Wrapping a C library in Lua | John's Blog

Introduction

An often overlooked feature of Lua is the C API. The most common use I've seen is to allow Lua scripts to use existing C libraries instead of reimplementing existing functionality in pure Lua (which is not always feasible). Fortunately, Lua has very strong integration with C and while not trivial, wrapping a C library is fairly straight foreword. That said Lua's C API isn't specific to this task, it's just a common use of the C API. For example, another use is to write performance critical code in C instead of Lua.

The C API has it's own way of doing things. Specifically using a virtual stack to marshal data between C and Lua. The stack is a bit odd to work with at first but efficient and highly necessary.

Since Lua's C API is robust there are a few different way to wrap a C library. I'm going to cover one way that I've found to be easy to understand for a C programmer. In this example I'm going to compile my C "library" directly into my Lua wrapper. Technically it's not wrapping in the sense that I'm not using a separate library with my C example code but the idea is the same. I'm treating everything as one project to focus on the wrapping. This isn't instructional in regard to linking libraries on different platforms.

The C Library We Want to Wrap

The C Library I'm using is a simple counter. You create a counter object with a given starting value. You can add, subtract, increment, decrement and get the current value. Once done you can/need to destroy the object.

counter.h

```
1  #ifndef __COUNTER_H__
2  #define __COUNTER_H__
3
4  struct counter;
5  typedef struct counter counter_t;
6
7  counter_t *counter_create(int start);
8  void counter_destroy(counter_t *c);
9
```

```
10  void counter_add(counter_t *c, int amount);
11  void counter_subtract(counter_t *c, int amount);
12
13  void counter_increment(counter_t *c);
14  void counter_decrement(counter_t *c);
15
16  int counter_getval(counter_t *c);
17
18  #endif /* __COUNTER_H__ */
```

The counter object is intended to be an opaque pointer with its data hidden. The counter's data is just an integer in a struct at this point. While this might not be the most efficient way to handle an int, this demonstrates working with complex (often opaque) types which typically have multiple data members.

counter.c

```
 1  #include <stdlib.h>
 2  #include "counter.h"
 3
 4  struct counter {
 5      int val;
 6  };
 7
 8  counter_t *counter_create(int start)
 9  {
10      counter_t *c;
11
12      c = malloc(sizeof(*c));
13      c->val = start;
14
15      return c;
16  }
17
18  void counter_destroy(counter_t *c)
19  {
20      if (c == NULL)
21          return;
22      free(c);
23  }
24
25  void counter_add(counter_t *c, int amount)
26  {
27      if (c == NULL)
28          return;
29      c->val += amount;
30  }
```

```
31
32  void counter_subtract(counter_t *c, int amount)
33  {
34      if (c == NULL)
35          return;
36      c->val -= amount;
37  }
38
39  void counter_increment(counter_t *c)
40  {
41      if (c == NULL)
42          return;
43      c->val++;
44  }
45
46  void counter_decrement(counter_t *c)
47  {
48      if (c == NULL)
49          return;
50      c->val--;
51  }
52
53  int counter_getval(counter_t *c)
54  {
55      if (c == NULL)
56          return 0;
57      return c->val;
58  }
```

The Wrapper

Now that we have our C object we need to wrap it in the Lua C API so we can access it from Lua.

Since Lua's C API uses a stack for passing values we need to wrap each public function for our library. Keep in mind there are ways to automate this task and ways code duplication in the below example can be reduced.

Essentially what happens is, you create a user data object which will hold a pointer to the allocated counter object. The user data is returned to Lua and when Lua calls a wrapper function the user data will be passed to the wrapper C function.

The below wrapper does a bit more than simply wrapping the C library. It extends it a bit by adding a name to the counter. This isn't required but an example of how a wrapper can be more complex than a one to one mapping.

