Working With Lua

A Tutorial with Effective Practices for Games

By:
Tommy A. Brosman IV
November 13, 2007
DigiPen Institute of Technology

Overview

Getting Lua for Windows, Linux, Etc.

http://www.lua.org is the official site for the Lua language. Source available from their site, along with a Makefile for POSIX-compatible systems.

http://luabinaries.luaforge.net/download.html for pre-compiled binaries of libraries/dlls. Also, MS Visual Studio project files for compiling the sourcecode.

Compiling Projects With Lua

GCC/G++

Link with -llua if you have it installed in the PATH
Otherwise, compile a library (*.a file) and link with -llua.a from a local directory
Under Cygwin, compile your *.a file manually by compiling with “make generic”

MSVC

Keep some file lua.lib or lua5.1.lib (or whatever your version is) in the source directory
command line: cl (FLAGS) (OBJECT FILES) lua5.1.lib
from Visual Studio .NET 2005:

1. Move lua5.1.lib to the same directory as your solution (.sln).
2. Open your solution.
3. Right click the active project in the Solution Explorer pane. Select Properties.
4. In the left pane, Configuration Properties->Linker->Input.
5. In the right pane, add lua5.1.lib

If you choose the static library (recommended), your executable size will be a few hundred K larger, but it will run faster and avoid messy DLL imports.

If you choose the DLL version of the binaries, your executable will be smaller, and you will have to remember to include the DLL in the executable directory.

Note that you must match the pre-compiled binary with the version of Visual Studio that you are using. Mismatches will cause conflicts with the CRT (C Runtime) library, as Lua is written completely in ANSI C, and different versions of MSVC do different things in terms of linking the CRT lib.

The Standalone Interpreter

lua.exe (on Windows, just “lua” on Linux) is the standalone Lua interpreter. After building Lua, it can be found in the /src directory under Lua's project folder. Invoke it from bash using ./lua.exe, or simply lua.exe from the non-POSIX Windows command prompt.

Here is your hello world program:

```
print("hello, world")
```

Type it in and press enter. Observe the results. ctrl-c to manually terminate the interpreter. It can also accept script files as input. Save the line of code to a file called test.lua, then run it using ./lua.exe test.lua

Crimson Editor is good for editing Lua files, and has excellent syntax highlighting.

A Quick Note

You do not use semicolons to seperate lines in Lua. Your script will blow up (if you're lucky) if you try to use them. Instead, linebreaks can be used. The linebreaks are optional, and multiple declarations can be combined into a single line, much like C.

Example: `a = 3 while a > 0 do print(a) b = 5 + a print(b) a = a-1 end`
Output:

```
3
8
2
7
1
6
```

Part I: The Language

Operators in Lua

Grouping: `( )`
Arithmetic: `+ - * / ^`
Relational: `== ~= < > <= >=`
Logical: `and or not ~`
String: `..`
Scope Resolution: `. :`

Logical, Relational, and Arithmetic: Differences from C Operators

The ^ operator is an exponent, not XOR.
The unary not symbol is a ~
There are not post/pre-decrement/increment symbols, since the “decrement” operator denotes a comment in Lua.
Logical operators are always written as words, with the exception of not, which can be written as a word or the unary not symbol.

The String Concatenation Operator

The string concatenation operator is a `..`
It is included in Lua as an operator, since strings are treated as first-class objects, just like numbers. You can concatenate mixed objects, just as long as the left-handed expression is a string (there is an exception for this, in the case of `nil`). Note that sometimes you must use parentheses to make sure this is the case.

Example:
```
print("this is the number "..2.."\n") -- won't work, left-hand expression in the right-most concatenation is a number, not a string
```

Example:
```
print(("this is the number "..2).."\n") -- will work, left-hand expression is a string
```

Scope Resolution

This will be covered more in the tables/metatables section. Since there are no pointers in Lua (there are, but they are either internal or you can't dereference them directly) both operators can be used on the same table. The difference is that the `:` operator implicitly passes a “self” pointer (like a “this” in C++). Think of the second dot as a physical representation of the extra parameter being passed.

```
A = {}
B = {}

function A.NormalFunction(a, b, c)
  print("self = "..tostring(self))
  print("a    = "..tostring(a))
  print("b    = "..tostring(b))
  print("c    = "..tostring(c))
end

function B:MemberFunction(a, b, c)
  print("self = "..tostring(self))
  print("a    = "..tostring(a))
  print("b    = "..tostring(b))
  print("c    = "..tostring(c))
end
```

