LUA in a Visual C++ 2003 based game engine

By Jeff Jacob

Last updated
Thursday, June 07, 2007

Table of Contents

Objective 3
What you need to know 3
What is LUA? 3
The Microsoft developer mythos 3
What is included 4
Some conventions 4
My setup 4
Compiling 5
The Basics 8
ex1.lua 10
ex1.cpp 10
Example 2 14
Ex2.lua 15
Ex2.cpp 15
Example 3 16
Ex3.lua 17
Ex3.cpp 17
References 19

Objective
To give a clean and concise tutorial on how to create a game engine to work with LUA version 5.0.2 using Microsoft Visual Studio 2003, C++.

The primary example used is a simple game engine for playing board games that involve a square grid, and have pieces moved in turn by two or more players. Examples of these types of games include checkers, snakes and ladders, and for this example chess. The theoretical game engine will implement functions like drawing a board and taking input, but the game mechanics will be in LUA. The advantage of this is that if the player wants to change something like make pawns able to move backwards they can simply modify a LUA script. There is no need to recompile.

As writing an entire game engine is beyond the scope of this paper I will be writing several smaller examples to guide you through the theory behind the LUA API while I try to teach the basic use of it.

What you need to know
Game engines and linking between two languages is a relatively advanced topic so it is recommend that you have moderate experience in programming both C and C++, as well a basic understanding of LUA would help. If you've read the first two parts of Programming in LUA (http://www.lua.org/pil/) you should be fine, if not this may go over your head but you are welcome to read it and keep it handy as a reference. Experience in windows is required, but knowledge of windows programming is not required.

What is LUA?

LUA is an embedded scripting language that is both an extension language and an extendable language, in that it can be expanded by your application by giving it new functions written in C to use and it can extend your application doing the job of internal code.

LUA to the game programmer is a way to allow others to help with the development. Games these days are far to complex to be built by one person. Very often a large team is working on a game together, a few programmers, a few level designers, an art and sound team. LUA will be primarily used by the level designers; the LUA API is used by the programmers.

For this paper we will think of ourselves largely as the game programmer, and as such there will be very little LUA code discussed.

The Microsoft developer mythos

As a Linux or command line programmer most people will be familiar with a few concepts of software development such as a "program" that is an end product, and a MAKEFILE, the file that describes how the source files go together to make a program.

The Microsoft mythos is a little different, at first glance, and possibly second and third glance, it seems like it was designed by marketing, but it does have its merits. Visual studio was built to make large applications, very often for deployment in house such as new software for a bank or billing software for a phone company.

To understand the Microsoft mythos we must think about why real programs are written and that is to solve a problem. Our problem for this paper is "Write a chess style board game that can be extended by LUA." The Microsoft buzz word filled answer is to create a "solution".

The Microsoft solution in visual studio is an interlinked collection of projects. For any simple command line program it is likely you will only have one project in a solution but in our case here we need at least two projects, one is the building of the LUA library. In Linux this would be a separate project with possibly separate make files but in VS we can group our projects together and Microsoft will keep track of all our dependencies for us.

The final thing you need to know about a solution is that it has a startup project, the actual exe file that is executed when you try to compile and run the solution.

What is included

To help out anyone that is picking this up I have created few resources that can go along with this tutorial, they include

- firstbuild.zip
  - A basic skeleton for LUA as a static library. And a copy of the 5.0.2 source code.
- Fulltutorial.zip
  - Full copy of the tutorial at time of first publishing

Some conventions

In order to make this as easy to read as possible some conventions are used along the way.

- Any series of clicks will be bold and linked with a → , for example **file→open.**
- Code examples will be in grey boxes
- Lua is in black and white
- C++ is in color

My setup

Because every system is a little different I can not guaranty that everything I say here will work on ever computer but the system I will be developing and testing this on is an AMD 1.1 ghz running windows XP PRO and Visual C version 7.1.3088 with .net 1.1.4322. These are both part of visual studio 2003. The version of LUA I am using is 5.0.2. At time of writing 5.1 has just been released but 5.0.2 can still be found here (http://www.lua.org/ftp/lua-5.0.2.tar.gz) or inside my zip files.

