WRAPPING LUA C IN C++

EFFICIENTLY, NICELY, AND WITH A TOUCH OF MAGIC

Boston C++ Meetup
November 8th, 2017

ThePhD, @thephantomderp
phdofthehouse@gmail.com | https://github.com/ThePhD/sol2

Design - Ideal

What we want out of this

Like the Language: Lua in C++

- What we need to mimic Lua in C++:
  - ☑ Primitive – string
    - Sized counted, const char*
  - ☑ Primitive – number
    - double covers everything (up until Lua 5.3)
  - ☒ Primitive – function
    - Created in Lua or bound from C++, not covered
  - ☒ Primitive – reference
    - All reference types in Lua (table, userdata, functions…)
    - std::shared_ptr to cover this?

Like the Language: C++ in Lua

- What we need to mimic C++ in Lua:
  - ☒ Primitive – classes
    - Member variable/function
    - Static functions
    - Inheritance?
  - ☒ Primitive – enumerations

- Will not be talking about this today

Perfect Interface

- int value = lua[“values”][2];
  - Access the VM, multi-depth query into state
- std::tie(a, b) = some_lua_function(“modulus”, 8, 3);
  - Retrieve function and call it
  - Be able to return multiple values
- lua[“my_function”] = [](std::string value, int append) { return value + “:” + std::to_string(append); };

Safety

- Lua is a dynamic language
  - Nothing informs you of really typical mistakes
  - Dynamite Development
- C++ rigorously checks types at compile-time
  - Reconcile rigorous C++ methods to a fast and loose runtime system

Implementation - Core

"… and on this rock I will build My church…"

Glossary I

- State – overall Lua state object
  - Contains everything
  - Not thread safe
  - All operations affect entire state

- Globals
  - global environment for everything
  - Accessible like a table
  - lua[“key”] means access the global table, for “key”

Glossary II

- Registry – designated place for C code storage
  - Mandatory for performant code outside the stack

- Stack – collection of working Lua values
  - Shared across entire state
  - Manipulated by iteration (!)

Glossary III

- L – lua_State*
  - Represents the state

- index – int
  - Stack: position on Lua's 1-based stack
  - Registry: a reference number in Lua's C registry

sol::stack

- The core of the API; usually never seen
  - sol::stack::get<Type>(L, stack_index)
  - sol::stack::push(L, obj);
  - sol::stack::check<Type>(L, stack_index);
- Defines fixed interop points:
  - struct sol::stack::getter<T, C = void>
  - struct sol::stack::pusher<T, C = void>
  - struct sol::stack::checker<T, sol::type, C = void>

sol::stack::getter

- Templated getter structure we can specialize
  - T - unqualified type
  - C - SFINAE-enabler

```
template <typename T, typename C = void>
struct getter {
        T get (lua_State* L, int index) {
                // ...
        }
};
```

sol::stack::getter<int>

- sol::stack::get<int>(L, 1);
  - int – the type we want to get
  - Purpose of C shown below:

```
template <typename T>
struct getter<T, std::enable_if_t<
        std::is_integral<T>::value
>> {
        int get (lua_State* L, int index) {
                return (T)lua_tointegerx(L, index, NULL);
        }
};
```

sol::stack::pusher

- Templated pusher structure we can specialize
  - T - unqualified type
  - C - SFINAE-enabler

```
template <typename T, typename C = void>
struct pusher {
        int push (lua_State* L, const T& object) {
                // ...
                return 1; // or # pushed onto stack
        }
};
```

sol::stack::pusher

- sol::stack::push(L, std::string("bark"));
  - T - unqualified type
  - C - SFINAE-enabler

```
template <>
struct pusher<std::string> {
        int push (lua_State* L, const std::string& s) {
                lua_pushlstring(L, s.c_str(), s.size());
                return 1;
        }
};
```

sol::stack::checker

- Templated checker structure we can specialize
  - T - unqualified type
  - C - SFINAE-enabler

```
template <typename T,
       sol::type expect = sol::lua_type_of<T>::value,
       typename C = void>
struct checker {
       template <typename H>
       bool checker (lua_State* L, int index, H&& handler) {
              if (/* type check fails */) {
                     handler( … ); return false;
              }
              return true;
       }
};
```

Extend as needed

- Generate getters for standard library and built-in types
  - Function types; `std::function<>`; `operator()( … )` types
  - Strings (c-string, std::string_view); integers; floats
    - utf-8/16/32 conversions at boundaries
  - Container types (`std::vector/map/forward_list`)

- Explicit and partial template specialization
  - Users can specialize for their own types – extensible!!

