

Integrating C/C++ with Lua

PV264 Advanced Programming in C++

Nikola Beneš Jan Koniarik

Faculty of Informatics, Masaryk University

Autumn 2020

- easily embeddable into C++ applications
- can be used to extend your app with plugins or as an interface to your code
- originally designed to work with C
 - we will mention some caveats of C++ usage

```
function fact(n)
    if i == 0 then
        return 0
    end
    return n * fact(n - 1)
end

print(fact(42))
```

History

- started at TeCGraf (university group)
- TeCGraf was asked to solve the problem of providing data for simulations for PETROBRAS
 - they created DEL, language for the description of datasets with ability to check properties of data ($x > z$ and so on)
- around the same time, PETROBRAS wanted configurable report generator of another application
 - TeCGraf created another language: SOL
- both languages were not the main language of the app, rather some form of extension
- they realized that these could be merged, which is preferable for longterm usage: Lua is born
 - key design thought: main users are not programmers, just engineers
- <https://www.lua.org/history.html>

International exposure

- 1996 - TecGraf published a paper about Lua:
 - Lua in Software: Practice & Experience
- after that, strong traction in the game industry as a scripting system
- nowadays, this includes Lucasarts, BioWare, Blizzard...
 - more on: https://en.wikipedia.org/wiki/Category:Lua-scripted_video_games

Lua language

- official manual at <https://www.lua.org/manual/5.4>
- dynamic scripting language, implemented as library executing the code
 - native ability to bind C functions to Lua code
- language has only a few types:
 - numbers (float/double), strings, nil, boolean
 - associative tables
 - userdata
 - function (both Lua and C)
 - thread
- supports anonymous functions, closures, coroutines and more
- indexing from 1

- the ability to create Lua context, in which we execute Lua code
 - the context is a “box”
 - we can have multiple separate Lua apps executed at once
- the library allows to:
 - load Lua code and execute it
 - bind C functions to the context, so they can be used in Lua code
 - call functions defined in loaded files from the C code
- the standard interpret of Lua is just an example usage of the library

Example of loading code

```
#include <lua.hpp>
#include <string>

std::string code = "print(\"test\")";

int main(int argc, char* argv[]) {
    // Create new Lua context
    lua_State* L = luaL_newstate();
    // Load standard Lua libraries into the context
    luaL_openlibs(L);
    // Load Lua code ( "vm" is just debugging mark )
    luaL_loadbuffer(L, code.c_str(), code.size(), "vm");
    // Execute the code
    lua_call(L, 0, 0);
}
```

- `lua_State` is a structure holding Lua context
- all functions from Lua library are prefixed with `lua_`
- all functions from the auxiliary library are prefixed with `luaL_`
 - this library is used to simplify usage of the standard API
- libraries loaded by `luaL_openlibs` are standard libraries such as `math` or `string`
- `lua_call(L, nargs, nreturns)` executes the loaded code¹

¹we will explain `nargs` and `nreturns` later

Compilation

- there is no magic for compilation of the example code
- Lua is a C library, present in most package managers
- as an alternative, you can use CMake to download its source and compile it
 - you can find such a cmake in this week's exercise

Interaction between Lua and C

- how to exchange data between the Lua code and C code?
- Lua is dynamic, C is not

Stack

- the key exchange mechanism used by the Lua is a stack
- the stack contains any of the data with types that Lua can work with
- Lua provides data to the C side pushed to the stack, alongside with API to work with the stack itself
- C side of the code works the same way, any data that should be passed to Lua side is put on the stack.
- `lua_call(L, nargs, nreturns)`
 - `lua_State* L` represents the stack
 - `nargs` represents the number of arguments
 - `nreturns` represents the number of return values
- when `lua_call(L, nargs, nreturns)` is executed
 - pops `nargs` values from the stack which are used as arguments
 - pops function from the stack
 - pushes back `nreturns` return values from a function call (filled with `nil` in case there is not enough of them)

Stack – example

- `luaL_newstate` creates new Lua state and `luaL_openlibs` loads standard Lua libraries
 - these do not affect the stack
- `luaL_loadbuffer(L, code.c_str(), code.size(), "vm")` loads the code, and pushes it as an executable function to the stack
- `lua_call(L, 0, 0)`
 - pops 0 arguments from the stack,
 - pops the code loaded in the previous step,
 - executes the code, and
 - pushes 0 return values to the stack;
 - in other words, it just executes the loaded code

Stack – return value

Lua code

```
function foo()  
    return 42  
end
```

C code

```
// Pushes function 'foo' to the stack from global space  
lua_getglobal(L, "foo");  
// Pops function from the stack and executes it  
lua_call(L, 0, 1);  
// Checks whenever the last item on the stack is number  
// and returns it  
double i = luaL_checknumber(L, -1);
```

Stack – arguments

```
// The function is pushed first on the stack  
lua_getglobal(L, "pow");  
// First argument  
lua_pushnumber(L, 2);  
// Second argument  
lua_pushnumber(L, 10);  
// Execute the function  
lua_call(L, 2, 1);  
// Get the return value  
int res = luaL_checkinteger(L, -1);
```