The directory structure used is simple. A directory c:\lua to store everything related to LUA. Extract the 5.0.2 source to C:\lua\lua-5.0.2 and create a directory C:\lua\Jeffscode to store all the sample code.

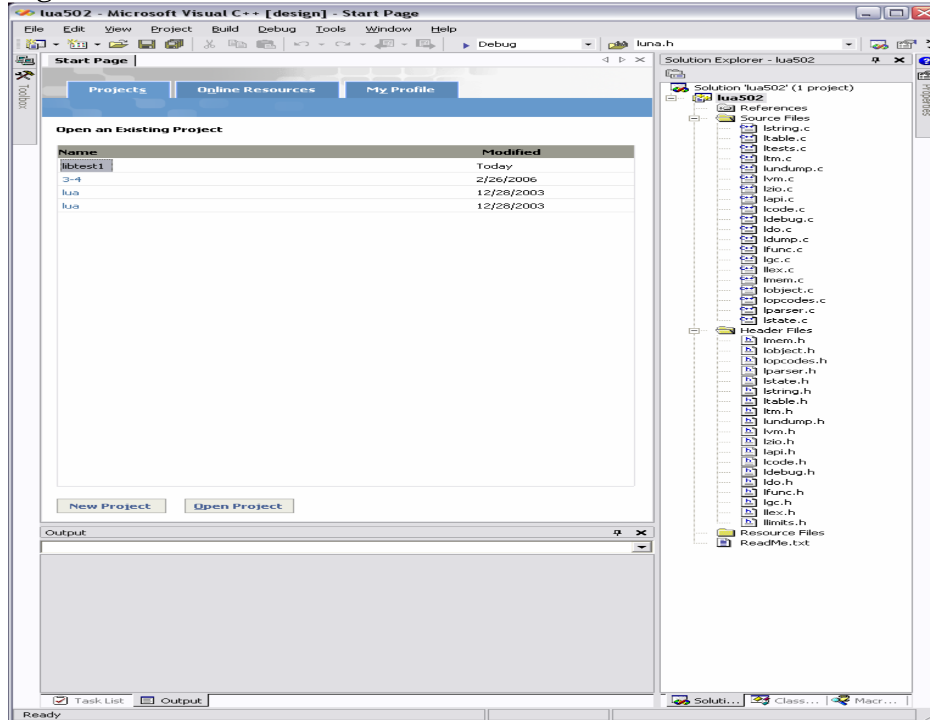
Compiling

First you will need to create a VS project to compile the LUA lib to do this you will have to let the compiler know about the include directory otherwise lines like #include "lua.h" will fail.

We will be building LUA as a static library so we will be going off the steps from (http://msdn2.microsoft.com/en-us/library/ms235627.aspx), unfortunately these are for VS 2005 and a few things have changed.

1. Start Visual c++ 2003
2. The New project start page should come up. Click **New Project** or click **file→new→project**
3. Show the advanced options and choose to create a directory for this solution, I called it **luatutorial**
4. We will first be building the static library for compiling LUA so on the right hand side select **Win32 Console Project** and then enter a name, such as lua502 and place it in a directory like C:\lua\jeffscode.
5. Click **ok**.
6. Win 32 application wizard will come up although it is not much of a wizard, on the left hand side click **Application Settings**
7. Select **Static Library** and uncheck **Precompiled header**
8. Click **Finish**
9. Welcome to an empty project. It is now time to add your source files. Open the source directory for LUA, for me it was at "C:\lua\lua-5.0.2\src"
10. The easiest way to add files to a VS project is drag and drop so select all the .c files and drag and drop them into source files
11. Drag all the .h files into header files.
12. Also add all the .c files from src/lib, these are for the various libraries suck as math and string. Your screen should look a little bit like **figure 1**.
13. Next you will have to make sure that the include paths are correct so right click on the **project name** and choose **properties**
14. Click on **Configuration drop down** and choose **all configurations**
15. click **c/c++→general→Additional Include Directories**
16. Add "C:\lua\lua-5.0.2\include"
17. Change **Detect 64-bit Portability issues** to **no**, otherwise it will give you lots of warnings
18. Click **ok**

Figure 1:



19. Right click on the project name again and choose build, This is probably one of the most difficult issues in getting it all to work. Even thought you tell VS to use C++ in **C/C++→Advanced→Compile** as you may find that it ignores this. You can check your buildlog.html if you see something like "/Od /I "C:\lua\lua-5.0.2\include" /D "WIN32" /D "_DEBUG" /D "_LIB" /D "_MBCS" /Gm /EHsc /RTC1 /MLd /Fo"Debug/" /Fd"Debug/vc70.pdb" /W3 /c /ZI **/TC**" then it compile in c not C++ if it was c++ it would have ended the line in **/TP**. To fix this go to **C/C++→Advanced→Compile** and choose **default**, then go to **C/C++→Command Line** and add **/TP**
20. If everything went well you should now have the following message in your build output

```
Creating library...

Build log was saved at "file://c:\lua\Jeffscode\luatutorial\lua502\Debug\BuildLog.htm"
    lua502 - 0 error(s), 0 warning(s)
```

21. You should have a file "lua502.lib" at "C:\lua\Jeffscode\luatutorial\lua502\Debug"
22. Now to add a test bench, **file→new→project**
23. Again we are going to select win32 console project but we are going to call it testbench
24. Make sure to check Application Settings to see that console application is selected and everything else if blank, you may not be able to uncheck precompiled header, don't worry about this.
25. Click **Add To Solution**
26. Click **finish**

27. You will find that this creates a testbench.cpp with a _tmain. Delete it everything in this file and you can completely remove the other files.
28. Now paste in some code to test it, I've have taken this code from Roberto Lerusalimschy's book and modified it to work in VS. It's a good book and worth buying or at least reading online.

testbench.cpp

```
#include <stdio.h>
#include <lua.h>
#include <lauxlib.h>
#include <lualib.h>
#include <string.h>// this line is added by me because strlen needs it


int main (void) {
      char buff[256];
      int error;
      lua_State *L = lua_open();    /* opens Lua */
      luaopen_base(L);              /* opens the basic library */
      luaopen_table(L);             /* opens the table library */
      luaopen_io(L);                /* opens the I/O library */
      luaopen_string(L);            /* opens the string lib. */
      luaopen_math(L);              /* opens the math lib. */

      while (fgets(buff, sizeof(buff), stdin) != NULL) {
            error = luaL_loadbuffer(L, buff, strlen(buff), "line") ||
                  lua_pcall(L, 0, 0, 0);
            if (error) {
                  fprintf(stderr, "%s", lua_tostring(L, -1));
                  lua_pop(L, 1);  /* pop error message from the stack
*/
            }
      }

      lua_close(L);
      return 0;
}
```

29. Right click on the project name **testbench** and choose **set as startup project**.
30. Right click on the project name **testbench** and choose **properties**
31. Click on **Configuration drop down** and choose **all configurations**
32. click **c/c++→general→Additional Include Directories**
33. Add "C:\lua\lua-5.0.2\include"
34. Go to **C/C++→Precompiled headers** and turn them **off**
35. Click **OK**
36. Right click on the project name **testbench** and choose **add reference**
37. Select "lua502" from the projects list and click **Select**
38. Click **ok**
39. At the top menu click **Build→Build Solution**
40. Click **Debug→Start without Debugging** or press Ctrl+F5

Welcome to a simple LUA interpreter.

You will find a copy of everything we've done up till now in "firstbuild.zip" this will be available on my website http://www.phansoft.ca/lua/ .

Also included are two test files demo1.lua and demo1_one_line.lua, because our program here doesn't support multiple line entry we can only enter in the one line version directly, if you want to run the other version try something like "dofile("c:/lua/jeffscode/demo1.lua");" depending on where you saved it.

Demo1_one_line.lua:

```
i = 1; -- a simple assignment
echo = print; -- functions are a first class variable type this will assign the print function
to echo, this will make all those perl programmers out ther happy.

--the next two lines are on one line
function tester(a,b,txt) b = b or 10 a = a or 1  txt = txt or "reasons to like LUA" for i=a,b
do echo (i,txt) end end
--now that we have the function lets make a few calls
tester(4,12);
tester();
tester(5);
tester(1,2,"happy days");
```

The Basics

Now that we have a copy of the LUA as a library we can work with we need to think about two things

A) What do we need to do as a game programmer and what can LUA do for us?
B) How do we talk to LUA and how does it talk back?

