The Lua Integration Guide

Marc Balmer marc@msys.ch

Table of Contents

The Lua Programming Language

C is hard to master. Pointers and memory management are just two of the many pitfalls. Lua, on the other hand, is very easy to learn, reasonably fast and comes with automatic memory management. Lua is the ideal language for the occasional programmer or a person who merely wants to change the behaviour of a program or extend an existing program.

This makes Lua an ideal scripting language.

Not everything can be written in Lua. A lot of software, especially libraries, is available only in C, or C is used to implement some low-level stuff like a binary protocol or direct hardware access.

Lua as an Orchestration Tool

Lua is ideal to orchestrate software parts written in C. Complex or computing intensive parts can be written in C, which Lua is used to drive the whole.

Lua as an Extension Language

Existing software can be changed in behaviour or be extended using Lua scripts. For this to happen, the software must be prepared for such extension.

Multilanguage Development

Programmers have a tendency of favoring a particular programming language and sometimes defend their choice with almost religious zeal. And then they try to solve every problem in that very language.

This is not always a good idea. Especially not when different people with different skill levels work together. Image a senior C programmer with years long of experience and the student who just started learning to program. Giving the experienced programmer C and the student Lua will make them both productive and let them collaborate. Multilanguage development bridges the skill and experience gap.

The art of Lua integration is to make available complex software, written in C, to Lua in a way a Lua programmer would expect it. Clean and clear syntax, using tables as the only data structure, automatic memory management, automatic cleaning up of ressources, and, using Lua paradigms.

The Lua Integration Guide Covers Lua 5.4

Lua 5.4 was released in june 2020. This version introduces new concepts to the language and the C API, like e.g. constant values, to-be-closed values, and, multiple uservalues. These new additions to the language can be helpful when integrating C code with Lua, so I decided to base the guide on Lua 5.4. Most concepts, patterns, and, best practices presented here, however, apply to older versions of Lua as well.

Making Software Scriptable

The crucial step to make software that is written in C scriptable with Lua is to integrate the Lua language into the software written in C.

Lua is a small library that the C program must be linked with. The C program, which we call the host program from now on, must then create one or more Lua states. A Lua state is an isolated Lua execution environment which means that if the host program creates more than one Lua state, these Lua states can not see each other nor can they exchange any data (unless the host program provides means for such data exchange.)

Linking With Lua

Link Lua to the host program in the usual way of your development environment.

It is assumed that the Lua library liblua.a resides in the path */usr/local/lua* and the Lua header files in */usr/local/lua/include*:

```
$ cc -o host host.c -I/usr/local/lua/include -L/usr/local/lua -llua
```

Creating a Lua State

In the host program, we must include the Lua header files.

```
#include <lualib.h> ①
#include <lauxlib.h> ②
```

① This is the main Lua header file.

② This is the auxiliary Lua header file, it provides macros that usually come in handy then working with the Lua C library.

Now anywhere in the host program we have to create a Lua state and keep a reference to it. The function that creates the Lua state returns a pointer to a lua_State variable, which we can store let's say in a global variable:

```
lua_State *L;

void
main(int argc, char *argv[])
{
    /* Create a Lua state */
    L = luaL_newstate(); ①

    /* Check the return value */
    if (L == NULL) {
        fprintf(stderr, "Lua: cannot initialize\n");
        return -1;
    }

}
```

① `luaL_newstate()` is actually a macro from lauxlib.h, making it very easy to create a new Lua state. The `lua_newstate()` function would require us to pass a memory allocation function.

Opening the Lua Libraries and Execute Lua Code

The newly created Lua state is completely empty, not even the Lua standard libraries are available. To make them available, we must call the `luaL_openlibs()` convenience function, again from lauxlib.h:

```
    luaL_openlibs(L);
```

We are now ready to call some Lua code, e.g. from a file named *script.lua*:

```
    luaL_dofile(L, "script.lua");
```

When we are done with our Lua state, we call `lua_close()` to finish the Lua state. Remind yourself that Lua is garbage collected language and calling `lua_close()`will allow it to run the garbage collector one last time, freeing all resources.

```
    lua_close(L);
```

We see a pattern here: All functions of the C API, besides the functions that create a new Lua state — `luaL_newstate()` and `lua_newstate()` — expect a lua_State * to be passed as the first parameter.

The complete short example now looks like this:

```c
#include <stdio.h>

#include <lualib.h>
#include <lauxlib.h>

lua_State *L;

void
main(int argc, char *argv[])
{
    /* Create a Lua state */
    L = luaL_newstate();

    /* Check the return value */
    if (L == NULL) {
        fprintf(stderr, "Lua: cannot initialize\n");
        return -1;
    }

    /* Provide the Lua standard libraries to the Lua state */
    luaL_openlibs(L);

    /* Execute a Lua program in script.lua */
    luaL_dofile(L, "script.lua");

    /* Close the Lua state */
    lua_close(L);

    return 0;
}
```

We have seen how to link Lua with a host program, create an empty Lua state, open the standard libraries, and, executing Lua code.

But this is a rather boring example, since all it does is to run a Lua script in complete isolation from the host program. There is absolute no interaction between the host program and the Lua script. This is where the Lua C API enters the stage.

