Sol Documentation

Release 2.20.5

ThePhD

Oct 10, 2022

Contents

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.

CHAPTER 1

get going:

1.1 tutorial: quick 'n' dirty

These are all the things. Use your browser's search to find things you want.

Note: After you learn the basics of sol, it is usually advised that if you think something can work, you should TRY IT. It will probably work!

Note: All of the code below is available at the sol2 tutorial examples.

1.1.1 asserts / prerequisites

You'll need to `#include <sol.hpp>`/`#include "sol.hpp"` somewhere in your code. Sol is header-only, so you don't need to compile anything. However, **Lua must be compiled and available**. See the *getting started tutorial* for more details.

The implementation for `assert.hpp` with `c_assert` looks like so:

```
#ifndef EXAMPLES_ASSERT_HPP
#define EXAMPLES_ASSERT_HPP

#   define m_assert(condition, message) \
    do { \
        if (! (condition)) { \
            std::cerr << "Assertion `" #condition "` failed in " << __FILE__ \
                      << " line " << __LINE__ << ": " << message << std::endl; \
            std::terminate(); \
        } \
    } while (false)
```

(continues on next page)

(continued from previous page)

```

#   define c_assert(condition) \
    do { \
        if (! (condition)) { \
            std::cerr << "Assertion `" #condition "` failed in " << __FILE__ \
                      << " line " << __LINE__ << std::endl; \
            std::terminate(); \
        } \
    } while (false)
#else
#   define m_assert(condition, message) do { if (false) { (void)(condition);␣
↪(void)sizeof(message); } } while (false)
#   define c_assert(condition) do { if (false) { (void)(condition); } } while (false)
#endif

#endif // EXAMPLES_ASSERT_HPP
```

This is the assert used in the quick code below.

1.1.2 opening a state

```
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

#include <iostream>
#include "../../assert.hpp"

int main(int, char*[]) {
        std::cout << "=== opening a state ===" << std::endl;

        sol::state lua;
        // open some common libraries
        lua.open_libraries(sol::lib::base, sol::lib::package);
        lua.script("print('bark bark bark!')");

        std::cout << std::endl;

        return 0;
}
```

1.1.3 using sol2 on a lua_State*

For your system/game that already has Lua or uses an in-house or pre-rolled Lua system (LuaBridge, kaguya, Luwra, etc.), but you'd still like sol2 and nice things:

```
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

#include <iostream>

int use_sol2(lua_State* L) {
        sol::state_view lua(L);
        lua.script("print('bark bark bark!')");
```

(continues on next page)

(continued from previous page)

```
        return 0;
}

int main(int, char*[]) {
        std::cout << "=== opening sol::state_view on raw Lua ===" << std::endl;

        lua_State* L = luaL_newstate();
        luaL_openlibs(L);

        lua_pushcclosure(L, &use_sol2, 0);
        lua_setglobal(L, "use_sol2");

        if (luaL_dostring(L, "use_sol2()")) {
                lua_error(L);
                return -1;
        }

        std::cout << std::endl;

        return 0;
}
```

1.1.4 running lua code

```
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

#include <fstream>
#include <iostream>
#include "../../assert.hpp"

int main(int, char*[]) {
        std::cout << "=== running lua code ===" << std::endl;

        sol::state lua;
        lua.open_libraries(sol::lib::base);

        // load and execute from string
        lua.script("a = 'test'");
        // load and execute from file
        lua.script_file("a_lua_script.lua");

        // run a script, get the result
        int value = lua.script("return 54");
        c_assert(value == 54);
```

To run Lua code but have an error handler in case things go wrong:

```
        auto bad_code_result = lua.script("123 herp.derp", [](lua_State*,
↪sol::protected_function_result pfr) {
                // pfr will contain things that went wrong, for either loading or
↪executing the script
                // Can throw your own custom error
                // You can also just return it, and let the call-site handle the
↪error if necessary.
```

(continues on next page)

(continued from previous page)

```
                return pfr;
        });
        // it did not work
        c_assert(!bad_code_result.valid());

        // the default handler panics or throws, depending on your settings
        // uncomment for explosions:
        //auto bad_code_result_2 = lua.script("bad.code", &sol::script_default_on_
↪error);
        return 0;
}
```

1.1.5 running lua code (low-level)

You can use the individual load and function call operator to load, check, and then subsequently run and check code.

Warning: This is ONLY if you need some sort of fine-grained control: for 99% of cases, *running lua code* is preferred and avoids pitfalls in not understanding the difference between script/load and needing to run a chunk after loading it.

```
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

#include <fstream>
#include <iostream>
#include <cstdio>
#include "../../assert.hpp"

int main(int, char*[]) {
        std::cout << "=== running lua code (low level) ===" << std::endl;

        sol::state lua;
        lua.open_libraries(sol::lib::base);

        // load file without execute
        sol::load_result script1 = lua.load_file("a_lua_script.lua");
        //execute
        script1();

        // load string without execute
        sol::load_result script2 = lua.load("a = 'test'");
        //execute
        sol::protected_function_result script2result = script2();
        // optionally, check if it worked
        if (script2result.valid()) {
                // yay!
        }
        else {
                // aww
        }

        sol::load_result script3 = lua.load("return 24");
        // execute, get return value
```

(continues on next page)

(continued from previous page)

```
        int value2 = script3();
        c_assert(value2 == 24);

        return 0;
}
```

1.1.6 set and get variables

You can set/get everything using table-like syntax.

```
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

#include "../../assert.hpp"

int main(int, char*[]) {
        sol::state lua;
        lua.open_libraries(sol::lib::base);

        // integer types
        lua.set("number", 24);
        // floating point numbers
        lua["number2"] = 24.5;
        // string types
        lua["important_string"] = "woof woof";
        // is callable, therefore gets stored as a function that can be called
        lua["a_function"] = []() { return 100; };
        // make a table
        lua["some_table"] = lua.create_table_with("value", 24);
```

Equivalent to loading lua values like so:

```
        // equivalent to this code
        std::string equivalent_code = R"(
                t = {
                        number = 24,
                        number2 = 24.5,
                        important_string = "woof woof",
                        a_function = function () return 100 end,
                        some_table = { value = 24 }
                }
        )";

        // check in Lua
        lua.script(equivalent_code);
```

You can show they are equivalent:

```
        lua.script(R"(
                assert(t.number == number)
                assert(t.number2 == number2)
                assert(t.important_string == important_string)
                assert(t.a_function() == a_function())
                assert(t.some_table.value == some_table.value)
        )");
```

Retrieve these variables using this syntax:

```
        // implicit conversion
        int number = lua["number"];
        c_assert(number == 24);
        // explicit get
        auto number2 = lua.get<double>("number2");
        c_assert(number2 == 24.5);
        // strings too
        std::string important_string = lua["important_string"];
        c_assert(important_string == "woof woof");
        // dig into a table
        int value = lua["some_table"]["value"];
        c_assert(value == 24);
        // get a function
        sol::function a_function = lua["a_function"];
        int value_is_100 = a_function();
        // convertible to std::function
        std::function<int()> a_std_function = a_function;
        int value_is_still_100 = a_std_function();
        c_assert(value_is_100 == 100);
        c_assert(value_is_still_100 == 100);
```