To answer point b in short we talk to LUA through a series of C commands that push and pop data off the LUA stack, If you've never worked with passing data to a function via the stack I recommend that you find any good book on stack programming. The basic concept is that functions take all their arguments off the stack and push back their return values. I will cover this more as we go through each example, but for LUA this is very advantageous because it can push more than one return value and it can deal more easily with the differences in types.

The main question to be answered is what can LUA do for a game programmer?

There are a few things that commonly change in a game after it has been compiled

1. Level scripting. Levels in games may have specific scripts or actions that require automation, these can be anything from "when player steps on pad open the door" to "when play lands on a square in snakes and ladders containing a ladder he is moved to the top of the ladder"

2. Artificial Intelligence, AI. The ability to tweak the reactions of characters in the game is very important. Slight changes can allow a game world to seem more real. At its crudest level if we want to decide that a pawn can now move forwards or backwards it's just a few lines of external script to change and no need to re compile. This is what lets us turn checkers into chess, or expand a chess game to allow En passant (http://en.wikipedia.org/wiki/En_passant**).**
3. Configuration and setup. This can be anything from defining a high score table to choosing the starting position of all the chess pieces. Although this can be done with XML or a simple parsed txt file the LUA parser can parse a lua table like {board.rows = 8, board.cols = 8} just as easily.

I will demonstrate the second and third point bellow, but first there are some basic concepts we will need to know

- Load a LUA library. The end-user will write the library and save it in a predefined file name such as "config.lua" "ai.lua" "level1.lua" in a real level format you would most likely encode the script into the level file and pass it as a string but this is a trivial difference.
- Call a LUA function, passing it information and reading back the result. Here we have a piece that was defined as "pawn" by the LUA config script. If you want to check where it can move to you might call a LUA function:
  move_piece(type, source_row, source_col, dest_row, dest_col);
  that would return true or false depending on if it was a valid move.
  We will build this function later.
- Lastly we will need some C functions that LUA might call like:
  int square_controlled(double row, double col);
  This function would return the owner of the current square, so 0 for empty 1 for player one and 2 for player two. As stated earlier we won't be building these functions because they are overly complex but this should give you something to think about.

Example 1:

Loads a LUA file "ex1.lua" and reads from it the number of rows and cols then prints the values and closes LUA down.

This will illustrate basic use of the LUA API and its use in config files. It will try to show the power of LUA in this respect by having a config file that actually does calculations.

First we need to create a new project. We will be using the same solution so if you've closed it down open "luatutorial" again.

1. Click **file→new→project**
2. Select **add to solution**
3. Name it **ex1**
4. Make sure **console application** is selected and click **ok.**
5. Click on **Application settings** and check empty project.
6. Click **ok**

7. Right click on **source files** and choose **add→add new item**
8. Choose **C++ file** and call it **ex1**
9. Right click on the project name and go to **C/C++→general→Additional Include Directories**
10. Add "C:\lua\lua-5.0.2\include"
11. Right click on **references** and choose **add reference**
12. Double click **lua502** and click ok, this will add a reference to our library.
13. Click **ok**

That's it we are done, these will be the standard steps for creating a new c++ lua project

If you want to use Visual Studio to edit the LUA file as well you can add it as a txt file.
1. Right click the project name,
2. Select **add→add new item**
3. Select **utility**
4. Select **Text file.**
5. Name it **ex1.lua**
6. Click **Ok**

SciTe is a recommend LUA editor (http://www.scintilla.org/SciTE.html) it does syntax highlighting for LUA. Now on with the code.