The Lua C API

The Lua C API provides a rich and very carefully designed set of functions for a host program to interact with a Lua script.

Using the Lua C API it is not only possible to create or close a Lua state, open the standard Lua libraries and execute Lua code from a file, but a lot, lot more.

With the Lua C API you can:

- Manage Lua states, creating and closing them.
- Load libraries into a Lua state.
- Create variables in a Lua state.
- Access variables in a Lua state.
- Run Lua code from C.
- Run C from Lua by providing C functions to the Lua state.
- Create uservalues, i.e. values that hold C data structures.
- And much more.

The Virtual Lua Stack

Lua uses a virtual stack to pass values between C and Lua. That means that the C function has to pop its arguments from the stack and push and return values to the stack, returning an int indicating the number of return values. Each element on the stack has a numerical index, starting at 1.

Passing Values Between Lua and C

Arguments and return values are passed between Lua and C using the virtual stack. That means that a C function has to pop its arguments from the stack and push and return values to the stack, returning an int indicating the number of return values. Each element on the stack has a numerical index, starting at 1. Consider the following Lua call:

```
local n = myFunction('miller', 42, 'south', 'north')
```

The stack will then look as follows:

Index	Content
1	"miller"
2	42
3	"south"
4	"north"

To return values to Lua, the C function pushes the values to the stack in the order they should be returned. Consider the following stack and assume the function returns the integer 3, indicating three result values:

Index	Content
1	"west"
2	"east"
3	42

The calling Lua function will receive this values as three return values:

```
local dir1, dir2, media = myFunction()  -- 'west', 'east', 42
```

So dir1 contains 'west, dir2 'east, and, media the integer 42.

The Lua C API provides a rich set functions for querying and manipulating the stack. The Lua auxiliary library provides additional functions which not retrieve values from the stack, but check for the expected type on go.

The functions for retrieving values from the stack are named `lua_totype()` while the functions in the auxiliary library are named `luaL_checktype()`.

To push values to the stack, functions named `lua_pushtype()` are provided by the Lua C API.

Calling Lua Code From C

Lua provides basically two functions to call Lua code from C, `lua_call()` and `lua_pcall()`. They differ in how error situations are handled.

`lua_call()` is used to call a Lua function in unprotected mode, meaning the host program will simply be terminated by a call to `exit()` in the case of an error, e.g. the Lua code calling the `error()` function.

`lua_pcall()`on the other hand calls a Lua function in protected mode, which means it will not terminate the calling program in case of an error. It will rather signal through its return value that there has been a problem and the Lua error message will be provided to the host program conveniently in the first element of the virtual stack. Thus we can call `lua_tostring(L, 1)` to retrieve the Lua error message.

Using `lua_pcall()` is usually the preferred way of calling Lua code.

To call a Lua function, the Lua virtual stack is again used to pass arguments and retrieve return values. To call a Lua function we first push onto the stack the function to be called, then the arguments to be passed to the function. Once the stack is setup, we call either `lua_pcall()` or `lua_call()`, passing the number of arguments be pushed onto the stack and the number of return values we expect:

```
lua_call(lua_State *L, int nargs, int nresults);
```

As an example, we want to call the `print()`function with three arguments, "miller", "north", and, the number 42. The following C code would prepare our stack:

```
    lua_getglobal(L, "print"); ①
    lua_pushstring(L, "miller");
    lua_pushstring(L, "north");
    lua_pushinteger(L, 42);
```

① `lua_getglobal()` pushes onto the stack the value of the global passed as argument, in this case the print() function (remember that in Lua functions are just ordinary values, so they can be put onto the stack).

Our stack now looks as follows:

Index	Content
1	The Lua `print()` function
2	The string "miller"
3	The string "north"
4	The integer 42

We can now call the function, expecting no result values:

```
    lua_call(L, 3, 0);
```

Calling C Code From Lua

For Lua to be able to call C code, the C code must be made available to Lua. In the simplest form, the host program will create a global variable holding the C function to be called.

Any C function that will be called from Lua must follow this protocol:

- It receives a lua_State * as its sole argument
- It pulls the arguments, if any, from the Lua stack.
- It does whatever it's purpose is.
- It pushes any return values to the Lua stack.
- It returns the number of return values as an int.

The signature of a C function that will be called from Lua looks as follows:

```
int
myFunction(lua_State *L)
{
}
```

Once we have written such a function it must be made known to the Lua state. A simple way is to define a global name for the function.

```
    ...

    lua_pushcfunction(L, myFunction);
    lua_setglobal(L, "myFunction");

    ...
```

In Lua, the function can now be called as `myFunction()`.

A simple, Yet Complete, Example

In this simple example we will provide a `greetings()` function to Lua which expects a string argument, the name of the person to greet, and which will return the number of invocations back to Lua. The function will output a greeting message to the console.

```
/* The function that will be called from Lua */
static int
greetings(lua_State *L)
{
    const char *name;
    static int invocations = 1;

    name = luaL_checkstring(L, 1); ①
    printf("greetings, %s\n", name);
    lua_pushinteger(L, invocations++);
    return 1;
}


    ...

    /* Register the function */
    lua_pushcfunction(L, greetings);
    lua_setglobal(L, "greetings");
```

① We use the auxiliary function `luaL_checkstring()` here which will ensure that the stack element 1 is actually a string. It will throw an error otherwise.

