

Lua y C++

Integración

• © Diego Garcés

10/04/2016 •

Lua como librería

- El lenguaje Lua esta pensado como una extensión
- Un lenguaje empotrado (embedded)
- Realmente es una librería que podemos usar
- Lua.exe es sólo una aplicación que usa esa librería
- Moai.exe es otra aplicación
 - Por eso podemos pasar un .lua normal a moai.exe y funciona

• © Diego Garcés

10/04/2016 •

Lua como librería

```
graph LR; LuaLib[Lua.lib] --> LuaExe[Lua.exe]; LuaLib --> MoaiExe[Moai.exe];
```

• © Diego Garcés

10/04/2016 •

Lua como librería

- Permite la comunicación C++ → Lua
- Acceder a variables globales de Lua
- Llamar a funciones de Lua
- Ejecutar código Lua
- Proporciona una API de Lua

• © Diego Garcés

10/04/2016 •

C++ y Lua: diferencias

- C++ gestiona memoria explícitamente: new y delete
- Lua tiene un garbage collector

• © Diego Garcés

10/04/2016 •

Lua garbage collector

- Proceso que libera memoria no usada
- Permite olvidarse de la gestión de memoria
- Elimina los memory leaks
- No se borran cosas
- Simplemente se dejan de usar

¿Suena bien, no?

• © Diego Garcés

10/04/2016 •

Lua garbage collector

- `criatura = { pos = {x = 20, y = 50}, items = { "sword", "shield" } }`

```
graph LR; criatura[criatura] -- pos --> pos_table[pos]; criatura -- items --> items_table[items]; pos_table -- x --> 20[20]; pos_table -- y --> 50[50]; items_table -- 1 --> sword[sword]; items_table -- 2 --> shield[shield];
```

• © Diego Garcés

10/04/2016 •

Lua garbage collector

- `criatura.items[2] = nil`

```
graph LR; criatura[criatura] -- pos --> pos_table[pos]; criatura -- items --> items_table[items]; pos_table -- x --> 20[20]; pos_table -- y --> 50[50]; items_table -- 1 --> sword[sword];
```

• © Diego Garcés

10/04/2016 •

Lua garbage collector

- `Criatura.pos = nil`

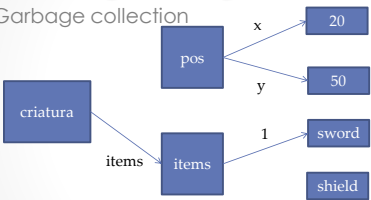
```
graph LR; criatura[criatura] -- items --> items_table[items]; items_table -- 1 --> sword[sword];
```

• © Diego Garcés

10/04/2016 •

Lua garbage collector

• Garbage collection



GC: Inconvenientes

- No hay control sobre el proceso
 - Compromiso entre transparencia y control
- Proceso algo costoso
 - Recorrer todos los datos para ver quién esta huérfano
 - Liberar toda esa memoria
- Puede suponer tirones de frame-rate
 - No abusar
 - Invocar al proceso de garbage collection explícitamente
 - Garbage collector incremental: Libera lo que de tiempo

C++ y Lua: Diferencias

- C++ es estaticamente tipado
- Lua es dinámicamente tipado
- Cuidado con pasar la información correctamente

Librería de Lua

- lua.h
 - Cabecera de la librería principal de lua
 - Funciones para todo tipo de operaciones con Lua
 - Prefijo lua_
- lauxlib.h
 - Cabecera con funciones de alto nivel
 - Usan las funciones de la API normal
 - Hacen más fácil ciertas operaciones
 - Prefijo luaL_

• © Diego Garcés

10/04/2016 •

Solución: Pila Virtual

- Es el mecanismo de intercambio de información entre C++ y Lua
- Permite reconciliar las diferencias

```
graph LR; Cplusplus[C++] <--> Pila[Pila]; Pila <--> Lua[Lua];
```

• © Diego Garcés

10/04/2016 •

Entorno de Lua

- No hay variables globales en librería
- Todo almacenado en una estructura
 - Tipo lua_State
 - Dinámica: Se añaden y borran datos
 - Todas las funciones de la librería reciben este parámetro
 - Así es posible usarlas desde varios threads
 - Si no tuviera cada thread su propio estado podría haber colisiones

