Embedding Lua in C

EXTENDING YOUR TOOLS AND APPS WITH LUA SCRIPTS

Updated in 2023-10-08 · First published in 2019-02-02 · By Lucas Klassmann

Introduction

Allowing users to extend their apps with scripts is an amazing feature. There are many languages for this purpose, but Lua is the choice today, it is powerful and easy to embed.

Lua is an important tool for many industries, being used inside **Game Engines**, Databases like

Redis, and HTTP servers like **Nginx**, for powering users to extend their features.

An introduction on how to embed it into a C application will be shown, with simple uses, but complete examples of how to work with Lua. I may not cover all concepts around Lua, installation, and other topics like **LuaJIT**, but it may serve as a starting point for you.

Another thing that I will not cover is how to compile and link your app with the Lua library, if you know a little about C you can do it with almost no effort using the official manual.

The goal is to take you through examples of common cases of using Lua. You may use it as a configuration script, allowing users to configure the properties of your application as you may wish. It is possible to invoke custom functions written by users to react to certain events of your application, which can be used for customizing logic or returning data from them, or maybe calling another application when yours finishes a task. All those possibilities will be covered here.

Installation

I do not cover the installation of Lua development libraries, because it varies and depends on your platform. You can look at the installation guide on the official website **here**.

Some Lua Concepts

There are three important data structures in Lua: the *State*, the *Virtual Stack*, and the *Registry Table*.

Lua State

When a new state(`lua_State`) is created, a new interpreter thread is started and associated with it. It also has a new registry table, where the global definitions(variables and functions) will be for being used by Lua and C. When calling a Lua API C function, it also stores the

virtual stack. It glues everything together and is the main parameter for the API calls.

Registry Table

It is the place where all global data were registered to be available to Lua code in a certain state, it is where Lua keeps the global variables and functions. In order to expose a variable from C to Lua, you need to register a new record inside this table. If you need the get the value from a variable or a reference of a Lua function to be called from C, this is the place where you will find it too.

Virtual Stack

The virtual stack is where the data is temporarily stored between calls, you put data in it when you want to pass data to Lua and get from it when Lua returns data to your code in C.

Lua C API

In order to use almost all C API functions in Lua, you need to put the necessary data into the virtual stack and call the function that you need. It is important you check the manual to know which parameters are required for each function. Some values need to be on the stack before calling the function.

As an example of how the API works take a look at this function:

```
void lua_setglobal (lua_State *L, const char *name);
```

It is used to define a new global variable in the registry table. It only needs two parameters, the `lua_State* L` and the name that will identify the value in Lua. There is no information about the value it will store because the function is independent of the type. We need to push a value of the type we want before calling it. After calling `lua_setglobal`, it will pop the value on top of the stack and create a new record in the registry with the name that we want.

This is how almost all functions in the Lua C API work, they are simple calls and usually based on the stack values and less on parameters in the function. It makes the API simple and decoupled from an eventual change in the type system.

Another example: If you want to expose to Lua script a variable called `WEBSITE_URL` with the value of `http://localhost:8000` you should do this:

```
lua_pushstring(L, "http://localhost:8000")  // Put it on the stack
lua_setglobal(L, "WEBSITE_URL")             // Removed from the stack and register
```

What happens here is, again, `lua_pushstring` pushes a string value into the stack and `lua_setglobal` pops the value on top of the stack to register the name `WEBSITE_URL` inside the *Registry Table*.

You are going to see other examples of Lua returning values from the stack. This usually happens as a result of calling a Lua function. The operation is similar, but you have to pop the values from the stack and check if the type of the returned data is expected.

> One important point is, whenever Lua calls C, it gets a new stack, which is independent of previous stacks and the stacks of C functions that are still active. **Manual 4.2**

You are also responsible for the correct use of the stack, ensuring its consistency and using its size correctly.

Starting Lua VM

Creating a new interpreter requires you to create a new state as mentioned before.

The basic code that we need to call when we use Lua C API is as follows. We start declaring a `lua_State` pointer and initialize it with the `luaL_newstate` function.

> Note that versions before 5.3 lua_open() is used instead.

It is our pointer to our **Lua** Virtual Machine and this `lua_State` store all data that we share between Lua and C. We can have many **Lua** states to run scripts with different contexts.

```
lua_State *L = luaL_newstate();
```

After opening a state, we are ready to call functions from Lua **API**.

One important function to be called is `luaL_openlibs`, it makes available the **Lua** Standard Library for the code that we will run afterward. It is not a requirement, but without this, you will

not be able to call use libraries like math, io, string, and utf8, inside Lua code.

```
luaL_openlibs(L);
```

You can also open only the libraries that you know it will be useful or safe to allow scripts to call, see the example bellow, more information **here**.

```
// Allows only math and string libraries to be used
luaopen_math(L);
luaopen_string(L);
```

After using the lua state, when you do not need to execute anything else, you have to close the state with:

```
lua_close(L);
```

Initial Examples

I wrote some examples of basic operations like running **Lua** code from a string, loading a script file, exposing variables and C functions to **Lua** code, recovering the values from global variables inside the **Lua** code and calling Lua function in C.