Once this has been setup, Lua can call the function in the usual way:

```
local invocations = greetings 'miller' ①

print('the greetings functions has been called ' .. invocations .. ' times.')
```

① If the sole argument to a Lua function is a string, then no parentheses are needed.

Userdata: Defining Your Own Data Types

Lua comes with basic data types like numbers, integers, strings, boolean etc. For more complex data structures it knows exactly one data type, the table.

C code, on the other hand, usually comes with complex data types and structures. As an example let's look at libuuid, a C library to generate and deal with universal unique identifiers, UUIDs (this library will also serve as an example in the chapter Binding Existing Libraries.)

libuuid defines an opaque data type `uuid_t`, which is used to hold a generated uuid. We have no idea how `uuid_t` is composed internally, neither do we need to. What we need is a way to tie a `uuid_t` C variable to a Lua value, so that Lua can deal with uuids and pass uuids to other functions.

The Lua C API provides us with userdata for exactly this purpose. A Lua userdata value is a Lua value that can be passed around just like any Lua value. Internally it is a piece of memory with a user defined size that we declare when we create the userdata value. Like all Lua values, userdata values get garbage collected eventually when they are no longer in use, the allocated memory is then automatically freed.

We create a userdata value by calling the `lua_newuserdatauv()` function, which returns a pointer to the allocated memory. As the Lua manual states, our code can freely use this memory:

```
    uuid_t uuid, *u;

    uuid_generate(uuid); ①
    u = lua_newuserdatauv(L, sizeof(uuid_t), 0); ②
    uuid_copy(*u, uuid); ③
```

① Call `uuid_generate()` from libuuid to generate a uuid.

② Allocate enough memory to hold a variable of the `uuid_t` type. Ignore the third argument for now.

③ Copy the previously generated uuid into the userdata memory.

In short, userdata is application memory that is managed by Lua, with automatic memory management for free.

Userdata for Application Memory

It is actually a good practice to use Lua userdata for application memory, even if that memory is never used as a Lua value. Consider the following fictional code fragment:

```
    char *text;

    text = malloc(1024); ①

    memcpy(text, "Hello, world!");
    lua_getlobal(L, "print");
    lua_pushstring(L, text);
    lua_call(L, 1, 0); ②
    free(text); ③
```

① Allocate memory using the standard `malloc()`function.

② We call the Lua `print()` function with one argument and expect no results.

③ Free the allocated memory.

Why is this bad?

Remember that Lua uses a `longjmp()` call when it encounters an error internally? And in fact, all three functions involved in the example above, `lua_getglobal()`, `lua_pushstring()`, and, `lua_call()` may raise an error, causing our program to `longjmp()` to a location that Lua defined using a `setjmp()` call earlier.

The `longjmp()` call will unwind the stack, but it will not free the allocated memory. We have just created a potential memory leak.

We can, however, leave memory management to the Lua C API and be completely safe, even if any of the `lua_` functions raises an error:

```
    char *text;

    text = lua_newuserdatauv(L, 1024, 0); ①

    memcpy(text, "Hello, world!");
    lua_getlobal(L, "print");
    lua_pushstring(L, text);
    lua_call(L, 1, 0); ②
    /* free(text); */ ③
```

① Allocate memory using the `lua_newuserdata()`function.

② We again call the Lua `print()` function with one argument and expect no results.

③ Don't free the allocated memory. Lua will take care of it.

Metatables

Values in Lua can be assigned a metatable which defines the behaviour of the value. Metatables are ordinary Lua tables that contain functions, so-called metamethods, which have predefined names starting with two underscores.

You can not just invent any name for a metamethod, there is a list of metamethods than can be defined and in which situation they are called by the Lua core.

Table 1. Complete list of metamethods

Metamethod	Used for	Operator
__add	The addition operation	+
__sub	The subtraction operation	-
__mul	The multiplication operation	*
__div	The division operation	/
__mod	The modulo operation	%
__pow	The exponentiation operation	^
__unm	The negation operation (unary minus)	unary -
__idiv	The floor division	//
__band	The bitwise and operation	&
__bor	The bitwise or operation	|
__bxor	The bitwise xor operation	binary ~
__bnot	The bitwise not operation	unary ~
__shl	The bitwise shift left operation	<<
__shr	The bitwise shift right operation	>>
__concat	The concatenation operation	..
__len	The length operation	#
__eq	Test for equality	==
__lt	Test for less than	<
__le	Test for less or equal	⇐
__index	The value is being indexed	[]
__newindex	A value is being set at a new index	[] =
__call	Call operation (execute as function)	()
__tostring	Used by tostring() to convert the value to a string	n/a

Metamethod	Used for	Operator
__close	The to-be-closed value is going out of scope	n/a
__gc	The value is about to be garbage collected	n/a

Once a metatable has been assigned to a value, Lua will call the metamethods whenever needed under the hood and completely invisible to the Lua program.

Metatables are one of the most powerfull concepts in Lua, especially when combined with userdata, one can do magic. The concept of metatables underline one strongest paradigm in Lua, namely to provide mechanisms, not policies.

For example, while Lua itself is not an object-oriented language, it can quite easily be made one by proper use of metatables.