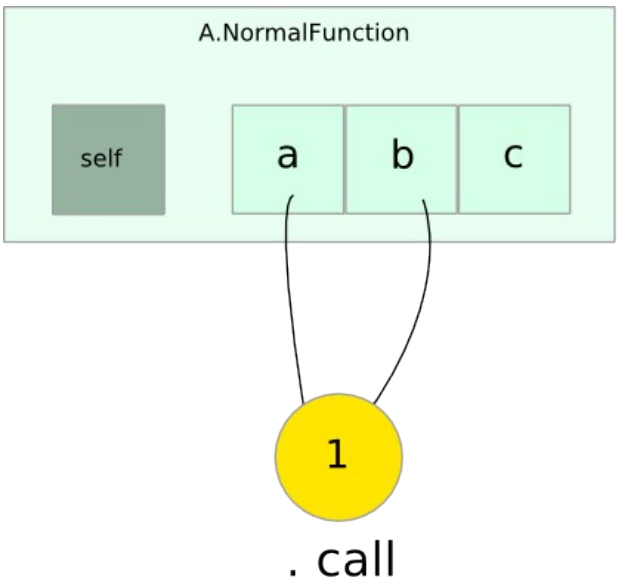
```
print("---"..1)
A.NormalFunction("test", 1)
print("\n---"..2)
A:NormalFunction("test", 2)
print("\n---"..3)
B.MemberFunction("test", 3)
print("\n---"..4)
B:MemberFunction("test", 4)
```

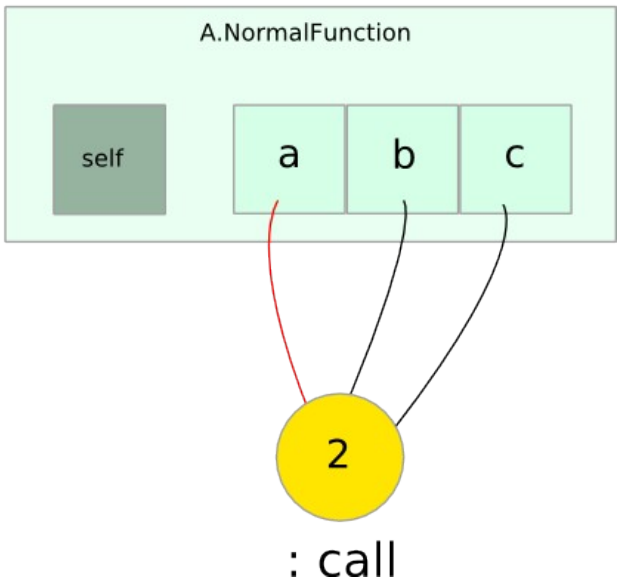
Output:

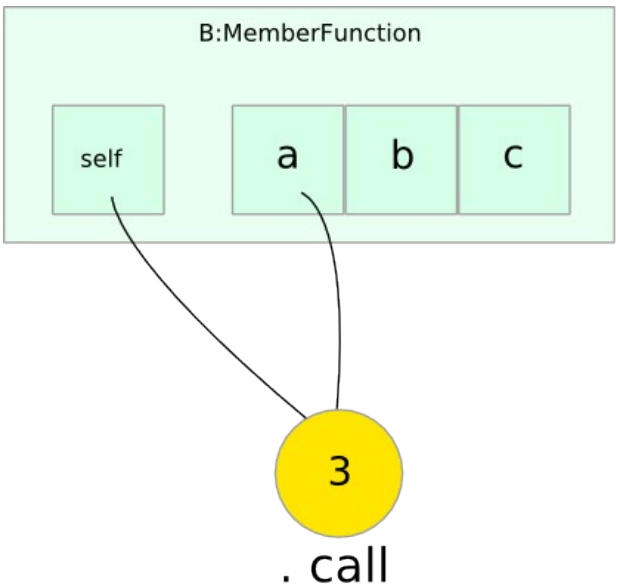
```
---1
self = nil
a    = test
b    = 1
c    = nil

---2
self = nil
a    = table: 0x686290
b    = test
c    = 2
```

