Howling at the Moon:

Lua for C++ Programmers

Andreas Weis

BMW AG

CppCon, September 29, 2017

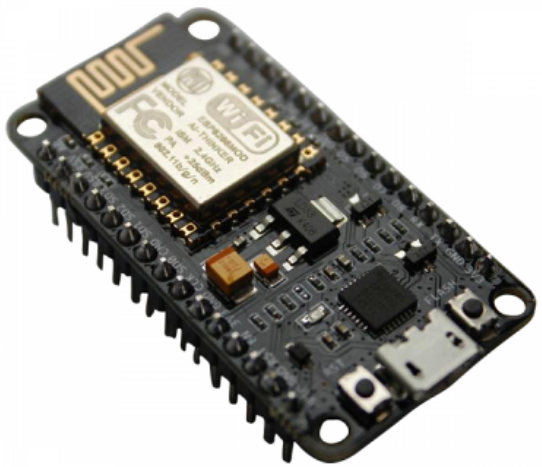
About me

-    Known as ComicSansMS on most sites
-  @DerGhulbus on Twitter
-  Co-organizer of the Munich C++ User Group
- Currently working as a Software Architect for BMW

  BMW GROUP  


Lua

Lua in the wild

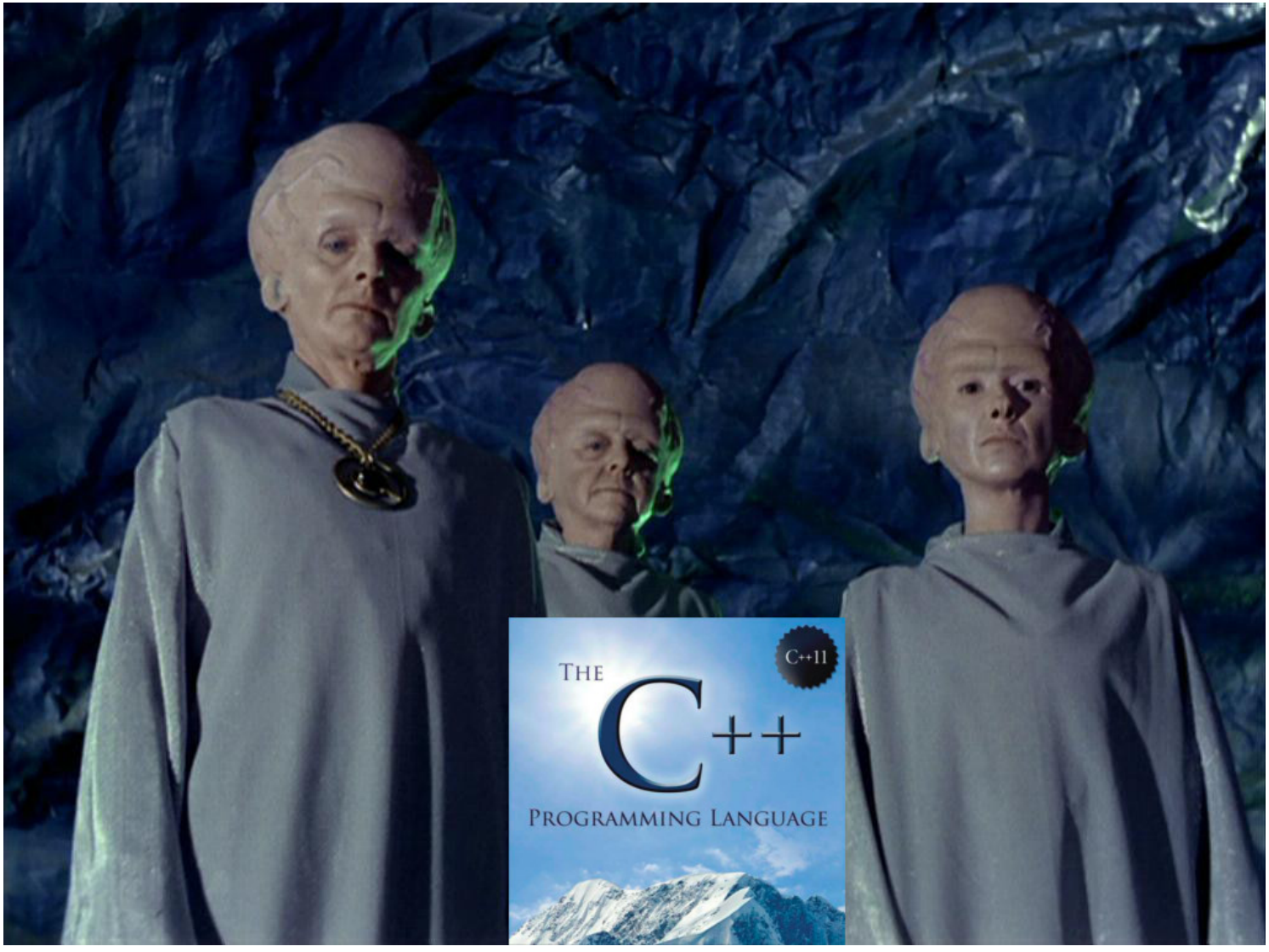


Lua is small

```
chunk ::= block

block ::= {stat} [retstat]

stat ::=  ';' |
        varlist '=' explist |
        functioncall |
        label |
        break |
        goto Name |
        do block end |
        while exp do block end |
        repeat block until exp |
        if exp then block {elseif exp then block} [else block] end |
        for Name '=' exp ',' exp [',' exp] do block end |
        for namelist in explist do block end |
        function funcname funcbody |
        local function Name funcbody |
        local namelist ['=' explist]

retstat ::= return [explist] [';']

label ::= '::' Name '::'

funcname ::= Name {'.' Name} [':' Name]

varlist ::= var {',' var}

var ::=  Name | prefixexp '[' exp ']' | prefixexp '.' Name

namelist ::= Name {',' Name}

explist ::= exp {',' exp}

exp ::=  nil | false | true | Numeral | LiteralString | '...' | functiondef |
        prefixexp | tableconstructor | exp binop exp | unop exp

prefixexp ::= var | functioncall | '(' exp ')'

functioncall ::=  prefixexp args | prefixexp ':' Name args

args ::=  '(' [explist] ')' | tableconstructor | LiteralString

functiondef ::= function funcbody

funcbody ::= '(' [parlist] ')' block end

parlist ::= namelist [',' '...'] | '...'

tableconstructor ::= '{' [fieldlist] '}'

fieldlist ::= field {fieldsep field} [fieldsep]

field ::= '[' exp ']' '=' exp | Name '=' exp | exp

fieldsep ::= ',' | ';'

binop ::=  '+' | '-' | '*' | '/' | '//' | '^' | '%' |
