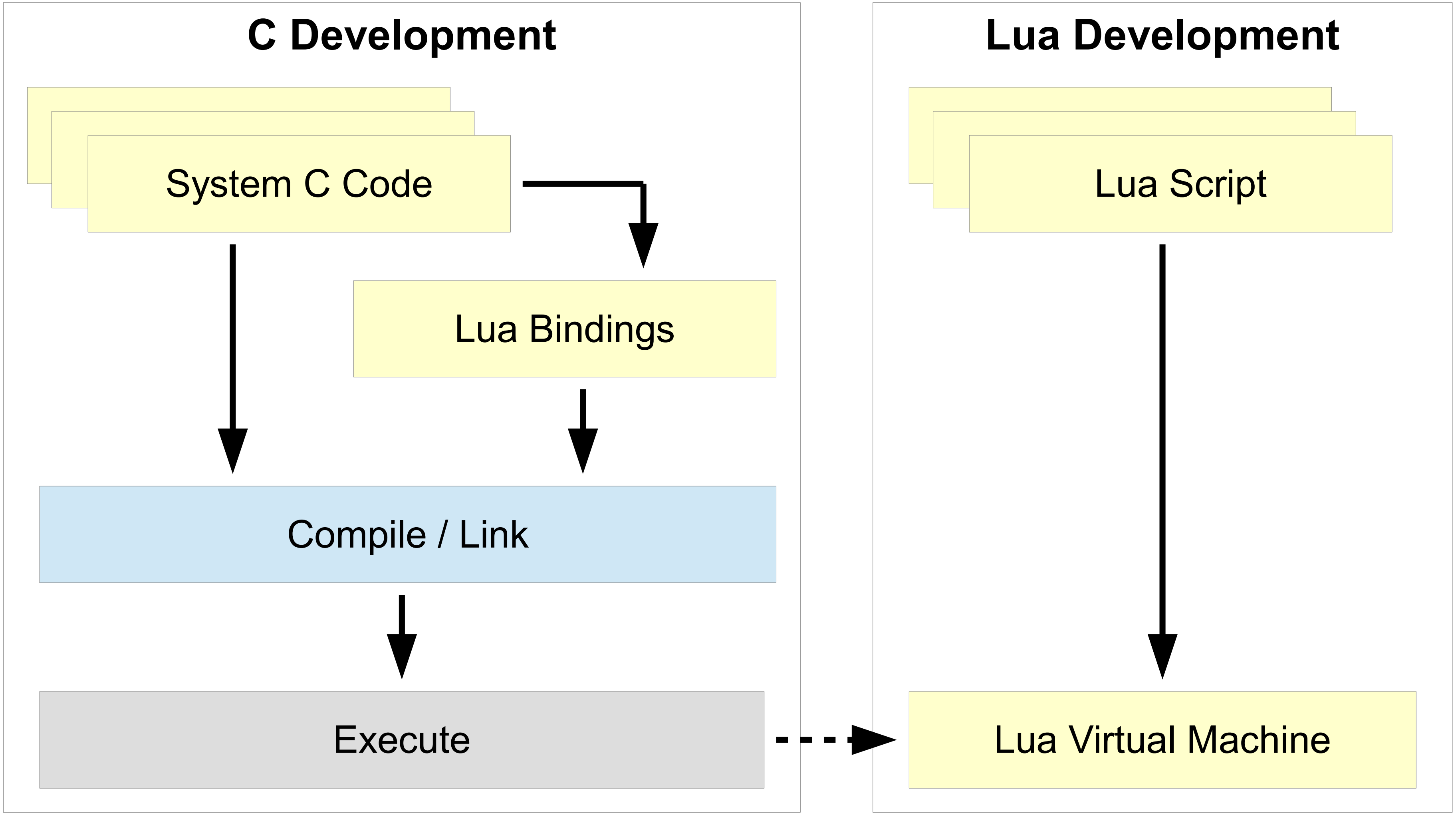
Lua Binding


C Development
System C Code
Lua Bindings
Compile / Link
Execute
Lua Development
Lua Script
Lua Virtual Machine

Contents

- What is a Scripting Language?
- Why use a Scripting Language?
- Why use Lua?
- Examples of Lua-Scripted Games
- The Lua C API
  - Initialization and Shutdown
  - The Lua Stack
  - Calling Lua Functions From C
  - Calling C Functions From Lua
  - Error Handling

What is a Scripting Language?