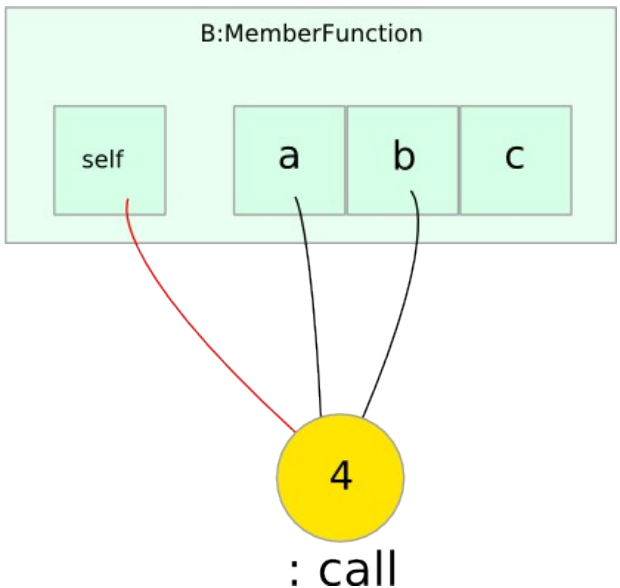
```
---3
self = test
a    = 3
b    = nil
c    = nil

---4
self = table: 0x6866a8
a    = test
b    = 4
c    = nil
```

Note that `tostring()` is used above, since it is not possible for lua to concatenate `nil` objects to a string. Also notice that all functions are automatically “overloaded”, and allow for any number of the given parameters to be specified.


A.NormalFunction
self
a
b
c
1
. call


A.NormalFunction
self
a
b
c
2
: call


B:MemberFunction
self
a
b
c
3
. call


B:MemberFunction
self
a
b
c
4
: call

Other Tokens

Comments

Comments are prefixed with the `--` token, which is the equivalent of the `//` token in C++. There are no multi-line comments in Lua, and block comments are usually written like this:

```
--------------------------------------------------------------------------------
-- block comment
--------------------------------------------------------------------------------
```

While Lua code is relatively easy to read, use file headers at the very least.

Variadic Parameter

The variadic parameter token `...` allows for multiple arguments to be passed as a table, much in the same way the `va_arg` macros work in C. When the function is called, a table called `arg[]` is generated, with the first parameter at `arg[1]` (Lua table indexes start at 1).

Tables and Indexes

Table declaration has a very simple syntax. The tables allow for mixed types, including functions and other tables. You can also declare empty tables (the most common declaration), then add members later.

```
myTable = {}
```

By default, tables have numerical keys starting at 1. You can declare new members implicitly by simply naming them.

```
myTable[1] = "hi"
myTable[2] = 23
```

Any object can be used as a key. The index operator is the default access method, although the dot operator can be used for string keys.

```
myTable["test"] = function() print("a test") end

myTable.test() -- a test
myTable["test"]() -- a test
```

Scope

Function Blocks

The basic syntax for a function with no parameters and no returns:

```
function test()
  --code goes here
end
```

Parameters, as with objects, are untyped. Returns are also untyped.

```
function test2(a, b)
  return a
end
```

Note that the preceding function will also work with only one parameter.
Returns are variadic, and you can return as many parameters as you want. You don't have to use all the returns, you can ignore them just as you would with a single parameter in C/C++.

```
function test3(a)
  return a, 2, "test"
end

test3("testing")
test3()
a, b = test3()
print(a, b) -- nil 2
a, b, c = test3("testing")
print(a, b, c) -- testing 2 test
```

Do Blocks

do-end blocks are the basis for all loops in Lua. They are the equivalent of `{ }` tokens in C/C++, and allows for scope resolution. They can be used with or without loops. The `local` keyword allows for objects to be declared only within a specific scope. This is good practice, as helps the automatic garbage collection and increases the execution speed.

```
a = 4

print(a) -- 4

do
  local a = 6
  print(a) -- 6
end

print(a) -- 4
```

Automatic Garbage Collection

Lua is an interpreted language. It is extremely fast, and most of its speed has to do with its reduction of OOP-specific objects (all non-primitive objects are implemented as arrays/tables, and metatable functions emulate classes), and also with its garbage collection algorithms. Lua will automatically deallocate an object once no remaining references to the object remain within any of the visible scopes. As a result, if you wish to deallocate an object explicitly (not recommended), you can simply assign a `nil` object to it.

Loops

While

```
while expr do
  -- loop code here
end
```

Another difference from C syntax is that parentheses are not required around the loop expression.