        '&' | '~' | '|' | '>>' | '<<' | '..' |
        '<' | '<=' | '>' | '>=' | '==' | '~=' |
        and | or

unop ::= '-' | not | '#' | '~'
```

- Compiled binary is $<$ 180KB
- Reference Manual 82 A4 pages
- 8 basic data types
- Batteries not included!

The whole language fits into your head



Disclaimer

This talk is trying too hard to be clever.
Keep simple things simple.

Hello World!

```
print("Hello World!");
```

All functions are lambdas

```
function f(a1, a2, a3)
  -- [...]
end
```

is just syntax sugar for

```
f = function(a1, a2, a3)
  -- [...]
end
```

Functions are true first-class values in Lua.

Replacing functions is trivial

```
print("Vanilla print");

print = function(...)
      -- my_print implementation
      -- [...]
   end;
print("My print");
```

Function Hooking – Counting print calls

```
count = 0;
old_print = print;
print = function(...)
    count = count + 1;
    old_print(...);
  end;
```

Capturing state with function closures

Instead of explicit Lambda captures, Lua has full lexical scoping.

```
function enable_counting()
  local count = 0;
  local old_print = print;
  print = function(...)
      count = count + 1;
      old_print(...);
    end;

  return function() return count; end;
end
```

Read `function` as closure construction.

Tables

The only complex data structure in the language.

```
local t = {};

local array = { 5, 4, 3, 2, 1 };
assert(array[2] == 4);    -- indices are 1-based

local dict = { the_answer = 42 };
assert(dict["the_answer"] == 42);
```

Tables can use any type of values as keys or values.

```
dict[print] = "function as key";
```

Tables (contd.)

Tables have reference semantics:

```
local list = { value = "foo", next = nil };
list.next = { value = "bar", next = nil  };
```

Read {} as table construction.