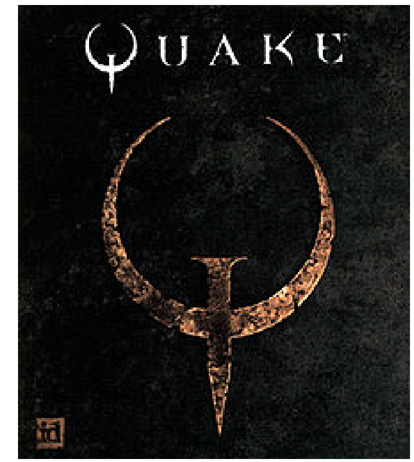
- A *Scripting Language* (SL) is a high-level programming language that is interpreted by another program at runtime.
  - Often embedded within a native application
  - May be interpreted directly or compiled to bytecode
  - Typically intended to be very fast to pick up and author programs in
  - Relatively simple syntax and semantics
  - SLs abstract their users from variable types and memory management
  - May be designed for use by end users of a program or may be only for internal use by developers
- The first SL used in a game was most likely SCUMM
  - Script Creation Utility for Maniac Mansion
  - Created during the developed of Maniac Mansion to create locations, dialogue, objects, puzzles, etc. without having to touch the 6502 assembly code

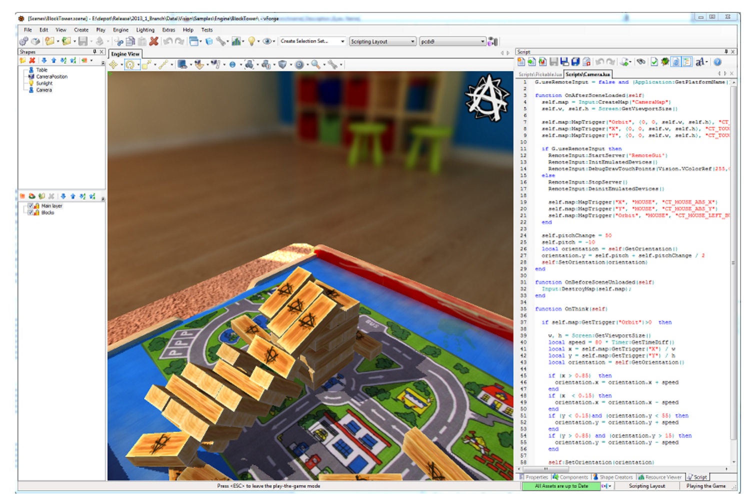
Why use a scripting language? (1/2)

- Rapid prototyping
  - It is much faster to script prototypes than to code them natively
- Short turnaround times
  - Scripts can be reloaded at runtime
  - The program can recover from errors and (probably) continue
- Less programming skill is needed
  - E.g. Level designers can create, maintain, and balance scripts
- Can be embedded into other file types (XML, CSV, etc.)
  - E.g. Conditions within a XML dialogue tree
- Users can modify certain parts of a program in a save environment
  - E.g. Custom user interfaces, button-mappings, or short-cuts
- Drawback: Scripting languages are generally slow compared to native code and take up more memory.

Why use a scripting language? (2/2)

- Game engines with scripting support (examples)
- Quake Engine: QuakeC
  - Developed in 1996 by John Carmack
- Unreal Engine: UnrealScript (or UScript)
  - Developed by Tim Sweeney
  - Used for authoring game code and gameplay events
- CryENGINE: Lua
  - Used for game rules, AI, inventory, networking, etc.
  - Can also be programmed in flow graphs
- Project Anarchy: Lua
  - Gameplay programming and triggering or changing the states of game entities
  - Can also be used for low-level game programming





Why use Lua?

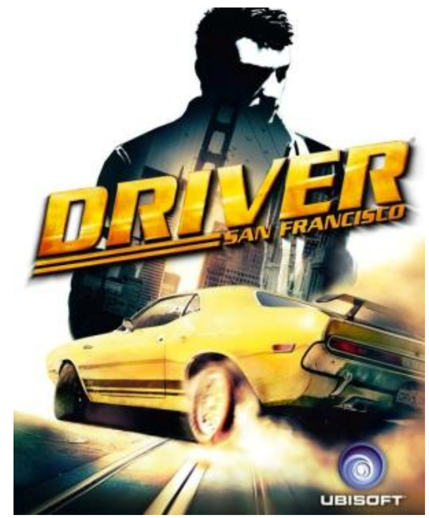
- Lua is created by a team at PUC-Rio, the Pontifical Catholic University of Rio de Janeiro in Brazil in 1993.
- Lua (pronounced LOO-ah) means "Moon" in Portuguese
- Characteristics
  - Lua has a deserved reputation for performance
    - To claim to be "as fast as Lua" is an aspiration of other scripting languages
  - Lua is portable and compiles out-of-the box on almost any platform
    - Only a standard C compiler is required
  - Lua has small footprint that you can embed easily into your application
    - E.g. Squeezed into 1 MB in *Der Fluch der Osterinsel* (Nintendo Wii)
  - Lua provides meta-mechanisms for implementing features, instead of providing a host of features directly in the language
  - Lua is small (source code and documentation takes 246kB compressed)
  - Lua is free open-source software, distributed under the MIT license

