Kurs języka Lua

08 – C API i dziedziczenie, cd.

Jakub Kowalski

Instytut Informatyki,
Uniwersytet Wrocławski

2017

Plan na dzisiaj

Opowiemy sobie dokładniej o

1. C API: tablicach i funkcjach
2. Dziedziczeniu

Uwaga

Za tydzień nie ma wykładu ani pracowni
(deadline na kolejną listę jest 08-09.05)

- Następna lista będzie ostatnią/przedostatnią
- Czas zacząć myśleć o projektach

C API

W ramach przypomnienia

Potrafimy

- Otwierać i zamykać wirtualne maszyny Lua
- Wstawiać na stos wartości globalne w Lua (podstawowych typów)
- Manipulować elementami na stosie
- Sprawdzać zawartość stosu i pobierać elementy stosu (podstawowych typów) jako wartości C

Stos

-1	`false`	4
-2	`'hello'`	3
-3	`4`	2
-4	`nil`	1
	$\perp$	

Wczytywanie tablic

Załóżmy, że chcemy trzymać kolor tła jako RBG

- W Lua jako float [0, 1]
- W C jako int [0, 255]

```
width  = 200
height = 300
background_red   = 0.30
background_green = 0.10
background_blue  = 0
```

Oczywiście nie chcielibyśmy robić tego tak jak wyżej:

- rozwlekłe, nienaturalne, brak możliwości predefiniowania kolorów

```
background = {red=0.30, green=0.10, blue=0}
BLUE = {red=0, green=0, blue=1.0}
background2 = BLUE
```

Wczytywanie tablic

```
lua_getglobal(L, "background");
if (!lua_istable(L, -1))
  error(L, "'background' is not a table");

red = getcolorfield(L, "red");
green = getcolorfield(L, "green");
blue = getcolorfield(L, "blue");
```

- getcolorfield napiszemy sobie sami bazując na podstawowych funkcjach C API

Wczytywanie tablic

```
#define MAX_COLOR 255

int getcolorfield(lua_State *L, const char *key)
{
  int result, isnum;
  lua_pushstring(L, key);
  lua_gettable(L, -2);
  result = (int)
    (lua_tonumberx(L, -1, &isnum) * MAX_COLOR);
  if (!isnum)
    error(L, "invalid color component '%s'", key);
  lua_pop(L, 1);
  return result;
}
```

Wczytywanie tablic

```
#define MAX_COLOR 255

int getcolorfield(lua_State *L, const char *key)
{
  int result, isnum;
  lua_getfield(L, -1, key);
  // if (lua_getfield(L, -1, key) != LUA_TNUMBER) error...
  result = (int)
    (lua_tonumberx(L, -1, &isnum) * MAX_COLOR);
  if (!isnum)
    error(L, "invalid color component '%s'", key);
  lua_pop(L, 1);
  return result;
}
```

Wczytywanie tablic

1. W funkcji zakładamy, że tablica znajduje się na szczycie stosu
2. Kładziemy klucz na stos (spychając tablicę w dół na indeks -2)
3. Rzutujemy wartość na liczbę i sprawdzamy poprawność konwersji
4. Zabieramy wartość ze stosu, zostawiając na szczycie tablicę

```
int lua_gettable (lua_State *L, int index);
```

- Jeśli na szczycie stosu znajduje się klucz `k`, a na zadanym indeksie tablica `t`, funckja zabiera klucz i kładzie na stos wartość `t[k]`
- Działa dla wszystkich typów wartości i kluczy (wczytując dane musimy dbać o prawidłowe konwersje), zwraca typ wartości

```
int lua_getfield (lua_State *L, int index,
                  const char *k);
```

- Wstawia na stos wartość `t[k]` dla tablicy `t` znajdującej się pod zadanym indeksem

