Preface

The guide is for experienced C/C++ programmers, who want to understand Lua, or quickly grasp the key points and conventional programming patterns of Lua. Therefore it's not intended to teach its audiences those as obvious as the grammars for if-conditions and function definitions, and those concepts as basic as Variable and Function common in any modern languages. Instead, the guide is only intended to tell you what distinctive characters Lua has against C/C++, and what widely different programming methodologies Lua brings to us. Do not look down them upon: they are to potentially convert you of your conventional programming world view of C/C++.

The guide is divided into three parts as rudimentary, higher and advanced topics, with each consisting of several chapters. The audiences should read sequentially from the start, but those chapters marked with a "*" (which are discussions on OO implementations in Lua) could be skipped without misunderstanding others (However, I don't suggest you do so). When the first two parts are finished, you will be competent for most Lua programming tasks. The third part is optional.

The guide is not intended to replace the Lua Reference Manual, or a complete Lua textbook. So it doesn't make enough explanations even for some important Lua functions mentioned in it. You should refer to the Lua Reference Manual and/or other materials during or after reading it. (The appendix lists some most useful materials' web links.)

Please access the online edition of this guide for an up-to-date version. In addition, the author has an open-source Lua debugger, RLdb, and a web site discussing Lua. Welcome to access!

Please write to me for any feedback!
Rudimentary Topics

- Data Types
- Function
- Table
- Simple Object Implementation*
- Simple Inheritance*
Data Types

8 types:

- **Number**
  Numeric value represented in double in Lua standard implementation.

- **String**
  A sequence of arbitrary characters (including zero) ending with zero, not equivalent to C string, but indeed its superclass.

- **Boolean**
  Logic type with only two values: "true" and "false".

- **Function**
  First-class object in Lua, not equivalent to C function or function pointer; one of the key concepts of Lua.

- **Table**
  Heterogeneous hash table; also a key concept of Lua.

- **Userdata**
  C data structures defined by C users, accessible but not definable from script.

- **Thread**
  Lua coroutine, a cooperative threading, different from the preemptive threading that most modern OS take.

- **Nil**
  Nothing, different from any other types, and somewhat analogous to NULL in C, but not being an empty pointer!
Function

function foo(a, b, c)
    local sum = a + b
    return sum, c  -- A function can return multiple values.
end

r1, r2 = foo(1, '123', 'hello')  -- Parallel assignment
print(r1, r2)

output:
124 hello
Function definition
Define a function with key word "function" and end it with "end".

Returning multi values from a function
return a, b, c, ...

Parallel assignment
a, b = c, d

Local variable
Variable defined with key word "local". A variable becomes global if no "local" prefix it, even when it's defined within a function body!

Global variable
Any variable defined without a "local" prefixing it (this is not always true: as you will see later, Lua has a third variable scope type, the external local variable). The previous code defined THREE global variables: foo, r1 and r2.
Table

a = { }
b = { x = 1, ["hello, "] = "world!" }
a.astring = "ni, hao!"
a[1] = 100
a["a table"] = b

function foo()
end
function bar()
end
a[foo] = bar

--enumerate out table a and b
for k, v in pairs(a) do
    print(k, "=>", v)
end
print("-----------------------------")
for k, v in pairs(b) do
    print(k, "=>", v)
end

output:
1 => 100
a table => table: 003D7238
astring => ni, hao!
function:
    003DBCE0 => function:
    003DBD00
-----------------------------

hello, => world!
x => 1
- Defining a table
  \( a = \{ \}, b = \{ \ldots \} \)
- Accessing a table's members
  Table members can be accessed via "." or "[]" operators. Note that expression "a.b" is equivalent to "a["b"]", but not equivalent to "a[b]".
- Table entry's key and value
  A variable of any type except type nil, can be used as the key or value of a table entry. Assigning nil to a table entry's value means removing that entry from table. For example, given "a.b = nil", then the entry in table a with its key equal to "b" is removed from a. In addition, accessing a non-existed table entry will get nil. For example, given "c = a.b", if there's no entry in table a with its key equal to "b", then c gets nil.
Simple Object Implementation*

```lua
function create(name, id)
    local obj = { name = name, id = id }
    function obj:SetName(name)
        self.name = name
    end
    function obj:GetName()
        return self.name
    end
    function obj:SetId(id)
        self.id = id
    end
    function obj:GetId()
        return self.id
    end
    return obj
end

o1 = create("Sam", 001)
print("o1's name:", o1:GetName(),
"o1's id:", o1:GetId())
o1:SetId(100)
o1:SetName("Lucy")
print("o1's name:", o1:GetName(),
"o1's id:", o1:GetId())
输出结果:
o1's name: Sam o1's id: 1
o1's name: Lucy o1's id: 100
```
Simple Object Implementation*
(continued)

