Lua API++ library:
Lua binding to C++11

Nikolay Zykov
2014

What is Lua API++

- Purpose: embed Lua into C++ application
  - create and expose functions and data to Lua
- Easy to use
  - automatic stack management
  - expressive OO syntax
  - natural expressions: calls, indexing, arithmetics and comparison
  - automatic function wrapping
- Lightweight, little overhead
  - no dynamic polymorphism
  - header-only mode available
- Requires C++11
- No external dependencies
- Compatible with Lua 5.1 (yes, LuaJIT too) and 5.2
- Open-source (MIT license)

Main players

State

- owns Lua state (create/destroy)
- setup the environment (execute files, strings, C functions)

LFunction

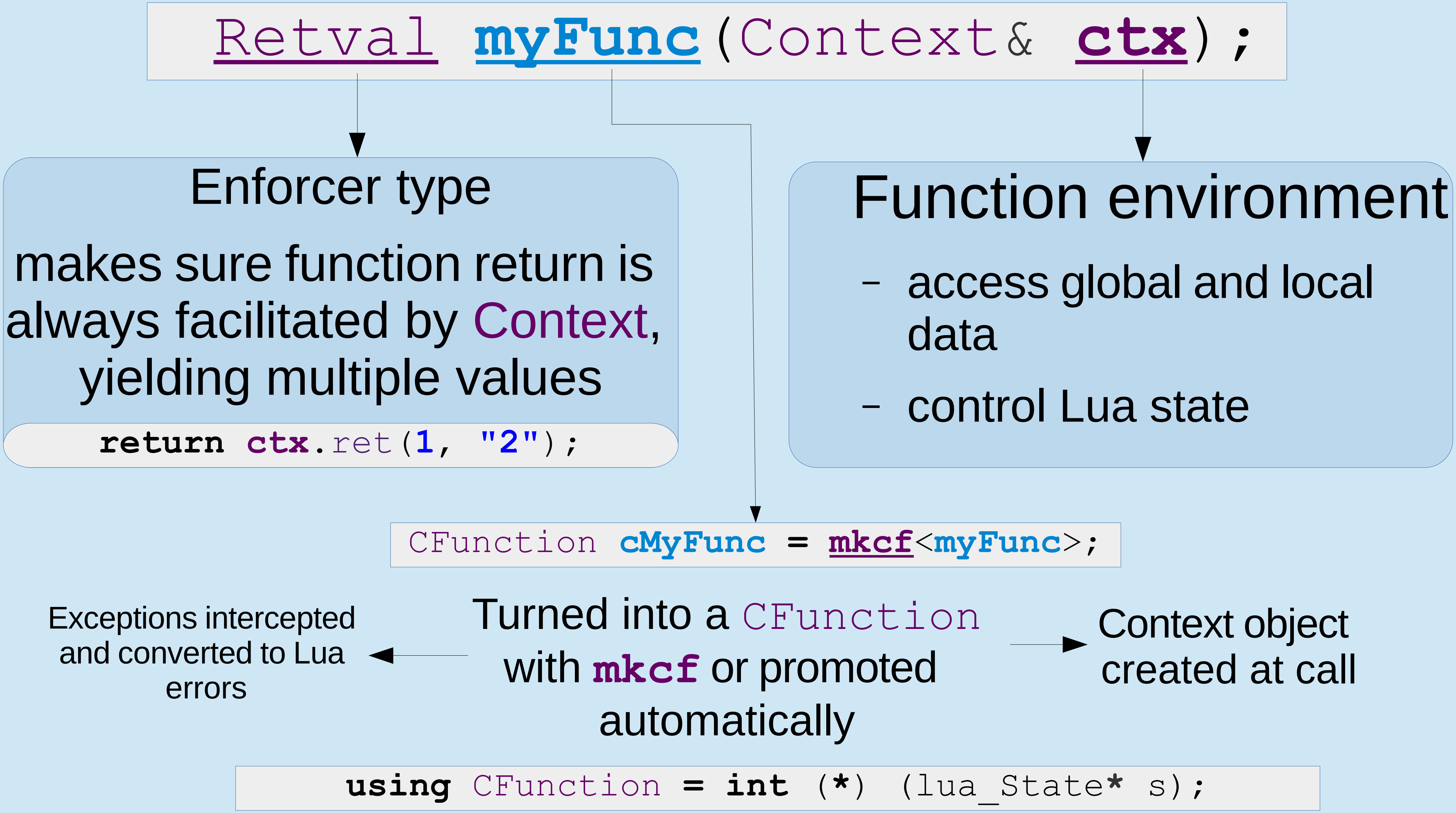
Context

- access the environment
  - the data: global, registry, arguments, upvalues
  - general control: GC and other
- control the flow
  - return
  - signal errors
- create values
  - wrap C functions

