NOV 26TH, 2012

First Steps in LUA-C++ Integration

EDIT: Part 2

Long time without posting, busy at work, family, etc.

Today I decided to write a bit on Lua.

Lua is a scripting language with an emphasis on being easily embeddable. Lua is used as the scripting language in Photoshop CS, and World of Warcraft, for example. So if you are looking into adding scriptability to your C or C++ applications Lua is quite suited for the task.

In order to learn Lua (both the language itself and how to embed it into your app) I recommend you Programming in Lua by one of the Lua authors.

You can find tons of tutorials on how to get started with Lua. So I will focus here on how to integrate with C++. Most of the sources I've found about Lua - C++ integrations take the approach of frameworks to automatically wrap your objects/classes to be usable from within Lua. I find this approach confusing, I think it's better to learn how to do it manually, and that's what I will do here.

Step 0. Compile Lua itself

Download Lua 5.2.1 and compile it. Usually it enough with `make macosx` or `make linux`. That will generate `liblua.a`, the library that we will link to.

Step 1. Create a simple host app

We need a simple host app. Our host app will simply setup Lua and run a script from it. This script will have access to a `std::queue` from the host app. This will illustrate how you can share objects with the Lua part. Later we will take a more complex example.

Let's start with the basic skeleton of a lua environment with some communication with its host:

The Makefile

(Makefile) download

```
1 LUAHOME=$(HOME)/tmp/lua-5.2.1/src
2 all: sampleluahost
3 sampleluahost: sampleluahost.cpp
4   g++ -g sampleluahost.cpp -llua -L$(LUAHOME) -I$(LUAHOME) -o sampleluahost
```

It uses `LUAHOME` that should point to the directory containing both `liblua.a` and the `lua*.h` files.

The samplehost application

(sampleluahost.cpp) download

```cpp
#include <iostream>
#include <lua.hpp>


extern "C" {
  static int l_cppfunction(lua_State *L) {
    double arg = luaL_checknumber(L,1);
    lua_pushnumber(L, arg * 0.5);
    return 1;
  }
}

using namespace std;
int main()
{
  cout << "** Test Lua embedding" << endl;
  cout << "** Init Lua" << endl;
  lua_State *L;
  L = luaL_newstate();
  cout << "** Load the (optional) standard libraries, to have the print function" << endl;
  luaL_openlibs(L);
  cout << "** Load chunk. without executing it" << endl;
  if (luaL_loadfile(L, "luascript.lua")) {
    cerr << "Something went wrong loading the chunk (syntax error?)" << endl;
    cerr << lua_tostring(L, -1) << endl;
    lua_pop(L,1);
  }

  cout << "** Make a insert a global var into Lua from C++" << endl;
  lua_pushnumber(L, 1.1);
  lua_setglobal(L, "cppvar");

  cout << "** Execute the Lua chunk" << endl;
  if (lua_pcall(L,0, LUA_MULTRET, 0)) {
    cerr << "Something went wrong during execution" << endl;
    cerr << lua_tostring(L, -1) << endl;
    lua_pop(L,1);
  }

  cout << "** Read a global var from Lua into C++" << endl;
  lua_getglobal(L, "luavar");
  double luavar = lua_tonumber(L,-1);
  lua_pop(L,1);
  cout << "C++ can read the value set from Lua luavar = " << luavar << endl;

  cout << "** Execute a Lua function from C++" << endl;
  lua_getglobal(L, "myluafunction");
  lua_pushnumber(L, 5);
  lua_pcall(L, 1, 1, 0);
  cout << "The return value of the function was " << lua_tostring(L, -1) << endl;
  lua_pop(L,1);

  cout << "** Execute a C++ function from Lua" << endl;
  cout << "**** First register the function in Lua" << endl;
  lua_pushcfunction(L,l_cppfunction);
  lua_setglobal(L, "cppfunction");

  cout << "**** Call a Lua function that uses the C++ function" << endl;
  lua_getglobal(L, "myfunction");
  lua_pushnumber(L, 5);
  lua_pcall(L, 1, 1, 0);
  cout << "The return value of the function was " << lua_tonumber(L, -1) << endl;
  lua_pop(L,1);

  cout << "** Release the Lua enviroment" << endl;
  lua_close(L);
}
```

(luascript.lua) download

```lua
print("Hello from Lua")
print("Lua code is capable of reading the value set from C++", cppvar)
luavar = cppvar * 3

```

```
function myluafunction(times)
  return string.rep("(-)", times)
end

function myfunction(arg)
  return cppfunction(arg)
end
```

The code explained step by step

Initialization

```
lua_State *L;
L = luaL_newstate();
luaL_openlibs(L);
if (luaL_loadfile(L, "luascript.lua")) {
  cerr << "Something went wrong loading the chunk (syntax error?)" << endl;
  cerr << lua_tostring(L, -1) << endl;
  lua_pop(L,1);
}
```

That creates a `lua_State` loads the standard libs in it and also loads the code in `luascript.lua`.