Initializing Lua VM

Here is our first example, it is a starting point for using Lua with C.

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

int main(int argc, char ** argv) {
```

```
    lua_State *L = luaL_newstate();
    luaL_openlibs(L);

    // Work with lua API

    lua_close(L);
    return 0;
}
```

Let's dissect the code structure. We start adding the headers:

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

Start a new Lua state:

```
lua_State *L = luaL_newstate();
```

With `L` state you're now able to call Lua C API and interact with Lua VM. Next, open the Lua standard libraries:

```
luaL_openlibs(L);
```

And when you application finishes, do not forget to close the state:

```
lua_close(L);
```

Running Lua code

This example shows how to load a **Lua** code from string and run it.

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

int main(int argc, char ** argv) {

    lua_State *L = luaL_newstate();
    luaL_openlibs(L);

    // Our Lua code, it simply prints a Hello, World message
    char * code = "print('Hello, World')";

    // Here we load the string and use lua_pcall for run the code
    if (luaL_loadstring(L, code) == LUA_OK) {
        if (lua_pcall(L, 0, 0, 0) == LUA_OK) {
            // If it was executed successfuly we
            // remove the code from the stack
            lua_pop(L, lua_gettop(l));
        }
    }

    lua_close(L);
    return 0;
}
```

The new things here are the API functions: `luaL_loadstring`, which is in charge of loading the code chunk on the top of the stack, `lua_pcall` is in charge of running the code in the stack, and if the execution is successful, we remove the code from the top(`lua_gettop`) of the stack with `lua_pop`.

Note about the updates in the article

Before continuing with the examples I have to explain some updates I made in the article.

After the feedback that I received, I decided to make the following examples more clear and simple using some macros available from Lua API.

The following changes were made:

- Instead of using `lua_pcall`, `luaL_loadstring`, and `luaL_loadfile`, I modified the code to use:
  - `luaL_dostring`, which is equivalent to calling `lua_pcall` and `luaL_loadstring`
  - `luaL_dofile`, which is equivalent to calling `lua_pcall` and `luaL_loadfile`
  - use `lua_pcall` only when I do not run the function just after loading a file or a string.

All those macros use `LUA_MULRET`, which is an argument that tells `lua_pcall` that it must expect a variable number of returned values.

Another point is that when we call `lua_pcall` or its variants, a result is returned indicating if the call was successfully executed or not. We can use the constant `LUA_OK` to check if there is a success or not.

When an error happens during `lua_pcall`, it returns a different value of `LUA_OK` and puts the error on the top of the stack. We are able to get this error using `lua_tostring(L, lua_gettop(L))`.

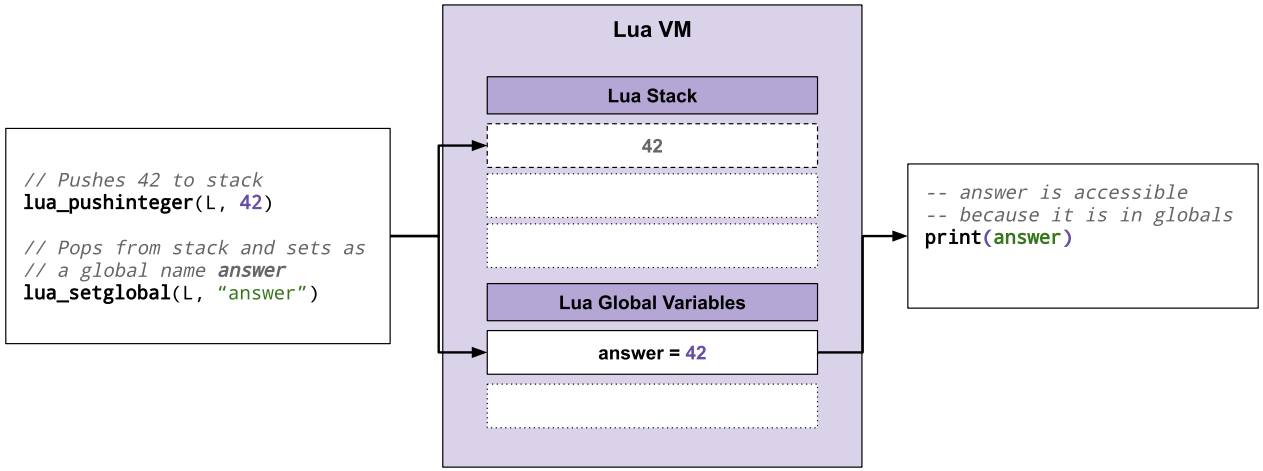
I do not use `lua_call` because it is a *non-protected* way to call a function, its use implies that Lua will call a panic function and then call abort. It is possible to handle those errors by setting a new panic function with `lua_atpanic`, but it is not covered in this article.

