

Lua Seminar

January 20th, 2009 –Randall Knapp

This and more information available at <http://randallknapp.com/lu>

Introduction

These notes are for a seminar given by Randall Knapp at DigiPen Institute of Technology on January 20th, 2009. This seminar is designed to give a brief overview of the Lua programming language and an introduction to using it with existing C or C++ code. The first part of this seminar assumes the participant is very familiar with programming in general (no previous knowledge in any specific language required to learn Lua) and the second part assumes they have extensive knowledge of and are proficient in C or C++.

Keywords:

```
if
then
else
elseif
end
for
while
do
until
repeat
break
true
false
function
local
in
return
not
or
and
nil
```

Variables, Values and Types

Lua has five types: Nil (a special type), Number, String, Table and Function. Only Values have types, Variables have names and an associated value.

- **Nil:** a special representation of nothing.
- **Number:** any numerical value (implemented in C as doubles).
- **String:** a string of characters.

- **Table:** the only container type in Lua. It is an associative container, with a key, value pair for every entry.
- **Function:** a Lua function. These are first-class values, so they may be assigned to.

Variables have names, and any variable is declared like this:

```
name = value
```

(No semicolons are ever required in Lua. If you put them in, Lua will ignore them, but you still shouldn't.)

Variables declared in such a way are always global, and accessible anywhere in the code. To avoid this (and you should avoid it whenever possible) declare variables with the keyword `local`. Then the variable only has scope in the block it was declared in.

Remember, variables do not have types, so the following code is completely ok:

```
local Var          -- by default, uninitialized variables are nil

Var = nil          -- but we can still set it to nil
Var = 5            -- a number
Var = "Hello Lua!" -- a string
Var = { 2, 3 }     -- a table (more on this later)
Var = print        -- a function (print is a global function)
```

Numbers

A number is any numerical value. They may or may not include the decimal point, and there is no need to mark it like a float (5.3f) or anything like that. 5, 7.2, 0.6, .8 are all properly declared numbers.

Boolean values `true` and `false` are actually numbers.

Lua has arithmetic operators as follows: the binary + (addition), - (subtraction), * (multiplication), / (division), % (modulo), and ^ (exponentiation); and unary - (negation). The operands may be numbers or string representations of numbers.

Strings

A string is any characters in double or single quotes. "My string" and 'another one' are both examples of proper strings. You can do normal escape sequences inside of strings: \ and \" are necessary only inside the same type of string definition. Strings can be concatenated using the special string concatenation operator, double-dots:

```
local S3 = S1 .. S2    -- S3 = "Hello world!"
```

Tables

Tables are special. A table is declared using curly-braces:

```
local MyTable = { 3, 4, 5 }
```

However, any type can be used for either the key or the value for each item in a table:

```
local MyTable2 = { nil, 2, "Test", MyTable, print }
```

Tables are indexed starting with 1 (remember this!) so from the previous example:

```
MyTable2[1]    -- is nil
MyTable2[2]    -- is 2
MyTable2[3]    -- is "Test"
MyTable2[4]    -- is { 3, 4, 5 }
MyTable2[5]    -- is print (the global function)
```

You can also declare tables and give both the key and value using the '=' operator:

```
local T = { [3] = "words", ["test"] = print }
```

You'll notice that you have to put the key in square brackets when doing this. You may use a key without brackets or quotes (similar to declaring a member datum in C). This will create a string keyed element with the given key.

```
local Q = { alpha = 5, beta = print, gamma = "Lua sucks!" }
Q["alpha"] -- is 5
Q["beta"]  -- is print
Q["gamma"] -- is "Lua sucks!"
```

You can also index into a table using the dot operator. The identifier following the dot is a string index into the table:

```
Q.alpha -- is equivalent to Q["alpha"]
Q.beta  -- Q["beta"]
Q.gamma -- Q["gamma"]
```

Be careful, when you index using the dot operator, the string has to be a valid identifier, so it can't start with a number.

```
Q.3 -- This is not ok!
```

Tables have a special operator, the length operator. This returns the number of elements in tables indexed by sequential numbers only!

```
local X = { "a", "b", "c" }
local Y = { "x" = 3, "y" = 7, "z" = 4.5 }

#X -- is 3
#Y -- is nil
```

Functions

Functions are declared one of the following two ways:

```
function FuncName( Params )
    block
end
```

```
FuncName = function( Params )
    block
end
```

They are syntactically identical. The parameters are simply a list of variable names separated by commas. The block is any valid Lua code. You can create local functions by adding the `local` keyword before the declaration. More on functions later.

Comments

You have seen the first type of comment in Lua already, the double-dash. This works just like the double slash in C++, it comments out the entire line after it. The other comment type is the block comment. The syntax is pretty simple:

```
--[[
This is commented out.
--]]

This = "isn't."
```

There are a couple more cool things about block comments, so if you want to know more, ask me later.