ex1.lua

```
function plustwo (x)
	local a;
	a = 2
	return x+a;
end
--here we see a simple function but it could be very complex,
--even check for user input
rows = 6; -- comments in a config file can be handy
cols = plustwo(rows);--now we can have functions in our config files.
```

ex1.cpp

```
#include <lua.h>
/*lua.h is the leader file that contains all of the basic API calls,
All of the function we will see that begin with lua_ are defined in
here*/
#include <lauxlib.h>
/*The Auxiliary library for LUA dfine all the luaL_ functions.
It will help us simplify the interface*/

#include <lualib.h>
/*The LUA library, it contains all of the functions to load the extra
bits of LUA, like math, string, and IO. We could do without these but
lua would not be very useful with out somewhere to start*/
```

```
#include <iostream> /*unlike most LUA examples I've found we will be
using c++ */
using namespace std;

/*This programs reads in a configuration file*/




int main()
{
	lua_State *L;
	/* lua is multi thread compatible and allows multiple threads to
us it at the same time. The way it implements this is by having
something called a lua_State. The lua_State holds all the information
about what LUA has been doing, think of it as the ram on your lua
running machine, A pointer to this state is passed as the first
argument to any function. This lets us create multiple states, for
example we could have each enemy in a game have it's own thread and own
LUA so we could define a LUA function call move_enemy globally in LUA
and have it different for each enemy. Probably not the most efficient
method though as it is uncommon to have a thread for each enemy and
each lua state (even in a single thread) would have some over head.*/

	L = lua_open();
	/*So now we've initialized the LUA state the LUA machine is
booted up and ready to go but it does not yet have any libraries to
work with. */

	luaopen_base(L);		/* opens the basic library */
	luaopen_table(L);		/* opens the table library */
	luaopen_io(L);			/* opens the I/O library */
	luaopen_string(L);		/* opens the string library. */
	luaopen_math(L);		/* opens the math library. */
	/*Although in this example we will not use all of the libraries
it is good practice to load them all anytime the end user might want
access to any of the functions in them. The best place I've found for a
quick list of what is in each library is the code itself. For example
in lbaselib.c you will find:
	static const luaL_reg base_funcs[] = {
	{"error", luaB_error},
	{"getmetatable", luaB_getmetatable},
	{"setmetatable", luaB_setmetatable},
	{"getfenv", luaB_getfenv},
	{"setfenv", luaB_setfenv},
	{"next", luaB_next},
	{"ipairs", luaB_ipairs},
	{"pairs", luaB_pairs},
	{"print", luaB_print},
	{"tonumber", luaB_tonumber},
	{"tostring", luaB_tostring},
	{"type", luaB_type},
	{"assert", luaB_assert},
	{"unpack", luaB_unpack},
	{"rawequal", luaB_rawequal},
	{"rawget", luaB_rawget},
```

```
	{"rawset", luaB_rawset},
	{"pcall", luaB_pcall},
	{"xpcall", luaB_xpcall},
	{"collectgarbage", luaB_collectgarbage},
	{"gcinfo", luaB_gcinfo},
	{"loadfile", luaB_loadfile},
	{"dofile", luaB_dofile},
	{"loadstring", luaB_loadstring},
	{"require", luaB_require},
	{NULL, NULL}
	};

	This is a list of the base functions that are exported to LUA,
the source code is there as well*/
	int temp_int; // used to store an int for checking return values.
	temp_int = luaL_loadfile(L,"ex1.lua");
	/*unfortunately it appears that aside from its use in
"Programming in LUA" no luaL_ function has official documentation, the
expression "lual_" does not even appear in the manual. So, I must turn
to the best and most absolute resource there is, the source code
"lauxlib.h".
LUALIB_API int luaL_loadfile (lua_State *L, const char *filename);
	This begs the question what is LUALIB_API, I had to trace it back
to lua.h but it turns out to be just an extern. Looking deeper into the
code in "lauxlib.c" we see it's just a wrapper for lua_load()
unfortunately due to the number of calls that the return value does
