LUA C API

Note Title 11/28/2010

→ Lua never keeps pointers to any external objects except C functions which are static. So once anything is pushed on to the Lua Stack Lua makes a copy of it and it can be modified in the C program.

→ Lua State is a dynamic structure & can be viewed as an instance of the Lua interpreter

```
lua_State * luaVM = lua_open();
```

↳ A new lua state created

↳ This state contains no predefined functions not even print

→ Libraries need to be loaded on the lua state to add predefined functions

```
luaL_openlibs(luaVM);
```

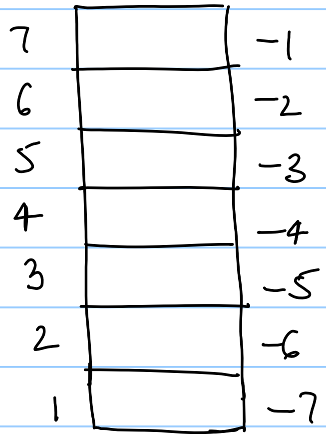
LUA STACK

→ Stack for Lua since Lua operates it like a stack LIFO (Last In first Out) for C it is more like an array accessed by indices

→ Each slot in the stack can hold any lua value

→ Index 1 points to the last element of the stack (that was pushed 1st). The Index -1 points to the first element on the stack (that was pushed last)

→ C program has to ensure stack has number of slots free before pushing things onto it. When lua calls C or when lua starts there are at least 20 free slots in stack set by (LUA_MINSTACK in lua.h)



STACK

PUSHING ELEMENTS

```
void lua_pushnil (lua_State * luaVM);
void lua_pushboolean (lua_State * luaVM, int bool);
void lua_pushnumber (lua_State * luaVM, double n);
void lua_pushlstring (lua_State * luaVM, const char *s,
            size_t length);
```

↳ Push an arbitrary length string

```
void lua_pushstring (lua_State * luaVM, const char *s);
```

↳ Push a zero terminated string

```
void lua_pushcfunction (lua_State * luaVM, lua_Cfunction f);
void lua_pushcclosure (lua_State * luaVM, lua_Cfunction f,
            int n);
```

QUERYING ELEMENTS

→ `int lua_is☒(lua_State *luaVM, int index)`
where ☒ is any of the 8 lua types.
index is the stack index for which the query is made.

→ If the queried slot is of that type or can be converted to that type then function returns 1 else 0

→ a Number on the stack would return 1 for lua_isnumber & lua_isstring

→ To get the actual type use

`int lua_type(lua_State *luaVM, int index)`

This returns one of these 9 constants defined in lua.h :

LUA_TNIL, LUA_TNUMBER, LUA_TBOOLEAN, LUA_TSTRING
LUA_TTABLE, LUA_TFUNCTION, LUA_TTHREAD, LUA_TUSERDATA
LUA_TLIGHTUSERDATA

→ To get a value from the stack use: lua_to☒
If the index does not point to that type the functions return a 0 or a NULL

OTHER STACK FUNCTIONS

`int   lua_gettop (lua_State *L);` → number of elements in stack

`void  lua_settop (lua_State *L, int index);` → sets the number of elements

if new no > old ⇒ nils & pushed

if new no < old ⇒ values from top discarded

If index < 0 then top is set to given index

∴ to pop n elements → `lua_settop(luaVM, -n-1);`

`void  lua_pushvalue (lua_State *L, int index);` → pushes on the stack a copy of the element at index

`void  lua_remove (lua_State *L, int index);` → removes the element at the given index & values on top fall down to fill the gap.

