Lua Tutorials for Korea F/W Team

Lua : https://en.wikipedia.org/wiki/Lua_(programming_language)

Lua programs are not interpreted directly from the textual Lua file, but are compiled into bytecode which is then run on the Lua virtual machine.

The compilation process is typically invisible to the user and is performed during run-time, but it can be done offline in order to increase loading

performance or reduce the memory footprint of the host environment by leaving out the compiler.

Lua bytecode can also be produced and executed from within Lua, using the `dump` function from the string library and the `load/loadstring/loadfile` functions, respectively.

Main site : http://www.lua.org/

User Wiki site : http://lua-users.org/wiki/

Manual for Lua 5.1 : http://www.lua.org/manual/5.1/

This tutorial is based on Michael's blog (http://nova-fusion.com/blog/)

1. General Syntax

```
--[[
This is
a block comment
]]

if state == 5 then
  doSomething() -- this is a line comment
elseif foo then
  a = true
else
  b = false
end
```

Lua doesn't use semicolons, nor curly braces.

2. Variables and Types

Lua is typeless, meaning that declaring a variable is as sample as **var = value.**
This creates the variable *in the global scope ,* we'll soon how to declare local variables.
You can also multiple variables at once : **var1, var2, var3, var4 = val1, val2, val3, val4**.

- **nil** is the nothing value, and also represents undefined variables.
- **true** and **false** are present. (false and nil —> false, everything else is considered true.)

3. Operators

```
or
and
<     >     <=    >=    ~=    ==
..
+     -
*     /     %
not   #     - (unary)
^
```

Here are the differences between standard C based operators set :

- || , && , ! —> or, and, not
- The inequality operator is ~= instead of !=.
- .. is the operator for string concatenation.
- ^ raises a number to a power , for example 10 ^ 2 would yield 100. In most other languages, this performs the binary XOR operation.
- # is an operators used to get the length of tables and strings.

If you are wondering whether bitwise operators are, there aren't any, However, Lua 5.2 added the bit32 library which you can use for bitwise operations.
Please refer to the below site if you are using the earlier version of Lua.
http://lua-users.org/wiki/BitwiseOperators

In Lua 5.1, you can install the below packages to use it.
liblua5.1-bitop-dev : fast bit manipulation library devel files for the Lua language version 5.1
liblua5.1-bitop0 : fast bit manipulation library for the Lua language version 5.1

4. Loops

`while`

```
i = 1

while i <= 5 do
  i = i + 1
  print(i)
end
```

`repeat`

```
i = 5

repeat
  i = i - 1
  print(i)
until i == 1
```

Numeric `for`

```
x = 1

for i = 1, 11, 2 do
  x = x * i
  print(x)
end
```

Loop Termination

```
while true do
  if condition then
    x = x ^ 2
  else
    break
  end
end
```

Lua doesn't have the **continue** keyword that many language have.

Functions

```lua
function foo()
  local x, y = something(4, 5)
  return x ^ y
end

function something(arg1, arg2)
  local ret1 = (arg1 * arg2) ^ 2
  local ret2 = (arg1 - arg2) ^ 2
  return ret1 + ret2, ret1 * ret2
end
```

As you can see, the order of declaration in a file doesn't matter as it does in some compiled language such as C and C++.
There are three things I want to point out :

- Lua is a bit different in how local variables are declared. If you want to do so, you need to prefix the declaration with the **local** keyword.
- Functions can return multiple values. foo shows how to capture these values.
- Default arguments aren't directly supported, but there are ways of achieving the same effect which we'll see in next section.

5. Tables

Tables are essentially associative arrays which can have keys and values of any type.
Tables are flexible and are the only data structure in Lua, so knowledge of them is important.

Here is an example of how to use them :

```lua
x = 5
a = {} -- empty table
b = { key = x, anotherKey = 10 } -- strings as keys
c = { [x] = b, ["string"] = 10, [34] = 10, [b] = x } -- variables and literals

-- assignment
a[1] = 20
a["foo"] = 50
a[x] = "bar"

-- retrieval
print(b["key"]) -- 5
print(c["string"]) -- 10
print(c[34]) -- 10
print(c[b]) -- 5
```

Curly brackets are used to create them.
You can use any kind of expression as a key as long as it's surrounded by square brackets.
If you'd like the key to be a string then you don't need square brackets or quotes, as demonstrated by the definition of `b`.

5.1 Syntactic Sugar

```lua
t = { foo = 1, bar = 2 }
print(t.foo) -- 1
t.bar = 3
```

This makes many uses of tables a lot more visually pleasing.
One example is the Lua standard libraries; instead of writing something like `math["sqrt"](x)` we can write `math.sqrt(x)`.