To get an idea of how powerful metatables are, consider the following scenario: We have a userdata value that is not the actual data, but merely a pointer to the real data.

In the example that follows, a PostgreSQL result set is returned by a query function. The actual size of the data is not known, so we have to store a pointer to it. Our userdata value effectively becomes a pointer to a pointer:

```
    PGresult **res;

    res = lua_newuserdata(L, sizeof(PGresult *));
    *res = PQexec(db, "select name from people");
```

What happens if this userdata value is garbage collected? The pointer to the pointer will be freed by Lua, but not the pointer itself, leading to a memory leak (in PostgreSQL, you have to call `PQfreemem(*res)` to free the result set.)

If, however, we define a table with a `__gc()` function and set this table as the metatable of above userdata value, then Lua would call the `__gc()` function just before the value is garbage collected. In the `__gc()` function, which receives the to-be-freed value on the virtual stack, we can then pop our value from the stack and call `PQfreemeem()`, releasing the memory held by the result set.

To-be-closed Variables

As Lua is a garbage collected language, the Lua programmer does not have to care about memory management at all. Every value, once it is no longer used, will eventually be garbage collected and the memory it occupied will be be freed. This is also true for values that have been created by C code as userdata values.

While garbage collection is a great concept from the Lua programmers point of view, it is not optimal from the view of a C programmer who writes a Lua module in C. There is no control over when a value actually gets garbage collected. This can lead to situations where a lot of no longer used — and no longer accessible — memory is still being held by the process running the Lua interpreter.

As an example, imagine a PostgreSQL database interface being used in a Lua program that runs thousands of SQL queries per second. Each query returns a PostgreSQL result set that is essentially allocated memory that has to be explicitely freed by a call to the `PQfree()` C function. Whe could free the memory with an explicit call from the Lua program, but a Lua programmer might forget to call the free function, assuming everything is garbage collected.

On the other hand, if we free the memory at garbage collection time only, we have exactly the situation where potentially a lot of no longer used memory remains allocated.

Database result sets are only one example, we can think of a lot of other resources that are release to late or later than needed.

To address this problem, Lua 5.4 introduced to-be-closed variables, a mechanism to free memory or otherwise release resources at the very moment the variable goes out of scope.

To do so, two requirements must be met:

- The underlying userdata value must have a `__close` metamethod.
- The Lua program must annotate a variable as to-be-closed.

Consider the following function:

```
local function getName(db)
    local res = db:exec('select name from account')
    return res[1].name
end
```

The memory used by the variable `res` will only be freed at garbage collection time (whenever that will be.)

If, however, `res` is annotated as to-be-closed using the new keyword `<close>`, the underlying memory can be freed in the `__close` metamethod, which gets called as soon a the variable goes out of scope, i.e. when the `getName()` function returns:

```
local function getName(db)
    local res <close> = db:exec('select name from account') ①
    return res[1].name
end
```

① `res` is annoted as to-be-closed with the `<close>` keyword.

If you try to annotate a variable as to-be-closed when the underlying userdata has no `__close` metamethod, Lua will throw an error.

For the curious, this is the C function that is called in the Lua PostgreSQL interface to free memory either when `__close` is called or when the garbage collector runs:

```
static int
res_clear(lua_State *L)
{
    PGresult **r;

    r = luaL_checkudata(L, 1, RES_METATABLE);
    if (*r) { ①
        PQclear(*r);
        *r = NULL;
    }
    return 0;
}
```

① Prevent double free of the memory as this function can be called more than one time.

Binding Existing Libraries

When binding an existing library, you make its functionality available to Lua. And there is an awful lot of libraries out there that provide functionality of all sorts and that usually have seen the proof of time and are well maintained.

Thanks to the well designed Lua C API this process is not very hard. A bit of planning ahead is nevertheless needed. C libraries are designed with the C programmer in mind, using C programming paradigms. In Lua things are quite different, e.g.:

- In C we can define complex data structures and our own complex types. Lua has only tables as data structures.
- In C we must carefully manage memory. Lua has automatic garbage collection and to-be-closed variables.

Desiging a Lua Binding

The first step to designing a good Lua binding is to truly understand the C library, how it works on the C side, and what its data structures are. You must be able to use the library in your C code if you want to create a binding for it.

Next you must think about how you would like to use the libraries functionality in Lua. I wrote **to use the libraries functionality** on purpose, and not **provide the libraries functions to Lua** because that's what designing good Lua bindings is all about: Build a bridge between the Lua and the C world.

Of course you could just write a binding that is a one-to-one mapping of the C functions to corresponding Lua functions, but with this approach you'll end up with a Lua binding that is non-intuitive to use in many cases.

The libyaml library serves us as an example for a design decision. libyaml is used to parse YAML data in a C program. It works in a way where the C program repeatetly calls a function which returns an event. Such an event can be that a data element has been found in the YAML stream, or that the start of an array has been detected etc. The events itself are not important here, but the fact that libyaml works by returning a stream of events is.

