Lua Object-Oriented Programming

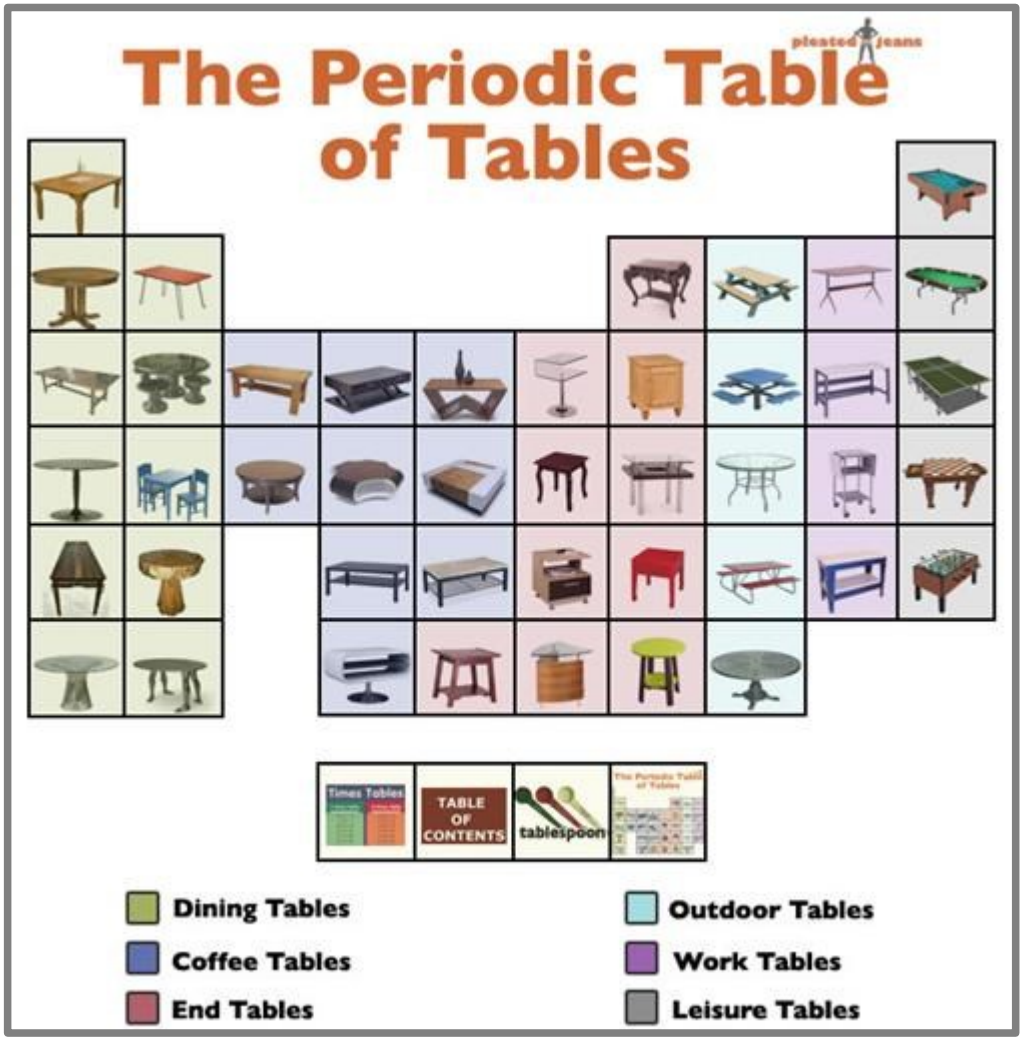


Image source: http://cheezburger.com/5076968704, 2014

metatable, table stores information about other tables

Contents

- Repeat: Lua Tables with 'Member' Functions
- Lua Metatables and Metamethods
  - Operator Overloading
  - `__index` and `__newindex`
- Manipulating Lua Tables from C
  - Creating Tables
  - Reading Table Elements
  - Handling Metatables
- Binding C++ Classes to Lua
  - Reference Types
  - Value Types

Repeat: Lua Tables with 'Member' Functions

```
--
-- tables and self
--
enemy = { }
enemy["hp"] = 1
function enemy:heal()
     self.hp = self.hp + 20 -- self is an implicit parameter
end
function enemy:status()
     print("I have " .. self.hp .. " hp")
end
enemy:heal()
enemy:status()
```

```
Output:

I have 21 hp
```

- Problem: How can we instantiate another enemy?
  - 'enemy2 = enemy' would just create a reference to the same instance.
- We need to separate the functions from the data!