wrap.c

```c
#include <stdlib.h>
#include <string.h>

#include <lua.h>
#include <lauxlib.h>

#include "counter.h"

/* Userdata object that will hold the counter and name. */
typedef struct {
    counter_t *c;
    char      *name;
} lcounter_userdata_t;

static int lcounter_new(lua_State *L)
{
    lcounter_userdata_t *cu;
    const char          *name;
    int                  start;

    /* Check the arguments are valid. */
    start = luaL_checkint(L, 1);
    name  = luaL_checkstring(L, 2);
    if (name == NULL)
        luaL_error(L, "name cannot be empty");

    /* Create the user data pushing it onto the stack. We als
     * the member of the userdata in case initialization fai
     * that happens we want the userdata to be in a consiste
    cu       = (lcounter_userdata_t *)lua_newuserdata(L, size
    cu->c    = NULL;
    cu->name = NULL;

    /* Add the metatable to the stack. */
    luaL_getmetatable(L, "LCounter");
    /* Set the metatable on the userdata. */
    lua_setmetatable(L, -2);

    /* Create the data that comprises the userdata (the count
    cu->c    = counter_create(start);
    cu->name = strdup(name);

    return 1;
}

static int lcounter_add(lua_State *L)
```

```
47  {
48      lcounter_userdata_t *cu;
49      int                  amount;
50
51      cu     = (lcounter_userdata_t *)luaL_checkudata(L, 1, "LC
52      amount = luaL_checkint(L, 2);
53      counter_add(cu->c, amount);
54
55      return 0;
56  }
57
58  static int lcounter_subtract(lua_State *L)
59  {
60      lcounter_userdata_t *cu;
61      int                  amount;
62
63      cu     = (lcounter_userdata_t *)luaL_checkudata(L, 1, "LC
64      amount = luaL_checkint(L, 2);
65      counter_subtract(cu->c, amount);
66
67      return 0;
68  }
69
70  static int lcounter_increment(lua_State *L)
71  {
72      lcounter_userdata_t *cu;
73
74      cu = (lcounter_userdata_t *)luaL_checkudata(L, 1, "LCount
75      counter_increment(cu->c);
76
77      return 0;
78  }
79
80  static int lcounter_decrement(lua_State *L)
81  {
82      lcounter_userdata_t *cu;
83
84      cu = (lcounter_userdata_t *)luaL_checkudata(L, 1, "LCount
85      counter_decrement(cu->c);
86
87      return 0;
88  }
89
90  static int lcounter_getval(lua_State *L)
91  {
92      lcounter_userdata_t *cu;
93
94      cu = (lcounter_userdata_t *)luaL_checkudata(L, 1, "LCount
```

```
95      lua_pushinteger(L, counter_getval(cu->c));
96
97      return 1;
98  }
99
100 static int lcounter_getname(lua_State *L)
101 {
102     lcounter_userdata_t *cu;
103
104     cu = (lcounter_userdata_t *)luaL_checkudata(L, 1, "LCount
105     lua_pushstring(L, cu->name);
106
107     return 1;
108 }
109
110 static int lcounter_destroy(lua_State *L)
111 {
112     lcounter_userdata_t *cu;
113
114     cu = (lcounter_userdata_t *)luaL_checkudata(L, 1, "LCount
115
116     if (cu->c != NULL)
117         counter_destroy(cu->c);
118     cu->c = NULL;
119
120     if (cu->name != NULL)
121         free(cu->name);
122     cu->name = NULL;
123
124     return 0;
125 }
126
127 static int lcounter_tostring(lua_State *L)
128 {
129     lcounter_userdata_t *cu;
130
131     cu = (lcounter_userdata_t *)luaL_checkudata(L, 1, "LCount
132
133     lua_pushfstring(L, "%s(%d)", cu->name, counter_getval(cu
134
135     return 1;
136 }
137
138 static const struct luaL_Reg lcounter_methods[] = {
139     { "add",          lcounter_add       },
140     { "subtract",     lcounter_subtract  },
141     { "increment",    lcounter_increment },
142     { "decrement",    lcounter_decrement },
```

```
143         { "getval",      lcounter_getval    },
144         { "getname",     lcounter_getname   },
145         { "__gc",        lcounter_destroy   },
146         { "__tostring",  lcounter_tostring  },
147         { NULL,          NULL               },
148     };
149
150     static const struct luaL_Reg lcounter_functions[] = {
151         { "new", lcounter_new },
152         { NULL,  NULL         }
153     };
154
155     int luaopen_lcounter(lua_State *L)
156     {
157         /* Create the metatable and put it on the stack. */
158         luaL_newmetatable(L, "LCounter");
159         /* Duplicate the metatable on the stack (We know have 2)
160         lua_pushvalue(L, -1);
161         /* Pop the first metatable off the stack and assign it to
162          * of the second one. We set the metatable for the table
163          * This is equivalent to the following in lua:
164          * metatable = {}
165          * metatable.__index = metatable
166          */
167         lua_setfield(L, -2, "__index");
168
169         /* Set the methods to the metatable that should be acces
170         luaL_setfuncs(L, lcounter_methods, 0);
171
172         /* Register the object.func functions into the table that
173          * stack. */
174         luaL_newlib(L, lcounter_functions);
175
176         return 1;
177     }
```

The Wrapper Explained

User Data

Lets break the wrapper code down a bit and start by looking at the lcounter_userdata_t struct.

```
1   ...
2   typedef struct {
```

```
3       counter_t *c;
4       char      *name;
5   } lcounter_userdata_t;
6   ...
```

We have a struct which stores a pointer to our counter object. It also stores a pointer to our name. If we didn't have the name and we were only referencing the counter we could have the user data be a pointer to the counter object instead.

