LabLua
programming language research

# Programming in Lua – C API Basics

Fabio Mascarenhas

http://www.dcc.ufrj.br/~fabiom/lua

# Extension and Extensible

- Lua is an *extensible language* – we can extend its functionality using *libraries* written in other languages (mostly C)
- Lua is an *extension language* – we can extend the functionality of *applications* written in other languages with Lua code
- In Lua, these are two sides of the same coin!
- The same API that we use to call Lua from C, for extending an application, is the API we use to call C from Lua, to implement C modules
- All of the Lua standard library, and the standalone interpreter/REPL, are implemented using this API

# C API

- The C API has a few dozen functions to read and write global variables, call functions and chunks, create new tables, read and write table fields, export C functions to Lua, etc.

- Functions of the C API are *unsafe*: it is the responsibility of the programmer to make sure they are called with the right arguments, and in the correct context

- This is C programming, so segmentation faults and memory corruption await the careless!

- The API is simple and flexible, but it is a powerful tool, and is not easy to use

# A simple REPL

- The code below implements a very primitive REPL:

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

int main (void) {
  char buff[256]; int error;
  lua_State *L = luaL_newstate();   /* opens Lua */
  luaL_openlibs(L);                 /* opens standard libraries */
  printf("> ");
  while (fgets(buff, sizeof(buff), stdin) != NULL) {
    error = luaL_loadstring(L, buff) || lua_pcall(L, 0, 0, 0);
    if (error) {
      fprintf(stderr, "%s\n", lua_tostring(L, -1));
      lua_pop(L, 1);  /* pop error message from the stack */
    }
    printf("> ");
  }
  lua_close(L);
  return 0;
}
```

# Compiling and running

- We can compile and link the code with the Lua:

```
$ cc –o repl repl.c –llua
$ repl
> print(2 + 2)
4
> a = 5
> print(a)
5
> print("foo" + 5)
[string "print("foo" + 5)..."]:1: attempt to perform arithmetic on a string value
```

- Depending on the system, you may need to pass an include path and library path, and use a different name for the Lua library (`-llua`)

# Lua states

- The Lua interpreter does not define any C global variables; it keeps its state in a data structure called just a *Lua state*
- Calling `luaL_newstate` instantiates a new Lua interpreter and its corresponding state; all other API functions take this state as the first argument
- A fresh state does not have any Lua global variables defined, not even the built-in functions; `luaL_openlibs` loads all built-in functions and modules in the new state
- We will use a single Lua state in our examples, but an application is free to have multiple Lua states, and they are completely independent

# Loading and calling a chunk

- The `luaL_loadstring` function loads a chunk of Lua code
- If there are no syntax errors, this function returns 0 and pushes a function that executes the code in the *Lua stack*
- If the chunk has syntax errors, luaL_loadstring returns and error code, and pushes the error message in the stack
- `lua_pcall` is the C API analogue of `pcall`, and pops the function from the stack and calls it; if there were errors it returns an error code and pushes the error message in the stack
- In case of errors, the error message will be on the top of the stack; we get it with `lua_tostring` and pop it before looping

# The Lua stack

- All communication between Lua and C code is done through the *Lua stack*
- The stack holds Lua values, and C API functions usually pop values they need from the stack and push values they produce on the stack
- Using the stack may seem awkward at first, but it greatly simplifies both the API and the Lua implementation, specially garbage collection
- It is your responsibility to make sure the stack has enough “slots” to do what you want, and a fresh stack begins with space for 20 slots; if you need more, use `lua_checkstack`:

```
sucess = lua_checkstack(L, 50); /* make sure there is space to push 50 values */
```

# Pushing values

- The C API has functions to push atomic values:

```
void lua_pushnil     (lua_State *L);
/* 0 pushes false, anything else pushes true */
void lua_pushboolean (lua_State *L, int bool);
void lua_pushnumber  (lua_State *L, double n);
/* be careful in 64-bit platforms */
void lua_pushinteger (lua_State *L, ptrdiff_t i);
void lua_pushunsigned(lua_State *L, unsigned int u);
/* for strings with embedded zeros */
void lua_pushlstring (lua_State *L, const char *s, size_t len);
/* this just calls pushlstring with strlen(s) */
void lua_pushstring  (lua_State *L, const char *s);
```

- You can also push a fresh table with:

```
void lua_newtable(lua_State *L);
```

- Later we will see how we can push C functions, and arbitrary data using *userdata*

# Querying elements

- API functions follow a LIFO stack discipline, but Lua does not force it in the C code that manipulates the stack
- C code can reference any position in the stack with *indices*
- Positive indices (from 1) count from the *bottom* to the stack and up; negative indices (from -1) count from the *top* of the stack and down; for example, -1 is always the top slot, -2 is the slot below the top, and so on
- The top is independent from how many slots the stack has available; a fresh stack has 20 available slots, but the top is 0, as there is nothing in the stack

# Type checking

- The `lua_type` function is the analogue of `type`:

```
int lua_type (lua_State *L, int index);
const char *lua_typename (lua_State *L, int type);
```

- `lua_type` returns a numeric code, but there are constants for the eight types: `LUA_TNIL`, `LUA_TBOOLEAN`, `LUA_TNUMBER`, `LUA_TSTRING`, `LUA_TTABLE`, `LUA_TTHREAD`, `LUA_TUSERDATA`, `LUA_TFUNCTION`

- `lua_typename` turns the numeric code in the same string returned by `type`

# Getting atomic values out

- The API has several functions to extract atomic values from the stack (while leaving them there):

```
int           lua_toboolean (lua_State *L, int index);
const char  *lua_tolstring (lua_State *L, int index, size_t *len);
double        lua_tonumber  (lua_State *L, int index);
ptrdiff_t     lua_tointeger (lua_State *L, int index);
unsigned int lua_tounsigned(lua_State *L, int index);
```

- `lua_toboolean` works for any type, with the usual Lua rules (anything is *true* except for `nil` and `false`)

- `lua_tolstring` returns `NULL` if the value is not a string, but it converts numbers to strings; the other functions return 0 if the value is not a number

- The pointer returned by lua_tolstring is only guaranteed to be valid as long as the value is in the stack, and the contents cannot be modified; make a copy if you want the string to survive the value, or want to change it

# Stack movement

- There are several functions to move the stack contents around, which is useful sometimes:

```
int  lua_gettop  (lua_State *L); /* index of top element */
/* sets the new top, popping values or pushing nils */
void lua_settop   (lua_State *L, int index);
/* pushes a copy of the value at index */
void lua_pushvalue(lua_State *L, int index);
/* removes the value at index, shifting down */
void lua_remove   (lua_State *L, int index);
/* pops the value at the top and inserts into index, shifting up */
void lua_insert   (lua_State *L, int index);
/* pops the value at the top and inserts into index, replacing what is there */
void lua_replace  (lua_State *L, int index);
/* copy the value at "from" to "to", replacing what is there */
void lua_copy     (lua_State *L, int from, int to);
```

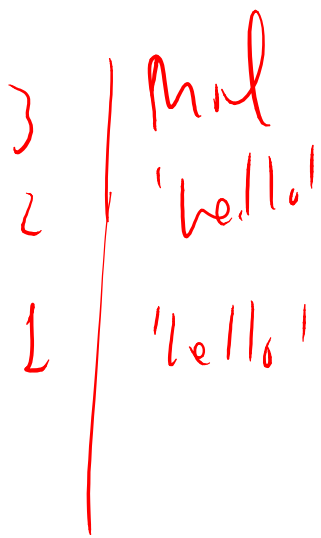
→ lua_pushvalue(L, from);
lua_replace(L, to);

- Remember that all indices can be positive or negative

# Quiz

- Assume the stack is empty. What will be its contents after the following sequence of calls?

```
lua_pushnumber(L, 3.5);
lua_pushstring(L, "hello");
lua_pushnil(L);
lua_pushvalue(L, -2);
lua_remove(L, 1);
lua_insert(L, -2);
```




SEE quiz.c