Examples of Lua-Scripted Games

- Grim Fandango
  - Considered the first use of Lua in gaming applications

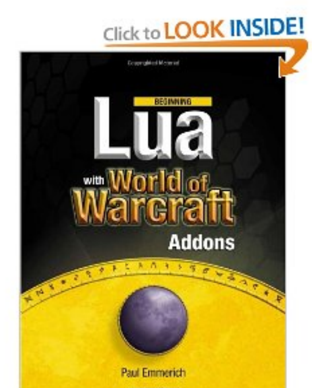
- Angry Birds

- Driver: San Francisco



- Don't Starve

- World of Warcraft
  - Interface customization



The Lua C API

- Lua is an embedded language and not a stand-alone program
  - The Lua library can be linked with other applications so as to incorporate Lua facilities into these applications.
    - E.g. Lua interpreter (lua.exe) or Lua compiler (luac.exe)
- The Lua C API is the set of functions (and macros) that allow C code to interact with Lua.
  - Read and write Lua global variables
  - Call Lua functions
  - Run pieces of Lua code
  - Register C functions so that they can later be called by Lua code
  - Most functions in the API don't check the correctness of their arguments
  - The API emphasizes flexibility and simplicity, sometimes at the cost of ease of use, which is why common tasks may involve several API calls.

    => Gives full control over all details like error handling, buffer sizes, etc.

Initialization and Shutdown

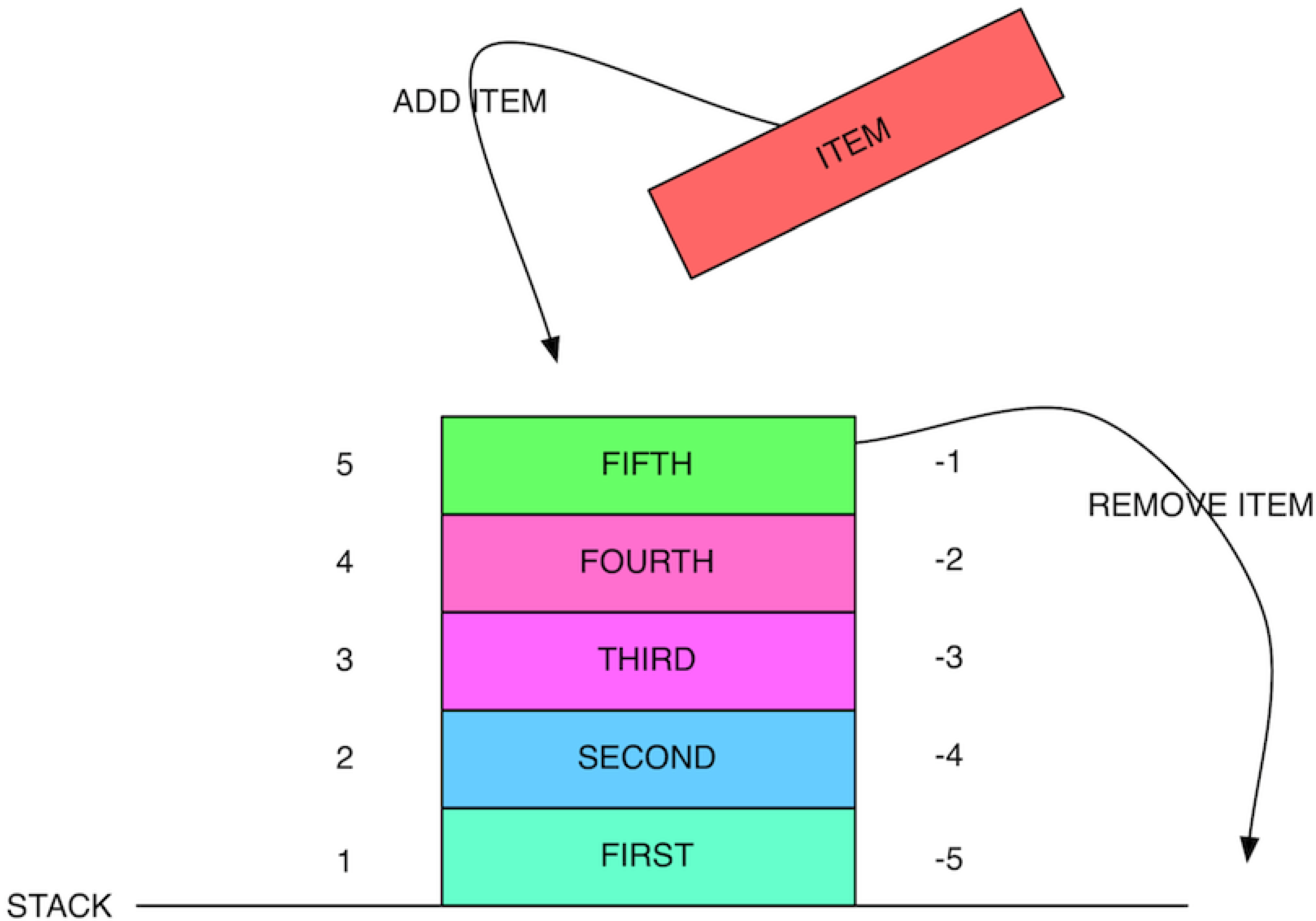
```
extern "C"              // Lua is written in C, so we must enfoce C linkage conventions
{
#include "lua.h"        // main Lua header file (functions, definitions, constants, etc.)
#include "lauxlib.h"    // auxiliary library, provides several convenience functions (luaL_)
#include "lualib.h"     // contains all Lua standard libraries (math, string, io, os, etc.)
}

int main(int argc, char* argv[])
{
    // create a new Lua state
    lua_State* L = luaL_newstate();

    // open all Lua standard libraries
    luaL_openlibs(L);

    // execute a string as Lua code
    luaL_dostring(L, "print('hello from Lua!')\n");

    // close the Lua state
    lua_close(L);

    return 0;
}
```

```
    Output:
    hello from Lua!
```