- Object factory pattern
  See the create function.
- Object Representation
  A table with data and methods in it represents an object. Although there's no way to hide private members in this implementation, it's good enough for simple scripts.
- Defining a member method
  "function obj:method(a1, a2, ...) ... end" is equivalent to
  "function obj.method(self, a1, a2, ...) ... end", which is equivalent to
  "obj.method = function (self, a1, a2, ...) ... end"
- Calling a member method
  "obj:method(a1, a2, ...)" is equivalent to
  "obj.method(obj, a1, a2, ...)"
function createRobot(name, id)
    local obj = { name = name, id = id }

    function obj:SetName(name)
        self.name = name
    end

    function obj:GetName()
        return self.name
    end

    function obj:GetId()
        return self.id
    end

    return obj
end

function createFootballRobot(name, id, position)
    local obj = createRobot(name, id)
    obj.position = "right back"

    function obj:SetPosition(p)
        self.position = p
    end

    function obj:GetPosition()
        return self.position
    end

    return obj
end
Simple Inheritance* (continued)

- **Pros:**
  Simple, intuitive

- **Cons:**
  Conventional, not dynamic enough
Higher Topics

- Function Closure
- Object Based Programming*
- Metatable
- Prototype Based Inheritance*
- Function Environment
- Package
function createCountdownTimer(second)
    local ms = second * 1000
    local function countDown()
        ms = ms - 1
        return ms
    end
    return countDown
end

timer1 = createCountdownTimer(1)
for i = 1, 3 do
    print(timer1())
end
print("------------")

timer2 = createCountdownTimer(1)
for i = 1, 3 do
    print(timer2())
end

output:
999
998
997

------------
999
998
997
Function closure (continued)

- **Upvalue**
  A local variable used in a function but defined in the outer scope of the function is an upvalue (also external local variable) to the function. In the previous code, variable `ms` is an upvalue to function `countDown`, but it's a common local variable to function `createCountdownTimer`. Upvalue is a special feature in Lua which has no counterpart in C/C++.

- **Function closure**
  A function and all its upvalues constitutes a function closure.

- **Function closure VS C function**
  A function closure has the ability to keep its status over callings, while a C function with static local variables can also keep status. However, the two things are quite different: the former is a first-class object in the language, but the latter is only a symbol name for a static memory address; the former can have several instances of the same class, with each having its own status, but the latter is static and thus not to mention instantiation.
Object Based Programming*

```lua
function create(name, id)
    local data = { name = name, id = id }
    local obj = {}
    function obj.SetName(name)
        data.name = name
    end
    function obj.GetName()
        return data.name
    end
    function obj.SetId(id)
        data.id = id
    end
    function obj.GetId()
        return data.id
    end
    return obj
end

o1 = create("Sam", 001)
o2 = create("Bob", 007)
o1.SetId(100)
print("o1's id:", o1.GetId(), "o2's id:", o2.GetId())
o2.SetName("Lucy")
print("o1's name:", o1.GetName(), "o2's name:", o2.GetName())
output:
o1's id: 100 o2's id: 7
o1's name: Sam o2's name: Lucy
```
Object Based Programming* (continued)

- Implementation
  Put private members in a table and use it as an upvalue for public member method, while put all the public members in another table as an object.

- Limitation
  Not flexible concerning inheritance and polymorphism. But it depends whether inheritance and/or polymorphism are required for script programming.
Metatable

t = {}
m = { a = " and ", b = "Li Lei", c = "Han Meimei" }

setmetatable(t, { __index = m }) -- Table { __index=m } is set as t's metatable.

for k, v in pairs(t) do -- Enumerate out table t.
    print(k, v)
end
print("----------")
print(t.b, t.a, t.c)

output:
----------
Li Lei and Han Meimei
function add(t1, t2)
    --Get table length via operator '#'.
    assert(#t1 == #t2)
    local length = #t1
    for i = 1, length do
        t1[i] = t1[i] + t2[i]
    end
    return t1
end

t1 = t1 + t2
for i = 1, #t1 do
    print(t1[i])
end

output:
11
22
33

--setmetatable returns the table set.
t1 = setmetatable({ 1, 2, 3}, { __add = add })
t2 = setmetatable({ 10, 20, 30 }, { __add = add })
Metatable(continued)

- Metatable
A common table usually with some special event callbacks in it, being set to another object via `setmetatable` and thus having effects on the object's behavior. These events (such as `__index` and `__add` in previous codes) are predefined by Lua and the callbacks are defined by script users, invoked by Lua VM when corresponding events happen. For the previous examples, table's addition operation produces an exception by default, while tables with a proper metatable set to them can make it correctly, for Lua VM will call the `__add` callback defined by user in those two tables' addition.

