Compile Fast
Run Faster
Scale Forever

A Look into the sol Lua Binding Library

ThePhD

May 10th, 2018

Why “ThePhD”?

- It’s a std::promise<> for my std::future<>
  - Finishing undergrad in about a year
  - Debating industry vs. graduate school
- Actually stands for “The Phantom Derpstorm”
  - ’cause bad at video games 😅

Lua

- Small scripting language used in tons of places
  - Databases (e.g. Redis)
  - Operating System components
  - Tons of game projects/engines that are not Unreal
  - High Performance Computing projects
  - GUI Scripting (Waze/OpenMPT)
  - Chat servers, Server management

- And so on and so forth…

sol2

- Lua <-> C++ interop library
  - Started by Danny “Rapptz” Y. (M.D.) as just **sol**
- C++14 and better
  - sol3: Making a break for C++17/20 soon
- Written on top of Lua C API
  - Provides C API compatibility layers

Established

- sol is Mature, used in many industries and projects
- Has competed against all other libraries (20+) and more or less survived + thrived
  - Except in the case of compilation speed

The Interface

What exactly would make a good interface for Lua in C++?

Language Parity

- Lua has….
  - Tables (serves as arrays, maps, class-proxies, …)
  - Numbers (always doubles until Lua 5.3, which introduced integers up to 64 bits signed)
  - Functions (as first class citizens, closures are easy)
  - Strings (Lua literals are encoded as utf8 by default)

Let me show you…

What would C++ look like…?

```
double timing                  = lua[“timing”];
function func                  = lua[“func”];
bool result                    = func(1, 2);
std::tuple<int, int> result2   = lua[“callable”](4, 2); // multiple returns

lua[“signal”]       = true;
lua[“signals”]      = make_new_table();
lua[“signals”][1]  = [](int v) { std::cout << “beep with” << v << ‘\n’; };

lua.script(“if signal then signals[1](20) else print(‘boop’)”);
```

“Pinching Point”

The stack abstraction and why it matters

Stacks!

- Lua’s C API is stack-based
  - Annoying to manage, even when understood
- Defines all interop for types
  - Primitives (numbers (integers), strings, tables, functions) to complex entities
  - Custom types (userdata, lightuserdata)

Good to use for simple things…

- my_table[“a”]
  - get ‘my_table’ global – lua_getglobal(L, “my_table”)
  - get field – lua_getfield(L, -1, “a”) // negative numbers count from top of stack
  - retrieve value: lua_to{x}(…) value (where x is number/userdata/string)

- my_func(2)
  - push ‘my_func’ global function – lua_getglobal(L, “my_func”)
  - push argument – lua_pushnumber(L, 2)
  - call, get return(s) – lua_pcall(…), lua_to{x}(…), lua_pop(L, …)

(╯°□°) ╯︵┻━━┻ !!

-
  ```
  other_func(
          my_table[“a”][“b”],
          my_func(2)
  )
  ```

- Lua’s C API does not scale with complexity
  - amount of necessary boilerplate
  - developer time

sol::stack

- Non-self-balancing, stack-changing API wrappers
  - sol::stack::get<Type>(L, stack_index, record);
  - int num_pushed = sol::stack::push(L, anything);
  - sol::stack::check<Type>(L, stack_index, handler, record);
  - sol::stack::check_get<Type>(L, stack_index, handler, record);
  - int res = stack::lua_call<…>(L, from, cpp_callable, extra_arguments…);
- record tracks how many items are used / pulled from the stack

Fixed interop points

- Each struct is a template that has a sole responsibility, can override for custom behavior
  - struct sol::stack::getter<T, C= void> (.get(…))
  - struct sol::stack::pusher<T, C= void> (.push(…))
  - struct sol::stack::checker<T, sol::type, C= void> (.check(…))
  - struct sol::stack::check_getter<T, sol::type, C= void> (.check_get(…))
- sol::stack::lua_call<…>(…) uses other functions to perform the call