Zapisywanie tablic

```
struct ColorTable {
  char *name;
  unsigned char red, green, blue;
} colortable[] = {
  {"WHITE", MAX_COLOR, MAX_COLOR, MAX_COLOR},
  {"RED",   MAX_COLOR, 0, 0},
  {"GREEN", 0, MAX_COLOR, 0},
  {"BLUE",  0, 0, MAX_COLOR},
  ...
  {NULL, 0, 0, 0}
};
```

```
WHITE = {red=1.0, green=1.0, blue=1.0}
RED   = {red=1.0, green=0.0, blue=0.0}
...
```

Zapisywanie tablic

- Chcemy zdefiniować jako wartości globalne pulę predefiniowanych kolorów, z których użytkownik będzie mógł korzystać w swoim skrypcie

```
void setcolorfield (lua_State *L,
                    const char *index, int value)
{
  lua_pushstring(L, index);
  lua_pushnumber(L, (double)value/MAX_COLOR);
  lua_settable(L, -3);
}
```

- Zakładamy, że tablica jest na szczycie stosu
- Wstawiamy do tablicy wartość o zadanym kluczu

Zapisywanie tablic

- Chcemy zdefiniować jako wartości globalne pulę predefiniowanych kolorów, z których użytkownik będzie mógł korzystać w swoim skrypcie

```
void setcolorfield (lua_State *L,
                    const char *index, int value)
{

  lua_pushnumber(L, (double)value/MAX_COLOR);
  lua_setfield(L, -2, index);
}
```

- Zakładamy, że tablica jest na szczycie stosu
- Wstawiamy do tablicy wartość o zadanym kluczu

Zapisywanie tablic

```
void lua_settable (lua_State *L, int index);
```

- Działa jak `t[k]=v` dla zadanej indeksem tablicy `t`, wartości `v` będącej na szczycie stosu (-1), i klucza `k` będącego bezpośrednio pod nim (-2)

```
void lua_setfield (lua_State *L, int index,
                   const char *k);
```

- Specjalna wersja `lua_settable`, klucz `k` będący napisem podajemy jako parametr

```
void lua_seti (lua_State *L, int index,
               lua_Integer n);
```

- Specjalna wersja `lua_settable`, klucz `n` będący liczbą podajemy jako parametr

Zapisywanie tablic

```
void setcolor (lua_State *L,struct ColorTable *ct)
{
  lua_newtable(L);     // creates a table
  setcolorfield(L, "red", ct->red);
  setcolorfield(L, "green", ct->green);
  setcolorfield(L, "blue", ct->blue);
  lua_setglobal(L, ct->name); // 'name' = table
}
```

- Definiujemy kolor umieszczając jako zmienną globalną Lua odpowiednio wypełnioną tablicę

- Pozostało zapisać wszystkie zdefiniowane kolory (przed uruchomieniem skryptu!)

```
int i = 0;
while (colortable[i].name != NULL)
  setcolor(L, &colortable[i++]);
```

Zapisywanie tablic

```
void setcolor (lua_State *L,struct ColorTable *ct)
{
  lua_createtable(L, 0, 3);     // creates a table
  setcolorfield(L, "red", ct->red);
  setcolorfield(L, "green", ct->green);
  setcolorfield(L, "blue", ct->blue);
  lua_setglobal(L, ct->name); // 'name' = table
}
```

- Definiujemy kolor umieszczając jako zmienną globalną Lua odpowiednio wypełnioną tablicę

- Pozostało zapisać wszystkie zdefiniowane kolory (przed uruchomieniem skryptu!)

```
int i = 0;
while (colortable[i].name != NULL)
  setcolor(L, &colortable[i++]);
```

Zapisywanie tablic

```
void lua_createtable (lua_State *L,
                      int narr, int nrec);
```

- Tworzy nową tablicę i wstawia na stos
- `narr` sugeruje wielkość części sekwencyjnej tablicy, `nrec` słownikowej

```
#define lua_newtable(L)  lua_createtable(L, 0, 0)
```