- Overriding operators
You may have realized from the example that the operators, such as `"+"` can be overridden in Lua! That's it! Not only `"+"`, but almost all the operators in Lua can be overridden! (That's one point for why I think Lua is a great script.)

- Metatable VS C++'s vtable
Metatable is a meta object used to affect the behaviors of another object, while vtable is a conceptual object used to point/locate certain behaviors(methods) of a real C++ object. A Lua object can have its metatable changed at runtime, while a C++ object can not change its vtable(if any) at all, for it's produced by a compiler and thus being static and unchangeable.

- More
Metatable is so significant to Lua that I strongly suggest you refer to the Lua Reference Manual for more information.
Prototype Based Inheritance*

Robot = { name = "Sam", id = 001 }

function Robot:New(extension)
    local t = setmetatable(extension or { }, self)
    self.__index = self
    return t
end

function Robot:SetName(name)
    self.name = name
end

function Robot:GetName()
    return self.name
end

function Robot:SetId(id)
    self.id = id
end

function Robot:GetId()
    return self.id
end

robot = Robot:New()
print("robot's name:", robot:GetName())
print("robot's id:", robot:GetId())
print("-----------------")

FootballRobot = Robot:New({position = "right back"})

function FootballRobot:SetPosition(p)
    self.position = p
end

function FootballRobot:GetPosition()
    return self.position
end

fr = FootballRobot:New()
print("fr's position:", fr:GetPosition())
print("fr's name:", fr:GetName())
print("fr's id:", fr:GetId())
print("-----------------"
fr:SetName("Bob")
print("fr's name:", fr:GetName())
print("robot's name:", robot:GetName())

output:
robot's name: Sam
robot's id: 1
-----------------
fr's position: right back
fr's name: Sam
fr's id: 1
-----------------
fr's name: Bob
robot's name: Sam
Prototype Based Inheritance*
(continued)

- Prototype pattern
  A common object is used as a prototype object to create the other objects. Dynamic changes to the prototype object reflect immediately on those created by the prototype object, also on those to be created by it. Moreover, an object created by some prototype object can override any methods or fields belonging to the prototype object, and it can also be a prototype object for creating other objects.
Function Environment

```lua
function foo()
    print(g or "No g defined!")
end

foo()

setfenv(foo, { g = 100, print = print })  -- Set { g=100, ...} as foo's environment.

foo()

print(g or "No g defined!")

output:
No g defined!
100
No g defined!
```
Function Environment (continued)

- Function environment
  A collection for all the global variables a function can access is called that function's environment, held in a table. By default, a function shares the environment of the function that defines it. But each function can have its own environment, set by `setfenv`.
  In the previous code, variable `g` is not defined in `foo`'s environment at first, so the first call to `foo` outputs "No g defined!". Later, an environment with `g` and `print` defined is set to `foo`, so the second call to `foo` outputs `g`'s value. However, `g` is never defined in the environment in which `foo` is defined.

- Application
  Function environment is another special feature of Lua with no counterpart in C/C++. You may wonder what the odd feature can do. Indeed it does a lot! For example, it can be used to implement a security sandbox for executing functions; it's also used to implement the Lua Package.
--testP.lua:

--import package "mypack"
pack = require "mypack"

print(ver or "No ver defined!")
print(pack.ver)

print(aFunInMyPack or "No aFunInMyPack defined!")
pack.aFunInMyPack()

print(aFuncFromMyPack or "No aFuncFromMyPack defined!")
aFuncFromMyPack()

--mypack.lua:

--define a package
module(..., package.seeall)

ver = "0.1 alpha"

function aFunInMyPack()
    print("Hello!")
end

_G.aFuncFromMyPack = aFunInMyPack
output of testP.lua:

No ver defined!
0.1 alpha
No aFunInMyPack defined!
Hello!
function: 003CBFC0
Hello!
Package(continued)

- **Package**
  A way to organize codes.