Safety

- On every sol::stack::get operation
  - Check if the desired specified C++ type matches what's stored using sol::stack::check, invoke default panic handler
  - Requires a safety #define to do this

- Everything runs through sol::stack::get/check/push
  - Definitive point of interop: lets us (and you) control everything

Ascending Higher

Higher level types

- Cannot work with sol::stack all the time
  - Too low-level for most programmers
  - Annoying to worry about push/pop counts and cleanup
- Need higher-level primitives
  - Things that automatically handle:
    - Registry lifetime
    - Stack push, pop and clean up

reference – the cornerstone

- Base primitive for all extended types
  - Only costs 1 `int` plus lua_State pointer
  - Our "rule of zero" type

- Implements std::shared_ptr-like details
  - Less overhead than std::`shared_ptr` with deleter
  - Does not need thread safety, bolted to registry
  - copy, move, deletion built into this type

reference - operations

- constructor (lua_State* L, int stack_index)
  - create from stack reference, save in registry
  - all our reference-based primitives need this constructor
- Basic observers
  - sol::type get_type() const; int registry_index() const;
- Stack manipulators, but really only used by library
  - int push() const; void pop ();

table, userdata, function, …

- Step 1: derive from sol::reference
- Step 2: add type assertion on construct
  - Make optional for performance nuts or potential unforseen future use cases
- Step 3: ????
- Step 4: Done!

```
class object: reference { … };
class table : reference { … };
class userdata : reference { … };
class function : reference { … };
```

Maximum "Rule of Zero"

- Step 3 may be more involved
  - No other extra data members needed, however
  - Everything is based on working with the stack, and that references the type in the registry
- Task: writing stack-manipulation functions that perform desired goal
  - table access, function call, value conversion…

sol::object

- General-purpose “thing”: can be checked and coverted
  - bool is_type = obj.is<type>();
  - type value = obj.as<type>();
- Considered the “any” type of the library
  - just represents some single thing

sol::table

- Object that an be accessed with keys
  - rhs = table.get<Type0, …, TypeN>(“key0”, …, “keyN”);
  - table.set (“key0”, value0, …, “keyN”, valueN);
- Just uses (multiple) sol::stack::get/sol::stack::push calls
  - Multiple types / values allow std::tie multiple objects from a tuple return
- sol::userdata is just a sol::table with a different type check

sol::function

- A callable object
  - func.call<result_type, …>(arg0, …, argN);
- Sequence of core stack operations
  - sol::stack::push for each arg, accumulate # pushed
  - Call function in VM, then sol::stack::get
- Variadic result specification
  - Produces a tuple, otherwise produces single type

Interface is bad

- Explicit function calls everywhere
- Types everywhere

```
std::string s = func.call<std::string>(“dog”);
my_table.set(“some_key”, 24);
int value = my_table.get<int>(“some_key”);
```

We want something better!

- We have effective syntax in both languages for this

```
std::string s = func(“dog”);
my_table[“some_key”] = 24;
int value = my_table[“some_key”];
```

```
s = func(“dog”)
my_table[“some_key”] = 24
value = my_table[“some_key”];
```

Implementation - Magic

the good stuff

Proxies

- We need to convert from some expression to some arbitrary type we want
  - We need in-between types to do the conversion
- Need multiple in-between types, to fit scenarios
  - operator[]: source and key-templated table proxy
  - func(…): function_result proxy
  - Other kinds for more advanced usages

The magic

- The proxy_base class, in its full glory…

```cpp

namespace sol {
    struct proxy_base_tag {};

    template <typename Super>
    struct proxy_base : proxy_base_tag {
        operator std::string() const {
            const Super& super = *static_cast<const Super*>(static_cast<const void*>(this));
            return super.template get<std::string>();
        }

        template <typename T, meta::enable<meta::neg<meta::is_string_constructible<T>>, is_proxy_primitive<meta::unqualified_t<T>>> = meta::enabler>
        operator T() const {
            const Super& super = *static_cast<const Super*>(static_cast<const void*>(this));
            return super.template get<T>();
        }

        template <typename T, meta::enable<meta::neg<meta::is_string_constructible<T>>, meta::neg<is_proxy_primitive<meta::unqualified_t<T>>>> = meta::enabler>
        operator T&() const {
            const Super& super = *static_cast<const Super*>(static_cast<const void*>(this));
            return super.template get<T&>();
        }

        lua_State* lua_state() const {
            const Super& super = *static_cast<const Super*>(static_cast<const void*>(this));
            return super.lua_state();
        }
    };
} // namespace sol