Retrieve Lua types using `object` and other `sol::` types.

```
        sol::object number_obj = lua.get<sol::object>("number");
        // sol::type::number
        sol::type t1 = number_obj.get_type();
        c_assert(t1 == sol::type::number);

        sol::object function_obj = lua["a_function"];
        // sol::type::function
        sol::type t2 = function_obj.get_type();
        c_assert(t2 == sol::type::function);
        bool is_it_really = function_obj.is<std::function<int()>>();
        c_assert(is_it_really);

        // will not contain data
        sol::optional<int> check_for_me = lua["a_function"];
        c_assert(check_for_me == sol::nullopt);

        return 0;
}
```

You can erase things by setting it to `nullptr` or `sol::lua_nil`.

```
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

#include "../../assert.hpp"

int main(int, char*[]) {
        sol::state lua;
        lua.open_libraries(sol::lib::base);

        lua.script("exists = 250");

        int first_try = lua.get_or("exists", 322);
```

(continues on next page)

(continued from previous page)

```
        c_assert(first_try == 250);

        lua.set("exists", sol::lua_nil);
        int second_try = lua.get_or("exists", 322);
        c_assert(second_try == 322);

        return 0;
}
```

Note that if its a *userdata/usertype* for a C++ type, the destructor will run only when the garbage collector deems it appropriate to destroy the memory. If you are relying on the destructor being run when its set to `sol::lua_nil`, you're probably committing a mistake.

1.1.7 tables

Tables can be manipulated using accessor-syntax. Note that *sol::state* is a table and all the methods shown here work with `sol::state`, too.

```
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

#include "../../assert.hpp"

int main(int, char*[]) {

        sol::state lua;
        lua.open_libraries(sol::lib::base);

        lua.script(R"(
                abc = { [0] = 24 }
                def = {
                        ghi = {
                                bark = 50,
                                woof = abc
                        }
                }
        )");

        sol::table abc = lua["abc"];
        sol::table def = lua["def"];
        sol::table ghi = lua["def"]["ghi"];

        int bark1 = def["ghi"]["bark"];
        int bark2 = lua["def"]["ghi"]["bark"];
        c_assert(bark1 == 50);
        c_assert(bark2 == 50);

        int abcval1 = abc[0];
        int abcval2 = ghi["woof"][0];
        c_assert(abcval1 == 24);
        c_assert(abcval2 == 24);
```

If you're going deep, be safe:

```
        sol::optional<int> will_not_error = lua["abc"]["DOESNOTEXIST"]["ghi"];
        c_assert(will_not_error == sol::nullopt);

        int also_will_not_error = lua["abc"]["def"]["ghi"]["jklm"].get_or(25);
        c_assert(also_will_not_error == 25);

        // if you don't go safe,
        // will throw (or do at_panic if no exceptions)
        //int aaaahhh = lua["boom"]["the_dynamite"];

        return 0;
}
```

1.1.8 make tables

There are many ways to make a table. Here's an easy way for simple ones:

```
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

#include "../../assert.hpp"

int main(int, char* []) {
        sol::state lua;
        lua.open_libraries(sol::lib::base);

        lua["abc_sol2"] = lua.create_table_with(
                0, 24
        );

        sol::table inner_table = lua.create_table_with("bark", 50,
                // can reference other existing stuff too
                "woof", lua["abc_sol2"]
        );
        lua.create_named_table("def_sol2",
                "ghi", inner_table
        );
```

Equivalent Lua code, and check that they're equivalent:

```
        std::string code = R"(
                abc = { [0] = 24 }
                def = {
                        ghi = {
                                bark = 50,
                                woof = abc
                        }
                }
        )";

        lua.script(code);
        lua.script(R"(
                assert(abc_sol2[0] == abc[0])
                assert(def_sol2.ghi.bark == def.ghi.bark)
        )");

```

(continues on next page)

(continued from previous page)

```
        return 0;
}
```

You can put anything you want in tables as values or keys, including strings, numbers, functions, other tables.

Note that this idea that things can be nested is important and will help later when you get into *namespacing*.

1.1.9 functions

They're easy to use, from Lua and from C++:

```
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

#include "../../assert.hpp"

int main(int, char*[]) {
        sol::state lua;
        lua.open_libraries(sol::lib::base);

        lua.script("function f (a, b, c, d) return 1 end");
        lua.script("function g (a, b) return a + b end");

        // sol::function is often easier:
        // takes a variable number/types of arguments...
        sol::function fx = lua["f"];
        // fixed signature std::function<...>
        // can be used to tie a sol::function down
        std::function<int(int, double, int, std::string)> stdfx = fx;

        int is_one = stdfx(1, 34.5, 3, "bark");
        c_assert(is_one == 1);
        int is_also_one = fx(1, "boop", 3, "bark");
        c_assert(is_also_one == 1);

        // call through operator[]
        int is_three = lua["g"](1, 2);
        c_assert(is_three == 3);
        double is_4_8 = lua["g"](2.4, 2.4);
        c_assert(is_4_8 == 4.8);

        return 0;
}
```

If you need to protect against errors and parser problems and you're not ready to deal with Lua's `longjmp` problems (if you compiled with C), use *sol::protected_function*.

You can bind member variables as functions too, as well as all KINDS of function-like things:

```
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

#include "../../assert.hpp"
#include <iostream>

void some_function() {
```

(continues on next page)

(continued from previous page)

```
        std::cout << "some function!" << std::endl;
}

void some_other_function() {
        std::cout << "some other function!" << std::endl;
}

struct some_class {
        int variable = 30;

        double member_function() {
                return 24.5;
        }
};

int main(int, char*[]) {
        std::cout << "=== functions (all) ===" << std::endl;

        sol::state lua;
        lua.open_libraries(sol::lib::base);

        // put an instance of "some_class" into lua
        // (we'll go into more detail about this later
        // just know here that it works and is
        // put into lua as a userdata
        lua.set("sc", some_class());

        // binds a plain function
        lua["f1"] = some_function;
        lua.set_function("f2", &some_other_function);

        // binds just the member function
        lua["m1"] = &some_class::member_function;

        // binds the class to the type
        lua.set_function("m2", &some_class::member_function, some_class{});

        // binds just the member variable as a function
        lua["v1"] = &some_class::variable;

        // binds class with member variable as function
        lua.set_function("v2", &some_class::variable, some_class{});
```

The lua code to call these things is:

```
        lua.script(R"(
        f1() -- some function!
        f2() -- some other function!

        -- need class instance if you don't bind it with the function
        print(m1(sc)) -- 24.5
        -- does not need class instance: was bound to lua with one
        print(m2()) -- 24.5

        -- need class instance if you
        -- don't bind it with the function
        print(v1(sc)) -- 30
```

(continues on next page)

(continued from previous page)

```
        -- does not need class instance:
        -- it was bound with one
        print(v2()) -- 30

        -- can set, still
        -- requires instance
        v1(sc, 212)
        -- can set, does not need
        -- class instance: was bound with one
        v2(254)

        print(v1(sc)) -- 212
        print(v2()) -- 254
        )");

        std::cout << std::endl;

        return 0;
}
```

You can use `sol::readonly( &some_class::variable )` to make a variable readonly and error if someone tries to write to it.