Using the protected call, any error will make Lua call long`jmp, recover from the most recent active recovery point, and set the error on top of the stack.

More examples

Exposing a Simple Variable

It is a really common need to expose some variables for Lua code, and it is simple to do this:




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

int main(int argc, char ** argv) {

    lua_State *L = luaL_newstate();
    luaL_openlibs(L);

    lua_pushinteger(L, 42);
    lua_setglobal(L, "answer");

    char * code = "print(answer)";

    if (luaL_dostring(L, code) == LUA_OK) {
        lua_pop(L, lua_gettop(L));
    }
```

```
    lua_close(L);
    return 0;
}
```

We use here `lua_pushinteger` to put an integer on the top of the stack and after it, we use `lua_setglobal` to get this value and set as a global variable named as **answer**.

After exposing the variable we can use it inside the **Lua** code.

> Note: For more complex types and structures, check Lua manual.

Exposing a Single Function to Lua

This example is a little bit more complex, I added some comments inside the code to explain the more important lines.

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

// Define our function, we have to follow the protocol of lua_CFunction that is
// typedef int (*lua_CFunction) (lua_State *L);
// When this function is called by Lua, the stack contains the arguments needed,
// what we need to do check if the arguments have the type that we expect.
int multiplication(lua_State *L) {

    // Check if the first argument is integer and return the value
    int a = luaL_checkinteger(L, 1);

    // Check if the second argument is integer and return the value
    int b = luaL_checkinteger(L, 2);

    // multiply and store the result inside a type lua_Integer
```

```
    lua_Integer c = a * b;

    // Here we prepare the values to be returned.
    // First we push the values we want to return onto the stack in direct order.
    // Second, we must return the number of values pushed onto the stack.

    // Pushing the result onto the stack to be returned
    lua_pushinteger(L, c);

    return 1; // The number of returned values
}

int main(int argc, char ** argv) {

    lua_State *L = luaL_newstate();
    luaL_openlibs(L);

    // Push the pointer to function
    lua_pushcfunction(L, multiplication);

    // Get the value on top of the stack
    // and set as a global, in this case is the function
    lua_setglobal(L, "mul");

    // we can use the function `mul` inside the Lua code
    char * code = "print(mul(7, 8))";

    if (luaL_dostring(L, code) == LUA_OK) {
        lua_pop(L, lua_gettop(l));
    }

    lua_close(L);
    return 0;
}
```

This example is similar to the exposing variables, but here we push a function pointer instead of an integer value.

Returning Values

An important note here is related to the value returned by the function multiplication, which represents the number of values pushed onto the stack as return values. The function exposed to Lua must follow the definition of the lua_CFunction type. A better explanation is found in the documentation:

> In order to communicate properly with Lua, a C function must use the following protocol, which defines the way parameters and results are passed: a C function receives its arguments from Lua in its stack in direct order (the first argument is pushed first). So, when the function starts, lua_gettop(L) returns the number of arguments received by the function. The first argument (if any) is at index 1 and its last argument is at index lua_gettop(L). To return values to Lua, a C function just pushes them onto the stack, in direct order (the first result is pushed first), and returns the number of results. Any other value in the stack below the results will be properly discarded by Lua. Like a Lua function, a C function called Lua can also return many results.

lua_register Macro

It is also possible to use the macro called `lua_register` which does the same work as `lua_pushcfunction` and `lua_setglobal`.

```
void lua_register (lua_State *L, const char *name, lua_CFunction f);
```

Instead of writing this:

```
lua_pushcfunction(L, multiplication);
lua_setglobal(L, "mul");
```

Using the macro you just need to write this:

```
lua_register(L, "mul", multiplication);
```

Exposing Functions to Lua with Namespace

We use a table to create a namespace, we put all functions inside this table.

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

// Here is the same function from the previous example
int multiplication(lua_State *L) {
    int a = luaL_checkinteger(L, 1);
    int b = luaL_checkinteger(L, 2);
    lua_Integer c = a * b;
    lua_pushinteger(L, c);
    return 1;
}
```

```c
int main(int argc, char ** argv) {

    lua_State *L = luaL_newstate();
    luaL_openlibs(L);

    // First, we need to define an array with
    // all functions that will be available inside our namespace
    const struct luaL_Reg MyMathLib[] = {
        { "mul", multiplication }
    };

    // We create a new table
    lua_newtable(L);

    // Here we set all functions from MyMathLib array into
    // the table on the top of the stack
    luaL_setfuncs(L, &MyMathLib, 0);

    // We get the table and set as global variable
    lua_setglobal(L, "MyMath");

    // Now we can call from Lua using the namespace MyMath
    char * code = "print(MyMath.mul(7, 8))";

    if (luaL_dostring(L, code) == LUA_OK) {
        lua_pop(L, lua_gettop(L));
    }

    lua_close(L);
    return 0;
}
```

Running a Lua Script

The only difference between running code from a string or file is that we use `luaL_loadfile` instead of `luaL_loadstring`.