Stack – table

- the stack can contain any of the basic Lua types, in case of the table, it also provides the ability to access the fields of the table
- let's look at the table more closely, it shows how Lua relies on the stack

Stack – table

Lua code

```
a = {}  
a["x"] = 10  
a["y"] = 20
```

C code

```
lua_getglobal(L, "a");  
// Push the key of the field we want to the stack  
lua_pushstring(L, "x");  
// `lua_gettable` pops item from the top of the stack and  
// uses it as key for the table on position -2  
//  
// Value of assigned to that key is then pushed to the stack  
lua_gettable(L, -2);  
// `x` is now 10  
int x = luaL_checkinteger(L, -1);
```


Lua code

```
a = {}  
a["x"] = 10  
a["y"] = 20
```

C code

```
lua_getglobal(L, "a");  
// We can also try to access keys that are not present  
// in the table  
lua_pushstring(L, "z");  
lua_gettable(L, -2);  
// This results in nil value  
bool is_nil = lua_isnil(L, -1);
```

Bind C function

- we know how to call Lua function and pass/receive data.
- let's do the opposite – let Lua call C function.
- each C function that is bound has same type:
`int foo(lua_State *)`
- when foo is called from Lua, the following happens:
 - Lua pushes all Lua arguments to a new stack
 - C function is called with that stack
 - it is expected that you put the result on the stack and return the number of returned values
 - the rest of the stack is garbage collected later

Bind example

```
int l_foo(lua_State*) {  
    std::cout << "wololo" << std::endl;  
    return 0;  
}  
  
// Pushes function pointer to the stack  
lua_pushcfunction(L, l_foo);  
// Assigns the pointer to global variable  
lua_setglobal(L, "foo");  
  
// Alternatively:  
// lua_register(L, "foo", l_foo);
```

Complex bind example

```
// usage in Lua:
//      quot, rem = divmod(25, 4)
int l_divmod(lua_State* L) {
    if (lua_gettop(L) != 2) {
        return luaL_error(L, "2 arguments expected");
    }
    int arg1 = luaL_checkinteger(L, 1);
    int arg2 = luaL_checkinteger(L, 2);

    div_t res = div(arg1, arg2);

    lua_pushinteger(L, res.quot);
    lua_pushinteger(L, res.rem);
    return 2;
}
```

Error handling

- what happens in case of an error?
 - what happens if `luaL_checkint(L, 1)` finds that the item on the stack is not a number?
- that results in `luaL_error(L, msg)` call
- Lua uses two C functions to handle errors:
 - `setjmp(env)` – stores actual environment of execution into variable `env`
 - `longjmp(env)` – stops execution and restores previously-stored environment of execution
- this makes it possible to stop execution at one point and jump to the previous state

lua_call vs lua_pcall

- lua_pcall is a modification that takes care of 'catching' Lua errors
- if you call lua_pcall(L, nargs, nreturns, errfunc) in C code:
 - the function on top of the stack is executed in the same way as lua_call(L, nargs, nreturns)
 - when an error happens, it is caught and an error message is put on the stack
 - errfunc is the index of the function on the stack that serves as an error handler
- you should have lua_pcall as the top call on the call tree, otherwise, the app just crashes
 - if you satisfy that, the error from any lua_call is simply caught at nearest lua_pcall

C++ implications

- what does this mean for C++?
- imagine your function with C++ code is called from Lua and error happens in `luaL_checkint(L, 1)` call
- what can cause problems in C++ ?

C++ implications

- problem is the execution of `longjmp`, which makes the code continue execution at the place of nearest `lua_pcall`
- this mechanism does not call destructors – potential leaks
- in case of the following code, the destructor of `unique_ptr` may not be called

```
int l_foo(lua_State * L){  
    auto my_obj_ptr = std::make_unique< MyObject >();  
    int i = luaL_checkinteger(L);  
  
    my_obj_ptr->process(i);  
  
    return 0;  
}
```


longjmp/setjmp solution

- the solution to the problem is pretty simple – do not use Lua library compiled as C code
- Lua now supports compilation of the library as C++ code
 - in this case, the library uses `throw/catch` to handle the errors instead of `setjmp/longjmp`
 - given that, destructors will work just fine
- warning: this problem may be relevant also with other C libraries!

Summary

- what you should take from this lecture:
 - general idea of how Lua C API works (stack)
 - how to bind C/C++ functions
 - key C++ related problem

Much more

- there is more that you can do
 - bind objects that behave like objects (userdata)
 - <https://www.lua.org/pil/28.html>
 - global registry of data accessible by each function
 - <https://www.lua.org/pil/27.3.1.html>
 - closure for each C function bind
 - <https://www.lua.org/pil/27.3.3.html>
 - various C++ libraries that simplifies work with the Lua
 - <https://github.com/ThePhD/sol2>