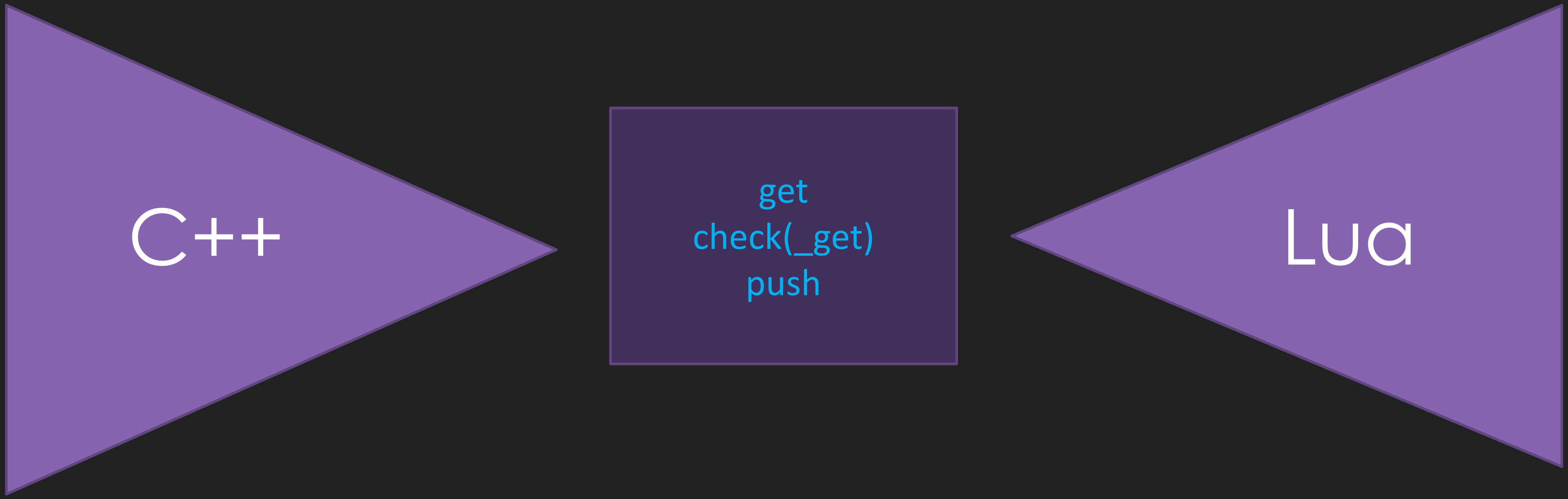
Scalability requires Defaults

- Problem: C++ has a lot more types than integers, floating point, strings, functions and table-alikes
- Need a sane default for some user-defined type T
  - Treated as userdata, which is a blob of memory

Some Types are Special

- std::pair / std::tuple
  - Lua has multiple returns, allow multiple-returns from C++ with these
- std::vector/std::list/std::map/ … - Lua has tables which emulates these
  - convert to table (expensive, but plays nice), or
  - store C++ container userdata (direct, fast, but plays less nice with Lua ecosystem)
- std::wstring/std::u16string/std::u32string
  - Unsurprisingly, people want these types to work – must UTF encode on push and on get.

What we are doing



C++

get
check(_get)
push

Lua

- Uniform conversions to and from, based on ***type***
- System is now well-defined for any given type, and easier to reason about

sol::reference

The cornerstone abstraction

Rule of 0 for Lua Binding

- sol::reference is a reference-counting object for something that is taken from Lua
  - Stored in the Lua registry, a heap of memory to keep Lua objects alive
  - Slower than stack, faster than literally any other serialization scheme

- Basically a Lua-specific version of the upcoming std::retain_ptr<T, R>
  - https://wg21.link/p0468r0

Formula for Success

- 1 – Derive from sol::reference
- 2 – Add no data members, just functionality and type-safety
- 3 – ???

4 – Profit

- sol::object – generic object for doing .is<T>() checks and .as<T>() conversions
- sol::table – allows operator[] indexing
- sol::function – allows operator() for calling in C++
- sol::thread – encapsulates a Lua thread (not like a C++ thread; it's separate stack space)
- sol::coroutine – like sol::function, but works off a stack space (thread)
- sol::state_view – cheap look at a Lua state, takes out a sol::table for registry and globals
- sol::state – sol::state_view + std::unique_ptr<lua_State*, lua_closer>

Magical Abstractions

Proxies, conversions and the missing Language Feature

Tables and []

- Need to be able to apply the access-operator [] on tables
  - Optimizations to be applied for nested lookups – my_table[“bark”][“woof”]

- Table lookup and global lookup actually have different C calls for Lua’s C API
  - Picking the right one / wrong one changes performances characteristics
  - … But gives same results (“API Trap”)

operator[]

- Lazily concatenates / saves keys, generating a new proxy type
- 1 tuple entry per operator [ ] lookup
- Commits lookup on any kind of assignment to proxy or implicit conversion of proxy

```
auto x = lua[“woof”][“bark”][1];
// decltype(x) == proxy<sol::global_table, const char*, const char*, int>
double value = x;
// triggers chained reads, attempts to conver to double
x = “woof”;
// triggers chained read into tables, then write into 1
```

proxy(_base) and friends

- Let’s take a peek…

What was all that SFINAE, exactly?