- Tworzymy tablicę bez prealokacji miejsca

Odczyt predefiniowanych kolorów

- A co gdyby (opcjonalnie) zastąpić nazwy kolorów napisami?
- Nie śmiecimy po przestrzeni globalnej
- W C byłoby to rozwiązanie dziwne, bo błędy nie byłyby wykryte w czas kompilacji
- W Lua ponieważ nie ma jasnego rozróżnienia między użytkownikiem a programistą, nie ma go również między błędami kompilacji i uruchomienia
- Więc dopóki autor skryptu ma dostęp do błędów to wszystko jest OK

```
background = {red=1.0, green=1.0, blue=1.0}
background = "WHITE"
background = 'White' --to też dałoby się obsłużyć
```

example01.cpp+example01.lua

Wywoływanie funkcji

- W naszym „pliku konfiguracyjnym” możemy również definiować funkcje, które mogą być uruchamiane z poziomu aplikacji

```
function f (x, y)
  return (x^2 * math.sin(y))/(1 - x)
end
```

example02.cpp+example02.lua

Wywoływanie funkcji

```
void lua_call (lua_State *L, int nargs, int nres);
int lua_pcall (lua_State *L, int nargs, int nres,
               int msgh);
```

- Wywołuje funkcję leżącą na stosie pod swoimi argumentami
- `nargs` definiuje liczbę argumentów
- `nres` definiuje liczbę zwracanych rezultatów
- Wywołanie usuwa ze stosu funkcję i wszystkie argumenty

`lua_pcall(L, 3, 2, 0)`

-1	arg3	N+3
-2	arg2	N+2
-3	arg1	N+1
-4	f	N
	...	

-1	res2	N+1
-2	res1	N
	...	

Wywoływanie funkcji

```
double f (lua_State *L, double x, double y)
{
  int isnum;  double z;
  lua_getglobal(L, "f"); // funkcja
  lua_pushnumber(L, x);  // pierwszy argument
  lua_pushnumber(L, y);  // drugi argument

  if (lua_pcall(L, 2, 1, 0) != LUA_OK)
    error(L, "error running function 'f'"<...>);

  z = lua_tonumberx(L, -1, &isnum);
  if (!isnum)
    error(L, "function 'f' must return a number");
  lua_pop(L, 1);
  return z;
}
```

Wywoływanie funkcji

Dynamiczna liczba zwracanych rezultatów

- `nres` może przyjąć wartość `LUA_MULTRET`
- Wywołanie funkcji włoży wtedy na stos wszystkie zwrócone rezultaty (normalnie ich liczba jest przycięta do `nres`)

message handler

- Jeśli `msgh` to 0, na stos kładziony jest obiekt błędu
- W przeciwnym przypadku `msgh` powinien wskazywać indeks na którym znajduje się funkcja obsługująca błąd – zostanie ona wywołana z obiektem błędu i na stosie wyląduje jej wynik

Kody błędów zwracane przez `lua_pcall`

- `LUA_OK (0)` – sukces
- `LUA_ERRRUN` – runtime error
- `LUA_ERRMEM` – błąd alokacji pamięci
- `LUA_ERRERR` – błąd message handlera
- `LUA_ERRGCMM` – błąd metametody `__gc`

Generyczne wywoływanie funkcji

- A gdyby tak opakować wywołania funkcji Lua w generyczny kod i móc pisać np.:

```
call_va(L, "f", "dd>d", 0.8, 2.0, &z);
```

example03.cpp+example03.lua

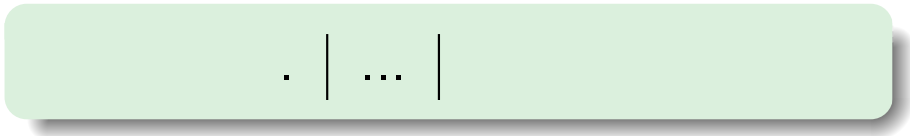
Istotne drobiazgi

- Ponieważ nie znamy liczby argumentów musimy zadbać o odpowiednią ilość miejsca na stosie
- Nie musimy ręcznie sprawdzać czy funkcja jest funkcją (zajmuje się tym `lua_pcall`)
- Nie możemy od razu zrzucać ze stosu rezultatów, bo mogą być napisami (chyba że dodalibyśmy ich kopiowanie)

Wszystko razem

```
a = f("how", t.x, 14)
```

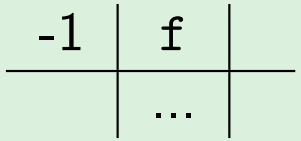
```
lua_getglobal(L, "f");
```



Wszystko razem

```
a = f("how", t.x, 14)
```

```
lua_getglobal(L, "f");
lua_pushliteral(L, "how");
```



Wszystko razem

```
a = f("how", t.x, 14)
```

```
lua_getglobal(L, "f");
lua_pushliteral(L, "how");
lua_getglobal(L, "t");
```

-1	"how"	
-2	f	
	...	