We create the lcounter_userdata_t object (which will hold pointers to our counter_t and name data) using the lua_newuserdata function. This will allocate memory to hold the user data object and this memory will be tracked by Lua as part of Lua's garbage collection system. We are creating a full user data object which is different from light user data. We want to use full user data because we can associate it with a metatable so we can verify the data type passed to our wrapper functions. Light user data cannot have a metatable associated with it. Meaning, when using light user data all we get/know (passed to a function) is a memory address which could be any arbitrary memory address (or object). This can cause all sorts of nasty behavior if someone were to abuse or misuse our wrapper.

The names full and light are somewhat misleading. The name for light user data being light implies that full is heavy. This isn't so. Full user data isn't heavy at all and in most cases is what should be used.

Wrapped Object Creation

Lets look at the lcounter_new function:

```
1   ...
2   static int lcounter_new(lua_State *L)
3   {
4   ...
5       /* Check the arguments are valid. */
6       start = luaL_checkint(L, 1);
7       name  = luaL_checkstring(L, 2);
8       if (name == NULL)
9           luaL_error(L, "name cannot be empty");
```

We start by pulling the arguments off of the stack using luaL_check*. This will check the type, check if set, and either return the argument or error out of the function entirely.

On error Lua uses longjmp as a sort of exception handler. The current function

won't continue running in case of error. Instead Lua will use longjmp to switch to an error handler function. It's very important to keep this in mind.

```
/* Create the user data pushing it onto the stack. We also pre
 * the member of the userdata in case initialization fails in 
 * that happens we want the userdata to be in a consistent sta
cu       = (lcounter_userdata_t *)lua_newuserdata(L, sizeof(*cu
cu->c    = NULL;
cu->name = NULL;
```

Now we create the user data object and add it to the stack. As the comment says we pre-initalize it with known values. Many Lua functions can fail and will prevent the current function from completing. We need the user data to be in a state that won't cause a crash if this happens before we actually set its data.

```
    /* Add the metatable to the stack. */
    luaL_getmetatable(L, "LCounter");
    /* Set the metatable on the userdata. */
    lua_setmetatable(L, -2);

    /* Create the data that comprises the userdata (the counte
    cu->c    = counter_create(start);
    cu->name = strdup(name);

    return 1;
}
...
```

Finally, we add the metatable (we create this during initialization later in the code) to the user data so we can verify it as well as call it's object:func functions. We then create the counter object, set the name and return telling Lua that we have placed one item on the stack.

The Wrapped Functionality Functions

Lets look at the other wrapped functions. Specifically, lcounter_add:

```
static int lcounter_add(lua_State *L)
{
    lcounter_userdata_t *cu;
    int                  amount;

    cu     = (lcounter_userdata_t *)luaL_checkudata(L, 1, "LC
    amount = luaL_checkint(L, 2);
    counter_add(cu->c, amount);
```

```
 9
10        return 0;
11    }
```

One thing worth mentioning at this point is all of our functions take the form:

```
1   static int (*func)(lua_State *L);
```

Since Lua uses a stack for passing data between C and Lua, all C functions that are callable by Lua will use this form. The arguments from Lua will be placed on the stack in the order they are passed. Meaning the first argument is on the bottom with the second on top of that and so forth.

This leads into these two lines:

```
1   cu      = (lcounter_userdata_t *)luaL_checkudata(L, 1, "LCounte
2   ...
3   amount = luaL_checkint(L, 2);
```

Here we check that the bottom most argument (the first one) is user data with the LCounter meta table. Basically, we're checking that this is valid data for the function. Then we check that the next argument is an integer. We could have used -2 and -1 respectively to check these instead of 1, and 2 because we can index from the top using negative values. That said, it makes more sense to check the arguments using positive values (1 and 2) because that's the order they're given in.

Working with the stack I've found it helpful to keep this in mind. Arguments are placed on the stack; function is called. At the start of the function the stack will have the arguments with the first argument at the bottom. You can check the arguments using luaL_check*(L, #) to check the first and all subsequent arguments. It's easiest to check using positive values.

Then you do your processing, and place the results on the stack. Anything you place on the stack is placed at the top. It's easier to index using negative values at this point. In this particular function we don't add anything to the stack so we return 0.

The lcounter_get* functions on the other hand do add a return value to the stack. These only add one value to the stack but it's perfectly acceptable to add multiple return values to the stack because Lua fully supports multiple return values. You'd add the values then return the count instead of 0 or 1.