5.2 Arrays and Indices

```lua
a = { 11, 22, "foo", "bar" }
a[3] = "foooo"

print(a[1]) -- 11
print(a[3]) -- foooo
print(#a) -- 4
```

Lua is different from most languages when it comes to indices; they start at 1 rather than 0.
That means that instead of half-open ranges (`[0, length)`), closed ranges (`[1, length]`) are used.

5.3 Generic for Loop

```lua
a = { x = 400, y = 300, [20] = "foo" }
b = { 20, 30, 40 }

for key, value in pairs(a) do
  print(key, value)
end

for index, value in ipairs(b) do
  print(index, value)
end
```

As Programming in Lua states, it "allows you to traverse all values returned by an iterator function."
`pairs` is an iterator function which loops through each item and gives you both the key and the value.
`ipairs` is like `pairs` except it only loops through items with indices, starting at 1 and continuing until it encounters a `nil` value.
Here's what that example outputs when run:

```
y    300
x    400
20   foo
1    20
2    30
3    40
```

5.4 table module

```lua
t = { 24, 25, 8, 13, 1, 40 }
table.insert(t, 50) -- inserts 50 at end
table.insert(t, 3, 89) -- inserts 89 at index 3
table.remove(t, 2) -- removes item at index 2
table.sort(t) -- sorts via the < operator
```

6. Strings

Strings work as you'd expect; you can declare them with double or single quotes, and use backslash to escape characters.
For strings spanning multiple lines you can use square brackets in the format `[[...]]` .
Here's an example:

```lua
s = "foo\nbar"
t = 'he said "hello world"'
u = "Hello \"world\""
v = [[
<html>
  <body>
    <p>Hello world!</p>
  </body>
</html>
]]
```

Programming in Lua's section on strings lists all the escape characters and gives more information.

6.1 string module

The standard library provides the `string` module for working with strings.
Let's take a look at a few of the more simple functions:

```
string.lower("HeLLO") -- hello
string.upper("Hello") -- HELLO
string.reverse("world") -- dlrow
string.char(87) -- W
string.sub("hello world", 2, 5) -- ello
```

There's also a number of functions that work with Lua's string patterns.
Strings patterns are somewhat like regular expressions, except less powerful and with a different syntax.

```
string.gsub("hello 42", "(%d+)", "%1 3") -- hello 42 3
string.gsub("heLLo", "(%u)", "") -- heo

-- 4 + 4 = 8
string.gsub("2 + 2 = 4", "(%d)", function(s)
  return s * 2
end)

-- prints each word
for w in string.gmatch("good morning chaps", "%w+") do
  print(w)
end
```

http://pgl.yoyo.org/luai/i/string.gsub
http://www.lua.org/pil/20.2.html

string.gsub (s, pattern, repl [, n])

Returns a copy of `s` in which all (or the first `n`, if given) occurrences of the `pattern` have been replaced by a replacement string specified by `repl`, which can be a string, a table, or a function. `gsub` also returns, as its second value, the total number of matches that occurred.

If `repl` is a string, then its value is used for replacement. The character `%` works as an escape character: any sequence in `repl` of the form *%n*, with *n* between 1 and 9, stands for the value of the *n*-th captured substring (see below). The sequence `%0` stands for the whole match. The sequence `%%` stands for a single `%`.

If `repl` is a table, then the table is queried for every match, using the first capture as the key; if the pattern specifies no captures, then the whole match is used as the key.

If `repl` is a function, then this function is called every time a match occurs, with all captured substrings passed as arguments, in order; if the pattern specifies no captures, then the whole match is passed as a sole argument.

If the value returned by the table query or by the function call is a string or a number, then it is used as the replacement string; otherwise, if it is **false** or **nil**, then there is no replacement (that is, the original match is kept in the string).

Here are some examples:

```
x = string.gsub("hello world", "(%w+)", "%1 %1")
--> x="hello hello world world"

x = string.gsub("hello world", "%w+", "%0 %0", 1)
--> x="hello hello world"

x = string.gsub("hello world from Lua", "(%w+)%s*(%w+)", "%2 %1")
--> x="world hello Lua from"

x = string.gsub("home = $HOME, user = $USER", "%$(%w+)", os.getenv)
--> x="home = /home/roberto, user = roberto"

x = string.gsub("4+5 = $return 4+5$", "%$(.-)%$", function (s)
      return loadstring(s)()
    end)
--> x="4+5 = 9"

local t = {name="lua", version="5.1"}
x = string.gsub("$name-$version.tar.gz", "%$(%w+)", t)
--> x="lua-5.1.tar.gz"
```