Conditional Statements

Just like in C there are conditional statements in Lua. Here are some examples of the syntax:

```
if condition then
    block
end

if condition then
    block
else
    block
end

if condition then
    block
elseif condition then
    block
elseif condition then
    block
else
    block
end
```

Conditional expressions use the following operators:

```
==  -- equals
```

```
~= -- not equals
>, >=, <, <= -- just like in C
not, or, and -- these are the logical operators
```

The blocks are any valid Lua code.

Loops

There are four kinds of loops in Lua: do, while, repeat and for. Syntax for do, while and repeat is as follows:

```
do
  block
end -- Yes that's it. I guess it's not really a loop

while condition do
  block
end

repeat
  block
until condition
```

For Loops

For loops are special and confusing, so I'll be concise. There are two kinds, a numeric for loop and a general for loop:

```
for VariableName = InitialValue, Limit, Step do
  block
end
```

You first initialize a variable, then set the limit you want to count to, then set the incremental value added to the variable each loop. Here is an example:

```
for i = 1, 4, 1 do
  print( i )
end
```

```
-- output:
-- 1
-- 2
-- 3
-- 4
```

The second type is the generic for loop. These are complicated; you can look up the full notation in the Lua Manual. The most common usage is as follows:

```
for Key, Value in pairs( TableName ) do
  block
end
```

`pairs()` is a function that iterates through a table in key/value pairs. The variables you name for *Key* and *Value* are local variables for the block. The Lua Interpreter given with this seminar includes a function called `PrintTable` that has a good example of a generic for loop.

Loops can all be broken out of using the `break` keyword. There is no continue keyword. Look up the `next()` function in the Lua Manual for more information on continue-like functionality.

Functions

Functions in Lua are most likely not like anything you've programmed before. Parameter names given in the function declaration are local variables for block of that function. The ... symbol may be given as the last or only parameter of a function's parameter list and denotes a variable number of function arguments. There is a default local table named `arg` in this case that is accessed like a normal table and contains all the arguments included in the ... parameter. Sometimes it is convenient to pass in all the values of a table as parameters for a function. There is a built in function called `unpack()` that does this for you.

```
function Func( ... )
  for i = 1, #arg, 1 do
    print( arg[i] )
  end
end

local T = {"Lua", "is", "great!"}

Func( T )
-- table: 0xGARBAGE
Func( unpack( T ) )
-- Lua
-- is
-- great!
```

Just a quick mention, the `type()` function is very useful; it returns a string equal to the type of the variable.

Not all of the parameters are required to be passed in to the function, and the remainder of the named arguments are equal to nil. If extra parameters are passed in, they are ignored.

Functions in Lua may return more than one value! For example:

```
function ReturnMore( foo )
  return foo, foo + 1, foo + 2
end

x, y, z = ReturnMore( 3 )
-- x = 3, y = 4, and z = 5
```

Eh, that is kind of a lame example, but you get the idea.

There is more to learn about functions including: closures, coroutines, lexical visibility, etc. It is up to you to find out more on your own. The Programming in Lua book will be a great help for this.

Some Useful Built-in Functions

- `type(any)` – returns the type of any variable as a string. Can be “nil”, “number”, “string”, “table”, or “function”.
- `dofile([string] filename)` – opens the file specified, compiles the Lua code and runs it.

The C API

The most important part about Lua is getting it to work with you existing C or C++ code. This is an INTRODUCTION to the C API, not a complete reference. The Lua Manual is very good and available online. I will cover topics relevant to beginning using the Lua C API but leave in-depth experimentation to you. This section assumes you are a proficient C or C++ programmer. Here we go...

The Stack

As Lua executes code it maintains a virtual stack to pass values back and forth between C and Lua. When a function is called, it gets a new stack that is independent from the main stack and any other stacks used by other functions. Each element on the stack is a Lua value (nil, number, string, table or function).

Stack Indexing

Instead of following typical stack discipline, the Lua stack allows you to index into the stack and access its values. Any index that is positive indexes starting at the “bottom” of the stack. Remember that indices start from 1 in Lua. So an index of 1 is the first value on the stack, and an index of 3 would be the 3rd value on the stack. Alternately, any index of negative value indexes from the top of the stack. So if a value is pushed on to the stack it is at index -1, and any value pushed on the stack will be at index -1 immediately afterward. Don't worry, you'll get the hang of it.

Stack Overflow

You will need to maintain stack consistency for yourself while using the Lua C API. The function `lua_pop()` will be extremely useful. The Lua Manual notes for each API call how many items they push on to the stack, and how many they pop off. Be sure that anything you push on to the stack (that isn't a return value, more on that below) in a function is popped off before the function returns. A function will always pop itself and its parameters off the stack when finished.

Lua Glue

Functions that are called from Lua that execute C code are commonly referred to as Lua Glue functions. This section will give an example of how to make Lua Glue Libraries.

Your First Lua Library

To start, you'll need to make your .h and .cpp files for your library, and be sure to #include "lua.hpp" at the top of both. The .h file will be minimal, and only include the declaration for your registration function.