• © Diego Garcés

10/04/2016 •

Entorno de lua

- luaL_newstate: crea un entorno nuevo. Vacío
- luaL_openlibs (luaolib.h): abre las librerías estándar
- El entorno está creado y listo para usarse

Código lua

- Siempre se trabaja con la pila
- Todos los datos hay que colocarlos en la pila
- luaL_loadbuffer: Compila código y coloca el programa resultante en la pila
 - Devuelve 0 si no ha habido error
 - Si hay error: devuelve distinto de 0 y coloca el mensaje de error en la pila
 - Alternativas: luaL_loadfile, luaL_loadstring dependiendo dónde este el código lua
- lua_pcall: Saca un programa de la pila y lo ejecuta
 - Devuelve 0 si no ha habido error
 - Si hay error: devuelve distinto de 0 y coloca el mensaje de error en la pila

Intérprete de lua

- Con esto ya podríamos hacer nuestro propio intérprete de lua

```
#include <stdio.h>
#include "lua.h"
#include "lauxlib.h"
#include "luaolib.h"
int main (void) {
    lua_State *L = luaL_newstate(); /* crea el entorno de lua */
    luaL_openlibs(L); /* abre las librerías */
    int error = luaL_loadfile(L, "test.lua"); /* carga el código en la pila */
    error |= lua_pcall(L, 0, 0, 0); /* ejecuta el código */
    if (error) {
        fprintf(stderr, "%s", lua_tostring(L, -1)); /* el mensaje de error está en la cima de la pila */
        lua_pop(L, 1); /* quitar el mensaje de error de la pila */
    }
    lua_close(L); /* cierra el entorno */
    return 0;
}
```

Pila

- Lua es un lenguaje dinámicamente tipado
- Para evitar tener que gestionar todos los tipos o definir un union para cualquiera
 - lua_settablevalue(lua_value table, lua_value field, lua_value value)
- Para evitar añadir mecanismos para el garbage collector
 - Un variable de C++, ¿cómo se sabe si se está referenciando para borrarla?
- Solución → Una pila de valores abstractos
 - La pila la maneja lua → puede comunicarse con el garbage collector
 - Sirve de intercambio de valores, en lugar de parámetros tipados

• © Diego Garcés

10/04/2016 •

Pila

- LIFO: last in first out
 - Añades valores encima de la pila (push)
 - Sacas valores de encima de la pila (pop)
 - El último en añadirse es el primero en sacarse
- Proceso:
 - Añadir valores a la pila
 - Llamar a código lua
 - Lua hace pop de los valores que necesita
 - Se ejecuta el código

• © Diego Garcés

10/04/2016 •

Pila: Añadir valores

- C++ es tipado → una función por cada tipo de dato de lua
- lua_pushnil: añade un nil
- lua_pushnumber: añade un double
- lua_pushinteger: añade un entero
- lua_pushboolean: añade un booleano
- lua_pushlstring: añade una cadena de caracteres, con una longitud
- lua_pushstring: añade una cadena de caracteres terminada en 0

• © Diego Garcés

10/04/2016 •

Pila: añadir valores


- Lua siempre hace una copia del valor en la pila
- También con cadenas
 - Podemos modificar la cadena de C++ después de añadirla a la pila
 - O liberarla
- La pila tiene un tamaño máximo
 - `LUA_MINSTACK` (`lua.h`) es el número mínimo de slots que tiene que tener la pila al crearse
 - `int lua_checkstack(lua_State* L, int sz)`: Para comprobar si hay espacio suficiente

• © Diego Garcés

10/04/2016 •

Pila: añadir valores

- `Lua_pushinteger(L, 20)`



• © Diego Garcés

10/04/2016 •

Pila: añadir valores

- `Lua_pushinteger(L, 20)`




• © Diego Garcés

10/04/2016 •

Pila: añadir valores


- `Lua_pushinteger(L, 20)`
- `Lua_pushstring(L, "hi")`



© Diego Garcés 10/04/2016

Pila: añadir valores

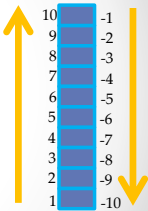
- `Lua_pushinteger(L, 20)`
- `Lua_pushstring(L, "hi")`