7. Block and Scope

Lua has “braces of a sort in the form of keywords like “then” and “do” and “end".
Much like a language with C-based syntax, these “braces” represent blocks, which are essentially sequence of statements.
Please refer to the below site to get more detailed information.
http://www.lua.org/manual/5.1/manual.html#2.4

```lua
if x then
  stuff()
  moreStuff()
end

for i = 1, 10 do
  local x = "foo"
end

function foo(x, y)
  -- ...
end

-- explicit block
do
  local x = 3
  local y = 4
end
```

Conditionals, loops, and functions are all blocks. You can explicitly define a block as shown by the last example,

Lua uses lexical scoping , which means that every block has its own scope.

```lua
x = 5 -- global

function foo()
  local x = 6
  print(x) -- 6

  if x == 6 then
    local x = 7
    y = 10 -- global
    print(x) -- 7
  end

  print(x, y) -- 6, 10

  do
    x = 3
    print(x) -- 3
  end

  print(x) -- 3
end

foo()
print(x, y) -- 5, 10
```

This example demonstrates the effect of variable shadowing.
It also demonstrates how global variables can be defined at any scope if the `local` keyword is absent and no local variables exist by that name (as is the case with `y`).

8. More On Functions

8.1 Default argumenets

Lua does not directly support default arguments, but there are ways of getting the same behavior.
If a value isn't supplied for a function argument it will default to nil, as all undefined variables do.
Therefore:

```
function func(x, y, z)
  if not y then y = 0 end
  if not z then z = 1 end
  -- code
end
```

A more common method is to use the `or` operator instead:

```
function func(x, y, z)
  y = y or 0
  z = z or 1
end
```

Logical AND (&&)

The following code shows examples of the && (logical AND) operator.

```
a1 = true  && true      // t && t returns true
a2 = true  && false     // t && f returns false
a3 = false && true      // f && t returns false
a4 = false && (3 == 4)  // f && f returns false
a5 = "Cat" && "Dog"     // t && t returns "Dog"
a6 = false && "Cat"     // f && t returns false
a7 = "Cat" && false     // t && f returns false
```

Logical OR (||)

The following code shows examples of the || (logical OR) operator.

```
o1 = true  || true      // t || t returns true
o2 = false || true      // f || t returns true
o3 = true  || false     // t || f returns true
o4 = false || (3 == 4)  // f || f returns false
o5 = "Cat" || "Dog"     // t || t returns "Cat"
o6 = false || "Cat"     // f || t returns "Cat"
o7 = "Cat" || false     // t || f returns "Cat"
```

8.2 Table Syntax

There's an interesting syntax for calling functions that take a table as their only argument:

```
function foo(t)
  return t[1] * t.x + t[2] * t.y
end

foo{3, 4, x = 5, y = 6} -- 39
```

==

```
foo({ 3, 4, x = 5, y = 6 })
```

8.3 Variable Arguments

Functions support the capturing of a varying number of arguments:

```lua
function sum(...)
  local ret = 0
  for i, v in ipairs{...} do ret = ret + v end
  return ret
end

sum(3, 4, 5, 6) -- 18
```

To do so, just put `...` at the end of your argument list.
You can access members of the list of arguments by putting them in a table (`{...}`) or through the `select` function:

```lua
function sum(...)
  local ret = 0
  for i = 1, select("#", ...) do ret = ret + select(i, ...) end
  return ret
end
```

Finally, just to be clear, `...` isn't a data structure of any kind, it's merely a list of values, like what you get from functions returning multiple values.

8.4 Self

Lua gives us a neat piece of syntactic sugar for calling and defining functions. When you have a function inside a table, you can do stuff like this:

```lua
t = {}

function t:func(x, y)
  self.x = x
  self.y = y
end

t:func(1, 1)
print(t.x) -- 1
```

The definition and call translate to:

```lua
function t.func(self, x, y)
  self.x = x
  self.y = y
end

t.func(t, 1, 1)
```

8.5 Loading Files

An important part of any language is how code is separated into multiple files. As you may know, Lua files generally end with the `.lua` extension.
There are a few ways to load and run a file from inside a Lua script. Two of those ways are `dofile` and `loadfile`:

```lua
dofile("test.lua")
loadfile("test.lua")()

func = loadfile("test.lua")
func()
```

All of those methods are essentially equivalent. Lua files are loaded as functions; this explains why `loadfile` returns a function.
In addition to doing the job of `loadfile`, `dofile` also calls the function (if no errors occurred when compiling the file).