1.1.10 self call

You can pass the `self` argument through C++ to emulate 'member function' calls in Lua.

```
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

#include <iostream>

int main() {
        std::cout << "=== self_call ===" << std::endl;

        sol::state lua;
        lua.open_libraries(sol::lib::base, sol::lib::package, sol::lib::table);

        // a small script using 'self' syntax
        lua.script(R"(
        some_table = { some_val = 100 }

        function some_table:add_to_some_val(value)
            self.some_val = self.some_val + value
        end

        function print_some_val()
            print("some_table.some_val = " .. some_table.some_val)
        end
        )");

        // do some printing
        lua["print_some_val"]();
        // 100

```

(continues on next page)

(continued from previous page)

```
        sol::table self = lua["some_table"];
        self["add_to_some_val"](self, 10);
        lua["print_some_val"]();

        std::cout << std::endl;

        return 0;
}
```

1.1.11 multiple returns from lua

```
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

#include "../../assert.hpp"

int main(int, char* []) {
        sol::state lua;

        lua.script("function f (a, b, c) return a, b, c end");

        std::tuple<int, int, int> result;
        result = lua["f"](100, 200, 300);
        // result == { 100, 200, 300 }
        int a;
        int b;
        std::string c;
        sol::tie(a, b, c) = lua["f"](100, 200, "bark");
        c_assert(a == 100);
        c_assert(b == 200);
        c_assert(c == "bark");

        return 0;
}
```

1.1.12 multiple returns to lua

```
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

#include "../../assert.hpp"

int main(int, char* []) {
        sol::state lua;
        lua.open_libraries(sol::lib::base);

        lua["f"] = [](int a, int b, sol::object c) {
                // sol::object can be anything here: just pass it through
                return std::make_tuple(a, b, c);
        };

        std::tuple<int, int, int> result = lua["f"](100, 200, 300);
```

(continues on next page)

(continued from previous page)

```
        const std::tuple<int, int, int> expected(100, 200, 300);
        c_assert(result == expected);

        std::tuple<int, int, std::string> result2;
        result2 = lua["f"](100, 200, "BARK BARK BARK!");
        const std::tuple<int, int, std::string> expected2(100, 200, "BARK BARK BARK!
↪");
        c_assert(result2 == expected2);

        int a, b;
        std::string c;
        sol::tie(a, b, c) = lua["f"](100, 200, "bark");
        c_assert(a == 100);
        c_assert(b == 200);
        c_assert(c == "bark");

        lua.script(R"(
                a, b, c = f(150, 250, "woofbark")
                assert(a == 150)
                assert(b == 250)
                assert(c == "woofbark")
        )");

        return 0;
}
```

1.1.13 C++ classes from C++

Everything that is not a:

- primitive type: `bool`, `char/short/int/long/long long`, `float/double`
- string type: `std::string`, `const char*`
- function type: function pointers, `lua_CFunction`, `std::function`, *sol::function/sol::protected_function*, *sol::coroutine*, member variable, member function
- designated sol type: *sol::table*, *sol::thread*, *sol::error*, *sol::object*
- transparent argument type: *sol::variadic_arg*, *sol::this_state*, *sol::this_environment*
- usertype<T> class: *sol::usertype*

Is set as a *userdata + usertype*.

```
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

#include "../../assert.hpp"
#include <iostream>

struct Doge {
        int tailwag = 50;

        Doge() {
        }

```

(continues on next page)

(continued from previous page)

```
        Doge(int wags)
        : tailwag(wags) {
        }

        ~Doge() {
                std::cout << "Dog at " << this << " is being destroyed..." <<␣
↪std::endl;
        }
};

int main(int, char* []) {
        std::cout << "=== userdata ===" << std::endl;

        sol::state lua;

        Doge dog{ 30 };

        // fresh one put into Lua
        lua["dog"] = Doge{};
        // Copy into lua: destroyed by Lua VM during garbage collection
        lua["dog_copy"] = dog;
        // OR: move semantics - will call move constructor if present instead
        // Again, owned by Lua
        lua["dog_move"] = std::move(dog);
        lua["dog_unique_ptr"] = std::make_unique<Doge>(25);
        lua["dog_shared_ptr"] = std::make_shared<Doge>(31);

        // Identical to above
        Doge dog2{ 30 };
        lua.set("dog2", Doge{});
        lua.set("dog2_copy", dog2);
        lua.set("dog2_move", std::move(dog2));
        lua.set("dog2_unique_ptr", std::unique_ptr<Doge>(new Doge(25)));
        lua.set("dog2_shared_ptr", std::shared_ptr<Doge>(new Doge(31)));

        // Note all of them can be retrieved the same way:
        Doge& lua_dog = lua["dog"];
        Doge& lua_dog_copy = lua["dog_copy"];
        Doge& lua_dog_move = lua["dog_move"];
        Doge& lua_dog_unique_ptr = lua["dog_unique_ptr"];
        Doge& lua_dog_shared_ptr = lua["dog_shared_ptr"];
        c_assert(lua_dog.tailwag == 50);
        c_assert(lua_dog_copy.tailwag == 30);
        c_assert(lua_dog_move.tailwag == 30);
        c_assert(lua_dog_unique_ptr.tailwag == 25);
        c_assert(lua_dog_shared_ptr.tailwag == 31);
        std::cout << std::endl;

        return 0;
}
```

`std::unique_ptr`/`std::shared_ptr`'s reference counts / deleters will *be respected*.

If you want it to refer to something, whose memory you know won't die in C++ while it is used/exists in Lua, do the following:

```cpp
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

#include "../../assert.hpp"
#include <iostream>

struct Doge {
	int tailwag = 50;

	Doge() {
	}

	Doge(int wags)
		: tailwag(wags) {
	}

	~Doge() {
		std::cout << "Dog at " << this << " is being destroyed..." << std::endl;
	}
};

int main(int, char* []) {
	std::cout << "=== userdata memory reference ===" << std::endl;

	sol::state lua;
	lua.open_libraries(sol::lib::base);

	Doge dog{}; // Kept alive somehow

	// Later...
	// The following stores a reference, and does not copy/move
	// lifetime is same as dog in C++
	// (access after it is destroyed is bad)
	lua["dog"] = &dog;
	// Same as above: respects std::reference_wrapper
	lua["dog"] = std::ref(dog);
	// These two are identical to above
	lua.set( "dog", &dog );
	lua.set( "dog", std::ref( dog ) );


	Doge& dog_ref = lua["dog"]; // References Lua memory
	Doge* dog_pointer = lua["dog"]; // References Lua memory
	Doge dog_copy = lua["dog"]; // Copies, will not affect lua
```

You can retrieve the userdata in the same way as everything else. Importantly, note that you can change the data of usertype variables and it will affect things in lua if you get a pointer or a reference:

```cpp
	lua.new_usertype<Doge>("Doge",
		"tailwag", &Doge::tailwag
	);

	dog_copy.tailwag = 525;
	// Still 50
	lua.script("assert(dog.tailwag == 50)");

```

(continues on next page)

(continued from previous page)

```
        dog_ref.tailwag = 100;
        // Now 100
        lua.script("assert(dog.tailwag == 100)");

        dog_pointer->tailwag = 345;
        // Now 345
        lua.script("assert(dog.tailwag == 345)");

        std::cout << std::endl;

        return 0;
}
```

1.1.14 C++ classes put into Lua

See this *section here*. Also check out a basic example, special functions example and initializers example! There are many more examples that show off the usage of classes in C++, so please peruse them all carefully as it can be as simple or as complex as your needs are.