The Setup

All the glue functions will go in your .cpp file. Every glue function has the same signature:

```
static int LuaFunctionName( lua_State* L );
```

The return value is an integer that indicates to Lua how many values you have returned to it (remember, you can return multiple values). The name is whatever you want to name it, but I like to use the following convention: `Lua_LibraryName_FunctionName` where *LibraryName* is the name of the library you registry with Lua and *FunctionName* is exactly the same as the name of the function you call in Lua.

The Register Function

This function needs to be called somewhere in your C code before you can use the functions you have defined. There are two parts to this, the registration array, and the registration function. The array looks like this:

```
static const lua_Reg LibraryNameLib[] =
{
    {"FunctionName", Lua_LibraryName_FunctionName},
    ...
    {NULL, NULL}
};
```

This is a null-terminated array of structures containing the string name of the function (how it will be called in Lua) and the function to call. The second part is the function. It can be of any form depending on how it gets called (it can even be a glue function itself, if you want to call it from Lua) but the body is important.

```
void RegisterMyLibrary( lua_State* L )
{
    lua_openlib( L, "LibraryName", LibraryNameLib, 0 );
}
```

The Functions

That's it. Now it's time to write those functions. One thing I like to do for my glue functions is make a comment in the function header showing how it will be called in Lua and what I expect the arguments to be. Let's make at an example glue function:

```
/// Test.IsAwesome( [string] Name )
static int Lua_Test_IsAwesome( lua_State* L )
{
    // The first value on the stack is the string we passed in as Name
    std::string Name = luaL_checkstring( L, 1 );

    // Lets do something with our string...
```

```

std::cout << Name << " is awesome!!!" << std::endl;

// We return 0 because we arent giving anything back to Lua
return 0;
}

```

As you can see, this function takes a string and prints it out with embellishments. Here is one of the first examples we have seen of Lua Stack manipulation. The `luaL_checkstring(L, 1)` function looks at the first value on the stack, makes sure it is a string and returns it.

Let's discuss some other topics in Lua stack manipulation. We can start with some function descriptions.

Check Functions: These functions return Boolean values for whether or not the value at the specified index is of the corresponding type. There are others, but here are the basics.

1. `lua_isnil(L, index)`
2. `lua_isstring(L, index)`
3. `lua_isnumber(L, index)`
4. `lua_istable(L, index)`
5. `lua_isfunction(L, index)`

To Functions: These functions take a value on the stack at the given index and return it as the requested type. It is better to use the `luaL_check` functions, because they check that the type is correct before conversion. Using these functions simply casts a void pointer, so they could be misused.

1. `lua_tonumber(L, index)`
2. `lua_tointeger(L, index)`
3. `lua_toboolean(L, index)`
 - The proper way to use this function is unintuitive. It actually returns a 0 or 1, not true or false. Try this to avoid a warning:
`(lua_toboolean(L, i))? true : false;`
4. `lua_tostring(L, index)`

Push Functions: These functions take values from C and push them onto the stack. There is a generic push value function that simply takes an index and re-pushes the value at that index on top of the stack.

1. `lua_pushinteger(L, i)`
2. `lua_pushnumber(L, n)`
3. `lua_pushstring(L, s)`
4. `lua_pushvalue(L, index)`

Table Manipulation: These functions are for dealing with tables.

1. `lua_getfield(L, index, name)` - pushes onto the stack a field from the table at the given index with the given name.
2. `lua_gettable(L, index)` - pushes onto the top of the stack the value from the table at the given index with a key that is currently on top of the stack.
3. `lua_next(L, index)` - Iterates through a table, here is how:

```
// Assume there is a table on top of the stack
lua_pushnil( L );
while( lua_next( L, -2 ) )
{
    // the key is now at -2, the value is at -1
    ...
    lua_pop(L, 1); // pop the value off
}
```

Other Functions:

1. `lua_pop(L, count)` - pops *count* values off of the stack
2. `lua_pcall(L, nargs, nresults, errorfunc)` - calls a function on the top of the stack with *nargs* pushed on before it. If an *errorfunc* is given, it is called if an error is encountered.
3. Go research and find out more interesting functions to use!

Lua Interpreter

Included in this seminar is a copy of the source for a very simple Lua Interpreter program. It can be found at <http://randallknapp.com/lua> and is available free of charge and may be used for any non-commercial reason. I encourage you to download the interpreter and try out what you have learned today. There is an example of a Lua file included, just type `dofile("test.lua")` in the interpreter and it will run the file. Also, in `main.cpp`, there are two examples of how to make Lua Glue functions. The first is just a macro that pushes the function into Lua as a global function with its name as the name of the function and the second is a small example of the way discussed in this seminar.

Conclusion

Well that's all I have for you. If you have more questions about Lua or the C API, please visit the [Lua 5.1 Manual](#) page or email me at randall@randallknapp.com. Thanks for your time, enjoy!