C-Style For

```
for i=a,b do
  -- loop code here
end
```

This is the equivalent of the C loop:

```
for(int i = a; a <= b; i++)
{
  // loop code here
}
```

To change the way the loop iterates, add a third argument:

```
for i=a,b,c do
  -- loop code here
end
```

This is the equivalent of the C loop:

```
if(c > 0)
{
  for(int i = a; a <= b; i += c)
  {
    // loop code here
  }
}
else
{
  for(int i = a; a >= b; i += c)
  {
    // loop code here
  }
}
```

Lua automatically adjusts the direction of the loop depending on the sign of the iterator.

For-In

for-in loops work a bit like C++ iterators.

```
-- t is some table
for k,v in pairs(t) do
  -- do something with the value v
end
```

Repeat-Until

```
repeat
  -- some code here
until expr
```

Essentially the opposite of a C do-while statement. Repeats until ***expr*** is true.

Objects

Nil, True, False

`nil` is the equivalent of the C-style `NULL`, but is different from `false`
`true` and `false` are first-class objects, and do not evaluate to integers.

Integers, Numbers, Strings, Booleans

Integers are whole numbers, and Numbers are doubles internally by default. At the script-level, there is no difference between the two.

Strings and string-literals are all first-class objects.

Booleans are either true or false.

Most type conversions take place implicitly, but can be performed implicitly using `tonumber()` and `tostring()`, both of which are built-in functions.

Strings and numbers can be added, multiplied, subtracted, etc and the string portions will be implicitly converted to numbers. This is why there is a concatenation operator seperate from the arithmetic operators.

```
print("4" + 6)
print(7 + "1")
print("3.2" + "0.8")
```

Output:

```
10
8
4
```

First-Class Objects

By definition, there are no primitives in Lua. Because of this, objects do not need to be declared with types. Also, being an interpreted language, there is no need to allocate/deallocate memory (the garbage collection mechanism handles that).

Functions as Objects

Functions in Lua are also objects. Because of this, you can nest functions, and assign them as table members (the way classes are done in Lua). Also, there are no datatypes in declarations, which means that all functions are overloaded for all types, and any errors in the handling of these types will not be caught until runtime.

```
function a()
  function b()
    return "function b"
  end

  print(b())
  return "function a"
end

print(a())
--print(b()) -- not available in this scope
```

Output:

```
function b
function a
```

Heterogeneous Tables

Tables in Lua are like a hybrid of C++ arrays and classes. Unlike C/C++ arrays and containers, Lua tables allow for mixed datatypes. Everything from primitives, to functions, to other tables can be stored in a Lua table. Each table entry is a pair consisting of a key and a value. Any datatype can be used for the key as well, but by default keys are integers starting from 1. Tables are not necessarily continuous, and may have gaps between entries, so use the for-in loop instead of the C-for loop to iterate through them. C-for loops can ONLY be used in a continuous table where all keys are integer values.

```
test = {alpha = 0.5,
        beta  = 1.3,
        gamma = 1.1,
        "string val"}

-- print everything, even non-numerical indices
PrintByPair(test)

-- will print "string val" only, it is the only numerical index in the table
PrintByIndex(test)

-- create a new member called "phi" that is a function
test.phi = function() return "phi" end
-- prints address of phi, does not call it, () operator needed
PrintByPair(test)


-- access using dot operator
print(test.phi())

-- access using array operator
print(test["phi"]())
```

Object-Oriented Programming In Lua (Metatables)

The double-dot operator is used for metatables to pass a “self” pointer. Metatables are an extension for tables, allowing for dynamic objects and operator overloading (not covered).

```
-- the original table
t = {}
-- overload the "index" [] (also dot) operator to access functions from the
-- original table (in other words, don't copy them, just make a reference to
-- the old table)
t_mt = { __index = t }

-- the "constructor"
function t:new()
  return setmetatable({value = 10}, t_mt)
end

-- some member functions
function t:IncValue()
  self.value = self.value + 1
end

function t:DecValue()
  self.value = self.value - 1
end
```

```
-- driver code
myTable = t:new()
myTable2 = t:new()

-- only prints the instance member "value"
for k,v in pairs(myTable) do print(k, v) end

print(myTable.value)

myTable:IncValue()
myTable:IncValue()

print("myTable.value:", myTable.value)    -- prints 12
print("myTable2.value:", myTable2.value) -- prints 10
```

Even though a table is already declared, a seperate metatable is needed as a kindof “template” for the instance version of the table. It overloads the index operator to allow access to the original table functions.