- Consider the simple case:

```
struct int_proxy {
        operator int () { return 2; }
};

int_proxy ip{};
int value = ip; // nice, conversion
const char* value_2 = ip; // boom, no conversion
```

Scaling up - 🦄 Proxy

```
struct unicorn_proxy {
        template <typename T>
        operator T () {
                /* arbitrary code can go here */
                return ...;
        }
};

unicorn_proxy up{};
int value = up; // nice, conversion
const char* value_2 = up; // yay!
```

Oh no! 🗡 🦄

```
struct unicorn_proxy {
        template <typename T>
        operator T () {
                /* arbitrary code can go here */
                return …;
        }
};

unicorn_proxy up{};
int a, b;
std::tie(a, b) = up; // Kabooooom!
```

Left Hand Side is Queen

- Implicit conversion operators take the type of the left hand side
    - Exactly, with no modifications
    - Cannot return a reference that is not fixed in memory
- ☠ Cannot SFINAE/change return type! ☠
    - Type “T” is not a regular return type
    - Cannot apply transformations not allowed by the language (only T& and T-style returns work)

Soon™ Paper: Extended Conversions

```
struct unicorn_proxy {
    template <typename T>
    int operator T () { // deduce from LHS…
        return 42;  // but return whatever you want
    }
};
```

function_result

- Just another kind of proxy that has the same issues, manifests in other ways

Lua

```
function f (v)
	return v, v * 2
end
```

C++

```
double a, b;
std::tie(a, b) = lua["f"](2); // error: std::tuple<int&, int&> return
sol::tie(a, b) = lua["f"](2); // ✓ : custom expansion and op=
```

Usertypes

A demo…

Overloading

Simple compile-time Overload Set reduction

Overloading

```
struct my_class {};
int bark ( int arg );
int woof ( std::string arg );
int bork ( int arg1, bool arg2, double arg3, std::vector<double> arg4 );
int borf ( bool arg );
int yip ( my_class& arg1, bool arg2 );

// create overloaded set
lua["f"] = sol::overload( bark, woof, bork, borf, yip );
```

- What kind of cost to select right overload if we do: f(my_class.new(), true) in Lua?

Simulate

Lua calls:

```
f(my_class.new(), true)
```

must match:

my_class&, bool (arity of 2)

bark	woof	bork	borf	yip
1 arg	1 arg	4 args	1 arg	2 args

Simulate

Lua calls:

```
f(my_class.new(), true)
```

must match:

my_class&, bool (arity of 2)

bark	woof	bork	borf	yip
1 arg	1 arg	4 args	1 arg	2 args

Arity != 1

Simulate

Lua calls:

    f(my_class.new(), true)

must match:

my_class&, bool (arity of 2)

bark	woof	bork	borf	yip
1 arg	1 arg	4 args	1 arg	2 args

Disallowed: std::integer_sequence<1>

Simulate

Lua calls:

```
f(my_class.new(), true)
```

must match:

my_class&, bool (arity of 2)

[illegible]	[illegible]	[illegible]	[illegible]	yip
[illegible]	[illegible]	4 args	[illegible]	2 args

Arity != 4

Disallowed: std::integer_sequence<1>

Simulate

Lua calls:

f(my_class.new(), true)

must match:

my_class&, bool (arity of 2)

[illegible]	[illegible]	[illegible]	[illegible]	yip
[illegible]	[illegible]	4 args	[illegible]	2 args

Arity == 2
Check types…

Disallowed: std::integer_sequence<1, 4>

Simulate

Lua calls:

```
f(my_class.new(), true)
```

must match:

my_class&, bool (arity of 2)

[illegible]	[illegible]	[illegible]	[illegible]	yip ✓
[illegible]	[illegible]	4 args	[illegible]	2 args ✓