This method of loading files as functions opens up some interesting possibilities:

```lua
local message = "Cheese for everyone!"
local t = { 1, 2, 3, 4, 5 }
print(message)
return t
```

As you can see, files can return values and define local variables. Here's how we might go about loading such a file:

```lua
x = dofile("cheese.lua")
```

`x` would be set to `t`, and the message from `cheese.lua` would be printed to the console.

8.6 Package System

A much better and more common way to load files is via the package system:

```lua
require("cheese")
require("folder.subfolder.file")
```

The `require` function takes a path to a Lua file and searches for it in a number of ways.
The manual explains the whole process, so I'll just tell you what you need to know. `require` replaces each dot with the directory separator ("/" on Unix systems).
It then looks for this file in the locations specified by `package.path`, which includes the current directory.

Once the file's been loaded, `require` adds the path you specified to the `package.loaded` table. `require` won't load any path found in this table, ensuring that files will only be loaded once.
That's the essence of the package system, but there's a lot more to it, so be sure to check out the documentation.

9. Module Definition

From a Table

```lua
-- mymodule.lua
local M = {} -- public interface

-- private
local x = 1
local function baz() print 'test' end

function M.foo() print("foo", x) end

function M.bar()
  M.foo()
  baz()
  print "bar"
end

return M

-- Example usage:
local MM = require 'mymodule'
MM.bar()
```

This is a common approach. It is simple, relies on no external code, avoids globals, and has few pitfalls. Externally facing variables are prefixed by "`M.`" and clearly seen.

See Also

- AlternativeModuleDefinitions

Please refer to the below wiki site which describes how to make a Lua modules.
http://lua-users.org/wiki/ModulesTutorial

10. Metatables

Metatables are a very powerful and flexible feature of Lua that allow you to modify how tables behave.
With them you can implement object-oriented programming, advanced data structures, and a lot more.

Every table can have a metatable attached to it.
A metatable is a table which, with some certain keys set, can change the behaviour of the table it's attached to. Let's see an example.

```lua
t = {} -- our normal table
mt = {} -- our metatable, which contains nothing right now
setmetatable(t, mt) -- sets mt to be t's metatable
getmetatable(t) -- this will return mt
```

In this case we could contract the first three lines into this :

```lua
t = setmetatable({}, {})
```

`setmetatable` returns its first argument, therefore we can use this shorter form.

Metatables can contain anything, but they respond to certain keys (which are strings of course) which always start with `__` (two underscores in a row), such as `__index` and `__newindex`.
The values corresponding to these keys will usually be functions or other tables.
An example:

```lua
t = setmetatable({}, {
  __index = function(t, key)
    if key == "foo" then
      return 0
    else
      return table[key] --> return rawget(t,key)
    end
  end
})
```

So as you can see, we assign a function to the `__index` key. Now let's have a look at what this key is all about.

10.1 __index

The most used metatable key is most likely `__index`; it can contain either a function or table.
When you look up a table with a key, regardless of what the key is (`t[4]`, `t.foo`, and `t["foo"]`, for example),
and a value hasn't been assigned for that key, Lua will look for an `__index` key in the table's metatable (if it has a metatable).
If `__index` contains a table, Lua will look up the key originally used in the table belonging to `__index`.
This probably sounds confusing, so here's an example:

```lua
other = { foo = 3 }
t = setmetatable({}, { __index = other })
t.foo -- 3
t.bar -- nil
```

If `__index` contains a function, then it's called, with the table that is being looked up and the key used as parameters.
As we saw in the code example above the last one, this allows us to use conditionals on the key, and basically anything else that Lua code can do.
Therefore, in that example, if the key was equal to the string "foo" we would return 0, otherwise we look up the `table` table with the key that was used;
this makes `t` an alias of `table` that returns 0 when the key "foo" is used.

10.2 __newindex

Next in line is `__newindex`, which is similar to `__index`. Like `__index`, it can contain either a function or table.
When you try to set a value in a table that is not already present, Lua will look for a `__newindex` key in the metatable.
It's the same sort of situation as `__index`; if `__newindex` is a table, the key and value will be set in the table specified:

```lua
other = {}
t = setmetatable({}, { __newindex = other })
t.foo = 3
other.foo -- 3
t.foo -- nil
```

As would be expected, if `__newindex` is a function, it will be called with the table, key, and value passed as parameters:

```lua
t = setmetatable({}, {
  __newindex = function(t, key, value)
    if type(value) == "number" then
      rawset(t, key, value * value)
    else
      rawset(t, key, value)
    end
  end
})

t.foo = "foo"
t.bar = 4
t.la = 10
t.foo -- "foo"
t.bar -- 16
t.la -- 100
```

