cfficloak Documentation

Release 0.1

Andrew Leech

Aug 09, 2016

Contents

CFFIwrap is a set of convenience functions and wrapper classes designed to make writing CFFI modules less tedious. The source tree contains a single python module called cfficloak.py. Simply install cfficloak with: Install with > pip install cfficloak or > pip install https://github.com/andrewleech/cfficloak/archive/master.zip For more examples take a look in the tests directory.

This project is licensed under the Apache License, version 2.0.

CHAPTER 1

Module documentation

A collection of convenience classes and functions for CFFI wrappers.

class `cfficloak.`**`CFunction`**(*ffi*, *cfunc*)

Bases: `object`

Adds some low-ish-level introspection to CFFI C functions.

Most other wrapper classes and fuctions expect API functions to be wrapped in a CFunction. See `wrapall()` below.

- `ffi`: The FFI object the C function is from.
- `cfunc`: The C function object from CFFI.

Attributes added to instances:

- `cfunc`: The C function object.
- `ffi`: The FFI object the C function is from.
- `typeof`: ffi.typeof(cfunc)
- `cname`: From typeof.
- `args`: From typeof.
- `kind`: From typeof.
- `result`: From typeof.

Callable: when called, the cfunc is called directly and it's result is returned. See `cmethod` for more uses.

`checkerr`(*cfunc*, *args*, *retval*)

Default error checker. Checks for NULL return values and raises NullError.

Can be overridden by subclasses. If `_checkerr` returns anything other than `None`, that value will be returned by the property or method, otherwise original return value of the C call will be returned. Also useful for massaging returned values.

`get_arrayptr`(*array*, *ctype=None*)

Get a CFFI compatible pointer object for an array.

Supported `array` types are:

- numpy ndarrays: The pointer to the underlying array buffer is cast to a CFFI pointer. Value returned from __call__ will be a pointer, but the numpy C buffer is updated in place, so continue to use the numpy ndarray object.

- CFFI CData pointers: If the user is already working with C arrays (i.e., `ffi.new("int[10]")`) these will be returned as given.
- Python ints and longs: These will be interpretted as the length of a newly allocated C array. The pointer to this array will be returned. `ctype` must be provided (CFunction's __call__ method does this automatically).
- Python collections: A new C array will be allocated with a length equal to the length of the iterable (`len()` is called and the iterable is iterated over, so don't use exhaustable generators, etc). `ctype` must be provided (CFunction's __call__ method does this automatically).

class `cfficloak.`**`CStruct`**(*ffi*, *struct*)

Bases: `object`

Provides introspection to an instantiation of a CFFI ``StructType``s and ``UnionType``s.

Instances of this class are essentially struct/union wrappers. Field names are easily inspected and transparent conversion of data types is done where possible.

Struct fields can be passed in as positional arguments or keyword arguments. `TypeError` is raised if positional arguments overlap with given keyword arguments.

The module convenience function `wrapall` creates `CStructs` for each instantiated struct and union imported from the FFI.

`enable_network_endian_translation`()

`set_py_converter`(*key*, *fn*)

class `cfficloak.`**`CUnion`**(*ffi*, *uniontype*)

Bases: *`cfficloak.CStruct`*

class `cfficloak.`**`CStructType`**(*ffi*, *structtype*)

Bases: `object`

Provides introspection to CFFI ``StructType``s and ``UnionType``s.

Instances have the following attributes:

- `ffi`: The FFI object this struct is pulled from.
- `cname`: The C name of the struct.
- `ptrname`: The C pointer type signature for this struct.
- `fldnames`: A list of fields this struct has.

Instances of this class are essentially struct/union generators. Calling an instance of `CStructType` will produce a newly allocated struct or union.

Struct fields can be passed in as positional arguments or keyword arguments. `TypeError` is raised if positional arguments overlap with given keyword arguments.

Arrays of structs can be created with the `array` method.

The module convenience function `wrapall` creates `CStructTypes` for each struct and union imported from the FFI.

`array`(*shape*)

Constructs a C array of the struct type with the given length.