Records

```
local complex = { real = 42.0, imag = 0.0 };
complex["real"] = 42.0;
complex.real = 42.0;

function fconjugate(c)
  c.imag = 0.0 - c.imag;
end

complex.conjugate = fconjugate;
complex.conjugate(complex);

complex:conjugate();
```

Object Construction

```
function build_complex(r, i)
  return { real = r, imag = i };
end

local c1 = build_complex(1, 0);
local c2 = build_complex(0, 1);

local sum = c1 + c2; -- ???
```

Metatables - Tables describing object properties

```
local mt = {};
mt.__add = function(c1, c2)
    return build_complex(c1.real + c2.real,
                         c1.imag + c2.imag);
  end;

function build_complex(r,i)
  local ret = {real = r, imag = i};
  setmetatable(ret, mt);
  return ret;
end
```

Encapsulation

```
function build_date(y, m, d)
  assert(validDate(y, m, d))
  local lself = { y=y, m=m, d=d }

  local lget_day = function() return lself.d end
  local lset_day = function(nd)
      assert(validDate(lself.y, lself.m, nd))
      lself.d = nd
    end

  return {
    set_day = lset_day,
    get_day = lget_day
  }
```

Reflection

All data structures are tables. Inspecting the fields of the table reveals everything we need to know about the type.

```
function is_complex(c)
  return type(c) == "table" and
         c.real and c.imag;
end

local tuple = {};
for k,v in pairs(t) do
  tuple[#tuple + 1] = v;
end
```

The environment

But what about global variables?

```
for k in pairs(_G) do
  print(k);
end
```

Constraining the environment

```
local foobar;
-- [...]
fobar = do_stuff();
```

Remember: `_G` is just a table.

```
setmetatable(_G,
  { __newindex = function(_, name)
      print(name .. " was not declared!");
    end
  });
```

Integration with C++

It's embedded — `main()` belongs to the enclosing program.

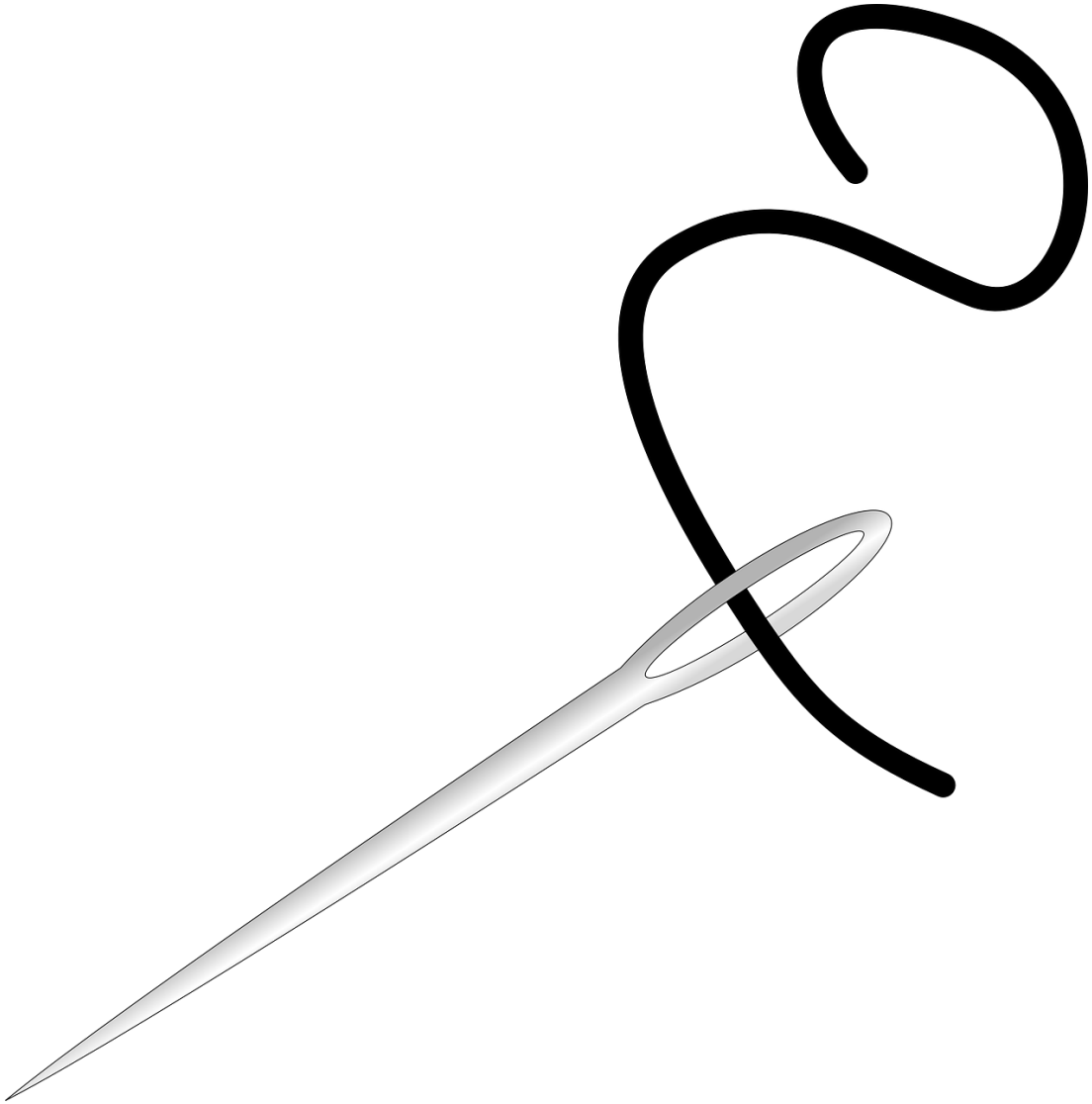
```
int main() {

  lua_State* l = luaL_newstate();

  luaL_dostring(l, R"(
      print("Hello World!");
    )" );

  lua_close(l);
}
```