10.3 __call

Next comes the `__call` key, which allows you to call tables as functions. A code example:

```lua
t = setmetatable({}, {
  __call = function(t, a, b, c, whatever)
    return (a + b + c) * whatever
  end
})

t(1, 2, 3, 4) -- 24
```

The function in call is passed the target table as usual, followed by the parameters that we passed in.
`__call` is very useful for many things.
One common thing it's used for is forwarding a call on a table to a function inside that table.
An example is found in kikito's tween.lua library, where `tween.start` can also be called by calling the table itself (`tween`).
Another example is found in MiddleClass, where a classes' `new` method can be called by just calling the class itself.

10.4 rawget and rawset

There are times when you need get and set a table's keys without having Lua do it's thing with metatables.
As you might guess, `rawget` allows you to get the value of a key without Lua using `__index`, and
`rawset` allows you to set the value of a key without Lua using `__newindex` (no these don't provide a speed increase to conventional way of doing things).
You'll need to use these when you would otherwise get stuck in an infinite loop.
For example, in that last code example, the code `t[key] = value * value` would set off the same `__newindex` function again, which would get you stuck in an infinite loop.
Using `rawset(t, key, value * value)` avoids this.
As you probably can see, to use these functions, for parameters we must pass in the target table, the key, and if you're using `rawset`, the value.

10.5 Metamethods

Method	Description
__index(*table*, *index*)	Fires when table[index] is indexed, if table[index] is nil. Can also be set to a table, in which case that table will be indexed.
__newindex(*table*, *index*, *value*)	Fires when table[index] tries to be set (table[index] = value), if table[index] is nil. Can also be set to a table, in which case that table will be indexed.
__call(*table*, ...)	Fires when the table is called like a function, ... is the arguments that were passed.
__concat(*table*, *value*)	Fires when the .. concatenation operator is used on the table.
__unm(*table*)	Fires when the unary – operator is used on the table.
__add(*table*, *value*)	The + addition operator.
__sub(*table*, *value*)	The – subtraction operator.
__mul(*table*, *value*)	The * mulitplication operator.
__div(*table*, *value*)	The / division operator.
__mod(*table*, *value*)	The % modulus operator.
__pow(*table*, *value*)	The ^ exponentiation operator.
__tostring(*table*)	Fired when tostring is called on the table.
__metatable	if present, locks the metatable so getmetatable will return this instead of the metatable and setmetatable will error. Non-function value.
__eq(*table*, *value*)	The == equal to operator*
__lt(*table*, *value*)	The < less than operator*; **NOTE:** Using the >= greater than or equal to operator will invoke this metamethod and return the opposite of what this returns, as greater than or equal to is the same as not less than.
__le(*table*, *value*)	The <= operator*; **NOTE:** Using the > greater than operator will invoke this metamethod and return the opposite of what this returns, as greater than is the same as not less than or equal to.
__mode	Used in weak tables, declaring whether the keys and/or values of a table are weak.
__gc(*table*)	Fired when the table is garbage-collected.
__len(*table*)	Fired when the # length operator is used on the Object. **NOTE:** Only userdatas actually respect the __len() metamethod in Lua 5.1

* Requires two values with the *same* metatable and basic type (table/userdata/etc.); does not work with a table and another random table, or with a userdata and a table.

10.6 Wrap-UP

To wrap everything up, we'll write a class encapsulating a 2D vector (thanks to hump.vector for much of the code).
It's too large to put here, but you can see the full code at gist #1055480.

```
Vector = {}
Vector.__index = Vector
```

This code sets up the table for the `Vector` class, and sets the `__index` key to point back at itself.
Now, what's going on here? You've probably noticed that we've put all the metatable methods inside the `Vector` class.
What you're seeing is the simplest way to achieve OOP (Object-Oriented Programming) in Lua.
The `Vector` table represents the class, which contains all the functions that instances can use. `Vector.new` (shown below) creates a new instance of this class.

```
function Vector.new(x, y)
  return setmetatable({ x = x or 0, y = y or 0 }, Vector)
end
```

It creates a new table with `x` and `y` properties, and then sets the metatable to the `Vector` class.
As we know, `Vector` contains all the metamethods and especially the `__index` key.
This means that we can use all the functions we define in `Vector` through this new table. We'll come back to this in a moment.
Another important thing is the last line:

```
setmetatable(Vector, { __call = function(_, ...) return Vector.new(...) end })
```