To call a function normally requiring a table instance, you can pass an extra parameter, passing a nil value before the normal parameters, using the single-dot operator to make the function call.

Other Objects

Userdata is data that is created outside of Lua, passed in as a pointer, and can be garbage-collected by Lua. It is rather difficult to use with classes, since classes need destructors to be called by a proper delete function, and Lua is built on ANSI C.

Light Userdata will be covered a bit more in the C++/Lua integration section, and is the same as Userdata, except that it is not subject to Lua's garbage collection.

Threads objects allow for a multithreaded environment. These are not covered, as they are not much use if you're only using Lua as a scripting language, and not to build the application itself.

Built-In Functions and Libraries

Lua's “includes” are done using the requires keyword. For example, to import the math library:

```
requires 'math'
```

Note the use of single-quotes instead of double-quotes. Math is essentially the same as C's math.h library. Most of the functions have the same name, except that you now put a “math.” before the function name.

Other libraries:

io
table
os

IO is mainly file operations and functions, and behaves like the C file interface. Table contains table-related functions, os contains things like time and date, and you can use the os.execute function to build shell-scripts.

Part II: Integration with C++

Lua Headers

There are 3 headers that are common to almost every application using Lua:

lua.h
lualib.h
lauxlib.h

lua.h contains all the basic integration functions, including ones for opening and closing the Lua state, performing stack operations, registering functions, and the interpreter itself.

lualib.h contains the libraries used by the runtime, such as math, io, os, and table.

lauxlib.h is a set of macros and functions that are made up of the functions in lua.h. Some are very useful for things like debugging. The source is readable, so if the manual fails, go into the lauxlib files and read through those.

There are other headers, but you won't necessarily need them depending on what you're doing.

A C Program with Lua

lua_context_test.c:

```c
#include "lua/lua.h"
#include "lua/lualib.h"
#include "lua/lauxlib.h"

int main()
{
  /* open the lua state */
  lua_State *Lstate = lua_open();

  /* open the standard libs and functions (needed for print) */
  luaL_openlibs(Lstate);

  /*
  * execute the global scope portion of the file (you can use this to load
  * files if your code is all inside functions)
  */
  luaL_dofile(Lstate, "lua_context_test.lua");

  /* close the state (have the interpreter clean up memory, etc) */
  lua_close(Lstate);
  return 0;
}
```

This is a very simple C program that executes a script.

This code assumes your libraries are in a local directory called “lua” (probably a good idea if you are going to be compiling on a machine that may or may not have Lua in its path). Here is the script portion:

lua_context_test.lua:

```
print("hello, world (from lua script)")
```

Compilation under GCC:

```
gcc -Wall -Wextra -ansi -pedantic lua_context_test.c lua/liblua.a -o
lua_context_test
```

(assuming liblua.a is the static lua library in the lua subdirectory)

Notice that Lua can use the same IO streams as your program (stdin, stdout, stderr). Of course, you need a console of some sort that is doing something with these streams (a WinMain, for example, won't process them).

Lua and C++

Since Lua was originally written in pure ANSI C, you will need an extern to include the headers from C++:

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

Binding C++ Functions To Lua

C/C++ functions that are registered with Lua must have a certain declaration:

```
int SomeCFunction(lua_State *)
```

The function is invoked by the interpreter, and the interpreter's context is passed to the function. The actual parameters are on the interpreter's stack, which can be accessed through the context. The return value of the C function is the number of returns pushed onto the Lua stack. The C function is responsible for cleaning the stack and making sure the parameters are of the correct type.