1.1.15 namespacing

You can emulate namespacing by having a table and giving it the namespace names you want before registering enums or usertypes:

```
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

#include <iostream>
#include "../../assert.hpp"

int main() {
        std::cout << "=== namespacing ===" << std::endl;

        struct my_class {
                int b = 24;

                int f() const {
                        return 24;
                }

                void g() {
                        ++b;
                }
        };

        sol::state lua;
        lua.open_libraries();

        // "bark" namespacing in Lua
        // namespacing is just putting things in a table
        // forces creation if it does not exist
        auto bark = lua["bark"].get_or_create<sol::table>();
        // equivalent-ish:
```

(continues on next page)

(continued from previous page)

```
        //sol::table bark = lua["bark"].force(); // forces table creation
        // equivalent, and more flexible:
        //sol::table bark = lua["bark"].get_or_create<sol::table>(sol::new_table());
        // equivalent, but less efficient/ugly:
        //sol::table bark = lua["bark"] = lua.get_or("bark", lua.create_table());
        bark.new_usertype<my_class>("my_class",
                "f", &my_class::f,
                "g", &my_class::g); // the usual

        // can add functions, as well (just like the global table)
        bark.set_function("print_my_class", [](my_class& self) { std::cout << "my_
↪class { b: " << self.b << " }" << std::endl; });

        // this works
        lua.script("obj = bark.my_class.new()");
        lua.script("obj:g()");

        // calling this function also works
        lua.script("bark.print_my_class(obj)");
        my_class& obj = lua["obj"];
        c_assert(obj.b == 25);

        std::cout << std::endl;

        return 0;
}
```

This technique can be used to register namespace-like functions and classes. It can be as deep as you want. Just make a table and name it appropriately, in either Lua script or using the equivalent Sol code. As long as the table FIRST exists (e.g., make it using a script or with one of Sol's methods or whatever you like), you can put anything you want specifically into that table using *sol::table's* abstractions.

1.1.16 there is a LOT more

Some more things you can do/read about:

- ***the usertypes page*** **lists the huge amount of features for functions**
  - *unique usertype traits* allows you to specialize handle/RAII types from other libraries frameworks, like boost and Unreal, to work with Sol. Allows custom smart pointers, custom handles and others
- *the containers page* gives full information about handling everything about container-like usertypes
- ***the functions page*** **lists a myriad of features for functions**
  - *variadic arguments* in functions with `sol::variadic_args`.
  - also comes with *variadic_results* for returning multiple differently-typed arguments
  - *this_state* to get the current `lua_State*`, alongside other transparent argument types
- *metatable manipulations* allow a user to change how indexing, function calls, and other things work on a single type.
- *ownership semantics* are described for how Lua deals with its own internal references and (raw) pointers.
- *stack manipulation* to safely play with the stack. You can also define customization points for `stack::get`/`stack::check`/`stack::push` for your type.

- *make_reference/make_object convenience function* to get the same benefits and conveniences as the low-level stack API but put into objects you can specify.
- *stack references* to have zero-overhead Sol abstractions while not copying to the Lua registry.
- *resolve* overloads in case you have overloaded functions; a cleaner casting utility. You must use this to emulate default parameters.

1.2 tutorial

Take some time to learn the framework with these tutorials. But, if you need to get going FAST, try using the *quick 'n' dirty* approach and your browser's / editors search function. It will also serve you well to look at all the examples, which have recently gotten a bit of an overhaul to contain more relevant working examples and other advanced tricks that you can leverage to have a good time!

1.2.1 getting started

Let's get you going with Sol! To start, you'll need to use a lua distribution of some sort. Sol doesn't provide that: it only wraps the API that comes with it, so you can pick whatever distribution you like for your application. There are lots, but the two popular ones are vanilla Lua and speedy LuaJIT . We recommend vanilla Lua if you're getting started, LuaJIT if you need speed and can handle some caveats: the interface for Sol doesn't change no matter what Lua version you're using.

If you need help getting or building Lua, check out the Lua page on getting started. Note that for Visual Studio, one can simply download the sources, include all the Lua library files in that project, and then build for debug/release, x86/x64/ARM rather easily and with minimal interference. Just make sure to adjust the Project Property page to build as a static library (or a DLL with the proper define set in the `Preprocessor` step).

After that, make sure you grab either the single header file release, or just perform a clone of the github repository here and set your include paths up so that you can get at `sol.hpp` somehow. Note that we also have the latest version of the single header file with all dependencies included kept in the repository as well. We recommend the single-header-file release, since it's easier to move around, manage and update if you commit it with some form of version control. You can also clone/submodule the repository and then point at the single/sol/sol.hpp on your include files path. Clone with:

```
>>> git clone https://github.com/ThePhD/sol2.git
```

When you're ready, try compiling this short snippet:

Listing 1: test.cpp: the first snippet

```
#include <sol.hpp> // or #include "sol.hpp", whichever suits your needs

int main (int argc, char* argv[]) {

        sol::state lua;
        lua.open_libraries( sol::lib::base );

        lua.script( "print('bark bark bark!')" );

        return 0;
}
```

Using this simple command line:

```
>>> g++ -std=c++14 test.cpp -I"path/to/lua/include" -L"path/to/lua/lib" -llua
```

Or using your favorite IDE / tool after setting up your include paths and library paths to Lua according to the documentation of the Lua distribution you got. Remember your linked lua library (`-llua`) and include / library paths will depend on your OS, file system, Lua distribution and your installation / compilation method of your Lua distribution.

Note: If you get an avalanche of errors (particularly referring to `auto`), you may not have enabled C++14 / C++17 mode for your compiler. Add one of `std=c++14`, `std=c++1z` OR `std=c++1y` to your compiler options. By default, this is always-on for VC++ compilers in Visual Studio and friends, but g++ and clang++ require a flag (unless you're on GCC 6.0 or better).

If this works, you're ready to start! The first line creates the `lua_State` and will hold onto it for the duration of the scope its declared in (e.g., from the opening `{` to the closing `}`). It will automatically close / cleanup that lua state when it gets destructed.

The second line opens a single lua-provided library, "base". There are several other libraries that come with lua that you can open by default, and those are included in the *sol::lib* enumeration. You can open multiple base libraries by specifying multiple `sol::lib` arguments:

Listing 2: test.cpp: the first snippet

```
#include <sol.hpp>

int main (int argc, char* argv[]) {

        sol::state lua;
        lua.open_libraries( sol::lib::base, sol::lib::coroutine, sol::lib::string,␣
↪sol::lib::io );

        lua.script( "print('bark bark bark!')" );

        return 0;
}
```

If you're interested in integrating Sol with a project that already uses some other library or Lua in the codebase, check out the *existing example* to see how to work with Sol when you add it to a project (the existing example covers `require` as well)!

Note: After you learn the basics of sol, it is usually advised that if you think something can work, you should TRY IT. It will probably work!

Some more ways of loading scripts and handling errors is shown in this example! There is also a full, cross-platform example of loading a DLL.

Next, let's start *reading/writing some variables* from Lua into C++, and vice-versa!

1.2.2 integrating into existing code

If you're already using lua and you just want to use `sol` in some places, you can use `state_view`:

Listing 3: using state_view

```
int something_in_my_system (lua_State* L) {
        // start using Sol with a pre-existing system
        sol::state_view lua(L); // non-owning

        lua.script("print('bark bark bark!')");

        // get the table off the top of the stack
        sol::table expected_table(L, -1);
        // start using it...

        return 0; // or whatever you require of working with a raw function
}
```

sol::state_view is exactly like `sol::state`, but it doesn't manage the lifetime of a `lua_State*`. Therefore, you get all the goodies that come with a `sol::state` without any of the ownership implications. Sol has no initialization components that need to deliberately remain alive for the duration of the program. It's entirely self-containing and uses lua's garbage collectors and various implementation techniques to require no state C++-side. After you do that, all of the power of *Sol* is available to you, and then some!

`sol::state_view` is also helpful when you want to create a DLL that loads some Lua module via requires.