Values

- value interaction
  - conversion to native types
  - operations (calls, indexation, arithmetics, comparisons)
  - miscellaneous (type info, length, metatable)
- anchors
  - Value: all-purpose anchor
  - Table: specialized for table handling
  - Valset: dynamic STL-like value container
- maintain open borders
  - implicit data extraction
  - promotion of native values

LFunction anatomy



```
Retval myFunc(Context& ctx);
```

Enforcer type

makes sure function return is always facilitated by Context, yielding multiple values

```
return ctx.ret(1, "2");
```

Function environment

- access global and local data
- control Lua state

```
CFunction cMyFunc = mkcf<myFunc>;
```

Exceptions intercepted and converted to Lua errors

Turned into a CFunction with mkcf or promoted automatically

Context object created at call

```
using CFunction = int (*) (lua_State* s);
```

Value traversal

Conversion to native types

- implicit

```
double d = val;
const char* s = val;
```

- explicit

```
cout << val.cast<string>();
```

- failsafe

```
cout << val.optcast<string>("default");
```

- type check

```
if(val.is<double>())
```

- direct type query

```
if(val.type() == ValueTypes::Number)
```

Automatically promoted native types

- `Nil`
- `bool`
- `int, unsigned int`
- `long long, unsigned long long`
- `float`
- `double`
- `const char*`
- `std::string`
- `CFunction: int (*) (lua_State*)`
- `LightUserData: void*`
- registered userdata
- `LFunction`
- generic functions and member functions

Single value operations

calls

- natural form

```
fn();
int x = fn(3, "three");
```

- explicit

```
int x = fn.call(3, "three");
```

- protected

```
int x = fn.pcall(3, "three");
```

free nesting and chaining

```
fn(val[1](), val[2][1])["f"]();
```

indexing

- read

```
int x = val[1];
```

- write

```
val["one"] = nil;
```

metatable

```
Table mt = val.mt();
val.mt() = nil;
```

length

```
size_t L = val.len();
```

Two value operations

arithmetics*(Lua 5.2 only)*

```
-fn(3 * (x ^ 2) - 2 * (x ^ 3));
```

- supported operations:
  - `-` unary minus
  - `+  -  *  /`
  - `%` modulus
  - `^` power
- natural priorities, except power

concatenation

```
strings = val & "strval"& 4;
```

- chained concatenations are coalesced into a single operation

comparisons

- produce **`bool`**
- supported operations:

`==   !=   >   >=   <   <=`

References, temporaries and anchors

- Valref: reference to an occupied stack slot
  - purpose: non-owning reference
- Temporary: result of an operation
  - purpose: handle value creation, use and removal
  - mimics Valref
  - esoteric actual type
- Anchor: owns a stack slot
  - Value: just nail some value to the stack
  - Table: special case for tables
  - Valset: STL-like container

Multiple value return

Call expression: `operator (...), call(...), pcall(...)`

```
function mrv() return 2, 3, 4; end
```

unused

```
mrv();
// no effect
```

- 0 values expected
- everything discarded

single value context

```
Value x = mrv();
// x == 2
```

- extra values trimmed
- **nil** if empty

sequence context

```
print(1, mrv(), 5);
// out: 1 2 3 4 5
```

- expands in sequence
- any suitable context

capture

```
Valset vs = mrv();
// vs.size() == 3
```

- all values anchored
- pcall status recorded

Valset as

a single value

- not allowed -

Valset expansion

```
print(1, vs, 5);
// out: 1 2 3 4 5
```

- any suitable context
- values are copied

Table handling

array literal

```
return ctx.ret(
  Table::array(ctx,
     "one", "two", "three"
  )
);
```

recordset literal

```
x.mt() = Table::records(ctx,
  "__index", xRead,
  "__newindex", xWrite,
  "__gc", xDestroy
);
```

iteration

```
Table t = ctx.global["myTable"];
t.iterate([&] (Valref k, Valref v)
{
  cout << int(k) << int(v);
});
```

raw access

```
Table t = ctx.global["myTable"];
const int x = t.raw["keystring"];
t.raw["keystring"] = nil;
```