Example:

lua_context2_test.cpp:

```
#include <iostream>

extern "C"
{
  #include "lua/lua.h"
  #include "lua/lualib.h"
  #include "lua/lauxlib.h"
}

int add(lua_State *Lstate)
{
  // check the parameters passed from left to right, then pull them off in
  // that order from the stack
  if(lua_isnumber(Lstate, 1) &&
     lua_isnumber(Lstate, 2))
  {
    double a = lua_tonumber(Lstate, 1);
    double b = lua_tonumber(Lstate, 2);

    using std::cout;
    using std::endl;
```

```
    cout << "(C++): add called with (" << a << ", " << b << ")" << endl;

    // clean up the stack
    lua_pop(Lstate, 2);

    lua_pushnumber(Lstate, (lua_Number) (a + b));
  }
  else
  {
    // throw some kind of error, the types don't match up
    // (possibly a corrupt Lua stack)
  }

  // 1 result
  return 1;
}
```

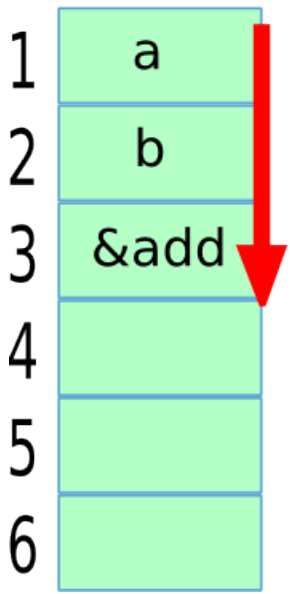
```
int main()
{
  // open the lua state
  lua_State *Lstate = lua_open();

  // open the standard libs and functions (needed for print)
  luaL_openlibs(Lstate);

  // register the C functions
  lua_pushcclosure(Lstate, add, 0);
  lua_setfield(Lstate, LUA_GLOBALSINDEX, "add");

  // execute the global scope portion of the file (you can use this to load
  // files if your code is all inside functions)
  luaL_dofile(Lstate, "lua_context2_test.lua");

  // close the state (have the interpreter clean up memory, etc)
  lua_close(Lstate);
  return 0;
}
```

lua_context2_test.lua:

```
print("(Lua): add(3, 5) = ", add(3, 5))
```

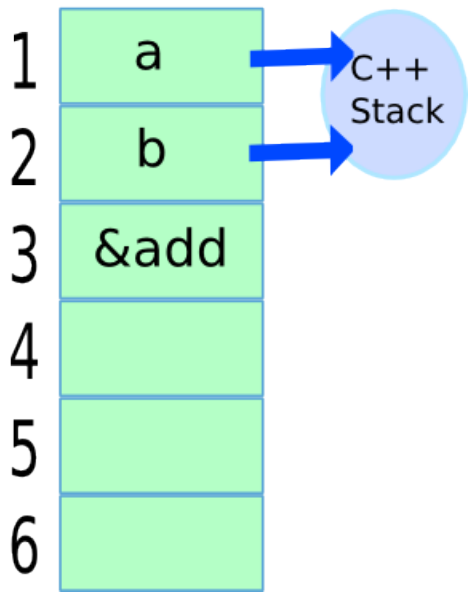
Lua Call:

```
ans = add(a, b)
```



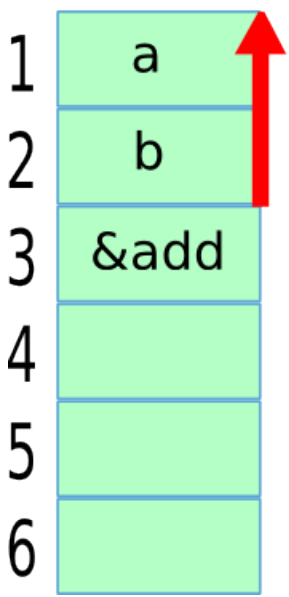
C++ Glue Function:

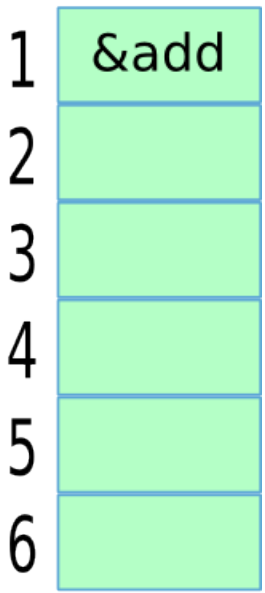
```
a = tostring(Lstate, 1)
b = tostring(Lstate, 2)
```



C++ Glue Function:

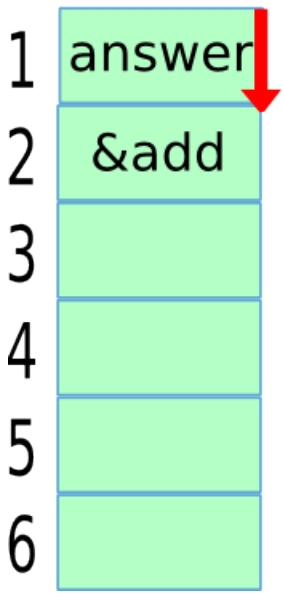
```
lua_pop(Lstate, 2)
```





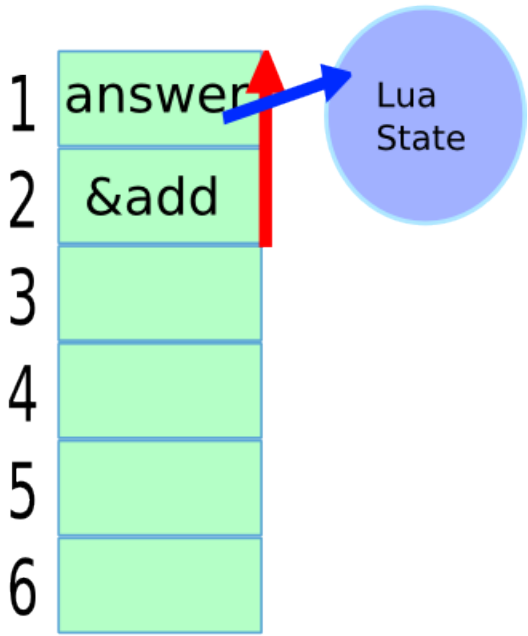
C++ Glue Function:

```
lua_pushnumber(Lstate, answer)
```



Lua State:

```
answer = add(a, b)
```



Compilation under G++:

```
g++ -Wall -Wextra -ansi -pedantic lua_context2_test.cpp lua/liblua.a -o
lua_context2_test
```

Output:

```
(C++): add called with (3, 5)
(Lua): add(3, 5) =      8
```

Notice the order is left-to-right for parameters. This is important to remember, as it gives you an idea of how the stack flows in Lua. Also, notice the function is in a static context. That is, it is not a member of a class/struct. Instance functions will not bind with Lua, unless a 3rd-party library is used. Using a 3rd-party library is not a good idea, as they are version-specific, and usually written completely in a mix of C++ templates and internal Lua bytecode. In short, don't use it unless you know how it works.

A good method is to create a namespace for Lua “glue” functions, and call existing functions from the glue functions.

Calling Lua Functions From C++

This is a good thing to know, since most of your scripts will (hopefully) be divided up into functions. The standard way to call a function inside a script:

```
lua_getglobal(Lstate, "functionName");
// push parameters
int errMsg = lua_pcall(Lstate, paramNum, resultNum, errNum);
// pop results
```

The `errNum` is usually set to 0, but can be used for tracing in some cases. `lua_getglobal` can also be used to get variables, etc that are registered in the global scope. `lua_pcall` simply treats whatever is below the parameters as a function, then calls it with those parameters, popping them from the stack. Then, the `resultNum` results are pushed back the the stack from the Lua interpreter, and the program must pop them off to prevent stack issues. This allows functions with variadic returns to be called without corrupting the stack.

Error Tracing and Debugging

As shown in the earlier C++ glue function `add`, you should always validate the parameter types (if you can afford the speed). Make functions that validate them for you, and throw exceptions if they're not correct, otherwise returning the value. Also, register some sort of output function in Lua that allows you to show debug messages somewhere in a Windows context (popup windows are kindof a last resort, but they work). You can always use ifndef's to change `WinMain` to a standard `main` function, but that will require recompilation of your project's source to turn debugging on and off.

When an error is thrown, you want to know what's on the stack. The easiest way to do this is to write a stacktrace function.

```
string TraceStack(lua_State *Lstate, int depth)
{
  using std::ostringstream;
  ostringstream buffer;

  buffer << "Lua Stack Trace\n";

  // traces upwards from the specified depth, with the greatest index first
  for(int i = -1; i >= -depth; i--)
  {
    if(lua_isstring(Lstate, i))
    {
      buffer << i << ":\t" << lua_tostring(Lstate, i) << '\n';
    }
    else if(lua_isnumber(Lstate, i))
    {
      buffer << i << ":\t" << double(lua_tonumber(Lstate, i)) << '\n';
    }
    // ...
  }

  return buffer.str();
}
```

Visual Studio's debugger cannot debug Lua bytecode, so this is a necessary step for scripts of any complexity.