The Lua Stack (1/2)

- Lua uses a *virtual stack* to pass values to and from C
  - Each stack element represents a Lua value (nil, number, string, etc.)
- Whenever Lua calls C, the called function gets a new stack
  - The stack initially contains any arguments to the C function and it is where the C function pushes its results
- Most queries can refer to any element in the stack by using an index
  - A positive index represents an absolute stack position (starting at 1)
  - A negative index represents an offset relative to the top of the stack
- Valid vs. acceptable indices
  - A *valid index* is an index that refers to a real position within the stack, that is, it lies between 1 and the stack top (1 ≤ abs(index) ≤ top)
  - An *acceptable index* can be any *valid index*, but it also can be any positive index after the stack top within the space allocated for the stack
    - E.g., a C function can query its third argument without the need to first check whether there actually is a third argument on the stack.

The Lua Stack (2/2)



Calling Lua Functions From C (1/2)

```lua
-- ScriptCode.lua

function testFunction(a, b, c)
    print(type(a), tostring(a))
    print(type(b), tostring(b))
    print(type(c), tostring(c))
    return 1, 2, 3
end
```

```c
// load and run the Lua file
luaL_dofile(L, "ScriptCode.lua");

// push the function on top of the stack
lua_getglobal(L, "testFunction");

// check, if the function is now on top of the stack
if (!lua_isfunction(L, -1))
    printf("error: function not found\n");

// call the function with 0 arguments and 0 return values
lua_call(L, 0, 0);
```

```
Output:
nil     nil
nil     nil
nil     nil
```

Calling Lua Functions From C (2/2)

```
// push the function on top of the stack
lua_getglobal(L, "testFunction");

// push two arguments on the stack
lua_pushinteger(L, 7);
lua_pushstring(L, "some text");
// third parameter will be nil

// call the function with 2 arguments and 3 return values
lua_call(L, 2, 3);

// convert all three return values to int
if (!lua_isnumber(L, -3)) printf("error: expected number\n");
int i = lua_tointeger(L, -3);
if (!lua_isnumber(L, -2)) printf("error: expected number\n");
int j = lua_tointeger(L, -2);
if (!lua_isnumber(L, -1)) printf("error: expected number\n");
int k = lua_tointeger(L, -1);

// print the return values
printf("i %i, j %i, k %i\n", i, j, k);
```

```
Output:
number  7
string  some text
nil     nil
i 1, j 2, k 3
```