it's hard to tell what the possible values are. What we do know is that
is can fail if there is an error in the LUA script and return true.
Lua_loadfile leaves the code at the top of the stack as an unnamed
function.*/
	if (temp_int)
	{
		cout << "cannot run configuration file."
			<< lua_tostring(L, -1) <<endl;
		/* lua_tostring is the most important command here so I
will break it down. "Lua_tostring(L"all functions take the state, in
this case L. The first argument,"-1" is where the magic happens, on the
stack.
Stack indexing works in two ways first we have absolute indexing where
1 = first item on stack (bottom) and so on up to the top of the stack
-1 = last item on stack (top), -2 is the second to last and so on.
Almost all API calls use the stack so you will become very accustomed
to this.
The reasoning behind the stack is that LUA and C do not share data
types, also LUA is dynamically typed and has garbage collection.
Therefore a pointer directly into LUA would be of no use. The above
call gets the string off the top of the stack (if it is not a string
then it returns NULL) and converts it to a C NULL terminated string.
One warning LUA strings are binary and can contain \0 inside them. You
can use
size_t lua_strlen (lua_State *L, int index);
and size_t strlen(char* s); to check for this, if the values don't
match then there is likely an extra \0. Never store the pointer to a
LUA string outside the function that called it. Use string copy, other
wise the garbage collector might cause you errors.*/
		lua_close(L);
		return 1;
```

```
	}
	temp_int = lua_pcall(L,0,0,0);
	/*
	LUA_API int lua_pcall (lua_State *L, int nargs, int nresults, int
errfunc)
This function does a protected call thus giving us some error handing,
nargs is the number of arguments that you pushed onto the stack. All
arguments and the function values are popped from the stack, and the
function results are pushed back on. The number of results is adjusted
to nresults. If there is an error it pushes the error message onto the
stack and returns an error number. We will just ignore the errfunc for
now you can check it out in the manual section 3.15. Some of my
comments are copied right out of the manual because it does explain
some thing pretty well, sadly not everything is explained or even
documented. The lua_pcall function returns 0 in case of success or one
of the following error codes (defined in lua.h):

    * LUA_ERRRUN --- a runtime error.
    * LUA_ERRMEM --- memory allocation error.
*/
	if (temp_int)
	{
		cout << "error with pcall"
			<< lua_tostring(L, -1)<< temp_int <<endl;
		lua_close(L);
		return 1;
	}
	lua_getglobal(L,"rows");
/*lua_getglobal(L,s) is not in the manual so we turn again to the
source and find out it isn't a function at all but a macro that gets
the global variable from the global variable table and pushes it on the
stack. In this case we push rows on the stack.    */
	lua_getglobal(L,"cols");
	if (!lua_isnumber(L, -2))
	{
/* there is a lua_is* function for every type in LUA, they are the best
way to check the type of an item on the stack, in this case we are
looking at -2 so the second to last item we pushed onto the stack. This
item is "rows".*/
		cout << "rows is not a number"<<endl;
		lua_close(L);
		return 1;
	}
	if (!lua_isnumber(L,-1))
	{
		cout << "cols is not a number" << endl;
		lua_close(L);
		return 1;
	}
	/*Now we have read the file and know that the variable rows and
cols are on the stack all we have left to do is read them into C++ and
print them out*/
	cout << "Rows:"
		<< (int) lua_tonumber(L,-2)
		<< "Cols:"
		<< lua_tonumber(L,-1) <<endl;
```

```
	/* lua.h states "typedef double lua_Number;" and lua_tonumber
returns a lua_Number, or basically this function takes the top value
off the stack and tries to convert it to a double and return it. On
failure it returns 0*/
	lua_pop(L,2);
	/*remove our top two values from the LUA stack, just a little
cleanup work that really isn't required as we are about to exit*/
	lua_close(L);
	/*shut down the LUA virtual machine*/
	return 0;
	/*and that's it that's all*/
}
```