Lua Metatables and Metamethods

- Every value in Lua can have a *metatable*.
- This *metatable* is an ordinary Lua table that defines the behaviour of the original value under certain special operations.
- The keys in a *metatable* are derived from the event names; the corresponding values are called *metamethods*.
- The key for each operation is a string with its name prefixed by two underscores, '__'; e.g., the key for operation "add" is the string "__add".
- Mathematic Operators
  - add (+), sub (-), mul (*), div (/), mod (%)
  - pow (^), unm (unary -), concat (..)
- Equivalence Comparison Operators
  - eq (==), lt (<), le (<=)
- Other Event Names
  - index, newindex, mode, call, metatable, tostring, len (#), gc

something()

garbage collection, when struct is beeing garbage collected

Operator Overloading

```
Vec2Meta = { }                                  -- metatable

function Vec2(_x, _y)                           -- constructor
     local vec = {x=_x, y=_y}
     setmetatable(vec, Vec2Meta)
     return vec
end

Vec2Meta["__add"] = function(self, other)       -- addition
     local result = Vec2(0, 0)
     result.x = self.x + other.x
     result.y = self.y + other.y
     return result
end

Vec2Meta["__eq"] = function(self, other)        -- equality
     return (self.x == other.x and self.y == other.y)
end

v1 = Vec2(4, 2)
print(v1.x, v1.y)
v2 = Vec2(1, 3)
print(v2.x, v2.y)
v3 = v1 + v2
print(v3.x, v3.y)
v4 = Vec2(5, 5)
print(v4.x, v4.y)
print(v1 == v2, v2 == v3, v3 == v4)
```

```
Output:

4       2
1       3
5       5
5       5
false   false   true
```

__index and __newindex (1/3)

- __index → Control 'prototype' inheritance.
  - When accessing "myTable[key]" and the key does not appear in the table, but the *metatable* has an __index property:
    - if the value is a function, the function is called, passing in the table and the key; the return value of that function is returned as the result.
    - if the value is another table, the value of the key in that table is asked for and returned
    - (and if it doesn't exist in that table, but that table's metatable has an __index property, then it continues on up)
- __newindex → Control property assignment.
  - When calling "myTable[key] = value", if the *metatable* has a __newindex key pointing to a function, call that function, passing it the table, key, and value.

when something is written in a table, you can control what is done.

Image source: http://lostinrehearsal.blogspot.de/2013/10/the-legend-of-zelda-wind-waker-hd.html, 2014

__index and __newindex (2/3)

```
Vec2Meta = { }
Vec2Meta["__newindex"] = function(t, k, v)    -- __newindex event handler
     print("Vec2Meta:__newindex event handler")
     print(tostring(t) .. ", k: " .. k .. ", v: " .. v)
     rawset(t, k, v) -- don't forget to add the value to the table!
end
Vec2Meta["__index"] = function(t, k)          -- __index event handler: function
     print("Vec2Meta:__index event handler")
     print(tostring(t) .. ", k: " .. k .. " was not found")
     return function() print("dummy function") return -1 end
end
```

```
vec = { }
setmetatable(vec, Vec2Meta)
vec["y"], vec["x"] = 7, 5
print("vec:length() returns " .. vec:length() .. "\n")
```

```
Output:

Vec2Meta:__newindex event handler
table: 0067FCA0, k: x, v: 5
Vec2Meta:__newindex event handler
table: 0067FCA0, k: y, v: 7
Vec2Meta:__index event handler
table: 0067FCA0, k: length was not found
dummy function
vec:length() returns -1
```

__index and __newindex (3/3)

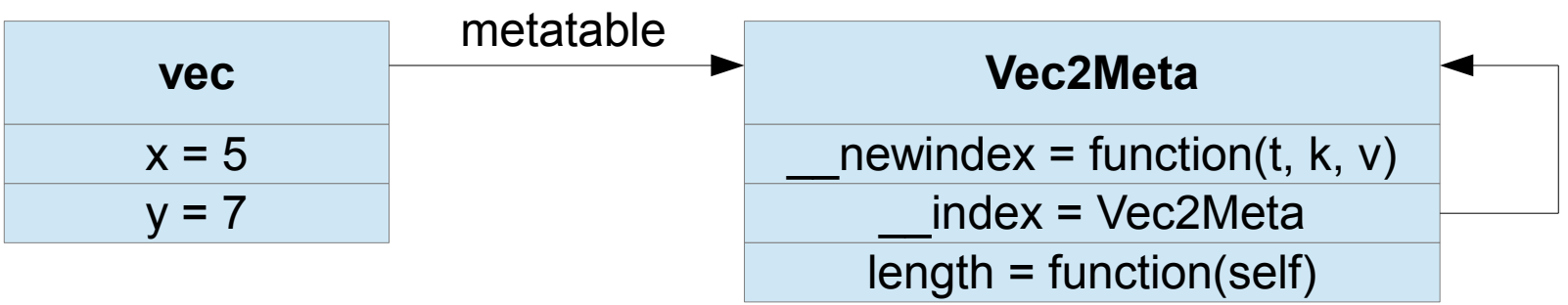
```
--
-- ... code continued from previous slide
--

Vec2Meta["__index"] = Vec2Meta            -- __index event handler: table
function Vec2Meta:length()
     return math.sqrt(self.x^2 + self.y^2)
end

print("vec:length() returns " .. vec:length())
```

```
Output:

vec:length() returns 8.6023252670426
```