Disallowed: std::integer_sequence<1, 4>

Safety is Optional

But not std::optional

Queries can be made safe…

```
int value = lua[“value”];
my_class my_obj = lua[“my_obj”];

my_class& my_obj_r = lua[“my_obj”]; // can manipulate memory directly
my_class* my_obj_p = lua[“my_obj”]; // can manipulate memory directly


sol::function func = lua[“func”];
double x = f();
```

By slapping optional on it / checking

```
sol::optional<int> safe_value = lua[“value”];
sol::optional<my_class> safe_my_obj = lua[“my_obj”];

sol::optional<my_class&> safe_my_obj_r = lua[“my_obj”]; // nil = unengaged
sol::optional<my_class*> safe_my_obj_p = lua[“my_obj”]; // nil = engaged

sol::function func = lua[“func”];
if (!func.valid()) { throw std::runtime_error(“aaah”); }
sol::optional<double> x = f();
```

std::optional does NOT cut it

- For the reference case, would have to use some non_null<T*> struct and put that in optional
    - gsl::non_null is an alias, not a real struct – cannot control Proxy expressions based on it
    - Overhead for the struct + boolean (optional<T&> is compact)
- Breaks library teaching:
    - "If you want safety, just wrap X in an optional", compared to
    - "If you want safety, just wrap X in an optional, unless it's a reference, then you need to use…"

Soon™ Paper: std::optional<T&>

- Rebind on assignment
  - Only sane behavior
- Do not allow rvalues to be assigned into optional reference
  - Prevents dangling lifetime issues
- Reduce internal boilerplate code

std::promise<sol>

What things are in the future for sol

Sol3: why?

https://github.com/ThePhD/sol2/issues/538

"I had spent a whole day for moving my binding from tolua++ to sol2, I found my xcode became very very lag and compile time is about 10 minutes with about 8G heap,so I have to abandon xcode for coding.

I had spent another whole day for moving my binding from sol2 to kaguya, compile time is about 2-3 seconds."

Compile Times MATTER

- Variadic templates lose absolutely 0 information in propagation
  - Can optimize the entire run time like crazy
- Overused, overzealous application: reduce with initializer_list and other techniques
  - Saving compiler performance is a must
  - Will lose users without it

if constexpr

- Probably the biggest thing that can be done
- There is a LOT of tag-dispatch and SFINAE that ultimate results in binary choices
  - Things with fallbacks are the perfect candidate

Bloatymcbloatface

- People have used this tool on executable which utilize sol2 and other analysis techniques on debug/release binaries
- The amount of symbols / spam is ENORMOUS

But the goal was runtime speed, right…?

- Right: http://sol2.readthedocs.io/en/latest/benchmarks.html

The Last and Most Important Thing

Super important, I swear

DOCUMENTATION!!!