- `shape`: Either an int for the length of a 1-D array, or a tuple for the length of each of len dimensions. I.e., [2,2] for a 2-D array with length 2 in each dimension. Hint: If you want an array of pointers just add an extra demension with length 1. I.e., [2,2,1] is a 2x2 array of pointers to structs.

No explicit initialization of the elements is performed, however CFFI itself automatically initializes newly allocated memory to zeros.

class `cfficloak.`**`CUnionType`**(*ffi*, *uniontype*)

Bases: *`cfficloak.CStructType`*

class `cfficloak.`**`CObject`**(**args*, ***kwargs*)

Bases: `object`

A pythonic representation of a C "object"

Usually representing a set of C functions that operate over a common peice of data. Many C APIs have lots of functions which accept some common struct pointer or identifier int as the first argument being manipulated. CObject provides a convenient abstrtaction to making this convention more "object oriented". See the example below. More examples can be found in the cfficloak unit tests.

Use `cproperty` and `cmethod` to wrap CFFI C functions to behave like instance methods, passing the instance in as the first argument. See the doc strings for each above.

For C types which are not automatically coerced/converted by CFFI (such as C functions accepting struct pointers, etc) the subclass can set a class- or instance-attribute named `_cdata` which will be passed to the CFFI functions instead of `self`. The CObject can also have a `_cnew` static method (see `cstaticmethod`) which will be called by the base class's `__init__` and the returned value assigned to the instance's `_cdata`.

For example:

libexample.h:

```
typedef int point_t;
point_t make_point(int x, int y);
int point_x(point_t p);
int point_y(point_t p);
int point_setx(point_t p, int x);
int point_sety(point_t p, int y);
int point_move(point_t p, int x, int y);

int point_x_abs(point_t p);
int point_movex(point_t p, int x);
```

Python usage (where `libexample` is an API object from `ffi.verify()`):

```
>>> from cfficloak import CObject, cproperty, cmethod, cstaticmethod
>>> class Point(CObject):
...     x = cproperty(libexample.point_x, libexample.point_setx)
...     y = cproperty(libexample.point_y, libexample.point_sety)
...     _cnew = cstaticmethod(libexample.make_point)
...
>>> p = Point(4, 2)
>>> p.x
4
>>> p.x = 8
>>> p.x
8
>>> p.y
2
```

You can also specify a destructor with a `_cdel` method in the same way as `_cnew`.

Alternatively you can assign a CFFI compatible object (either an actual CFFI CData object, or something CFFI automatically converts like and int) to the instance's _cdata attribute.

`cmethod` wraps a CFunction to provide an easy way to handle ‘output’ pointer arguments, arrays, etc. (See the `cmethod` documentation.):

```
>>> class Point2(Point):
...     move = cmethod(libexample.point_move)
...
>>> p2 = Point2(8, 2)
>>> p2.move(2, 2)
0
>>> p2.x
10
>>> p2.y
4
```

If _cdata is set, attributes of the cdata object can also be retrieved from the CObject instance, e.g., for struct fields, etc.

libexample cdef:

```
typedef struct { int x; int y; ...; } mystruct;
mystruct* make_mystruct(int x, int y);
int mystruct_x(mystruct* ms);
```

python:

```
>>> class MyStruct(CObject):
...     x = cproperty(libexample.mystruct_x)
...     _cnew = cstaticmethod(libexample.make_mystruct)
...
>>> ms = MyStruct(4, 2)
>>> ms.x  # Call to mystruct_x via cproperty
4
>>> ms.y  # direct struct field access
2
```

Note: stack-passed structs are not supported yet* but pointers to structs work as expected if you set the `_cdata` attribute to the pointer.

- https://bitbucket.org/cffi/cffi/issue/102

exception `cfficloak.`**`NullError`**

Bases: `exceptions.Exception`

`cfficloak.`**`cmethod`**(*cfunc=None*, *outargs=()*, *inoutargs=()*, *arrays=()*, *retargs=None*, *checkerr=None*, *doc=None*)

Wrap cfunc to simplify handling outargs, etc.

This feature helps to simplify dealing with pointer parameters which are meant to be “return” parameters. If any of these are specified, the return value from the wrapper function will be a tuple containing the actual return value from the C function followed by the values of the pointers which were passed in. Each list should be a list of parameter position numbers (0 for the first parameter, etc)..