You may also want to call `require` and supply a string of a script file or something that returns an object that you set equal to something in C++. For that, you can use the *require functionality*.

Remember that Sol can be as lightweight as you want it: almost all of Sol's Lua types take the `lua_State*` argument and then a second `int index` stack index argument, meaning you can use *tables*, *lua functions*, *coroutines*, and other reference-derived objects that expose the proper constructor for your use. You can also set *usertypes* and other things you need without changing your entire architecture in one go.

You can even customize it to work with an external Lua wrapper/framework/library.

Note that you can also make non-standard pointer and reference types with custom reference counting and such also play nice with the system. See *unique_usertype_traits<T>* to see how! Custom types is also mentioned in the *customization tutorial*.

There are a few things that creating a `sol::state` does for you. You can read about it *in the sol::state docs* and call those functions directly if you need them.

1.2.3 variables

Working with variables is easy with sol, and behaves pretty much like any associative array / map structure you might have dealt with previously.

reading

Given this lua file that gets loaded into sol:

```
config = {
        fullscreen = false,
        resolution = { x = 1024, y = 768 }
}
```

You can interact with the Lua Virtual Machine like so:

```cpp
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

#include <tuple>
#include "../assert.hpp"
#include <utility> // for std::pair

int main() {

	sol::state lua;
	lua.script_file("variables.lua");
	// the type "sol::state" behaves
	// exactly like a table!
	bool isfullscreen = lua["config"]["fullscreen"]; // can get nested variables
	sol::table config = lua["config"];
	c_assert(isfullscreen);
	return 0;
}
```

From this example, you can see that there's many ways to pull out the varaibles you want. For example, to determine if a nested variable exists or not, you can use `auto` to capture the value of a `table[key]` lookup, and then use the `.valid()` method:

```cpp
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

#include <tuple>
#include "../assert.hpp"
#include <utility> // for std::pair

int main() {

	sol::state lua;
	lua.script_file("variables.lua");

	// test variable
	auto bark = lua["config"]["bark"];
	if (bark.valid()) {
		// branch not taken: config and/or bark are not variables
	}
	else {
		// Branch taken: config and bark are existing variables
	}

	return 0;
}
```

This comes in handy when you want to check if a nested variable exists. You can also check if a toplevel variable is present or not by using `sol::optional`, which also checks if A) the keys you're going into exist and B) the type you're trying to get is of a specific type:

Listing 4: optional lookup

```cpp
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

#include <tuple>
```

(continues on next page)

(continued from previous page)

```
#include "../assert.hpp"
#include <utility> // for std::pair

int main() {

	sol::state lua;
	lua.script_file("variables.lua");

	// can also use optional
	sol::optional<int> not_an_integer = lua["config"]["fullscreen"];
	if (not_an_integer) {
		// Branch not taken: value is not an integer
	}

	sol::optional<bool> is_a_boolean = lua["config"]["fullscreen"];
	if (is_a_boolean) {
		// Branch taken: the value is a boolean
	}

	sol::optional<double> does_not_exist = lua["not_a_variable"];
	if (does_not_exist) {
		// Branch not taken: that variable is not present
	}
	return 0;
}
```

This can come in handy when, even in optimized or release modes, you still want the safety of checking. You can also use the *get_or* methods to, if a certain value may be present but you just want to default the value to something else:

Listing 5: optional lookup

```
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

#include <tuple>
#include "../assert.hpp"
#include <utility> // for std::pair

int main() {

	sol::state lua;
	lua.script_file("variables.lua");
	// this will result in a value of '24'
	// (it tries to get a number, and fullscreen is
	// not a number
	int is_defaulted = lua["config"]["fullscreen"].get_or(24);
	c_assert(is_defaulted == 24);

	// This will result in the value of the config, which is 'false'
	bool is_not_defaulted = lua["config"]["fullscreen"];
	c_assert(!is_not_defaulted);

	return 0;
}
```

That's all it takes to read variables!

writing

Writing gets a lot simpler. Even without scripting a file or a string, you can read and write variables into lua as you please:

```
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

#include <iostream>

int main() {

        sol::state lua;

        // open those basic lua libraries
        // again, for print() and other basic utilities
        lua.open_libraries(sol::lib::base);

        // value in the global table
        lua["bark"] = 50;

        // a table being created in the global table
        lua["some_table"] = lua.create_table_with(
                "key0", 24,
                "key1", 25,
                lua["bark"], "the key is 50 and this string is its value!");

        // Run a plain ol' string of lua code
        // Note you can interact with things set through Sol in C++ with lua!
        // Using a "Raw String Literal" to have multi-line goodness:
        // http://en.cppreference.com/w/cpp/language/string_literal
        lua.script(R"(

        print(some_table[50])
        print(some_table["key0"])
        print(some_table["key1"])

        -- a lua comment: access a global in a lua script with the _G table
        print(_G["bark"])

        )");

        return 0;
}
```

This example pretty much sums up what can be done. Note that the syntax `lua["non_existing_key_1"] = 1` will make that variable, but if you tunnel too deep without first creating a table, the Lua API will panic (e.g., `lua["does_not_exist"]["b"] = 20` will trigger a panic). You can also be lazy with reading / writing values:

```
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

#include <iostream>

int main() {

        sol::state lua;

```

(continues on next page)

(continued from previous page)

```
        auto barkkey = lua["bark"];
        if (barkkey.valid()) {
                // Branch not taken: doesn't exist yet
                std::cout << "How did you get in here, arf?!" << std::endl;
        }

        barkkey = 50;
        if (barkkey.valid()) {
                // Branch taken: value exists!
                std::cout << "Bark Bjork Wan Wan Wan" << std::endl;
        }

        return 0;
}
```

Finally, it's possible to erase a reference/variable by setting it to `nil`, using the constant `sol::nil` in C++:

```
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

int main() {

        sol::state lua;
        lua["bark"] = 50;
        sol::optional<int> x = lua["bark"];
        // x will have a value

        lua["bark"] = sol::nil;
        sol::optional<int> y = lua["bark"];
        // y will not have a value

        return 0;
}
```

It's easy to see that there's a lot of options to do what you want here. But, these are just traditional numbers and strings. What if we want more power, more capabilities than what these limited types can offer us? Let's throw some *functions in there C++ classes into the mix*!

1.2.4 functions and You

Sol can register all kinds of functions. Many are shown in the *quick 'n' dirty*, but here we will discuss many of the additional ways you can register functions into a sol-wrapped Lua system.