Manipulating Lua Tables from C

- Like any other Lua type, tables are manipulated using the stack
  - General procedure for writing:
    - push the table on the stack
    - push a key and a value to the stack
    - use an API call to set the key and the value as new table field
  - General procedure for reading:
    - push the table on the stack
    - use an API call to read from the table → result is pushed to the stack
    - obtain the results (a value and eventually a key) from the stack
- We have already manipulated Lua's 'global' table before!
  - Lua keeps all its global variables in one or more a regular table(s), called *environment(s)*.
  - Lua stores the *environment* itself in a global variable _G.

```
s = "hello"
print(_G["s"])          -- _G is the global table
print(_G._G._G._G._G)   -- LOL ;-)
```

```
Output:
hello
table: 003BB168
```

Creating Tables

```
function printTable(t) for k, v in pairs(t) do print("k: " .. k .. ", v: " .. v) end end
```

```
-- Lua

t = { }


t[1] = 10


t[2] = 20

t["a"] = 30

t["b"] = 40



printTable(t)
```

```
// C
lua_newtable(L);                    // push a new table on the stack
lua_setglobal(L, "t");              // t = table at -1 (pops table)
lua_getglobal(L, "t");              // push the table on the stack
lua_pushnumber(L, 1);               // push k = 1 on the stack
lua_pushnumber(L, 10);              // push v = 10 on the stack
lua_settable(L, -3);                // t[k] = v (pops k and v), where
                                    // t at -3, k at -2, v at -1
lua_pushnumber(L, 2);               // push k = 2 on the stack
lua_pushnumber(L, 20);              // push v = 20 on the stack
lua_settable(L, -3);                // t[k] = v (pops k and v), where
                                    // t at -3, k at -2, v at -1
lua_pushnumber(L, 30);              // push v = 30 on the stack
lua_setfield(L, -2, "a");           // t[k] = v (pops v), where
                                    // t at -2, k is "a", v at -1
lua_pushnumber(L, 40);              // push v = 40 on the stack
lua_setfield(L, -2, "b");           // t[k] = v (pops v), where
                                    // t at -2, k is "b", v at -1
lua_pop(L, 1);                      // pop the table from the stack
lua_getglobal(L, "printTable");     // push the function on the stack
lua_getglobal(L, "t");              // push the table on the stack
lua_call(L, 1, 0);                  // call printTable(t)
```

create new table entry, from top 2 stack elem (-1 ,-2)

-3 is where the table is on the stack

1 = how many elements will be poped

executes first stack item

```
Output:
k: 1, v: 10
k: 2, v: 20
k: a, v: 30
k: b, v: 40
```


```
Output:
k: 1, v: 10
k: 2, v: 20
k: a, v: 30
k: b, v: 40
```

Reading Table Elements (1/2)

```
int lua_CFunction_printVec2(lua_State* L)
{
  int n = lua_gettop(L);                  // get the number of parameters
  printf("n: %i\n", n);
  lua_getfield(L, 1, "x");                // pushes t[k] on the stack, where t at 1, k is "x"
  float x = (float)lua_tonumber(L, -1); // get v at -1
  lua_pop(L, 1);                          // pop v from the stack
  printf("x: %.2f\n", x);
  lua_getfield(L, 1, "y");                // pushes t[k] on the stack, where t at 1, k is "y"
  float y = (float)lua_tonumber(L, -1); // get v at -1
  lua_pop(L, 1);                          // pop v from the stack
  printf("y: %.2f\n", y);
  return 0;
}

  // inside main()...
  lua_pushcfunction(L, lua_CFunction_printVec2);
  lua_setglobal(L, "printVec2");
```

```
  vec = {x=2, y=4}
  printVec2(vec)
```

```
  Output:
  n: 1
  x: 2.00
  y: 4.00
```