Registering Table Members

You can use tables like namespaces, and register C++ functions inside of them. You can also create your own libraries (not covered), which is basically the same thing with a wrapper. Here is a table called `myMath` with two functions `sub` and `add`.

```
// create a table
lua_newtable(Lstate);
// keep a pointer to the table at the top of the stack
lua_pushvalue(Lstate, -1);
// set the table as a global
lua_setfield(Lstate, LUA_GLOBALSINDEX, "myMath");

// register C functions inside the table

// push the name of the function
lua_pushstring(Lstate, "add");
// push the C pointer to the function
lua_pushcclosure(Lstate, add, 0);
// bind it to the table that was at the top of the stack
lua_settable(Lstate, -3);

// repeat for the other functions
lua_pushstring(Lstate, "sub");
lua_pushcclosure(Lstate, sub, 0);
lua_settable(Lstate, -3);

// pop the table pointer off the stack
lua_pop(Lstate, 1);
```

Userdata

Normal Userdata, which is subject to Lua's garbage collection, is not a good idea for classes unless you find a way to hack in something that will call the constructors/destructors.

Light Userdata is a better option, as it allows you to handle the garbage collection, and invoke constructors/destructors. Both Userdata and Light Userdata objects can exist in the script as variables without functions or operators (kindof like a pointer). In the case of Light Userdata, the best way to use it is to pass it to and from C++ functions registered in Lua.

For example, say `myMath` is now a C++ class:

```
class myMath
{
  public:
    myMath() : data(0.0) {}
    void add(double a)
    {
      data += a;
    }

    void sub(double a)
    {
      data -= a;
    }

    double get()
    {
      return data;
    }

  private:
    double data;
};
```

Two extra functions are added to the table, `New` and `Delete`:

```
int New(lua_State *Lstate)
{
  // check the parameters passed from left to right, then pull them off in
  // that order from the stack
  if(true)
  {
    using std::cout;
    using std::endl;

    cout << "(C++): new myMath" << endl;

    myMath *m = new myMath();
    lua_pushlightuserdata(Lstate, (void *) m);
  }
  else
  {
    // throw some kind of error, the types don't match up
    // (possibly a corrupt Lua stack)
  }

  // 1 result
  return 1;
}
```

```
int Delete(lua_State *Lstate)
{
  // check the parameters passed from left to right, then pull them off in
  // that order from the stack
  if(lua_isuserdata(Lstate, 1))
  {
    void *p = lua_touserdata(Lstate, 1);
    myMath *m = static_cast<myMath *>(p);

    using std::cout;
    using std::endl;

    cout << "(C++): delete myMath" << endl;

    delete m;

    // clean up the stack
    lua_pop(Lstate, 1);
  }
  else
  {
    // throw some kind of error, the types don't match up
    // (possibly a corrupt Lua stack)
  }

  // 0 result
  return 0;
}
```

Also, both `add` and `sub` are modified to use the class' functions, and now accept a Userdata object as their first parameter. Here is the modified portion of `sub`:

```
if(lua_isuserdata(Lstate, 1) &&
   lua_isnumber(Lstate, 2))
{
  void *p = lua_touserdata(Lstate, 1);
  myMath *m = static_cast<myMath *>(p);
  double a = lua_tonumber(Lstate, 2);

  using std::cout;
  using std::endl;

  m->sub(a);

  cout << "(C++): myMath->sub called with (" << a << ")" << endl;

  // clean up the stack
  lua_pop(Lstate, 2);

  lua_pushnumber(Lstate, (lua_Number) (m->get()));
}
```

The first parameter must be statically cast to the class type, as it is returned as a void pointer and loses its type information. The driver code in the Lua file now looks like this:

lua_userdata_test.lua:

```
m = myMath.New()
print("m = ", m)
myMath.add(m, 3)
print("m.get = ", myMath.sub(m, 1))
myMath.Delete(m)
```

Output:

```
(C++): new myMath
m =     userdata: 0x6c5b08
(C++): myMath->add called with (3)
(C++): myMath->sub called with (1)
m.get =         2
(C++): delete myMath
```

Of course, the implementation could be cleaner, but this is a good starting point for creating your own. There are more advanced techniques, but this is the most practical to implement if you don't want to spend a lot of time hacking Lua. One disadvantage is that this allows for memory leaks at the script level. This can be problematic, but it's the price you pay for dynamic, non-hacked code.

Any Questions?