This means that we can create a new `Vector` instance by either calling `Vector.new` or just `Vector`.
The last important thing that you may not be aware of is the colon syntax. When we define a function with a colon, like this:

```
function t:method(a, b, c)
  -- ...
end
```

What we are really doing is defining this function:

```
function t.method(self, a, b, c)
  -- ...
end
```

This is syntactic sugar to help with OOP. We can then use the colon syntax when calling functions:

```lua
-- these are the same
t:method(1, 2, 3)
t.method(t, 1, 2, 3)
```

Now, how do we use this `Vector` class? Here's an example:

```lua
a = Vector.new(10, 10)
b = Vector(20, 11)
c = a + b
print(a:len()) -- 14.142135623731
print(a) -- (10, 10)
print(c) -- (30, 21)
print(a < c) -- true
print(a == b) -- false
```

Because of the `__index` in `Vector`, we can use all the methods defined in the class through the instances.

10.7 Lua OOP

http://lua-users.org/wiki/ObjectOrientedProgramming

11. Lua C-API Basics

11.1 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.

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

11.2 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 function 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

11.3 A simple REPL

- The code below implements a very primitive REPL:

```c
#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;
}
```

11.4 Lua states

- The Lua interpreter does not define any C global variable ; 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 example , but application is free to have multiple Lua states, and they are completely independent.

11.5 Loading and calling a chunk

- The **luaL_loadstring** function loads a chuck 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 chuck has syntax errors. I**uaL_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.

11.6 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**:

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

11.7 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);
```

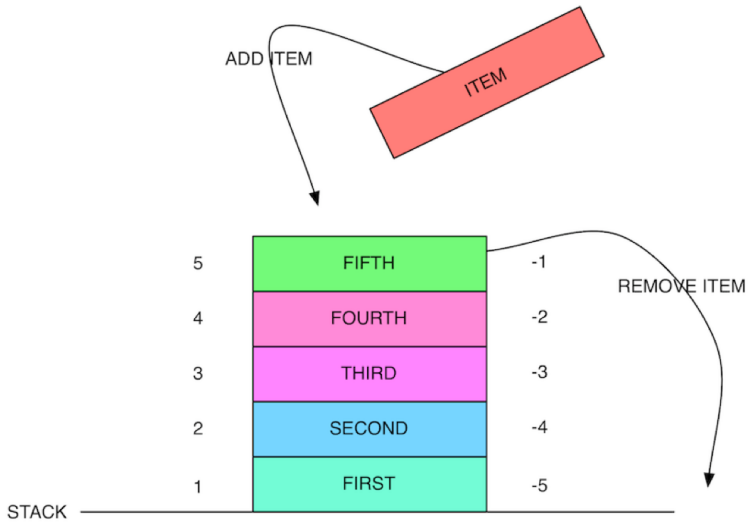
- You can also push a fresh table with:

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

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

11.8 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.



11. 9 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**.

11.10 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 point 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.

11.11 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

11.12 Quiz

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

Q1 :
lua_pushnumber(L, 3.5);

lua_pushstring(L, “hello”);
lua_pushnil(L);
lua_pushvalue(L, -2);
lua_remove(L, 1);
lua_insert(L, -2);

Q1 Answer :

Q2 :
lua_pushboolean(L, 1);
lua_pushnumber(L, 10);
lua_pushnil(L);
lua_pushstring(L, “hello”);

lua_pushvalue(L, -4);
lua_replace(L, 3);
lua_settop(L, 6);
lua_remove(L, -3);
lua_settop(L, -5);

Q2 Answer :

11.13 Lau C-API manual site
http://www.lua.org/pil/24.html

Calling a C function from Lua

- Function receives a Lua state (stack) and returns (in C) number of results (in Lua)
- Get arguments from the stack , do computation, push arguments into the stack.

```
static int l_sin (lua_State *L) {
  double d = lua_tonumber(L, 1);  /* get argument */
  lua_pushnumber(L, sin(d));  /* push result */
  return 1;  /* number of results */
}
```

Any function registered with Lua must have this same prototype, defined as `lua_CFunction` in `lua.h`:

```
typedef int (*lua_CFunction) (lua_State *L);
```

Before we can use this function from Lua, we must register it.
There are some way to register c-function, but we only investigate the below approach.

Lua 5.1 : luaL_register : http://www.lua.org/manual/5.1/manual.html#luaL_register

```
extern "C"
{
#include <lua.h>
#include <lauxlib.h>
#include <lualib.h>
}

#include <sstream>

namespace
{

int Hoge_hoge(lua_State* L)
{
    const char* s = luaL_checkstring(L, 1);
    std::stringstream ss;
    ss << "<hoge>" << s << "</hoge>";
    lua_pushstring(L, ss.str().c_str());
    return 1;
}

const luaL_Reg hogelib[] =
{
    { "hoge", Hoge_hoge },
    { NULL,   NULL      }
};