Reading Table Elements (2/2)

```
void printLuaType(lua_State* L, int idx)
{
  int type = lua_type(L, idx);
  switch(type)
  {
    case LUA_TNUMBER:    printf("%i", (int)lua_tonumber(L, idx));     break;
    case LUA_TSTRING:    printf("%s", lua_tostring(L, idx));          break;
    case LUA_TFUNCTION:  printf("()");                                break;
    case LUA_TTABLE:     printf("{}");                                break;
    default:             printf("error: unsupported type!\n");
  }
}

int lua_CFunction_printTable(lua_State* L)
{
  lua_pushnil(L);              // first key
  while (lua_next(L, 1) != 0) // reads from t at 1, pushes k at -2 and v at -1
  {
    const char* tk = lua_typename(L, lua_type(L, -2));
    const char* tv = lua_typename(L, lua_type(L, -1));
    printf("type(k): %s, type(v): %s\n", tk, tv);
    printf("k: "); printLuaType(L, -2); printf(", ");
    printf("v: "); printLuaType(L, -1); printf("\n");
    lua_pop(L, 1);  // remove v, keep k for next iteration
  }
  return 0;
}
```

```
  printTable({666, name="Beelzebub", age="unknown"})
  printTable({f=function() end, data={}})
```

```
Output:
type(k): number, type(v): number
k: 1, v: 666
type(k): string, type(v): string
k: age, v: unknown
type(k): string, type(v): string
k: name, v: Beelzebub

type(k): string, type(v): table
k: data, v: {}
type(k): string, type(v): function
k: f, v: ()
```

Handling Metatables

- Lua provides a *registry*, a predefined table that can be used by any C code to store whatever Lua values it needs to store.
- The *registry* table is always located at pseudo-index `LUA_REGISTRYINDEX`, which is a valid index.
- *Metatables* are typically stored in the *registry*.
  - `int luaL_newmetatable (lua_State *L, const char *tname);`
    - If the *registry* already has the key `tname`, returns 0.
    - Otherwise, creates a new table to be used as a *metatable* for *userdata*, adds it to the *registry* with key `tname`, and returns 1.

```
int result = luaL_newmetatable(L, "MyMetatable"); // create a new metatable and store it in the registry
if (result == 1)
  printf("MyMetatable was added to the registry\n");

lua_newtable(L);                                  // create a normal table and add some data
lua_pushstring(L, "some value");
lua_setfield(L, -2, "some key");

luaL_setmetatable(L, "MyMetatable");              // set the metatable for the table at -1
lua_setglobal(L, "MyTable");                      // create a global "MyTable" for the table at -1
```

Binding C++ Classes to Lua

- General Approach
  - C++ objects are represented by Lua tables
  - The member functions are stored in *metatables*
- Reference Types
  - Persistent instances on the heap (e.g. player, enemies, components, ...)
  - The pointer address is stored in a hidden field in the table
- Value Types
  - Temporary instances on the C/C++ stack (e.g. vectors, matrices, ...)
  - The entire data must be stored in the table
    - Either by using the Lua type *userdata* (memory block)
      - Advantage: better performance
    - Or by storing every member variable separately
      - Advantage: The members can be accessed from Lua

Reference Types

```
struct Enemy
{
	void sayHi() const { printf("Enemy 0x%X says hi!\n", this); }
	static void lua_bind(lua_State* L)
	{
		luaL_newmetatable(L, "EnemyMeta");          // push a new metatable on the stack
		lua_pushvalue(L, -1);                       // push the metatable on the stack (a reference)
		lua_setfield(L, -2, "__index");             // add the field "__index" to the metatable
		lua_pushcfunction(L, lua_CFunction_sayHi); // push the "sayHi" function on the stack
		lua_setfield(L, -2, "sayHi");               // add a field "sayHi" to store the function
		lua_pushcfunction(L, lua_CFunction_Enemy); // push the "Enemy" function on the stack
		lua_setglobal(L, "Enemy");                  // assign the function to the global "Enemy"
	}
	static int lua_CFunction_Enemy(lua_State* L)
	{
		lua_newtable(L);                            // push a new table on the stack
		lua_pushlightuserdata(L, new Enemy());      // push a new Enemy pointer to the stack (leaks!!!)
		lua_setfield(L, -2, "__ptr");               // add a hidden field "__ptr" to store the pointer
		luaL_setmetatable(L, "EnemyMeta");          // set the proper metatable
		return 1;
	}
	static int lua_CFunction_sayHi(lua_State* L)
	{
		lua_getfield(L, 1, "__ptr");                // read the hidden field "__ptr"
		Enemy* p = (Enemy*)lua_topointer(L, -1);    // get the pointer from the stack
		p->sayHi();                                 // call the member function
		return 0;
	}
};

	// inside main()...
	Enemy::lua_bind(L);
```

```
e1 = Enemy()
e1:sayHi()
e2 = Enemy()
e2:sayHi()
```

```
Output:

Enemy 0x75B6D0 says hi!
Enemy 0x75B700 says hi!
```