We must think about what should be achieved with a libyaml Lua binding. YAML describes data in an easy to read, easy to write text format. In the end, we probably want all that data to be available in a Lua table. Converting the YAML data into a (probably nested) table seems like a reasonable design:

Parsing YAML data into a Lua table

```
local yaml = require 'yaml'

local data = yaml.parsefile('mydata.yaml')
```

Clearly, the approach of simply exposing the event-returning function to Lua would not work well,

it would mean that we have to do a lot of overhead work on the Lua side. A better approach is to do all the processing in C, populating a table while the events are processed. This means more C code (in fact a lot more) than if we just exposed the event-returning function, but it leads to a very straightforward usage in Lua.

If you are interested in the full code of luayaml, you may fetch it from Github at https://github.com/arcapos/luayaml.

luauuid, Working with UUIDs in Lua

luauuid is a Lua binding for the libuuid UUID library by Theodore Y. Ts'o. A UUID (Universally Unique ID) is an ID that consists of 128-bits and comes in different flavors: It can contain purely random bits or it can contain parts that are based e.g. on the current time. A UUID can be represented as a 36 character long string in the form `4309877e-29c7-4d7a-b4a3-4382ea7deead`.

The purpose of the Lua UUID binding is to create and parse UUIDs, compare them against each other or the UUID null value. Let's first have a look at the uuid test program *testuuid.lua* to show its usage:

```
local uuid = require 'uuid'

local function test_1 ()
    local myuuid <close> = uuid.generate_random()
    print(myuuid:unparse())
end

local function test_2 ()
    local myuuid <close> = uuid.generate_random()
    print(myuuid:unparse())
end
print 'test_1'
test_1()

print 'test_2'
test_2()
```

In this binding we use Lua userdata to store a pointer to a `uuid_t`, the internal C representation of a UUID. We define a metatable name in the headerfile *luauuid.h*:

```
#ifndef __LUAUUID_H__
#define __LUAUUID_H__

#define UUID_STR_SIZE   36
#define UUID_METATABLE  "uuid"

#endif /* __LUAUUID_H__ */
```

When a uuid is created using one of the libuuid generator functions, teh code will actually create a new userdata value, store the uuid in it and set the metatable of the new value to the UUID_METATABLE. As libuuid offers different uuid-generating functions, helper functions are used that are called by Lua and which hand the actual generator function to the `lua_uuid_generator()` function:

```
static int
lua_uuid_generator(lua_State *L, void (*generator)(uuid_t)) ①
{
    uuid_t uuid, *u;
    const char *format;
    char str[UUID_STR_SIZE + 1];

    generator(uuid);
    format = lua_tostring(L, 1);

    if (format && *format == 't') {  ②
        uuid_unparse(uuid, str);
        lua_pushlstring(L, str, UUID_STR_SIZE);
    } else {
        u = (uuid_t *)lua_newuserdata(L, sizeof(uuid_t)); ③
        uuid_copy(*u, uuid);
        luaL_setmetatable(L, UUID_METATABLE);
    }
    return 1;
}

static int
lua_uuid_generate(lua_State *L)
{
    return lua_uuid_generator(L, uuid_generate);
}

static int
lua_uuid_generate_random(lua_State *L)
{
    return lua_uuid_generator(L, uuid_generate_random);
}

static int
lua_uuid_generate_time(lua_State *L)
{
    return lua_uuid_generator(L, uuid_generate_time);
}
```

① The `lua_uuid_generator()` is used only internally and not exposed to Lua.

② A UUID in text form has been requested, there is no need to store it, instead the string representation is returned.

③ Allocate userdata to hold the actual uuid.

The latter three functions will be exposed to Lua in `luaopen_uuid()`:

```
int
luaopen_uuid(lua_State *L)
{
    struct luaL_Reg luauuid[] = {
        { "generate",       lua_uuid_generate },
        { "generate_random",    lua_uuid_generate_random },
        { "generate_time",  lua_uuid_generate_time },
        /* more functions to come */
        { NULL, NULL }
    };

    luaL_newlib(L, luauuid);
    return 1;
}
```

The more interesting part is the uuid metatable which defines all the functions and metamethods we can call on a uuid value. It is also defined in `luaopen_uuid()`:

```
    struct luaL_Reg uuid_methods[] = {
        { "clear",      lua_uuid_clear },
        { "compare",        lua_uuid_compare },
        { "data",       lua_uuid_data },
        { "is_null",        lua_uuid_is_null },
        { "time",       lua_uuid_time },
        { "unparse",        lua_uuid_unparse },
        { "__eq",       lua_uuid_equal },
        { "__lt",       lua_uuid_less },
        { "__le",       lua_uuid_less_or_equal },
        { "__gc",       lua_uuid_clear },
        { "__close",        lua_uuid_clear },
        { "__tostring",     lua_uuid_unparse },
        { "__concat",       lua_uuid_concat },
        { "__len",      lua_uuid_length },
        { NULL, NULL }
    };
    if (luaL_newmetatable(L, UUID_METATABLE)) {
        luaL_setfuncs(L, uuid_methods, 0);

        lua_pushliteral(L, "__index");
        lua_pushvalue(L, -2);
        lua_settable(L, -3);

        lua_pushliteral(L, "__metatable");
        lua_pushliteral(L, "must not access this metatable");
        lua_settable(L, -3);
    }
    lua_pop(L, 1);
```