Wszystko razem

```
a = f("how", t.x, 14)
```

```
lua_getglobal(L, "f");
lua_pushliteral(L, "how");
lua_getglobal(L, "t");
lua_getfield(L, -1, "x");
```

-1	t	
-2	"how"	
-3	f	
	...	

Wszystko razem

```
a = f("how", t.x, 14)
```

```
lua_getglobal(L, "f");
lua_pushliteral(L, "how");
lua_getglobal(L, "t");
lua_getfield(L, -1, "x");
lua_remove(L, -2);
```

-1	t.x	
-2	t	
-3	"how"	
-4	f	
	...	

Wszystko razem

```
a = f("how", t.x, 14)
```

```
lua_getglobal(L, "f");
lua_pushliteral(L, "how");
lua_getglobal(L, "t");
lua_getfield(L, -1, "x");
lua_remove(L, -2);
lua_pushinteger(L, 14);
```

-1	t.x	
-2	"how"	
-3	f	
	...	

Wszystko razem

```
a = f("how", t.x, 14)
```

```
lua_getglobal(L, "f");
lua_pushliteral(L, "how");
lua_getglobal(L, "t");
lua_getfield(L, -1, "x");
lua_remove(L, -2);
lua_pushinteger(L, 14);
lua_call(L, 3, 1);
```

-1	14	
-2	t.x	
-3	"how"	
-4	f	
	...	

Wszystko razem

```
a = f("how", t.x, 14)
```

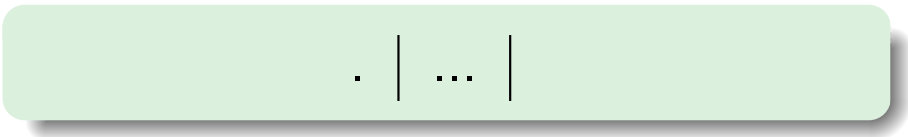
```
lua_getglobal(L, "f");
lua_pushliteral(L, "how");
lua_getglobal(L, "t");
lua_getfield(L, -1, "x");
lua_remove(L, -2);
lua_pushinteger(L, 14);
lua_call(L, 3, 1);
lua_setglobal(L, "a");
```

-1	?	
	...	

Wszystko razem

```
a = f("how", t.x, 14)
```

```
lua_getglobal(L, "f");
lua_pushliteral(L, "how");
lua_getglobal(L, "t");
lua_getfield(L, -1, "x");
lua_remove(L, -2);
lua_pushinteger(L, 14);
lua_call(L, 3, 1);
lua_setglobal(L, "a");
```



Dziedziczenie

Dziedziczenie

- Zdefiniujmy naszą klasę bazową Account

```
Account = {balance = 0}
function Account:new (o)
  o = o or {}
  self.__index = self
  setmetatable(o, self)
  return o
end
function Account:deposit (v)
  self.balance = self.balance + v
end
function Account:withdraw (v)
  if v > self.balance then
    error "insufficient funds" end
  self.balance = self.balance - v
end
```

Dziedziczenie

- Chcemy stworzyć podklasę SpecialAccount

```
SpecialAccount = Account:new()
```

Magia

```
s=SpecialAccount:new{limit=1000.00}
<--> s=Account.new(SpecialAccount, {limit...})
-- getmetatable(s) = SpecialAccount
-- s.__index = SpecialAccount

s:deposit(100)
--> s.deposit(s, 100) -- nil
--> SpecialAccount.deposit(s, 100) -- nil
--> Account.deposit(s, 100)  -- OK!
```