`void  lua_insert (lua_State *L, int index);` → moves the top element at given index

`void  lua_replace (lua_State *L, int index);` → pops the top element & overwrites it over the element at index

MANIPULATING TABLES

→ void lua_gettable (lua_State*luaVM, int index);

(Pops key, pushes the value on top)

↳ table index

↳ key is at -1

→ void lua_settable (lua_State * luaVM, int index);

(Pops out both key & value)

↳ table index

↳ key is at index -2

↳ value is at index -1

→ void lua_newtable (lua_State *L); → creates a new table & pushes on to the stack

USERDATA

→ This is a memory block that can be declared using

void *lua_newuserdata (lua_State *L, size_t size);

Since it returns a void pointer it can be type casted into a valid type pointer

eg. int * n = (int *) lua_newuserdata (luaVM, sizeof(n) * 1000);

↳ creates space for 1000 integers & returns the pointer to n

↳ Also pushes this userdata on the stack

→ Userdata can have metatables but lua code cannot change the metatable of userdata, it can although read it or change its key value pairs.

→ lightuserdata is just a C/C++ pointer that can be passed to & fro between lua & C++

→ Adding a metatable to light we can expose a C++ class into lua

CALLING LUA FUNCTIONS/CHUNKS FROM C
& lua_pcall

As an example, let us assume that our configuration file has a function like

```
function f (x, y)
  return (x^2 * math.sin(y))/(1 - x)
end
```

and you want to evaluate, in C, z = f(x, y) for given x and y. Assuming that you have already opened the Lua library and run the configuration file, you can encapsulate this call in the following C function:

```
/* call a function `f' defined in Lua */
double f (double x, double y) {
  double z;

  /* push functions and arguments */
  lua_getglobal(L, "f");  /* function to be called */
  lua_pushnumber(L, x);   /* push 1st argument */
  lua_pushnumber(L, y);   /* push 2nd argument */

  /* do the call (2 arguments, 1 result) */
  if (lua_pcall(L, 2, 1, 0) != 0)
    error(L, "error running function `f': %s",
             lua_tostring(L, -1));

  /* retrieve result */
  if (!lua_isnumber(L, -1))
    error(L, "function `f' must return a number");
  z = lua_tonumber(L, -1);
  lua_pop(L, 1);  /* pop returned value */
  return z;
}
```

You call lua_pcall with the number of arguments you are passing and the number of results you want. The fourth argument indicates an error-handling function; we will discuss it in a moment. As in a Lua assignment, lua_pcall adjusts the actual number of results to what you have asked for, pushing nils or discarding extra values as needed. Before pushing the results, lua_pcall removes from the stack the function and its arguments. If a function returns multiple results, the first result is pushed first; so, if there are n results, the first one will be at index -n and the last at index -1.

If there is any error while lua_pcall is running, lua_pcall returns a value different from zero; moreover, it pushes the error message on the stack (but still pops the function and its arguments). Before pushing the message, however, lua_pcall calls the error handler function, if there is one. To specify an error handler function, we use the last argument of lua_pcall. A zero means no error handler function; that is, the final error message is the original message. Otherwise, that argument should be the index in the stack where the error handler function is located. Notice that, in such cases, the handler must be pushed in the stack before the function to be called and its arguments.

For normal errors, lua_pcall returns the error code LUA_ERRRUN. Two special kinds of errors deserve different codes, because they never run the error handler. The first kind is a memory allocation error. For such errors, lua_pcall always returns LUA_ERRMEM. The second kind is

LUA REGISTRY

→ This is a table to store C function states for a lua instance. It is accessed from the lua stack using the pseudo index → LUA_REGISTRYINDEX

→ It cannot be accessed by the lua Instance so its a safe place to store lua data across C function calls

→ Functions to access the registry are:

```
int   luaL_ref ( lua_State * luaVM, LUA_REGISTRYINDEX) ;
void  lua_rawgeti(lua_State * luaVM, LUA_REGISTRYNDEX, int referance);
void  luaL_unref (lua_State * luaVM, LUA_REGISTRYINDEX, int referance);
```

→ Registry can also be accessed as a normal table on the stack

```
void  lua_gettable (lua_State * luaVM , LUA_REGISTRYINDEX);
```