Value Types (1/2): *userdata*

```
struct Vec2 {
  float m_x, m_y;     // public member variables
  float length() const { return sqrt(m_x*m_x + m_y*m_y); }
  static void lua_bind(lua_State* L) {
    luaL_newmetatable(L, "Vec2Meta");           // push a new metatable on the stack
    lua_pushvalue(L, -1);                       // the metatable is at -1
    lua_setfield(L, -2, "__index");             // add __index to the metatable
    lua_pushcfunction(L, lua_CFunction_length); // push the "length" function on the stack
    lua_setfield(L, -2, "length");              // add a field "length" to store the function
    lua_pushcfunction(L, lua_CFunction_Vec2);   // push the "constructor" function on the stack
    lua_setglobal(L, "Vec2");                   // assign the function to the global "Vec2"
  }
  static int lua_CFunction_Vec2(lua_State* L) {
    float x = (float)lua_tonumber(L, 1);        // get the constructor parameters from the stack
    float y = (float)lua_tonumber(L, 2);        // TODO should also work for tables {x, y}
    Vec2 v = {x, y};                            // create an instance of Vec2 on the C/C++ stack
    lua_newtable(L);                            // push a new table on the stack
    void* ud = lua_newuserdata(L, sizeof(Vec2));// creates a new buffer and pushes it on the stack
    memcpy(ud, &v, sizeof(Vec2));               // copy the vectors data into the buffer
    lua_setfield(L, -2, "__data");              // add a hidden field “__data” to store the data
    luaL_setmetatable(L, "Vec2Meta");           // set the proper metatable
    return 1;
  }
  static int lua_CFunction_length(lua_State* L) {
    lua_getfield(L, 1, "__data");          // read the hidden field “__data”
    void* ud = lua_touserdata(L, -1);      // get the buffer from the stack
    Vec2 v;                                // create an instance of Vec2 on the C/C++ stack
    memcpy(&v, ud, sizeof(Vec2));          // copy the buffer data over the vector instance
    lua_pushnumber(L, v.length());         // call the member function and push the result on the stack
    return 1;
  }
};
```

```
vec = Vec2(2, 4); print(vec:length())
vec = Vec2(4, 8); print(vec:length())
print(vec.x, vec.y)
```

```
Output:    4.4721360206604
           8.9442720413208
           nil     nil
```

Value Types (2/2): member variables

```
struct Vec2 {

  // ... same as on previous slide ...

  static int lua_CFunction_Vec2(lua_State* L) {
    float x = (float)lua_tonumber(L, 1);   // get the constructor parameters from the stack
    float y = (float)lua_tonumber(L, 2);   // TODO should also work for tables {x, y}
    Vec2 v = {x, y};                       // create an instance of Vec2 on the C/C++ stack
    lua_newtable(L);                       // push a new table on the stack
    lua_pushnumber(L, x);                  // push the value of x on the stack
    lua_setfield(L, -2, "x");              // store the value of x in the field "x"
    lua_pushnumber(L, y);                  // push the value of y on the stack
    lua_setfield(L, -2, "y");              // store the value of y in the field "y"
    luaL_setmetatable(L, "Vec2Meta");      // set the proper metatable
    return 1;
  }
  static int lua_CFunction_length(lua_State* L) {
    lua_getfield(L, 1, "x");               // read the field "x"
    float x = (float)lua_tonumber(L, -1);  // get the value of "x" from the stack
    lua_pop(L, 1);                         // pop the value of "x" from the stack
    lua_getfield(L, 1, "y");               // read the field "y"
    float y = (float)lua_tonumber(L, -1);  // get the value of "y" from the stack
    lua_pop(L, 1);                         // pop the value of "y" from the stack
    Vec2 v = {x, y};                       // create an instance of Vec2 on the C/C++ stack
    lua_pushnumber(L, v.length());         // call the member function and push the result on the stack
    return 1;
  }
};
```

```
vec = Vec2(2, 4); print(vec:length())
vec = Vec2(4, 8); print(vec:length())
print(vec.x, vec.y)
```

```
Output:  4.4721360206604
         8.9442720413208
         4      8
```

References

- http://www.lua.org
- http://lua-users.org/wiki/MetatableEvents
- http://www.lua.org/pil/13.4.2.html
- http://www.lua.org/pil/14.html