Setting a new function

Given a C++ function, you can drop it into Sol in several equivalent ways, working similar to how *setting variables* works:

Listing 6: Registering C++ functions

```
#include <sol.hpp>

std::string my_function( int a, std::string b ) {
        // Create a string with the letter 'D' "a" times,
```

(continues on next page)

(continued from previous page)

```
        // append it to 'b'
        return b + std::string( 'D', a );
}

int main () {

        sol::state lua;

        lua["my_func"] = my_function; // way 1
        lua.set("my_func", my_function); // way 2
        lua.set_function("my_func", my_function); // way 3

        // This function is now accessible as 'my_func' in
        // lua scripts / code run on this state:
        lua.script("some_str = my_func(1, 'Da')");

        // Read out the global variable we stored in 'some_str' in the
        // quick lua code we just executed
        std::string some_str = lua["some_str"];
        // some_str == "DaD"
}
```

The same code works with all sorts of functions, from member function/variable pointers you have on a class as well as lambdas:

Listing 7: Registering C++ member functions

```
struct my_class {
        int a = 0;

        my_class(int x) : a(x) {

        }

        int func() {
                ++a; // increment a by 1
                return a;
        }
};

int main () {

        sol::state lua;

        // Here, we are binding the member function and a class instance: it will
↪call the function on
        // the given class instance
        lua.set_function("my_class_func", &my_class::func, my_class());

        // We do not pass a class instance here:
        // the function will need you to pass an instance of "my_class" to it
        // in lua to work, as shown below
        lua.set_function("my_class_func_2", &my_class::func);

        // With a pre-bound instance:
        lua.script(R"(
```

(continues on next page)

(continued from previous page)

```
                first_value = my_class_func()
                second_value = my_class_func()
        )");
        // first_value == 1
        // second_value == 2

        // With no bound instance:
        lua.set("obj", my_class(24));
        // Calls "func" on the class instance
        // referenced by "obj" in Lua
        lua.script(R"(
                third_value = my_class_func_2(obj)
                fourth_value = my_class_func_2(obj)
        )");
        // first_value == 25
        // second_value == 26
}
```

Member class functions and member class variables will both be turned into functions when set in this manner. You can get intuitive variable with the `obj.a = value` access after this section when you learn about *usertypes to have C++ in Lua*, but for now we're just dealing with functions!

Another question a lot of people have is about function templates. Function templates – member functions or free functions – cannot be registered because they do not exist until you instantiate them in C++. Therefore, given a templated function such as:

Listing 8: A C++ templated function

```
template <typename A, typename B>
auto my_add( A a, B b ) {
        return a + b;
}
```

You must specify all the template arguments in order to bind and use it, like so:

Listing 9: Registering function template instantiations

```
int main () {

        sol::state lua;

        // adds 2 integers
        lua["my_int_add"] = my_add<int, int>;

        // concatenates 2 strings
        lua["my_string_combine"] = my_add<std::string, std::string>;

        lua.script("my_num = my_int_add(1, 2)");
        int my_num = lua["my_num"];
        // my_num == 3

        lua.script("my_str = my_string_combine('bark bark', ' woof woof')");
        std::string my_str = lua["my_str"];
        // my_str == "bark bark woof woof"
}
```

Notice here that we bind two separate functions. What if we wanted to bind only one function, but have it behave

differently based on what arguments it is called with? This is called Overloading, and it can be done with *sol::overload* like so:

Listing 10: Registering C++ function template instantiations

```
int main () {

        sol::state lua;

        // adds 2 integers
        lua["my_combine"] = sol::overload( my_add<int, int>, my_add<std::string,␣
↪std::string> );

        lua.script("my_num = my_combine(1, 2)");
        lua.script("my_str = my_combine('bark bark', ' woof woof')");
        int my_num = lua["my_num"];
        std::string my_str = lua["my_str"];
        // my_num == 3
        // my_str == "bark bark woof woof"
}
```

This is useful for functions which can take multiple types and need to behave differently based on those types. You can set as many overloads as you want, and they can be of many different types.

As a side note, binding functions with default parameters does not magically bind multiple versions of the function to be called with the default parameters. You must instead use *sol::overload*.

As a side note, please make sure to understand Make sure you understand the *implications of binding a lambda/callable struct in the various ways* and what it means for your code!

Getting a function from Lua

There are 2 ways to get a function from Lua. One is with *sol::function* and the other is a more advanced wrapper with *sol::protected_function*. Use them to retrieve callables from Lua and call the underlying function, in two ways:

Listing 11: Retrieving a sol::function

```
int main () {

        sol::state lua;

        lua.script(R"(
                function f (a)
                        return a + 5
                end
        )");

        // Get and immediately call
        int x = lua["f"](30);
        // x == 35

        // Store it into a variable first, then call
        sol::function f = lua["f"];
        int y = f(20);
        // y == 25
}
```

You can get anything that's a callable in Lua, including C++ functions you bind using `set_function` or similar.

`sol::protected_function` behaves similarly to `sol::function`, but has a *error_handler* variable you can set to a Lua function. This catches all errors and runs them through the error-handling function:

Listing 12: Retrieving a sol::protected_function

```
int main () {
	sol::state lua;

	lua.script(R"(
		function handler (message)
			return "Handled this message: " .. message
		end

		function f (a)
			if a < 0 then
				error("negative number detected")
			end
			return a + 5
		end
	)");

	sol::protected_function f = lua["f"];
	f.error_handler = lua["handler"];

	sol::protected_function_result result = f(-500);
	if (result.valid()) {
		// Call succeeded
		int x = result;
	}
	else {
		// Call failed
		sol::error err = result;
		std::string what = err.what();
		// 'what' Should read
		// "Handled this message: negative number detected"
	}
}
```

Multiple returns to and from Lua

You can return multiple items to and from Lua using `std::tuple`/`std::pair` classes provided by C++. These enable you to also use *sol::tie* to set return values into pre-declared items. To recieve multiple returns, just ask for a `std::tuple` type from the result of a function's computation, or `sol::tie` a bunch of pre-declared variables together and set the result equal to that:

Listing 13: Multiple returns from Lua

```
int main () {
	sol::state lua;

	lua.script("function f (a, b, c) return a, b, c end");

	std::tuple<int, int, int> result;
	result = lua["f"](1, 2, 3);
	// result == { 1, 2, 3 }
	int a, int b;
```

(continues on next page)

(continued from previous page)

```
        std::string c;
        sol::tie( a, b, c ) = lua["f"](1, 2, "bark");
        // a == 1
        // b == 2
        // c == "bark"
}
```

You can also return mutiple items yourself from a C++-bound function. Here, we're going to bind a C++ lambda into Lua, and then call it through Lua and get a `std::tuple` out on the other side:

Listing 14: Multiple returns into Lua

```
int main () {
        sol::state lua;

        lua["f"] = [](int a, int b, sol::object c) {
                // sol::object can be anything here: just pass it through
                return std::make_tuple( a, b, c );
        };

        std::tuple<int, int, int> result = lua["f"](1, 2, 3);
        // result == { 1, 2, 3 }

        std::tuple<int, int, std::string> result2;
        result2 = lua["f"](1, 2, "Arf?")
        // result2 == { 1, 2, "Arf?" }

        int a, int b;
        std::string c;
        sol::tie( a, b, c ) = lua["f"](1, 2, "meow");
        // a == 1
        // b == 2
        // c == "meow"
}
```

Note here that we use *sol::object* to transport through "any value" that can come from Lua. You can also use `sol::make_object` to create an object from some value, so that it can be returned into Lua as well.

Any return to and from Lua

It was hinted at in the previous code example, but `sol::object` is a good way to pass "any type" back into Lua (while we all wait for `std::variant<...>` to get implemented and shipped by C++ compiler/library implementers).

It can be used like so, inconjunction with `sol::this_state`:

Listing 15: Return anything into Lua

```
sol::object fancy_func (sol::object a, sol::object b, sol::this_state s) {
        sol::state_view lua(s);
        if (a.is<int>() && b.is<int>()) {
                return sol::make_object(lua, a.as<int>() + b.as<int>());
        }
        else if (a.is<bool>()) {
                bool do_triple = a.as<bool>();
```

(continues on next page)

(continued from previous page)

```
                return sol::make_object(lua, b.as<double>() * ( do_triple ? 3 : 1 ) );
        }
        return sol::make_object(lua, sol::nil);
}

int main () {
        sol::state lua;

        lua["f"] = fancy_func;

        int result = lua["f"](1, 2);
        // result == 3
        double result2 = lua["f"](false, 2.5);
        // result2 == 2.5

        // call in Lua, get result
        lua.script("result3 = f(true, 5.5)");
        double result3 = lua["result3"];
        // result3 == 16.5
}
```

This covers almost everything you need to know about Functions and how they interact with Sol. For some advanced tricks and neat things, check out *sol::this_state* and *sol::variadic_args*. The next stop in this tutorial is about *C++ types (usertypes) in Lua*! If you need a bit more information about functions in the C++ side and how to best utilize arguments from C++, see *this note*.