© Diego Garcés 10/04/2016

Pila: recoger datos

- Para acceder elementos se usa su índice
- Índices positivos: cuentan desde la parte baja de la pila
- Índices negativos: cuentan desde la cima de la pila



© Diego Garcés 10/04/2016

Pila: tipos

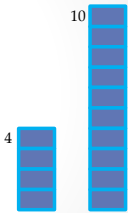
- La pila puede contener cualquier tipo de dato
- Saber que tipo tiene un dato de la pila → Una función para cada tipo
- `int lua_isnumber(lua_State* L, int index)`
- `lua_isstring`
- `lua_istable`
- ...
- Devuelven true si el valor se puede convertir a ese tipo
- `lua_type`: devuelve el tipo. Constantes definidas en `lua.h`

Pila: recoger datos

- Una función para cada tipo de dato
- `lua_Number lua_tonumber(lua_State* L, int index)`
- `int lua_toboolean`
- `lua_Integer lua_tointeger`
- `size_t lua_objlen`
- `const char* lua_tolstring(lua_State* L, int index, size_t* len)`
- Si no es del tipo correcto devuelven 0 (NULL)

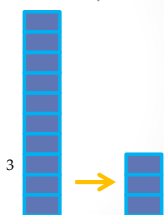
Pila: Manipulación

- `int lua_gettop(lua_State* L)`



Pila: Manipulación

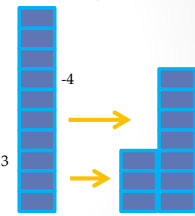
- `void lua_settop(lua_State* L, int index)`
- `lua_settop(state, 3)`



© Diego Garcés 10/04/2016

Pila: Manipulación

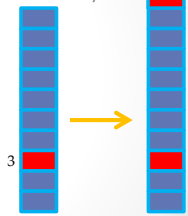
- `void lua_settop(lua_State* L, int index)`
- `lua_settop(state, 3)`
- `lua_settop(state, -4)`
- `lua_pop(state, 3)`



© Diego Garcés 10/04/2016

Pila: Manipulación

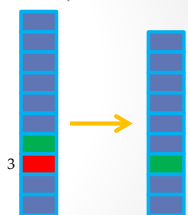
- `void lua_pushvalue(lua_State* L, int index)`
- `lua_pushvalue(state, 3)`



© Diego Garcés 10/04/2016

Pila: Manipulación

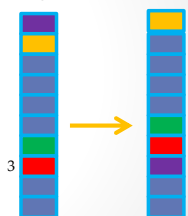
- void lua_remove(lua_State* L, int index)
- lua_remove(state, 3)



© Diego Garcés 10/04/2016

Pila: Manipulación

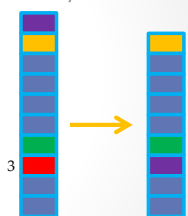
- void lua_insert(lua_State* L, int index)
- lua_insert(state, 3)



© Diego Garcés 10/04/2016

Pila: Manipulación

- void lua_replace(lua_State* L, int index)
- lua_replace(state, 3)



© Diego Garcés 10/04/2016

Lua ← C++: Uso

- Un uso de lua desde C++ es como lenguaje de configuración
- Ciertos parámetros de nuestro programa se definen en lua
- Como variables globales
- Más flexibilidad que con un .ini o un .txt

Lua ← C++: Ejemplo

- Parámetros de velocidad angular y aceleración angular máximas

```
-- Parametros para seek steering
max_angular_vel = 20
max_angular_acc = 40

-- Parametros para path following
```

Lua ← C++

- Pasos
 - Crear un entorno
 - Compilar el código
 - Ejecutar el código lua
 - Recuperar los valores de las variables
- Qué sabemos ya
 - luaL_newstate
 - luaL_loadbuffer
 - lua_pcall
 - ¿Poner variables globales en la pila?
 - lua_isnumber
 - lua_tonumber