} // namespace

extern "C" int luaopen_hoge(lua_State* L)
{
    luaL_register(L, "Hoge", hogelib);
    return 1;
}
```

Lua 5.2 :
luaL_setfuncs : http://www.lua.org/manual/5.2/manual.html#luaL_setfuncs
luaL_newlib : http://www.lua.org/manual/5.2/manual.html#luaL_newlib

```
extern "C" int luaopen_hoge(lua_State* L)
{
    luaL_newlib(L, hogelib);
    return 1;
}
```

Code for supporting Lua 5.1 and 5.2
Use LUA_VERSION_NUM to detect the different version

```
extern "C" int luaopen_hoge(lua_State* L)
{
#if LUA_VERSION_NUM >= 502
    luaL_newlib(L, hogelib);
#else
    luaL_register(L, "Hoge", hogelib);
#endif
    return 1;
}
```

[[Incompatibilities with the Previous Lua Version]]

Note : There is a little bit different to register c-library to a Lua environment between Lua 5.1 and other version , so you should read Lua 5.1 manual to do it.

The below site provides a great sample code which shows how to make a C-library module that is called by Lua script.
https://john.nachtimwald.com/2014/07/12/wrapping-a-c-library-in-lua/

```c
static const struct luaL_Reg lcounter_methods[] = {
    { "add",         lcounter_add       },
    { "subtract",    lcounter_subtract  },
    { "increment",   lcounter_increment },
    { "decrement",   lcounter_decrement },
    { "getval",      lcounter_getval    },
    { "getname",     lcounter_getname   },
    { "__gc",        lcounter_destroy   },
    { "__tostring",  lcounter_tostring  },
    { NULL,          NULL               },
};

static const struct luaL_Reg lcounter_functions[] = {
    { "new", lcounter_new },
    { NULL,  NULL         }
};

int luaopen_lcounter(lua_State *L)
{
    /* Create the metatable and put it on the stack. */
    luaL_newmetatable(L, "LCounter");
    /* Duplicate the metatable on the stack (We know have 2). */
    lua_pushvalue(L, -1);
    /* Pop the first metatable off the stack and assign it to __index
     * of the second one. We set the metatable for the table to itself.
     * This is equivalent to the following in lua:
     * metatable = {}
     * metatable.__index = metatable
     */
    lua_setfield(L, -2, "__index");

    /* Set the methods to the metatable that should be accessed via object:func */
    luaL_setfuncs(L, lcounter_methods, 0);

    /* Register the object.func functions into the table that is at the top of the
     * stack. */
    luaL_newlib(L, lcounter_functions);


    return 1;