- `outargs`: These will be omitted from the cmethod-wrapped function parameter list, and fresh pointers will be allocated (with types derived from the C function signature) and inserted in to the arguments list to be passed in to the C function. The pointers will then be dereferenced and the value included in the return tuple.
- `inoutargs`: Arguments passed to the wrapper function for these parameters will be cast to pointers before being passed in to the C function. Pointers will be unboxed in the return tuple.
- `arrays`: Arguments to these parameters can be python lists or tuples, numpy arrays or integers.

  - –Python lists/tuples will be copied in to newly allocated CFFI arrays and the pointer passed in. The generated CFFI array will be in the return tuple.
  - –Numpy arrays will have their data buffer pointer cast to a CFFI pointer and passed in directly (no copying is done). The CFFI pointer to the raw buffer will be returned, but any updates to the array data will also be reflected in the original numpy array, so it's recommended to just keep using that. (TODO: This behavior may change to remove these CFFI pointers from the return tuple or maybe replace the C array with the original numpy object.)
  - –Integers will indicate that a fresh CFFI array should be allocated with a length equal to the int and initialized to zeros. The generated CFFI array will be included in the return tuple.
- `retargs`: (Not implemented yet.) A list of values to be returned from the cmethod-wrapped function. Normally the returned value will be a tuple containing the actual return value of the C function, followed by the final value of each of the `outargs`, `inoutargs`, and `arrays` in the order they appear in the C function's paramater list.
- `doc`: Optional string/object to attach to the returned function's docstring

As an example of using `outargs` and `inoutargs`, a C function with this signature:

```
``int cfunc(int inarg, int *outarg, float *inoutarg);``
```

with an `outargs` of `[1]` and `inoutargs` set to `[2]` can be called from python as:

```
>>> wrapped_cfunc = cmethod(cfunc, outargs=[1], inoutargs=[2])
>>> ret, ret_outarg, ret_inoutarg = wrapped_cfunc(inarg, inoutarg)
```

Returned values will be unboxed python values unless otherwise documented (i.e., arrays).

`cfficloak.`**`cstaticmethod`**(*cfunc*, ***kwargs*)

Shortcut for staticmethod(cmethod(cfunc, [kwargs ...]))

`cfficloak.`**`cproperty`**(*fget=None*, *fset=None*, *fdel=None*, *doc=None*, *checkerr=None*)

Shortcut to create `cmethod` wrapped `property`s.

E.g., this:

```
>>> class MyCObj(CObject):
...     x = property(cmethod(get_x_cfunc), cmethod(set_x_cfunc))
```

becomes:

```
>>> class MyCObj(CObject):
...     x = cproperty(get_x_cfunc, set_x_cfunc)
```

If you need more control of the outargs/etc of the cmethods, stick to the first form, or create and assign individual cmethods and put them in a normal property.

`cfficloak.`**`wrap`**(*ffi*, *cobj*)

Convenience function to wrap CFFI functions structs and unions.

`cfficloak.`**`wrapall`**(*ffi*, *api*)

Convenience function to wrap CFFI functions structs and unions.

Reads functions, structs and unions from an API/Verifier object and wrap them with the respective wrapper functions.

- `ffi`: The FFI object (needed for it's `typeof()` method)
- `api`: As returned by `ffi.verify()`

Returns a dict mapping object names to wrapper instances. Hint: in a python module that only does CFFI boilerplate and verification, etc, try something like this to make the C values available directly from the module itself:

```
globals().update(wrapall(myffi, myapi))
```

`cfficloak.`**`wrapenum`**(*retval*, *enumTypeDescr*)

Wraps enum int in an auto-generated wrapper class. This is used automatically when cmethod() returns an enum type :param retval: integer :param enumTypeDescr: the cTypeDescr for the enum :return: subclass of Enum

`cfficloak.`**`carray`**(*items_or_size=None*, *size=None*, *ctype='int'*)

Convenience function for creating C arrays.

`cfficloak.`**`nparrayptr`**(*nparr*, *offset=0*)

Convenience function for getting the CFFI-compatible pointer to a numpy array object.

CHAPTER 2

Indices and tables

- genindex
- search

Python Module Index

Index