One last thing before you compile, go back into the project properties and add the include directory.

1. Right click on the project name
2. Select **Properties**
3. Go to **C/C++→General**
4. Add "C:\lua\lua-5.0.2\include" to the include directors so it can find lua.h
5. Click **ok**

To build it:

1. Right click on the project name
2. Select **Build**

You should get "`Build: 2 succeeded, 0 failed, 0 skipped`" now it's time to run the program.

1. Click **debug→start without debugging** or just hit ctrl+f5

The zipped up the code in "jeffscode" is included on the web site. Feel free to try it out.

Example 2

Loads a LUA file "ex2.lua" that contains a function "ex2f". This function takes three arguments a string and two numbers. It will take the string print it out and return the sum of the two numbers and the string "Thank you"

This is probably the most useful feature of LUA for a game programmer as it lets him/her call functions that are defined by the user. For any game to have long term popularity it needs to be expandable. Look at half life almost no one still plays the single player game but counter strike is still one of the most popular games online at time of this publication. Giving the player the ability to improve on what you create is very useful. Also this same power can be extended to level designers to let them create new effects and objects in game.

And now on to the code.

Ex2.lua

```
function ex2f (str_a,num_b,num_c)
    print(str_a);
    return num_b + num_c, "Thank you";
end
```

Ex2.cpp

```
#include <lua.h>
#include <lauxlib.h>
#include <lualib.h>
#include <iostream>
using namespace std;

int main()
{
      lua_State *L;
      L = lua_open();

      luaopen_base(L);              /* opens the basic library */
      luaopen_table(L);             /* opens the table library */
      luaopen_io(L);                /* opens the I/O library */
      luaopen_string(L);            /* opens the string library. */
      luaopen_math(L);              /* opens the math library. */

      int temp_int;
      temp_int = luaL_loadfile(L,"ex2.lua");
      if (temp_int)
      {
            cout << "cannot load file ex2.lua."
                  << lua_tostring(L, -1) <<endl;
            lua_close(L);
            return 1;
      }
      temp_int = lua_pcall(L,0,0,0);
/* again I will reiterate the lua_pcall function because it is so
important,LUA_API int lua_pcall (lua_State *L, int nargs, int nresults,
int errfunc).nargs is the number of arguments that you pushed onto the
stack.  All arguments and the function value are popped from the stack,
and the function results are pushed. The number of results is adjusted
to nresults. If there is an error it pushes the error message onto the
stack and returns an error number. Make sure you've memorized at least
this one call if nothing else.*/
      if (temp_int)
      {
            cout << "error with pcall"
                  << lua_tostring(L, -1)<< temp_int <<endl;
            lua_close(L);
            return 1;
      }

      lua_getglobal(L,"ex2f");
/*functions in lua are a first class data value, so much like rows = 5
ex2f = a function to perform some operations, ex2f is now on the top of
the stack*/
      lua_pushstring(L,"Lua will receive this");/* push 1st argument */
```

```
      lua_pushnumber(L, 3);   /* push 2nd argument */
    lua_pushnumber(L, 4);   /* push 3rd argument */

      /* do the call (3 arguments, 2 result) */
      if (lua_pcall(L, 3, 2, 0) != 0)
      {
            cout << "error with pcall of ex2f: "
                  << lua_tostring(L, -1)<< temp_int <<endl;
            lua_close(L);
            return 1;
      }
      /*check return types*/
      if (lua_isstring(L,-1) && lua_isnumber(L, -2))
      {
            cout << "return value 1 (string): (" << lua_tostring(L,-1)
<< ")" << endl;
            cout << "return value 2 (double): (" << lua_tonumber(L,-2)
<< ")" << endl;
            lua_pop(L,2);
/*here we pull the data off the stack, lua_to* is a peek function we
need to call pop to clean up the stack. pcall cleans up the stack of
the function and its arguments */
      }
      else
      {
            cout << "invalid values returned";
      }


      lua_close(L);
}
```

Hopefully from this example you can see how simple it is to let a user add there own parts to your game, I leave it up to you now to think about a function that would be used to check to see if a chess move was valid, lets assume you have a function in LUA like piece(x,y) that will return the team and piece type at a current location. Using this thing about how you would write validmove(x_source,y_source,x_dest,y_dest) that would check to see if piece at source can move to destination? It's not to hard to start. Add a lua function like removepiece(x,y) that lets you remove/capture an enemy piece. Now we must learn how to create the piece and removepiece functions in C and learn how to export them to LUA.

Example 3

Loads a LUA file "ex3.lua" that uses the C function double RMath(double a, double b); RMath is a function, in C, that takes two doubles and multiplies them then returns the result. The LUA function "ex3f" will take two numbers and perform RMath on them.

Now LUA can call C functions but not just any c function, there is a little bit of preparation work involved to make the function LUA ready. C functions in LUA must act like LUA functions, all data has to be passed via the stack but each function will have its

own stack including recursive calls where the C code calls the LUA function that called it.