Calling C Functions From Lua (1/2)

- A C function receives its arguments from Lua in its stack in direct order (the first argument is pushed first).
  - lua_gettop(L) returns the number of arguments received by the function.
    - i.e., lua_gettop(L) returns the index of the top element in the stack.
  - The first argument (if any) is at index 1.
  - The last argument is at index lua_gettop(L).
- To return values to Lua, a C function just pushes them onto the stack, in direct order (the first result is pushed first), and returns the number of results.
  - Any other value in the stack below the results will be discarded by Lua.
  - A C function called by Lua can also return many results.

```
// type definition of lua_CFunction in lua.h

typedef int (*lua_CFunction) (lua_State *L);
```

Calling C Functions From Lua (2/2)

```c
int lua_CFunction_multiply(lua_State* L)
{
      // get the number of arguments
      int n = lua_gettop(L);
      if (n != 2) printf("error: expected two arguments\n");

      // get the first argument
      if (!lua_isnumber(L, 1)) printf("error: expected number\n");
      float x = (float)lua_tonumber(L, 1);
      // get the second argument
      if (!lua_isnumber(L, 2)) printf("error: expected number\n");
      float y = (float)lua_tonumber(L, 2);

      // push the return value on the stack
      lua_pushnumber(L, x * y);
      // the function returns 1 value
      return 1;
}
```

```c
      // inside main() ...

      // push the function on top of the stack
      lua_pushcfunction(L, lua_CFunction_multiply);

      // pop the top value from the stack and set it as value of the global "multiply"
      lua_setglobal(L, "multiply");
```

```lua
      -- ScriptCode.lua

      z = multiply(3.5, 2)
      print(z)
```

```
      Output:

      7
```

Error Handling (1/2)

```
-- ScriptCode.lua
function testFunction(a)
     print(type(a), tostring(a))
end
print("success!")
```

```
-- ScriptCodeERROR.lua
function testFunction(a)
     print(type(a), tostring(a) -- missing )
end
print("success!")
```

```
void printLuaError(lua_State* L)
{
     // the top of the stack *should* now contain the error message
     const char* theError = lua_tostring(L, lua_gettop(L));
     lua_pop(L, 1);
     printf(theError); printf("\n");
}
```

```
     // inside main() ...

     // let's replace luaL_dofile with something safer
     int err = luaL_loadfile(L, "ScrcrcrcriptCode.lua");
     if (err != LUA_OK) printLuaError(L);

     // now load an existing file with a syntax error
     err = luaL_loadfile(L, "ScriptCodeERROR.lua");
     if (err != LUA_OK) printLuaError(L);

     // finally load the correct file
     err = luaL_loadfile(L, "ScriptCode.lua");
     if (err == LUA_OK)
     {
          // the top of the stack *should* be the function to call to run the file
          err = lua_pcall(L, 0, LUA_MULTRET, 0);
          if (err != LUA_OK) printLuaError(L);
     }
```

```
Output (with additional line breaks):
cannot open ScrcrcrcriptCode.lua:
No such file or directory
ScriptCodeERROR.lua:5:
')' expected (to close '(' at line 4) near 'end'
success!
```

Error Handling (2/2)

- `int luaL_error (lua_State *L, const char *fmt, ...);`
  - Raises an error.
  - The error message format is given by fmt plus any extra arguments.
  - It also adds at the beginning of the message the file name and the line number where the error occurred, if this information is available.
- `lua_CFunction lua_atpanic (lua_State *L, lua_CFunction panicf);`
  - Sets a new panic function and returns the old one.
  - If an error happens, Lua calls a panic function and then calls abort, thus exiting the host application.
  - A custom panic function can avoid this exit by never returning.
    - E.g. long jump, goto, throw, etc.
- Exception handling
  - `LUAI_THROW`/`LUAI_TRY` define how Lua does exception handling.
  - By default, Lua handles errors with exceptions when compiling as C++.

References

- http://www.lua.org
- http://en.wikipedia.org/wiki/Scripting_language
- http://realtimelogic.com/blog/2012/08/Lua-FastTracks-Embedded-Web-Application-Development
- http://dev.oz-apps.com/?p=331
- http://en.wikipedia.org/wiki/Category:Lua-scripted_video_games
- http://en.wikipedia.org/wiki/Grim_Fandango
- http://en.wikipedia.org/wiki/QuakeC
- http://en.wikipedia.org/wiki/UnrealScript
- http://docs.cryengine.com/display/SDKDOC5/Home
- http://www.projectanarchy.com/scripting-1