('module1.c', '../module2.c', MACRO1=1, MACRO2=None)
class TSSample(TestSetup):
    pass

ts = TSSample()
print(ts.mod1_var.c_str())    # content of module1's global var mod1_var
ts.__unload__()     # this call is not needed but recommended
```

In the above sample `module1.c` and `../module2.c` are compiled with the command line parameter "-DMACRO1=1" and "-DMACRO2". Please note that all C-file paths are relative to the directory of the python file of the testsetup. In this example the pyton file resides in a subdirectory of the corresponding C module (due to the directory prefix `..`).

When the testsetup object is not needed any more it is recommended to run `headlock.testsetup.TestSetup.__unload__()`. Although the `__del__()` method will call the `__unload__()` method implicitly, this is not a guaranteed approch.

Testsetup derived classes that want to run initialization routines that are relying on C code/mocks should **not** run this code in the `__init__()` method but use the `__startup__()` method for this purpose. This ensures that

- the testsetup is *comletely* initialized when the first C routine is called.
- running the initialization routines can be separated/delayed on demand

As `__startup__()` can be used to add custom initialization code, `__shutdown__()` should be used to run custom deinitialization code. It will be called implicitly by `__unload__()` if there was a call to `__startup__()` and the `__shutdown__()` method was not called already before.

For convenience it is possible to use the testsetup as contextmanager (which returns the testsetup object itself). This contextmanger ensures that `__startup__()` is called at the beginning of the context (if not called already) and `__unload__()` at the end:

```
with TSSample() as ts:
    # ts __startup__ was called and __unload__ will be called
```

Note: By convention testsetup classnames always start with the letters `TS`. Furthermore headlock uses the name `ts` everywhere a instantiated testsetup is referred. This is also the case for the `self` parameter of methods of the testsetup! This is also the case for this documentation.

3.3 Bridging Function Calls Between C And Python

One of the main goals of headlock is to provide seamless integration of calling C functions from python and vice versa. Therefore calling C functions from python is as simple as calling python functions. The same is true if the C code calls a function that is not part of the testsetup (the C modules of your testsetup relies on a C modules which is not included into the testsetup).

3.3.1 Calling C Functions From Python

Headlock will add *C/Python Proxy Objects* for all C functions implemented in the MUT of the testsetup. Anyone of these proxy object is a callable that accepts proxy objects as parameters and forwards them to to the underlying C function. As it knows all parameter types and the return type (see chapter *C Type Objects*) it ensures that all passed parameters that are not proxy objects are casted automaticially to a corresponding temporary proxy object that lives for the time of the function call. If casting is not possible or a proxy of wrong type is passed a TypeError/ValueError is raised. This implies that the rules for casting are identical the the convertion rules for the .val attribute (see chapter *C/Python Proxy Objects*)

For example when calling a function that gets an pointer to a list of integers one can simply write:

```
ts.c_func([1, 2, 3])
```

This will cast the list `[1, 2, 3]` to an int array proxy object by passing it to the C type object of the first parameter. The C type object will create a proxy object including allocating and initializing the corresponding C int array. (A Pointer to) this C int array will be passed to the C function. After the function returned the created proxy object will be released and thus the C int array will also be released. This works also, if `c_func` requires a more complex data structure (i.e. an array of array of structs).

The above example is very convenient but can be only used for input parameters. To return data from the C function via output parameters (pointers) an already existing proxy object has to be passed. Otherwise the returned value will be destroyed immediately after the C function returned. The following code demonstrates how to return an integer via parameter:

```
int_obj = ts.int()          # create integer that will hold the returned value
ts.c_func(int_obj.ptr)      # pass pointer to this int to c_func
assert int_obj == 123       # process the returned value
```

The object returned on calling a function proxy object is always a proxy object. The only expection are functions of return type `void`, in which case `None` is returned.

3.3.2 Function Pointers

Function pointers can wrap either C function or python callables. As for data pointers there is a C data object and a python proxy object for every function pointer. Thus a function pointer allows both directions of bridging:

- Pointer to C functions

  Apart from running `func_ptr = ts.func.ptr` to get a pointer proxy to a C function‘‘func‘‘ one can instantiate the function pointer type object by passing the address of a C function to it. Usually the latter is not done explicitly but implicitly when receiving a function pointer object from the C code (i.e. `func_ptr = ts.get_pointer_to_func()`, where `get_pointer_to_func` is a C function that returns a function pointer).

  The returned proxy is a callable, that can be called from python like any function object. Of course it can be also passed to any C function or C global variable that requires a function pointer (as the .val attribute corresponds to the address which is passed to the function/variable then)

- Pointer to Python functions

  Get a function pointer that refers to a python functions and is callable from c is as simple as passing the python callable to a function pointer type object. This will create a C wrapper, that can be passed around to C functions/variables. When the wrapper is called from C it will bridge the call to the python callable.

  Please note that all parameters passed to the C wrapper are encapsulated into python proxies by the bridge. This means when the python callable which is backing the proxy object is called, all its parameters are python proxy objects. The return value of the python callable will be also cast automaticially to a python proxy object if it returns a standard python object instead of a matching proxy.

  Attention: In contrary functiom pointer proxies of C functions this case requires more careful resource management. The reason is, that the proxy object for python functions creates (and manages) the wrapper object that bridges between C and python. Thus you have to ensure the proxy object is not released (aka keep at least one reference to it) as long as a C function could call the wrapper.

3.3.3 Mocking C Function

When creating a testsetup with C modules that refers to other C modules which are **not** part of the testsetup headlock will automaticially creating function stubs. These stubs are very useful as they ensure that the testsetup compiles (although C function implementations are missing). In fact a stub will be generated for *any* non-static function declaration where no corresponding function implementation is found.

This works out already perfectly as long as no C or python function is calling the stubbed C functions. This is required in tests that are testing parts of the Module Under Test for which the stubbed functions are not relevant).

If the Module Under Tests requires one of the stubbed functions it is as simple as adding a python function with the same name but the postfix “_mock” to the testsetup class. Every call to the corresponding C function is bridged to this python function. As done when calling Python *Function Pointers* all parameters are wrapped into the corresponding

proxy objects when calling a mock function. The same rule applies to the return value of the mock function, before being passed back to the C function.

Note: Please note that for every mocked C function there are two function objects in the testsetup. One with the exactly same name as the C function and one with the postfix `_mock`. The first one can be used to call the latter one from python ensuring that the same automatic proxy encapsulation rules apply as when calling the function from C.

I.e. given the function signature `int func(int)`. When calling `ts.func(1)` will forward the to `ts.func_mock(ts.int(1))`. If this one returns `2` the result will be encapsulated into `ts.int(2)` before being returned by `ts.func`. This guarantees that when testing the mock functions from python they are working the same way as when being called from C.

What is making headlocks mocking really powerful is the fact, that there is no special magic bit it is totally conform to pythons semantic for methods. This means:

- you can derive the Testsetup class and add/overwrite mocked methods. The derived testsetup is usable like an independant testsetup (different mock functionality although same C module):

```
@CModule('test.c')
class TSTest(TestSetup):
    def test_func_mock(ts, param1, param2):
        return param1 + param2

class TSTest2(TSTest):
    def test_func_mock(ts, *params):
        return 99
```

- you can mixin Mock-classes, that implements mocked functions for a specific underlying C module this is not included in the testsetup but required by the MUT. This mixing could be reused for all testsetups relying on the underlying C module.

```
class UnderlyingModuleMock(TestSetup):
    def test_func_mock(ts, param1, param2):
        return param1 + param2

@CModule('test.c')
class TSTest(UnderlyingModuleMock, TestSetup):
    pass
```

- you can even add/replace mock functions after testsetup instantiation. I.e. one can utilitize the powerful `unittest.mock` Module:

```
from unittes.mock import Mock

@CModule('test.c')
class TSTest(UnderlyingModuleMock, TestSetup):
    pass

ts = TSTest()
ts.test_func_mock = Mock(return_value=99)
ts.c_func_that_calls_test_func()
ts.test_func_mock.assert_called_once_with(3, 4)
```

CHAPTER 4

Reference

TBD

CHAPTER 5

Current Status Of Development

Attention: The currently implementation of headlock is an alpha version. Although it is already in production use at http://www.baltech.de it must be noted that the **the API is not stable yet**!

The current status of the project (preliminary limitations/not yet implemented features) is shown in the following list:

- Does not work on macOS
- Works only with GCC/Mingw64
- Requires LLVM
- Does not support specifying packing of structures in C sources (`#pragma pack`). As workaround it is possible to specify packing on a per-C-file basis in the Testsetup.
- No Support yet for: * enum * union * float/double * calling/mocking inline functions
- Does not support running testsetups on external process / other machine / embdedded system

5.1 Bugreports/Pullrequests

Bugreports/Pullrequests can be provided via the page of GitHub Page of the project