Dziedziczenie

Możemy nadpisywać metody i dodawać nowe

```
function SpecialAccount:withdraw (v)
  if v - self.balance >= self:getLimit() then
    error"insufficient funds"
  end
  self.balance = self.balance - v
end
function SpecialAccount:getLimit ()
  return self.limit or 0
end
```

```
s:withdraw(200.00)
-- s.withdraw(s, 200.00) -- nil
-- SpecialAccount.withdraw(s, 200.00) -- OK!
print(s.balance) --> -200
```

Dziedziczenie

- Czasami żeby wyspecyfikować nowe zachowanie nie potrzebujemy definiować całej klasy
- Jeśli modyfikacja dotyczy pojedynczych obiektów, to możemy nadpisać te zachowania bezpośrednio w obiekcie

```
function s:getLimit ()
  return self.balance * 0.10
end
```

```
s:getLimit()
-- s.getLimit(s) -- OK!
```

Interfejsy

- Dzięki dynamicznemu typowaniu i nierestrykcyjnemu modelowi obiektowemu interfejsy mamy darmowe i intuicyjne

```
ship = {row=y, col=x, speed=arg2, rum=arg3}
barrel = {row=y, col=x, rum=arg1}
mine = {row=y, col=x}

function HexLib.offset_to_cube(entity)
  local x = entity.row -
            (entity.col - (entity.col & 1)) / 2
  local z = entity.col
  local y = -x - z
  <...>
end

function compare_rum(e1, e2)
  return e1.rum < e2.rum
end
```

Wielokrotne dziedziczenie

- Istnieje wiele możliwości programowania w Lua w sposób obiektowy
- Ten który omówiliśmy generalnie cechuje się prostotą, wydajnoscia i elastycznością

Co gdybyśmy chcieli dopuścić dziedziczenie wielokrotne

- Idea – nadpisać metametodę `__index` tak, aby przeszukiwała wszystkie nadklasy w poszukiwaniu brakującego pola/funkcji
- Ponieważ teraz klasa nie posiada jednej nadklasy, nie możemy po prostu skorzystać z jej konstruktora do stworzenia klasy dziedziczącej
- Zamiast tego zdefiniujemy niezależną funkcję tworzącą nową klasę mając listę jej nadklas.

```
local function search (k, plist)
  for i=1, #plist do
    local v = plist[i][k]
    if v then return v end
  end
end
```

Wielokrotne dziedziczenie

```
function createClass (...)
  local c = {}           -- nowa klasa
  local parents = {...} -- lista rodziców
  setmetatable(c, {__index = function (t, k)
                    return search(k, parents)
                  end})
  -- c ma być metatabelą swoich instancji
  c.__index = c

  -- definiujemy konstruktor klasy
  function c:new (o)
    o = o or {}
    setmetatable(o, c)
    return o
  end
  return c -- zwracamy nową klasę
end
```

Wielokrotne dziedziczenie

```
Named = {}
function Named:getname ()
  return self.name
end
function Named:setname (n)
  self.name = n
end
```

```
NamedAccount = createClass(Account, Named)
account = NamedAccount:new{name = "Paul"}
print(account:getname())   --> Paul
-- account['getname'] --> NamedAccount['getname']
--> search('getname', {Account, Named})
  --> Account['getname']   --> nil
  --> Named['getname']     --> OK!
```

Wielokrotne dziedziczenie

- Wydajność takiego podejścia jest oczywiście istotnie gorsza
- Możemy to poprawić poprzez kopiowanie dziedziczonych metod

```
setmetatable(c, {__index = function (t, k)
                    local v = search(k, parents)
                    t[k] = v
                    return v
                  end})
```

- W ten sposób wszystkie zapytania poza pierwszym będą szybsze
- Tracimy jednak możliwość zmiany definicji w locie
- Wprowadzone modyfikacje nie są już propagowane zgodnie z hierarchią klas

Dziękuję za uwagę

Za **2 tygodnie**:
obiektowość cd.,
uruchamianie C z Lua,
...?