1.2.5 C++ in Lua

Using user defined types ("usertype"s, or just "udt"s) is simple with Sol. If you don't call any member variables or functions, then you don't even have to 'register' the usertype at all: just pass it through. But if you want variables and functions on your usertype inside of Lua, you need to register it. We're going to give a short example here that includes a bunch of information on how to work with things.

Take this `player` struct in C++ in a header file:

Listing 16: test_player.hpp

```
struct player {
public:
        int bullets;
        int speed;

        player()
        : player(3, 100) {

        }

        player(int ammo)
        : player(ammo, 100) {

        }

        player(int ammo, int hitpoints)
        : bullets(ammo), hp(hitpoints) {
```

(continues on next page)

(continued from previous page)

```
        }

        void boost () {
                speed += 10;
        }

        bool shoot () {
                if (bullets < 1)
                        return false;
                --bullets;
                return true;
        }

        void set_hp(int value) {
                hp = value;
        }

        int get_hp() const {
                return hp;
        }

private:
        int hp;
};
```

It's a fairly minimal class, but we don't want to have to rewrite this with metatables in Lua. We want this to be part of Lua easily. The following is the Lua code that we'd like to have work properly:

Listing 17: player_script.lua

```
-- call single argument integer constructor
p1 = player.new(2)

-- p2 is still here from being
-- set with lua["p2"] = player(0);
-- in cpp file
local p2shoots = p2:shoot()
assert(not p2shoots)
-- had 0 ammo

-- set variable property setter
p1.hp = 545;
-- get variable through property getter
print(p1.hp);

local did_shoot_1 = p1:shoot()
print(did_shoot_1)
print(p1.bullets)
local did_shoot_2 = p1:shoot()
print(did_shoot_2)
print(p1.bullets)
local did_shoot_3 = p1:shoot()
print(did_shoot_3)

-- can read
print(p1.bullets)
```

(continues on next page)

(continued from previous page)

```
-- would error: is a readonly variable, cannot write
-- p1.bullets = 20

p1:boost()
```

To do this, you bind things using the `new_usertype` and `set_usertype` methods as shown below. These methods are on both *table* and *state(_view)*, but we're going to just use it on `state`:

Listing 18: player_script.cpp

```
#include <sol.hpp>

int main () {
        sol::state lua;

        // note that you can set a
        // userdata before you register a usertype,
        // and it will still carry
        // the right metatable if you register it later

        // set a variable "p2" of type "player" with 0 ammo
        lua["p2"] = player(0);

        // make usertype metatable
        lua.new_usertype<player>( "player",

                // 3 constructors
                sol::constructors<player(), player(int), player(int, int)>(),

                // typical member function that returns a variable
                "shoot", &player::shoot,
                // typical member function
                "boost", &player::boost,

                // gets or set the value using member variable syntax
                "hp", sol::property(&player::get_hp, &player::set_hp),

                // read and write variable
                "speed", &player::speed,
                // can only read from, not write to
                "bullets", sol::readonly( &player::bullets )
        );

        lua.script_file("player_script.lua");
}
```

That script should run fine now, and you can observe and play around with the values. Even more stuff *you can do* is described elsewhere, like initializer functions (private constructors / destructors support), "static" functions callable with `name.my_function( ... )`, and overloaded member functions. You can even bind global variables (even by reference with `std::ref`) with `sol::var`. There's a lot to try out!

This is a powerful way to allow reuse of C++ code from Lua beyond just registering functions, and should get you on your way to having more complex classes and data structures! In the case that you need more customization than just usertypes, however, you can customize Sol to behave more fit to your desires by using the desired *customization and extension structures*.

You can check out this code and more complicated code at the examples directory by looking at the `usertype_`-

prefixed examples.

1.2.6 ownership

You can take a reference to something that exists in Lua by pulling out a *sol::reference* or a *sol::object*:

```
sol::state lua;
lua.open_libraries(sol::lib::base);

lua.script(R"(
obj = "please don't let me die";
)");

sol::object keep_alive = lua["obj"];
lua.script(R"(
obj = nil;
function say(msg)
        print(msg)
end
)");

lua.collect_garbage();

lua["say"](lua["obj"]);
// still accessible here and still alive in Lua
// even though the name was cleared
std::string message = keep_alive.as<std::string>();
std::cout << message << std::endl;

// Can be pushed back into Lua as an argument
// or set to a new name,
// whatever you like!
lua["say"](keep_alive);
```

Sol will not take ownership of raw pointers: raw pointers do not own anything. Sol will not delete raw pointers, because they do not (and are not supposed to) own anything:

```
struct my_type {
        void stuff () {}
};

sol::state lua;

// AAAHHH BAD
// dangling pointer!
lua["my_func"] = []() -> my_type* {
        return new my_type();
};

// AAAHHH!
lua.set("something", new my_type());

// AAAAAAHHH!!!
lua["something_else"] = new my_type();
```

Use/return a `unique_ptr` or `shared_ptr` instead or just return a value:

```
// :ok:
lua["my_func"] = []() -> std::unique_ptr<my_type> {
        return std::make_unique<my_type>();
};

// :ok:
lua["my_func"] = []() -> std::shared_ptr<my_type> {
        return std::make_shared<my_type>();
};

// :ok:
lua["my_func"] = []() -> my_type {
        return my_type();
};

// :ok:
lua.set("something", std::unique_ptr<my_type>(new my_type()));

std::shared_ptr<my_type> my_shared = std::make_shared<my_type>();
// :ok:
lua.set("something_else", my_shared);

auto my_unique = std::make_unique<my_type>();
lua["other_thing"] = std::move(my_unique);
```

If you have something you know is going to last and you just want to give it to Lua as a reference, then it's fine too:

```
// :ok:
lua["my_func"] = []() -> my_type* {
        static my_type mt;
        return &mt;
};
```

Sol can detect `nullptr`, so if you happen to return it there won't be any dangling because a `sol::nil` will be pushed.

```
struct my_type {
        void stuff () {}
};

sol::state lua;

// BUT THIS IS STILL BAD DON'T DO IT AAAHHH BAD
// return a unique_ptr still or something!
lua["my_func"] = []() -> my_type* {
        return nullptr;
};

lua["my_func_2"] = [] () -> std::unique_ptr<my_type> {
        // default-constructs as a nullptr,
        // gets pushed as nil to Lua
        return std::unique_ptr<my_type>();
        // same happens for std::shared_ptr
}

// Acceptable, it will set 'something' to nil
// (and delete it on next GC if there's no more references)
```

(continues on next page)

(continued from previous page)

```
lua.set("something", nullptr);

// Also fine
lua["something_else"] = nullptr;
```

1.2.7 adding your own types

Sometimes, overriding Sol to make it handle certain `struct`'s and `class`'es as something other than just userdata is desirable. The way to do this is to take advantage of the 4 customization points for Sol. These are `sol::lua_size<T>`, `sol::stack::pusher<T, C>`, `sol::stack::getter<T, C>`, `sol::stack::checker<T, sol::type t, C>`.