Lua ← C++: Globales

- lua_getglobal(lua_State* L, const char* varName)
- Pone el valor de la variable global de ese nombre en la cima de la pila

Lua ← C++: Ejemplo

- Añadir parametros para path following

```
-- Parametros para seek steering
seek_max_angular_vel = 20
seek_max_angular_acc = 40

-- Parametros para path following
path_follow_max_angular_vel = 10
path_follow_max_angular_acc = 20
```

Lua ← C++: ejemplo

- Mejor solución → Usar tablas

```
-- Parametros para seek steering
seek = {}
seek.max_angular_vel = 20
seek.max_angular_acc = 40

-- Parametros para path following
path_follow = {}
path_follow.max_angular_vel = 10
path_follow.max_angular_acc = 20
```

Lua ← C++: Tablas

- Objetivo: colocar el valor de un campo de una tabla en la pila
- `lua_gettable(lua_State* L, int index)`
 - Coloca en la cima de la pila el campo cuyo nombre está indicado en la cima de la pila
 - La tabla está en la posición indicada por index
- Objetivo: leer el valor del campo
 - Usar las funciones que ya conocemos para tratar con valores
 - `lua_is*`
 - `lua_to*`

© Diego Garcés

10/04/2016

Lua ← C++: Tablas

- `lua_getglobal(L, "criatura")`
- `lua_pushstring(L, "x")`
- `lua_gettable(L, -2)`

criatura

x = 20

y = 30

x

20

© Diego Garcés

10/04/2016

Lua ← C++

- Es tan común que hay definida una función para ello
- `lua_getfield(lua_State* L, int index, const char* key)`
 - `lua_pushstring(L, key)`
 - `lua_gettable(L, index)`
- `lua_getglobal(L, "criatura")`
- `lua_pushstring(L, "x")`
- `lua_gettable(L, -2)`
- `lua_getglobal(L, "criatura")`
- `lua_getfield(L, -1, "x")`

© Diego Garcés

10/04/2016

Lua ← C++

- Si queremos poner valores en una tabla
 - Por ejemplo para dar valores por defecto
- Equivalente a obtener valores de tablas
- `lua_setfield(lua_State* L, int index, const char* key)`
 - Coloca en la tabla de la posición index, con la clave key, el valor de la cima de la pila
 - Elimina el valor de la cima de la pila

• © Diego Garcés

10/04/2016 •

Lua ← C++

- Para valores por defecto hay que crear una tabla
- `lua_newtable(lua_State* L)`
 - Crea una nueva tabla y la coloca en la cima de la pila
- `lua_setglobal(lua_State* L, const char* name)`
 - Asigna el valor de la cima de la pila a la variable global con nombre name
 - Elimina el valor de la cima de la pila
 - Hay que rellenar la tabla antes de hacer el setglobal (para no tener que volver a traerla a la pila)

• © Diego Garcés

10/04/2016 •

Lua ← C++: Funciones

- **Pasos:**
- Poner la función en la pila
- Poner los argumentos en la pila
- Llamar a la función
- Recuperar el resultado de la pila

• © Diego Garcés

10/04/2016 •

Lua ← C++: Funciones

- Poner la función en la pila
- Las funciones son datos como otro cualquiera
- Poner un valor de una variable global en la pila
- lua_getglobal(lua_State* L, const char* name)
- Exactamente igual que para recuperar nuestros parámetros

Lua ← C++: Funciones

- Poner los parámetros en la pila
- Funciones lua_push*

Lua ← C++: Funciones

- Llamar a la función
- lua_pcall(lua_State* L, int num_parameters, int num_results, error_func)
- Las reglas de los parámetros son las mismas que en lua
 - Si faltan parámetros, se pasan nil
 - Si faltan resultados, se devuelven nil
- La llamada lua_pcall elimina de la pila
 - Función
 - Parámetros

Lua ← C++: Funciones

- Recuperar resultados
- Al llamar a la función, los resultados se añaden a la pila
- El primer resultado se añade el primero
- El último resultado acaba en la cima de la pila (posición -1)
- Habrá tantos resultados como le hayamos dicho a lua_pcall