Ex3.lua

```
function ex3f (num_a,num_b)
    print("lua pringing:",rmath(num_a,num_b))
    return rmath(num_a, num_b)
end
```

Ex3.cpp

```
#include <lua.h>
#include <lauxlib.h>
#include <lualib.h>
#include <iostream>
using namespace std;

double RMath(double a, double b);
int RMath_LUA(lua_State* L);

int main()
{
      lua_State *L;
      L = lua_open();

      luaopen_base(L);              /* opens the basic library */
      luaopen_table(L);             /* opens the table library */
      luaopen_io(L);                /* opens the I/O library */
      luaopen_string(L);            /* opens the string library. */
      luaopen_math(L);              /* opens the math library. */

      int temp_int;
      temp_int = luaL_loadfile(L,"ex3.lua");
      if (temp_int)
      {
            cout << "cannot load file ex3.lua."
                  << lua_tostring(L, -1) <<endl;
            lua_close(L);
            return 1;
      }
      temp_int = lua_pcall(L,0,0,0);
      if (temp_int)
      {
            cout << "error with pcall"
                  << lua_tostring(L, -1)<< temp_int <<endl;
            lua_close(L);
            return 1;
      }



      lua_pushcfunction(L,RMath_LUA);
      /*pushed the C function RMath_LUA, on to the top of the stack
where it can be accessed*/
    lua_setglobal(L, "rmath");
```

```
      /*once again a function is just a data type so pop it off the
stack and assign it to a global variable called rmath*/

      lua_getglobal(L,"ex3f");
/*functions in lua are a first class data value, so much like rows = 5
ex2f = a function to perform some operations ex2f is now on the top of
the stack*/



    lua_pushnumber(L, 3);   /* push 1st argument */
    lua_pushnumber(L, 4);   /* push 2nd argument */

      /* do the call (2 arguments, 1 result) */
      if (lua_pcall(L, 2, 1, 0) != 0)
      {
            cout << "error with pcall of ex3f: "
                  << lua_tostring(L, -1)<< temp_int <<endl;
            lua_close(L);
            return 1;
      }

      /*check return types*/
      if (lua_isnumber(L, -1))
      {
                  cout << "return value 1 (double): (" <<
lua_tonumber(L,-1) << ")" << endl;
            lua_pop(L,1);
/*here we pull the data off the stack, lua_to* is a peek function we
need to call pop to clean up the stack. pcall cleans up the stack of
the function and its arguments*/
      }
      else
      {
            cout << "invalid values returned";
      }
      lua_close(L);
}


/*this is a basic c function you don't have to know anythink about lua
to write it and as such it can be used freely by any c or c++ app*/
double RMath(double a, double b)
{
 return a*b;
}


/*any function that is called by LUA must have the following format:
int func_name (lua_State* L)*/
int RMath_LUA(lua_State* L)
{
      double a = lua_tonumber(L, 1);  /* get 1st argument  */
      double b = lua_tonumber(L, 2);  /* get 2nd argument  */
      lua_pushnumber(L,RMath(a,b)); /*push the results back on the
stack*/
      return 1; /*number of arguments returned*/
```

```
}
```

Hopefully now that you've read all this you should be able to start writing you own additions to LUA and thinking about how LUA can make your next game ever better. Remember give the player a game he plays it once, give the gamer community a tool and they will use it for years.

References

As I said the best source for this kind of information is the source code currently available at http://www.lua.org/ftp/lua-5.0.2.tar.gz with the aid of visual studio's right click "find definition" it is actually readable, ok mostly readable but better than you would expect for free and it is rather advanced source code.

For the weaker at heart we have the manual at http://www.lua.org/manual/5.0/ but as I've said many times before don't actually expect to find any documentation of half the functions I've used above.

Realistically the only good reference is the book p
(Programming in Lua)
by Roberto Ierusalimschy
Lua.org, December 2003
ISBN 85-903798-1-7
It can be found online at http://www.lua.org/pil/ or the new version for sale at http://www.amazon.com/exec/obidos/ASIN/8590379817/theprogrammil-20 it's a great read and worth buying if you plan on making any money off LUA. It was written by one of the creators. Sadly it only gives examples and almost no in depth explanation of all the finer points, I would love to see a reference as good as the MSDN http://msdn.microsoft.com or the php manual, but this is just a dream.

The last and final reference I used was http://lua-users.org/ which this will hopefully become part of, sadly I didn't find the information on there as useful as I would have liked but it's wiki so this will likely change.