The key part of the lcounter_getval function is this line which pushes the integer count value onto the stack:

```
1   lua_pushinteger(L, counter_getval(cu->c));
```

Realize that the lcounter_tostring function is very similar to the lcounter_get* functions. lcounter_tostring is similar in regard to lcounter_getname in the fact that it is an extension provided by the wrapper and not a wrapped part of the C library. Unlike lcounter_getname and the whole name addition, lcounter_tostring makes the library more Lua like when using it in Lua code.

Memory Management

One major difference between Lua and C is memory management. The counter object has create and destroy functions. The destroy function must be called in order to clean up and free the memory used by the counter object. In this example the destroy function just frees memory but other libraries may need to do things like close files or network sockets.

Lua uses garbage collection to determine if an object is no longer being used and at that point destroys it. We don't want to force explicit memory management in Lua; so we need to ensure we have the counter destroy function called automatically when Lua garbage collects the object.

This situation is handled by setting the metamethod __gc to the wrapped destroy function. Now whenever the garbage collector runs and decides to clean up the object the destroy function will be called. This can be verified by putting a printf in counter_destroy.

We could have an explicit entry in the metatable for destroy (we don't) if we needed one. In this example we can simply rely on __gc to take care of cleaning up. If the object used file or network access we may want to provide an explicit destroy (or close) function. While we could rely on the garbage collector we shouldn't for certain resources (file and sockets).

If we did have an explicit destroy we need to ensure that it works without causing a crash (or other unexpected and unwanted behavior) if called twice. The garbage collector will still call the destroy function even if we manually call it. This is why we free then set to NULL the data objects within the user data struct. This ensure's we only free the data once. Also, we don't free the memory for the user data itself (we only worry about what's inside of it) because that was created by Lua and will be freed by Lua. It's a small amount of memory, sizeof(lcounter_userdata_t), so it's not critical if it stays around until Lua frees it.

Registering The Wrapper For Use By Lua

The following structs and function is what allows the Lua C wrapper functions to be usable by Lua.

```
1   static const struct luaL_Reg lcounter_methods[] = {
2   static const struct luaL_Reg lcounter_functions[] = {
3   int luaopen_lcounter(lua_State *L)
4   ...
```

The structs define the names used by Lua and the underlying C function that should be called. All of these C functions need the “int (*func)(lua_State *L)” setup. You can’t use any arbitrary C function.

There are two structs of functions. We’re going to register one to the Lua object (module) and the other to the object’s metatable. lcounter_functions is only the new function while lcounter_methods are the metatable functions. Remember the metatable functions need to be called using object:func or object.func(object). While the module functions don’t need the reference to themselves. This is why we have new outside of the metatable so it can create a new object.

int luaopen_lcounter(lua_State *L) is documented to the point it should be clear how it works. That said, the luaopen_lcounter name isn’t arbitrary. Lua doesn’t need a public header to load a Lua C module. It looks for the function luaopen_NAME. Where NAME is the name of the C library it’s opening. As long as the names match Lua will be able to find the function and load the Lua C module.

Building

CMakeLists.txt

```
1   cmake_minimum_required (VERSION 3.0)
2   project (lcounter C)
3
4   find_package(Lua REQUIRED)
5
6   include_directories (
7       ${CMAKE_CURRENT_BINARY_DIR}
8       ${CMAKE_CURRENT_SOURCE_DIR}
9       ${LUA_INCLUDE_DIR}
10  )
11
12  set (SOURCES
13      counter.c
14      wrap.c
15  )
```

```
16
17  add_library (${PROJECT_NAME} SHARED ${SOURCES} ${LUA_LIBRARIES
18  target_link_libraries (${PROJECT_NAME} lua)
19  set_target_properties (${PROJECT_NAME} PROPERTIES PREFIX "")
```

Building:

```
1  $ mkdir build
2  $ cd build
3  $ cmake ..
4  $ make
5  $ cd ..
```

Using Lua

Here is a Lua script that will use our lcounter.

counter_test.lua

```
1  lcounter = require("lcounter")
2
3  c = lcounter.new(0, "c1")
4  c:add(4)
5  c:decrement()
6  print("val=" .. c:getval())
7
8  c:subtract(-2)
9  c:increment()
10 print(c)
```

The script creates a counter, with an initial state of 0. It will add 4 then decrement bring the count to 3. We print the value (3). Then we subtract -2, which is really an addition of 2, and increment. This bring the final count to 6. We call print on the object which prints the object (name and value) by calling the __tostring metatable function.

Running:

```
1  $ LUA_CPATH="./build/?.dylib;./build/?.so;./build/?.dll" lua t
2  val=3
3  c1(6)
```

As we can see the wrapper has the correct values and __tostring works as expected. Note that I'm setting the LUA_CPATH explicitly in this example simply

because I’m not installing the binary into the system search path. Otherwise that wouldn’t be necessary.