Adding variables from C++ into Lua

```
lua_pushnumber(L, 1.1);
lua_setglobal(L, "cppvar");
if (lua_pcall(L,0, LUA_MULTRET, 0)) {
  cerr << "Something went wrong during execution" << endl;
  cerr << lua_tostring(L, -1) << endl;
  lua_pop(L,1);
}
```

Then it sets a global variable in Lua from C++ code using `lua_setglobal`. If you don't know what are the `lua_pushxxxx` and the Lua stack, etc. I recoment that you take a look at the Lua Reference Manual and Programming in Lua. More or less Lua and the C++ communicate through the stack that lives in `lua_State` and there is bunch of function to manipulate that stack. So in order to set a global in Lua from C++ you must push the value into the stack and call `lua_setglobal` that will pop the value in the stack and assign it to the identifier provided inside the Lua environment.

After setting the global `cppvar` it executes the loaded chunk of code (that is in the stack) with `lua_pcall`. The Lua code is able to read and print the value of `cppvar`. The lua code will also set a new global `luavar` that we will access from C++.

Reading a Lua variable from C++

```
lua_getglobal(L, "luavar");
double luavar = lua_tonumber(L,-1);
lua_pop(L,1);
cout << "C++ can read the value set from Lua luavar = " << luavar << endl;
```

To get luavar from C++, we must first use `lua_getglobal` that will put the value associated with the identifier into the top of the stack and the `lua_tonumber` will transform whatever it's at the top of the stack into a double (well a `luaNumber`) and then we can use that double in our C++ code to print it.

Calling a Lua function from C++

```
cout << "** Execute a Lua function from C++" << endl;
lua_getglobal(L, "myluafunction");
lua_pushnumber(L, 5);
lua_pcall(L, 1, 1, 0);
cout << "The return value of the function was " << lua_tostring(L, -1) << endl;
lua_pop(L,1);
```

The example won't be complete without function calling so that's the next step. Calling a Lua function from C++ it's quite easy. Function in Lua are first class values, so that means that it's just a like reading a any other value. `lua_getglobal` will get the value and put it on the stack and then we push the function arguments into the stack and use `lua_pcall` to call the function (that is the stack). The returned value from the function will be pushed in the stack and that's were the C++ code will get it, `lua_tostring` and then it will remove from the stack with `lua_pop`.

Calling a C++ function from Lua

```
lua_pushcfunction(L,l_cppfunction);
lua_setglobal(L, "cppfunction");

lua_getglobal(L, "myfunction");
lua_pushnumber(L, 5);
lua_pcall(L, 1, 1, 0);
cout << "The return value of the function was " << lua_tonumber(L, -1) << endl;
lua_pop(L,1);
```

The other way around it's more complex. You can't just call *any* function from Lua. It has to has a special signature `lua_CFunction`, that is, `typedef int (*lua_CFunction) (lua_State *L)` a function that returns an int and takes a `lua_State`. This special funciton will communicate with Lua via the lua stack that resides in the `lua_State` parameter. The return value of the function tell lua how many value the function has pushed into the stack as result values for the function call.

So to make the function accesible from Lua, you create push the function into the stack with `lua_pushcfunction` and bind it to an identifier in lua with `lua_setglobal`. Then lua code will be able to invoke this function like any other function. In the example I call the `myfunction` (which is lua code) and `myfunction` in turn invokes `cppfunction` which is "bound" to C++ `l_cppfunction`. Ah, I almost forgot. `l_cppfunction` is declared as `extern "C"` telling the compiler to provide C linkage for this function so it can be called from a C library like Lua is.

Free Lua resources

```
lua_close(L);
```

`lua_close` will free all resources held by the `lua_State` L.

Wrap up

I will leave the part on how to wrap C++ class objects in Lua for a later post because I don't want to make this post too long. Hopefully I'll post it tomorrow.

Posted by Ruben Laguna • Nov 26th, 2012

« Sublime Text 2 RVM and RSpec

GDB posix_spawn failed on Mac OS X Mountain Lion »

Comments

Recent Posts

Accessing C++ objects from Lua

Sublime Text 2 integration with RVM and Rspec: Take number 2

GDB posix_spawn failed on Mac OS X Mountain Lion

First steps in LUA-C++ integration

Sublime Text 2 RVM and RSpec

GitHub Repos

dotfiles

jletty
LDAP server implemented in java

en4j
Java Desktop Client to Evernote

nevernote
Nevernote is an incomplete clone of Evernote designed to run on Linux. This repository is a development fork from original. You can see original distribution from http://nevernote.sourceforge.net/

rosert-petal-parser
An ANTLR based parser for RoseRT petal files

@ecerulm on GitHub

Latest Tweets

Status updating...

On Delicious

ICEpdf - Open Source Java PDF, Java PDF Viewer, Java PDF Rendering, Java PDF Extraction

Länna Sport, Stockholm sportbutik sportaffär golfbutik cykelbutik skor uteliv konfektion alpint

Windsurf sweden

My Delicious Bookmarks »

My Pinboard

Aho/Ullman Foundations of Computer Science

algorithms, theory, ullman, book, automata

GNU Octave

octave, manual, documentation

bubble bobble mcu decapping

bubble, bobble, rom, decapping

My Pinboard Bookmarks »