As an example of how we access the userdata value, let's look at the implementation of the `unparse()` function:

```
static int
lua_uuid_unparse(lua_State *L)
{
    char str[UUID_STR_SIZE + 1];
    uuid_t *u;

    u = luaL_checkudata(L, 1, UUID_METATABLE);
    uuid_unparse(*u, str);
    lua_pushstring(L, str);
    return 1;
}
```

The remaining functions are no magic either, they just call different functions of the libuuid library after they retrieved the `uuid_t *` pointer. The full source code of luauuid can be fetched from Github: https://github.com/arcapos/luauuid

With this explanations and examples you should now be able to create your own bindings. But before doing so, it is well worth checking if a binding for the library you have in mind doesn't already exist. You might be surprised how many different implementations e.g. of decoders for JSON you will find…

Lua Readers

To load Lua code into a Lua state, you will most likely load it from a file or string using either the `luaL_loadfile()` or `luaL_loadstring()` function from the Lua auxiliary library.

Lua itself, however, uses a different approach to load Lua code: It repeatetly calls a function to return the code piece by piece. This function is called a Lua reader and it can either return the next piece of code plus its size in bytes or signal that the end of the Lua code has been reached.

So in order to load Lua code using only functions from the core library, you use the `lua_load()` function, supplying it with a Lua reader:

```
int
lua_load(lua_State *L,
    lua_Reader reader, ①
    void *data, ②
    const char *chunkname,
    const char *mode);
```

① This is the Lua reader function that will repeatly be called.

② The `data` pointer will be handed to the reader function on each call. It is not used by Lua.

Please note that `lua_load()` only loads the Lua code, but does not run it. After `lua_load()` successfully returns (with return code `LUA_OK`) the loaded Lua code will be on top of the stack.

But how does the Lua reader function look like and how is it called? The function returns the next piece of Lua code as a `const char *` pointer and gets the Lua state, the data pointer and a pointer to a `size_t` as arguments:

```
const char *
myReader(lua_State *L, void *data, size_t size)
{
    /* Return a piece of Lua source and put its size in size */
}
```

When there is no more Lua code to be returned, the reader function returns either NULL or sets `size` to zero.

The functions found in the Lua auxiliary library to load Lua code are in fact unsurprisingly implemented as Lua readers.

What makes this approach interesting? It allows us to supply a function that returns Lua code from any source, e.g. a network connection, a HTTP request, an object stored in a database, or, even synthesized Lua code that is built on the fly from parsing a different code format.

Lua Templates use a Lua reader to process source templates

The Lua templates engine, used to intermix any source language with Lua expressions and Lua code, uses a Lua reader to parse a source file and convert it to Lua code. This happens only when a template is rendered for the first fime. The code will be compiled to bytecode and on subsequent calls of the same template, the Lua code is directly executed withouth the need for parsing or compiling it again. The Lua code produced by the reader function will output the template content and at the same time run the Lua code fragments that were present in the original template.

Reading Lua code from a .zip file

As an example of a working Lua reader, we assume that we want to load Lua code from a file that stored within a .zip file, without first decompressing the .zip file on the command line.

Let's assume that we have a *testzip.zip* file that contains an entry *testzip.lua* with the code we want to load:

```
% zip -sf testzip.zip
Archive contains:
  testzip.lua
Total 1 entries (1468 bytes)
%
```

We first define how we want our helper function to look like, i.e. the function which we call to load Lua from a .zip file. This means, we design a function similar to `luaL_loadfile()` and `luaL_loadstring()`.

One approach is to pass the path to the .zip file and the path to the Lua file contained within the .zip file to the function:

```
int lua_loadzip(lua_State *L, const char *zipfile, const char *path)
```

Before we look at the implementation of `lua_loadzip()`, lets write a small command line tool which hosts the Lua environment, loads and calls the Lua code contained in a .zip file. We call the utility `ziplua` and it is called with two arguments, the path to the .zip file and the path to the Lua code.

```
% ./ziplua
usage: ziplua <zipfile> <path>
% ./ziplua testzip.zip testzip.lua # Load and run testzip.lua
%
```

And here is the source code of the ziplua utility:

```c
#include <err.h>
#include <stdio.h>
#include <stdlib.h>

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

extern int lua_loadzip(lua_State *, const char *, const char *);

int
main(int argc, char *argv[])
{
    lua_State *L;

    if (argc != 3) {
        fprintf(stderr, "usage: ziplua <zipfile> <path>\n");
        exit(1);
    }

    L = luaL_newstate();
    if (L == NULL) {
        fprintf(stderr, "memory error\n");
        exit(1);
    }
    luaL_openlibs(L);

    switch (lua_loadzip(L, argv[1], argv[2])) {
    case LUA_ERRSYNTAX:
        fprintf(stderr, "syntax error: %s\n", lua_tostring(L, -1));
        exit(1);
    case LUA_ERRMEM:
        fprintf(stderr, "memory error\n");
        exit(1);
    case LUA_ERRFILE:
        fprintf(stderr, "file error: %s\n", lua_tostring(L, -1));
        exit(1);
    case LUA_OK:
        ;
    }

    switch (lua_pcall(L, 0, 0, 0)) {
    case LUA_ERRRUN:
        fprintf(stderr, "runtime error: %s\n",
            lua_tostring(L, -1));
        exit(1);
    case LUA_ERRMEM:
        fprintf(stderr, "memory allocation error: %s\n",
            lua_tostring(L, -1));
        exit(1);
    case LUA_ERRERR:
```

```
        fprintf(stderr, "error running message handler: %s\n",
            lua_tostring(L, -1));
        exit(1);
    case LUA_OK:
        ;
    }
    lua_close(L);

    return 0;
}
```