```

proxy_base: the reference of proxies

- Implement the desired function for proxy_base (get), and it handles conversions for us
  - operator[]-generated proxies override operator= for assignment purposes
  - Allows seamless conversion

- operator[] on a proxy just generates another proxy with the passed-in key

Proxies = 99% of the magic

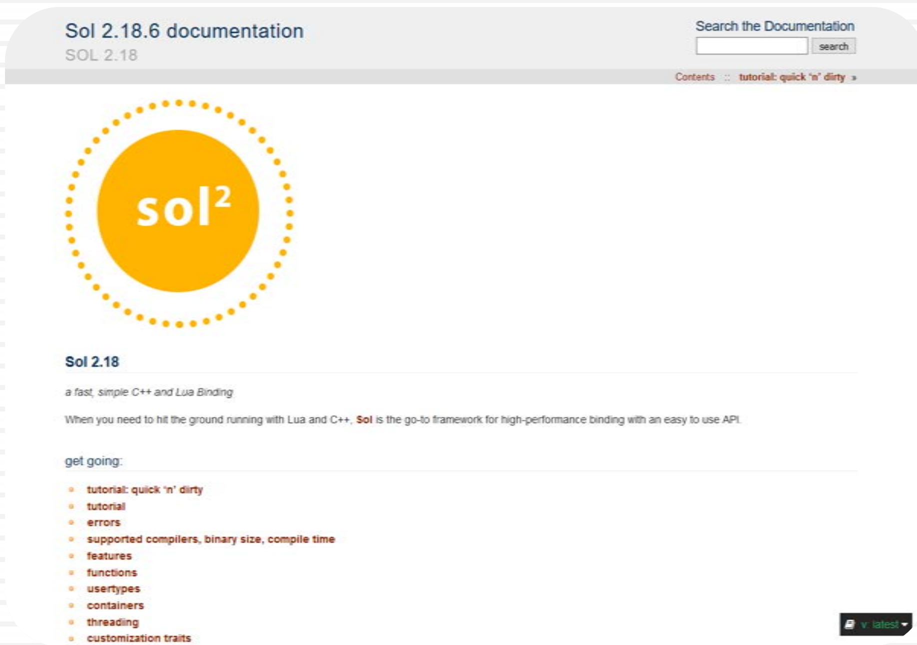
- proxy_base forms base of:
  - function_result/protected_function_result, stack_proxy, table_proxy, etc…

- This works exactly as advertised:

```
std::string s = func("dog");
my_table["some_key"] = 24;
int value = my_table["some_key"];

std::string s2 = other_func(my_table["key1"]["key2"]);
```

Documentation



Sol 2.18.6 documentation

SOL 2.18

Search the Documentation

search

Contents :: tutorial: quick 'n' dirty »

sol²

Sol 2.18

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

v: latest

Present since release of sol2

- Very explicit, lots of examples, lots of suggestions, searchable:
  - http://sol2.rtfd.io
  - https://github.com/ThePhD/sol2/tree/develop/examples

- Covers simple and advanced use cases
  - Attempts to group subject matter
  - Tutorials through the basics
  - Continuously adding examples from user feedback

Thanks To

- Professor Gail E. Kaiser
  - Coms E6156 – Advanced Software Engineering
  - Iris Zhang – Vetted library, improved Mac OSX story
- Kevin Brightwell (Nava2)
  - Took a great interest in sol2 before anyone else
  - Vastly improved CI (twice in a row!)
  - Submitted an upstream patch to Cmake for LuaJIT!

Thanks To

- Lounge<C++>
- EliasDaler (@EliasDaler), Eevee (@eevee)
  - Blogposts (https://eev.ee, https://elias-daler.github.io)
- Jason Turner (@lefticus)
  - Encouraged me to present at first and talk about Sol2
  - Runs CppCast (https://cppcast.com)

Thank you!

- Questions and/or Comments?
  - If you use sol2 or are going to use sol2, consider leaving some feedback: https://github.com/ThePhD/sol2/issues/189