Context: accessors

global

- indexed with strings
- produces temporaries

registry

- indexed with strings for userdata metatable access
- `store(value)` creates integer keys
- indexed with integer keys for stored value access
- produces temporaries

args

- is a Valset
- **0**-based numeric indices
- size known
- produces Valref

upvalues

- **1**-based numeric indices
- size unknown
- producesValref

Context:
returning values and reporting errors

```
return ctx.ret("one", "two");
```

- must be called to create Retval required as LFunction return type
- allows arbitrary number and type of return values
- effectively stops automatic stack management
- must be used only with **`return`**
- expands Valset and call expressions
- special case of single Valset: does not copy its content

```
return ctx.error("Fail");
```

- never returns
- creates Retval, can be used with **`return`** for clarity
- error description is promoted

```
ctx.where();
```

- describes current execution point for error messages
- produces temporary, allows concatenation

```
return ctx.error();
```

- error message defaults to the result of `where()`

Context: function handling

closures and chunks

```
Context::closure(CFunction, ...);
Context::closure(LFunction, ...);
```

creates a closure with provided upvalues

```
Context::chunk(const char* text);
Context::chunk(const string& text);
```

creates a chunk from text

```
Context::load(const char* fileName);
Context::load(const string& fileName);
```

creates a chunk from file

execution

```
Context::runString(const char* text);
Context::runString(const string& text);
```

executes the text

```
Context::runFile(const char* fileName);
Context::runFile(const string& fileName);
```

executes the file

C function wrapping

```
Value f = ctx.wrap(funcName);
```

creates closure with automatic wrapper

```
Value f2 = ctx.vwrap(funcName);
```

same as `wrap`, but the result is discarded

`LUAPP_AUTOWRAP` enables automatic C function promotion

Userdata support

- set up
  - register type ID

```
LUAPP_USERDATA(MyType, "MyTypeID")
```

  - assign metatable

```
ctx.mt<MyType>() =
Table::records(ctx);
```

- create
  - registered userdata is promoted just like native types
  - requires copy or move constructor
  - the metatable for new value is extracted from the registry

- retrieve
  - registered userdata can be explicitly cast to a reference

```
MyType& val = x.cast<MyType>();
```

  - no implicit cast

- type check
  - registered userdata is recognized by `is` function

```
if(x.is<MyType>())
```

  - the exact type matching is done by comparing value's metatable against one stored in registry

Conclusion

The Lua API++ library is available at

`https://github.com/OldFisher/lua-api-pp`

Motivational example

Interpreter

```
#include <iostream>
#include <luapp/lua.hpp>
using namespace std; using namespace lua;

void interpretLine(State& s, const string& line)
{
  try { s.runString(line); }
  catch(exception& e) { cerr<< e.what()<< endl; }
}

void interpretStream(State& s, istream& in) {
  string currentLine;
  while(!in.eof()) {
    getline(in, currentLine);
    interpretLine(s, currentLine);
  }
}

int main(int argc, const char* argv[]) {
  State state;
  state.call(mkcf<setup>);
  interpretStream(state, cin);
}
```

Userdata support

```
#include <vector>
using dvec = std::vector<double>;
LUAPP_USERDATA(dvec, "Number array")

dvec aCreate(size_t size) { return dvec(size); }

void aDestroy(dvec& self) { self.~dvec(); }

void aWrite(dvec& self, size_t index, double val)
{ self.at(index) = val; }

Retval setup(Context& c) {
  c.mt<dvec>() = Table::records(c,
    "__index",
    static_cast<double& (dvec::*)(size_t)>
        (&dvec::at),
    "__newindex", aWrite,
    "__len",       dvec::size,
    "__gc",        aDestroy  );
  c.global["createArray"] = aCreate;
  return c.ret();
}
```