}
```

11.14 Userdata and Light Userdata

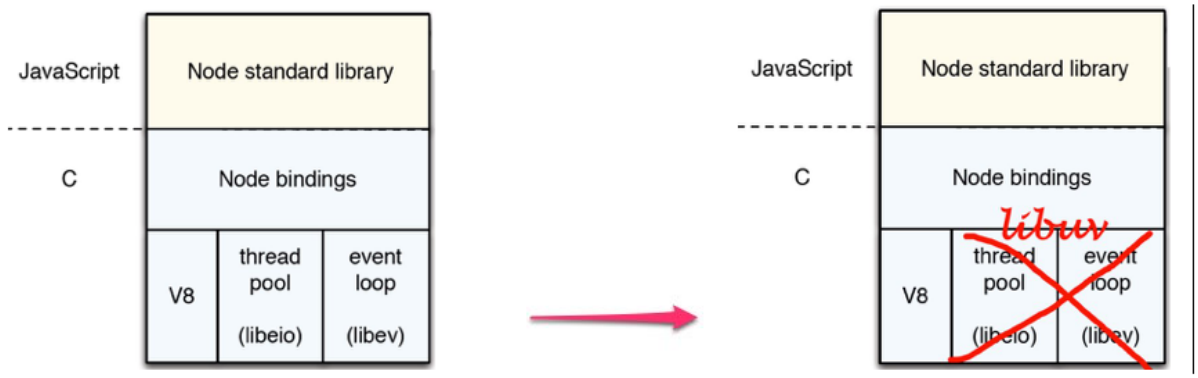
Userdata : http://www.lua.org/pil/28.1.html
Userdata sample : http://lua-users.org/wiki/UserDataExample
http://lua-users.org/wiki/UserDataWithPointerExample
Light Userdata : http://www.lua.org/pil/28.5.html

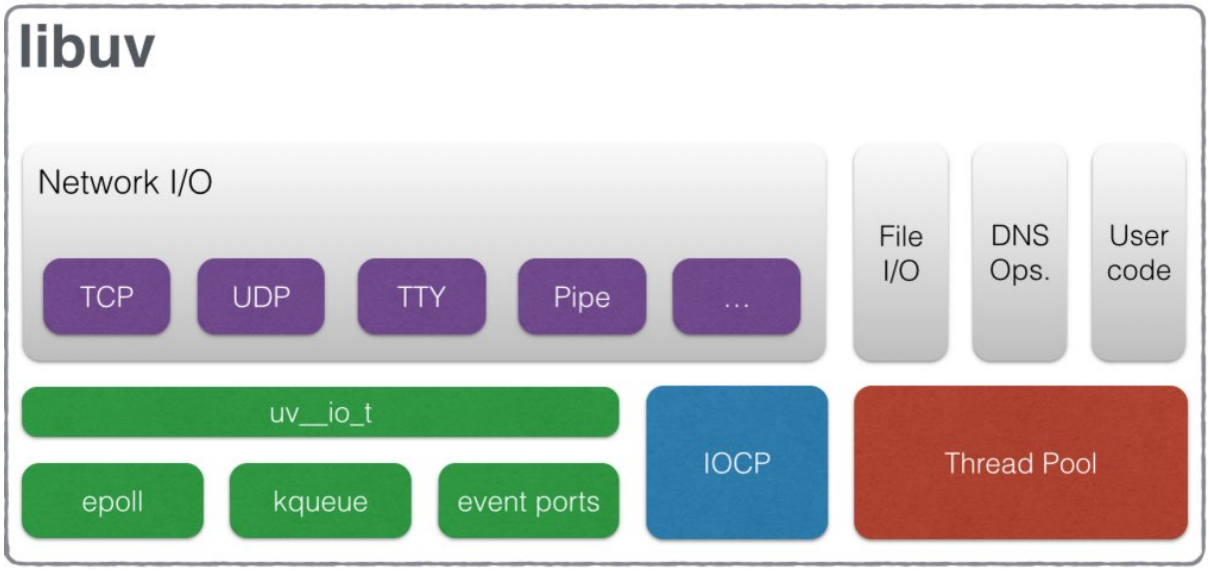
12. Interesting Lua Project

1. eLua project http://www.eluaproject.net/ : Embedded Lua for MCU
2. LuaJIT http://luajit.org/ : A Just-In-Time Compiler for Lua
http://luajit.org/ext_ffi.html : FFI (Foreign Function Interface) library allows calling external C functions and using C data

structure from pure Lua code.

3. Luvit https://luvit.io/ : Luvit implements the same APIs as Node.js, but in Lua!
   libuv : https://github.com/libuv/libuv





4. Lua LTN12 http://w3.impa.br/~diego/software/luasocket/ltn12.html : Network support for the Lua language
5. LuaRocks https://luarocks.org/ : **LuaRocks** is the package manager for Lua modules.
6. NodeMCU http://nodemcu.com/index_en.html
7. Mihini http://www.eclipse.org/proposals/technology.mihini/
8. LuaForge : http://luaforge.net/projects/

13. Please refer to the below site for for getting more knowledge about Lua.

13.1 Lua Metatable

http://www.lua.org/pil/13.html
http://lua-users.org/wiki/MetatableEvents
http://lua-users.org/wiki/MetamethodsTutorial
http://phrogz.net/lua/LearningLua_ValuesAndMetatables.html
http://stackoverflow.com/questions/10891957/difference-between-tables-and-metatables-in-lua
http://wiki.roblox.com/index.php/Metatable

13.2 Lua OOP

http://lua-users.org/wiki/ObjectOrientationTutorial

13.3 Lua Data structures

http://wiki.roblox.com/index.php?title=Tables#Arrays
http://wiki.roblox.com/index.php?title=Stack
http://wiki.roblox.com/index.php?title=Linked_lists
http://wiki.roblox.com/index.php?title=Set_(collection)

13.4 Lua Learning Reference site :

http://tylerneylon.com/a/learn-lua/ : 15-min lecture.
http://wiki.roblox.com/index.php?title=Function_dump/Core_functions
http://wiki.roblox.com/index.php?title=Function_dump/Coroutine_manipulation
http://wiki.roblox.com/index.php?title=Function_dump/String_manipulation
http://wiki.roblox.com/index.php?title=Function_dump/Table_manipulation
http://wiki.roblox.com/index.php?title=Tables