With that in place, let's develop the corresponding Lua reader and the `lua_loadzip()` function. For the actual processing of .zip files we will rely on the libzip library from https://libzip.org.

```
#include <stdlib.h>

#include <lua.h>
#include <lualib.h>
#include <lauxlib.h>
#include <zip.h>
```

We will need some state information for the actual reader function, which will be passed in the `data *` pointer. In our case we will need a pointer to the zip file from which we read plus the remaining size which we still have to read. We define a `zip_reader_stat_t` type to keep this information.

```
typedef struct {
    zip_file_t  *zf; ①
    size_t      size;
} zip_reader_stat_t;
```

① `zip_file_t` is a data type defined by libzip in zip.h.

In our `lua_loadzip()` function we initialize a `zip_reader_stat_t` variable and pass it to `loa_load()` alongside with our reader function:

```
    zip_reader_stat_t st;

    st.zf = zip_fopen(zip, path, 0); ①
    st.size = sb.size;

    error = lua_load(L, zip_Reader, &st, path, NULL);
```

① path contains the path to the Lua file within the .zip file.

`lua_load()` will now call our `zip_Reader()` function, which will try to read up to `st.size` bytes from the .zip file and return them to `lua_load`. As reading can return fewer bytes than requested, it will

also return the number of bytes actually read and decrement `st.size` by that number. Once `st.size` is zero, the function will return NULL, thus terminating the loading process.

The complete function with the `zip_Reader()` function now looks like this:

```
#include <stdlib.h>

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

typedef struct {
    zip_file_t  *zf;
    size_t       size;
} zip_reader_stat_t;

static const char *
zip_Reader(lua_State *L, void *data, size_t *size)
{
    zip_reader_stat_t *st = (zip_reader_stat_t *)data;
    static char *buf = NULL;

    free(buf);

    if (st->size == 0)
        return NULL;

    buf = malloc(st->size);
    *size = zip_fread(st->zf, buf, st->size);

    st->size -= *size;
    return buf;
}

int
lua_loadzip(lua_State *L, const char *zipfile, const char *path)
{
    zip_t *zip;
    zip_stat_t sb;
    zip_reader_stat_t st;
    int error;

    zip = zip_open(zipfile, ZIP_RDONLY, &error);

    if (zip == NULL) {
        switch (error) {
        case ZIP_ER_INVAL:
            lua_pushstring(L, "invalid path");
            break;
```

```
        case ZIP_ER_MEMORY:
            lua_pushstring(L, "memory error");
            break;
        case ZIP_ER_NOENT:
            lua_pushstring(L, "file not found");
            break;
        case ZIP_ER_NOZIP:
            lua_pushstring(L, "not a zip file");
            break;
        case ZIP_ER_OPEN:
            lua_pushstring(L, "file open error");
            break;
        case ZIP_ER_READ:
            lua_pushstring(L, "read error");
            break;
        default:
            lua_pushstring(L, "unknown zipfile related error");
        }
        return LUA_ERRFILE;
    }
    if (zip_stat(zip, path, 0, &sb)) {
        lua_pushstring(L, "can't stat zipfile entry");
        zip_close(zip);
        return LUA_ERRFILE;
    }

    st.zf = zip_fopen(zip, path, 0);
    st.size = sb.size;

    error = lua_load(L, zip_Reader, &st, path, NULL);

    zip_fclose(st.zf);
    zip_close(zip);
    return error;
}
```

Useful Helper Functions

lua_vpncall()

The `lua_vnpcall()`function calls a Lua function by name, specifying parameters and return values in a `printf()` like manner.

Parameters passed as variable arguments and the expected results must be indicated by strings in *arg* and *ret* containing the following characters:

Character	C Type	Lua Type
b	int	Lua
f	double	Number
d	int	Integer
l	long	Integer
L	long long	Integer
s	char *	String

To pass a string and number to a function called *test*, expecting an int to be returned, one would call the function as follows:

```
    int rv;

    if (lua_vnpcall(L, NULL, "test", "sd", "d", "Hello, Lua!", 42, &rv)) {
        /*
         * Something went wrong, add an error handler. The Lua error
         * message is on top the Lua stack.
         */
    }
```