```
-- script.lua
print("Hello, World from File")
```

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

int main(int argc, char ** argv) {
    lua_State *L = luaL_newstate();
    luaL_openlibs(L);

    if (luaL_dofile(L, "script.lua") == LUA_OK) {
        lua_pop(L, lua_gettop(L));
    }

    lua_close(L);
    return 0;
}
```

Getting a Global Variable from Lua

Retrieving values from a Lua script is a good way of configuring your app.

> Note that we can only get the variable after running the Lua code.

```
-- script2.lua
message = 'This message is stored inside Lua code'
```

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

int main(int argc, char ** argv) {

    lua_State *L = luaL_newstate();
    luaL_openlibs(L);

    if (luaL_dofile(L, "script2.lua") == LUA_OK) {
        lua_pop(L, lua_gettop(L));
    }

    lua_getglobal(L, "message");

    if (lua_isstring(L, -1)) {
        const char * message = lua_tostring(L, -1);
        lua_pop(L, 1);
        printf("Message from lua: %s\n", message);
    }
```

```
    lua_close(L);
    return 0;
}
```

Calling a Lua Function in C

Calling a function defined in a Lua code it is pretty similar of getting a variable, but to execute the function you must check if it is a function on the top of the stack with `lua_isfunction` and call it with `lua_pcall`. If it is successfully executed you have to remove from the stack with `lua_pop(l, lua_gettop(l))`.

```
-- script3.lua
function my_function()
    print("Hello from Function in Lua")
end
```

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

int main(int argc, char ** argv) {
    lua_State *L = luaL_newstate();
    luaL_openlibs(L);

    if (luaL_dofile(L, "script3.lua") == LUA_OK) {
        lua_pop(L, lua_gettop(l));
    }

    lua_getglobal(L, "my_function");
    if (lua_isfunction(L, -1)) {
        if (lua_pcall(L, 0, 1, 0) == LUA_OK) {
            lua_pop(L, lua_gettop(L));
        }
```

```
    }

    lua_close(L);
    return 0;
}
```

Calling a Lua Function in C with Arguments and Return Value

When you call a function defined in Lua, you can pass argument values and get the return value. To do that you need to put onto the stack the list of arguments after putting the function on the stack. After running the function you can check on the top of the stack a get the return value.

```
-- script4.lua
function my_function(a, b)
    return a * b
end
```

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

int main(int argc, char ** argv) {

    lua_State *L = luaL_newstate();
    luaL_openlibs(L);

    if (luaL_dofile(L, "script4.lua") == LUA_OK) {
        lua_pop(L, lua_gettop(L));
    }

    // Put the function to be called onto the stack
```

```
    lua_getglobal(L, "my_function");
    lua_pushinteger(L, 3);  // first argument
    lua_pushinteger(L, 4);  // second argument

    // Execute my_function with 2 arguments and 1 return value
    if (lua_pcall(L, 2, 1, 0) == LUA_OK) {

        // Check if the return is an integer
        if (lua_isinteger(L, -1)) {

            // Convert the return value to integer
            int result = lua_tointeger(L, -1);

            // Pop the return value
            lua_pop(L, 1);
            printf("Result: %d\n", result);
        }
        // Remove the function from the stack
        lua_pop(L, lua_gettop(L));
    }

    lua_close(L);
    return 0;
}
```

Example of Error Handling

A simple way to handle errors when calling a function:

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

int main(int argc, char ** argv) {
```

```
    lua_State *L = luaL_newstate();
    luaL_openlibs(L);

    char * code = "print(return)"; // intentional error

    if (luaL_dostring(L, code) != LUA_OK) {
        puts(lua_tostring(L, lua_gettop(L)));
        lua_pop(L, lua_gettop(L));
    }

    lua_close(L);
    return 0;
}
```

The End

There is much more about this topic, you can learn more about Lua in the official manual, LuaJIT is another important thing, as it is a way to use Lua with better performance. If you are interested in game development, check out Love2D, which is an amazing tool. Finally, check my repository with all examples. I hope it will be useful for you.

Thank you!

PS: Thanks for all fixes and improvements sent to me over time. They're welcome!

Resources

- **Repository with Examples**
- **Lua 5.3 Manual**
- **Lua Installation Guide**
- **Using Lua with SDL 2 - Example**
- **LuaJIT**

- LoSa 2D Game Framework

Software developer, retro computer enthusiast and game developer.