Lua API is prefixed `lua_`
Auxiliary library is prefixed `luaL_`

Exposing C functions to Lua

```
int my_function(lua_State* l) {
    // [...]
}
```

The Stack - The needle's eye



Pushing values on the stack

```
void push(lua_State* l, lua_Number n) {
  lua_pushnumber(l, n);
}

void push(lua_State* l, char const* s) {
  lua_pushstring(l, str);
}

template<typename... Ts>
void pushargs(lua_State* l, Ts... args) {
    ( push(l, args), ... );
}
```

Pushing values on the stack (2)

```
template<typename... Ts>
void pushargs(lua_State* l, Ts... args) {
    auto t = boost::hana::make_tuple(args...);
    boost::hana::for_each(t,
      [l](auto&& v) { push(l, v); });
}
```

Pushing values on the stack (3)

```
template<typename... Ts>
void pushargs(lua_State* l, Ts... args) {
  auto t = std::make_tuple(args...);
  push_helper(l, t,
    std::make_index_sequence<sizeof...(Ts)>());
}

template<typename... Ts, std::size_t... Is>
void push_helper(lua_State* l,
                 std::tuple<Ts...> const& t,
                 std::index_sequence<Is...>) {
    using expander = int [];
    (void) expander { 0,
      (push(l, std::get<Is>(t)), 0)... };
}
```

Getting values from the stack

```
??? getValueFromStack(lua_State* l, int idx)
{
  switch(lua_type(l, idx)) {
    case LUA_TNUMBER:
      return Number(lua_tonumber(l, idx));
    case LUA_TSTRING:
      return String(lua_tostring(l, idx));
    // [...]
  }
}
```

Representing values

```
enum class Type;

class Number { Type type() const; };
class String { Type type() const; };
// [...]

using Value =
  std::variant<Number, String /*, [...] */>;

Type getType(Value const& v) {
  return std::visit(
    [](auto x) { return x.type(); },
    v);
}
```

Calling functions

```
template<typename... Ts>
std::vector<Value> call(
    lua_State* l,
    char const* func,
    Ts... args)
{
  lua_getglobal(l, func);
  pushargs(l, args...);
  lua_call(l, sizeof...(Ts),
           LUA_MULTRET, 0);
  return getValuesFromStack(l);
}

call(l, "print", 42, "Hello World");
  // -> 42.0   Hello World
```

Constraining functions

```
template<typename... Ts>
std::vector<Value> call(lua_State*,
  char const*,
  Ts...);

std::array<Value, 2> call(lua_State*,
  char const*,
  Value, Value);

std::tuple<Number, String> call(lua_State*,
  char const*,
  Number, Number);
```

Wrapping up

Lua is a powerful, efficient, lightweight, embeddable scripting language.

Literature

- Lua 5.3 Reference Manual
- Programming in Lua (4th Ed.)
- The evolution of Lua (ACM HOPL 2007)
- Passing a language through the eye of a needle
- https://www.lua.org/docs.html

Thanks for your attention.