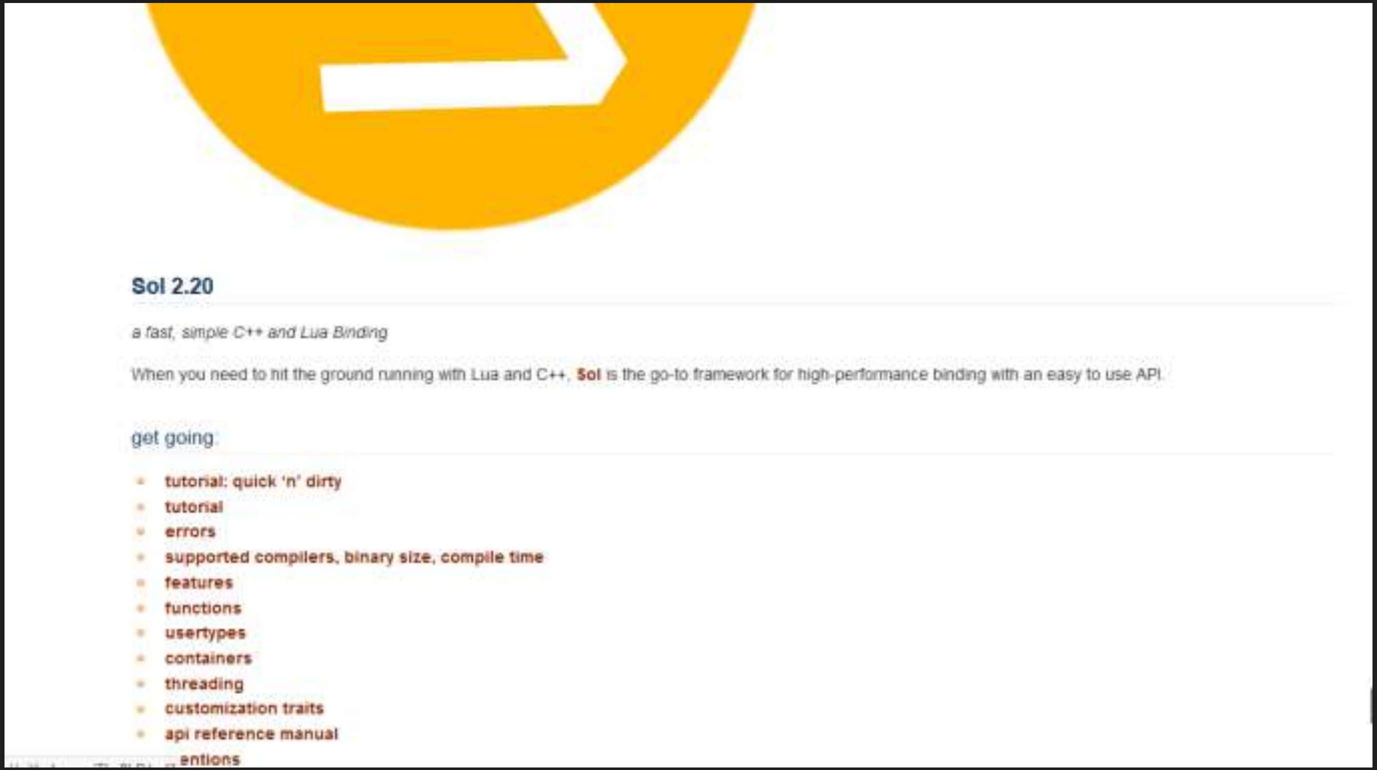
https://github.com/ThePhD/sol2/issues/36

"Greetings. I used to use Sol but could not figure out how it works … and thus quickly switched over to Selene, since on its main page it had a much better tutorial/how-to-manual. However now I'm currently using Selene and thinking about switching to Sol2 (because it supports LuaJit, being able to switch between luajit and lua5.3 for comparison is quite nice) and i ***think*** has more features."

The Backbone of Any Project

- Some projects are the “only alternative” so rather than reinvent
  - People muck through it and class APIs
  - Join an IRC to understand
  - Read the library’s tests to understand
- sol has 20+ competitors, with more NIH Syndrome spawns more bindings
  - Bled users everywhere because of no docs

http://sol2.rtfd.io/



Sol 2.20

a fast, simple C++ and Lua Binding

When you need to hit the ground running with Lua and C++, **Sol** is the go-to framework for high-performance binding with an easy to use API.

get going:

- tutorial: quick 'n' dirty
- tutorial
- errors
- supported compilers, binary size, compile time
- features
- functions
- usertypes
- containers
- threading
- customization traits
- api reference manual
- entions

Thanks and Shilling

- Support me and my family
  - Donation Links at the bottom of Docs Front Page and Readme
  - Donations have kept me fed for this trip, woo!

- THANK YOU!:
  - Donators: Robert Salvet, Ορφέας Ζαφείρης, Michael Waller, Elias Daler and Johannes Schultz
  - All of sol2’s users over the years

My Gratitude

- Mark Zeren of VMWare, Simon Brand (@TartanLlama) of Codeplay
  - Pushed me to apply as a student Volunteer
  - Words of encouragement are powerful things ♡

- Jason Turner (@lefticus)
  - Spoke about sol before I ever had plans for it
  - Really encouraged me to speak and finally got to meet him 😁
  - I’m going to appear on CppCast! Monday, May 21st, 2018

More Gratitude

- Hipony (Alexandr Timofeev) and kyzo (Alexander Scigajlo) for helping me bikeshed the logo in the Cpplang Slack!

- #include
  - for showing me that even if there might not be people like me in many of the places I am going and want to go, that they will accept me as a regular human being all the same

- Lounge<C++>
  - For always dragging me back in and being all around amazing nerds with great senses of humor

Questions? Comments?

- E-mail: phdofthehouse@gmail.com
- Twitter: @thephantomderp
- Linkedin: https://www.linkedin.com/in/thephd/
- Repository: https://github.com/ThePhD/sol2