- **Implementing a package**
  A package is usually a Lua file start with a call to `module`, which defines a new environment for that file. At this point, it's necessary to make it clear that a Lua file's content is treated as a function body by Lua VM (which means you can return something from the file or receive some parameters from outside!). Given a new environment, a Lua file (a function) has all its globals going into that environment table.

  Taking the previous code for example, "module(..., package.seeall)" means defining a package and enabling it to "see" all the globals in the environment of the function `require`'s the package (without `package.seeall`, `print` is unavailable).

- **Using a package**
  `require` imports a package, which must have been on the package path already. The path is similar to that in Java, set via environment or `package.path`. Usually the current working directory is included in the path by default.

- **More**
  Please refer to the Lua Reference Manual for more.
Advanced Topics

- Iteration
- Coroutine
function enum(array)
    local index = 1
    --return the iterating function
    return function()
        local ret = array[index]
        index = index + 1
        return ret
    end
end

function foreach(array, action)
    for element in enum(array) do
        action(element)
    end
end

foreach({1, 2, 3}, print)

output:
1
2
3
Iteration(continued)

- **Iteration**
  A special form of *for* statement, through which an iterating function is called repeatedly on a given collection to traverse it. The formal and complete grammar of the *for* statement is complicated. Please refer to the Lua Reference Manual for details.

- **Implementation**
  Taking the previous code for example: `enum` return an anonymous iterating function, which is called repeatedly by the *for* statement, returning a value held by `element`. If element gets nil, then the *for* loop ends.
Coroutine

function producer()
    return coroutine.create(
        function (salt)
            local t = { 1, 2, 3 }
            for i = 1, #t do
                salt =
                    coroutine.yield(t[i] + salt)
            end
        end
    )
end

output:
11
102
10003
END!

function consumer(prod)
    local salt = 10
    while true do
        local running, product =
            coroutine.resume(prod, salt)
        salt = salt * salt
        if running then
            print(product or "END!")
        else
            break
        end
    end
end

consumer(producer())
Coroutine(continued)

- Coroutine
  A Lua thread type. Rather than preemptive threading, Lua takes a cooperative threading, in which each thread runs until it yield the processor itself.

- Creating a coroutine
coroutine.create creates a coroutine. The function requires a parameter of type function as the thread body and returns a thread object.

- Starting/Resuming a thread
coroutine.resume starts or resumes a thread. The function requires a thread object as the first parameter, and accepts other optional parameters to pass to the thread. When a thread is started, the thread function starts from its beginning and the optional parameters passed to resume are passed to the thread function's parameter list; when a thread is resumed, the thread function continues right after yield, and these optional parameters are returned by yield.

- Yielding
A thread calls coroutine.yield to yield the processor, returning control to the thread who starts/resumes the yielding thread. The yielding thread can pass some values through yield to the thread the control is returned to. The values passed through yield is returned by resume.
function instream()
  return coroutine.wrap(function()
    while true do
      local line = io.read("*l")
      if line then
        coroutine.yield(line)
      else
        break
      end
    end
  end)
end

function filter(ins)
  return coroutine.wrap(function()
    while true do
      local line = ins()
      if line then
        line = '** ' .. line .. ' **
        coroutine.yield(line)
      else
        break
      end
    end
  end)
end

function outstream(ins)
  while true do
    local line = ins()
    if line then
      print(line)
    else
      break
    end
  end
end

outstream(filter(instream()))

input/output:
abc
** abc **
123
** 123 **
^Z
Coroutine (continued)

- Unix pipes and Stream IO
  It's handy to make Unix pipe style or Stream IO style design with coroutine.
Coroutine(continued)

function enum(array)
    return coroutine.wrap(function()
        local len = #array
        for i = 1, len do
            coroutine.yield(array[i])
        end
    end)
end

function foreach(array, action)
    for element in enum(array) do
        action(element)
    end
end

foreach({1, 2, 3}, print)

output:
1
2
3
An alternative implementation for the iterating function
Coroutine can be used to implement the iterating function for the `for`
statement. Although it's unnecessary for traversing simple arrays, but what
about a complicated data collection, say, a binary tree? How can you write
code as simple as "foreache(tree, action)" to traverse the tree while doing
something on each tree node? In this case, coroutine can help you a lot.
Appendix: Usefull Material Links

- **Lua Reference Manual** (authoritative Lua document)
- **Programming in Lua** (also authoritative Lua textbook)
- **Lua official website's document page** (containing many valuable links)
- **lua-users wiki** (most complete wiki for Lua)
- **LuaForge** (most rich open-source code base for Lua)