And here is the function itself:

```
#include <errno.h>
#include <stdarg.h>
#include <stdlib.h>
#include <string.h>

#include <lua.h>
#include <lualib.h>

int
lua_vnpcall(lua_State *L, char *table, char *func, char *arg, char *ret, ...)
{
    va_list ap;
    int n, retval, args, rets, pop;
```

```
    char *types, *t, *fnam, *f;

    fnam = strdup(func);
    if (fnam == NULL) {
        lua_pushstring(L, strerror(errno));
        return LUA_ERRMEM;
    }

    pop = 0;
    if (table != NULL) {
        lua_getfield(L, -1, table);
        if (lua_isnil(L, -1)) {
            lua_pushstring(L, "no such table");
            lua_pop(L, 1);
            return LUA_ERRRUN;
        }
        pop++;
    }

    f = fnam;
    while ((t = strsep(&f, ".")) != NULL) {
        lua_getfield(L, -1, t);
        if (lua_isnil(L, -1)) {
            free(fnam);
            lua_pushstring(L, "no such function");
            if (pop)
                lua_pop(L, pop);
            return LUA_ERRRUN;
        }
        pop++;
    }
    free(fnam);

    va_start(ap, ret);
    args = 0;
    if (arg != NULL) {
        for (types = arg; *types; types++)
            switch (*types) {
            case 'b':
                lua_pushboolean(L, va_arg(ap, int));
                args++;
                break;
            case 'f':
                lua_pushnumber(L, va_arg(ap, double));
                args++;
                break;
            case 'd':
                lua_pushinteger(L, va_arg(ap, int));
                args++;
                break;
            case 'l':
```

```
                lua_pushinteger(L, va_arg(ap, long));
                args++;
                break;
            case 'L':
                lua_pushinteger(L,
                    va_arg(ap, long long));
                args++;
                break;
            case 's':
                lua_pushstring(L, va_arg(ap, char *));
                args++;
                break;
            default:
                lua_pop(L, 1 + args + pop);
                lua_pushstring(L, "unknown return type");
                va_end(ap);
                return LUA_ERRRUN;
            }
    }
    rets = ret != NULL ? strlen(ret) : 0;
    if ((retval = lua_pcall(L, args, rets, 0))) {
        va_end(ap);
        if (pop)
            lua_pop(L, pop);
        return retval;
    }

    if (ret != NULL) {
        for (n = rets, types = ret; *types; n--, types++)
            switch (*types) {
            case 'b':
                *(va_arg(ap, int *)) =
                    lua_toboolean(L, -n);
                break;
            case 'f':
                *(va_arg(ap, double *)) =
                    lua_tonumber(L, -n);
                break;
            case 'd':
                *(va_arg(ap, int *)) =
                    lua_tointeger(L, -n);
                break;
            case 'l':
                *(va_arg(ap, long *)) =
                    (long)lua_tointeger(L, -n);
                break;
            case 'L':
                *(va_arg(ap, long long *)) =
                    (long long)lua_tointeger(L, -n);
                break;
            case 's':
```

```
                *(va_arg(ap, char **)) =
                    (char *)lua_tostring(L, -n);
                break;
            default:
                lua_pop(L, lua_gettop(L));
                lua_pushstring(L, "unknown return type");
                va_end(ap);
                return LUA_ERRRUN;
            }
        if (rets)
            lua_pop(L, rets);
    }
    va_end(ap);

    if (pop)
        lua_pop(L, pop);

    return 0;
}
```

lua_vpcall()

The `lua_vpcall()` function is similar to `lua_vnpcall()`, but instead of passing a function name, it expects the function to be on top of the Lua stack.

```
#include <stdarg.h>
#include <string.h>

#include <lua.h>
#include <lualib.h>

int
lua_vpcall(lua_State *L, char *arg, char *ret, ...)
{
    va_list ap;
    int n, retval, args, rets;
    char *types;

    va_start(ap, ret);
    args = 0;
    if (arg != NULL) {
        for (types = arg; *types; types++)
            switch (*types) {
            case 'b':
                lua_pushboolean(L, va_arg(ap, int));
                args++;
                break;
            case 'f':
                lua_pushnumber(L, va_arg(ap, double));
```

```
                args++;
                break;
            case 'd':
                lua_pushinteger(L, va_arg(ap, int));
                args++;
                break;
            case 'l':
                lua_pushinteger(L, va_arg(ap, long));
                args++;
                break;
            case 'L':
                lua_pushinteger(L,
                    va_arg(ap, long long));
                args++;
                break;
            case 's':
                lua_pushstring(L, va_arg(ap, char *));
                args++;
                break;
            default:
                lua_pop(L, 1 + args);
                lua_pushstring(L, "unknown parameter type");
                va_end(ap);
                return LUA_ERRRUN;
            }
    }
    rets = ret != NULL ? strlen(ret) : 0;
    if ((retval = lua_pcall(L, args, rets, 0))) {
        va_end(ap);
        return retval;
    }
    if (ret != NULL) {
        for (n = rets, types = ret; *types; n--, types++)
            switch (*types) {
            case 'b':
                *(va_arg(ap, int *)) =
                    lua_toboolean(L, -n);
                break;
            case 'f':
                *(va_arg(ap, double *)) =
                    lua_tonumber(L, -n);
                break;
            case 'd':
                *(va_arg(ap, int *)) =
                    lua_tointeger(L, -n);
                break;
            case 'l':
                *(va_arg(ap, long *)) =
                    (long)lua_tointeger(L, -n);
                break;
            case 'L':
```

```
                *(va_arg(ap, long long *)) =
                    (long long)lua_tointeger(L, -n);
                break;
            case 's':
                *(va_arg(ap, char **)) =
                    (char *)lua_tostring(L, -n);
                break;
            default:
                lua_pop(L, lua_gettop(L));
                lua_pushstring(L, "unknown return type");
                va_end(ap);
                return LUA_ERRRUN;
            }
        if (rets)
            lua_pop(L, rets);
    }
    va_end(ap);
    return 0;
}
```