These are template class/structs, so you'll override them using a technique C++ calls *class/struct specialization*. Below is an example of a struct that gets broken apart into 2 pieces when going in the C++ –> Lua direction, and then pulled back into a struct when going in the Lua –> C++:

```
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

#include <iostream>
#include "assert.hpp"

struct two_things {
        int a;
        bool b;
};

namespace sol {

        // First, the expected size
        // Specialization of a struct
        template <>
        struct lua_size<two_things> : std::integral_constant<int, 2> {};

        // Then, specialize the type
        // this makes sure Sol can return it properly
        template <>
        struct lua_type_of<two_things> : std::integral_constant<sol::type, sol::type::poly> {};

        // Now, specialize various stack structures
        namespace stack {

                template <>
                struct checker<two_things> {
                        template <typename Handler>
                        static bool check(lua_State* L, int index, Handler&& handler, record& tracking) {
                                // indices can be negative to count backwards from the top of the stack,
                                // rather than the bottom up
                                // to deal with this, we adjust the index to
                                // its absolute position using the lua_absindex function
                                int absolute_index = lua_absindex(L, index);
```

(continues on next page)

(continued from previous page)

```
                                // Check first and second second index for being the␣
↪proper types
                                bool success = stack::check<int>(L, absolute_index,␣
↪handler)
                                        && stack::check<bool>(L, absolute_index + 1,␣
↪handler);
                                tracking.use(2);
                                return success;
                        }
                };

                template <>
                struct getter<two_things> {
                        static two_things get(lua_State* L, int index, record&␣
↪tracking) {
                                int absolute_index = lua_absindex(L, index);
                                // Get the first element
                                int a = stack::get<int>(L, absolute_index);
                                // Get the second element,
                                // in the +1 position from the first
                                bool b = stack::get<bool>(L, absolute_index + 1);
                                // we use 2 slots, each of the previous takes 1
                                tracking.use(2);
                                return two_things{ a, b };
                        }
                };

                template <>
                struct pusher<two_things> {
                        static int push(lua_State* L, const two_things& things) {
                                int amount = stack::push(L, things.a);
                                // amount will be 1: int pushes 1 item
                                amount += stack::push(L, things.b);
                                // amount 2 now, since bool pushes a single item
                                // Return 2 things
                                return amount;
                        }
                };

        }
}
```

This is the base formula that you can follow to extend to your own classes. Using it in the rest of the library should then be seamless:

```
int main() {
        std::cout << "=== customization ===" << std::endl;
        std::cout << std::boolalpha;

        sol::state lua;
        lua.open_libraries(sol::lib::base);

        // Create a pass-through style of function
        lua.script("function f ( a, b ) print(a, b) return a, b end");

        // get the function out of Lua
```

(continues on next page)

(continued from previous page)

```
        sol::function f = lua["f"];

        two_things things = f(two_things{ 24, false });
        c_assert(things.a == 24);
        c_assert(things.b == false);
        // things.a == 24
        // things.b == true

        std::cout << "things.a: " << things.a << std::endl;
        std::cout << "things.b: " << things.b << std::endl;
        std::cout << std::endl;

        return 0;
}
```

And that's it!

A few things of note about the implementation: First, there's an auxiliary parameter of type *sol::stack::record* for the getters and checkers. This keeps track of what the last complete operation performed. Since we retrieved 2 members, we use `tracking.use(2);` to indicate that we used 2 stack positions (one for `bool`, one for `int`). The second thing to note here is that we made sure to use the `index` parameter, and then proceeded to add 1 to it for the next one.

You can make something pushable into Lua, but not get-able in the same way if you only specialize one part of the system. If you need to retrieve it (as a return using one or multiple values from Lua), you should specialize the `sol::stack::getter` template class and the `sol::stack::checker` template class. If you need to push it into Lua at some point, then you'll want to specialize the `sol::stack::pusher` template class. The `sol::lua_size` template class trait needs to be specialized for both cases, unless it only pushes 1 item, in which case the default implementation will assume 1.

Note: It is important to note here that the `getter`, `pusher` and `checker` differentiate between a type `T` and a pointer to a type `T*`. This means that if you want to work purely with, say, a `T*` handle that does not have the same semantics as just `T`, you may need to specify checkers/getters/pushers for both `T*` and `T`. The checkers for `T*` forward to the checkers for `T`, but the getter for `T*` does not forward to the getter for `T` (e.g., because of `int*` not being quite the same as `int`).

In general, this is fine since most getters/checkers only use 1 stack point. But, if you're doing more complex nested classes, it would be useful to use `tracking.last` to understand how many stack indices the last getter/checker operation did and increment it by `index + tracking.last` after using a `stack::check<..>( L, index, tracking)` call.

You can read more about the structs themselves *over on the API page for stack*, and if there's something that goes wrong or you have anymore questions, please feel free to drop a line on the Github Issues page or send an e-mail!

1.3 errors

1.3.1 how to handle exceptions or other errors

Here is some advice and some tricks for common errors about iteration, compile time / linker errors, and other pitfalls, especially when dealing with thrown exceptions, error conditions and the like in Sol.

1.3.2 Running Scripts

Scripts can have syntax errors, can load from the file system wrong, or have runtime issues. Knowing which one can be troublesome. There are various small building blocks to load and run code, but to check errors you can use the overloaded *script/script_file functions on sol::state/sol::state_view*, specifically the `safe_script` variants. These also take an error callback that is called only when something goes wrong, and Sol comes with some default error handlers in the form of `sol::script_default_on_error` and `sol::script_pass_on_error`.

1.3.3 Compiler Errors / Warnings

A myriad of compiler errors can occur when something goes wrong. Here is some basic advice about working with these types:

- If there are a myriad of errors relating to `std::index_sequence`, type traits, and other `std::` members, it is likely you have not turned on your C++14 switch for your compiler. Visual Studio 2015 turns these on by default, but g++ and clang++ do not have them as defaults and you should pass the flag `--std=c++1y` or `--std=c++14`, or similar for your compiler.
- If you are pushing a non-primitive type into Lua, you may get strange errors about initializer lists or being unable to initializer a `luaL_Reg`. This may be due to *automatic function and operator registration*. *Disabling it* may help.
- Sometimes, a generated usertype can be very long if you are binding a lot of member functions. You may end up with a myriad of warnings about debug symbols being cut off or about `__LINE_VAR` exceeding maximum length. You can silence these warnings safely for some compilers.
- Template depth errors may also be a problem on earlier versions of clang++ and g++. Use `-ftemplate-depth` compiler flag and specify really high number (something like 2048 or even double that amount) to let the compiler work freely.
- When using usertype templates extensively, MSVC may invoke compiler error C1128 , which is solved by using the /bigobj compilation flag.
- If you have a move-only type, that type may need to be made `readonly` if it is bound as a member variable on a usertype or bound using `state_view::set_function`. See *sol::readonly* for more details.
- Assigning a `std::string` or a `std::pair<T1, T2>` using `operator=` after it's been constructed can result in compiler errors when working with `sol::function` and its results. See this issue for fixes to this behavior.
- Sometimes, using `__stdcall` in a 32-bit (x86) environment on VC++ can cause problems binding functions because of a compiler bug. We have a prelimanry fix in, but if it doesn't work and there are still problems: put the function in a `std::function` to make the compiler errors and other problems go away. Also see this __stdcall issue report for more details.
- If you are using `/std:c++latest` on VC++ and get a number of errors for `noexcept` specifiers on functions, you may need to file an issue or wait for the next release of the VC++ compiler.

1.3.4 Mac OSX Crashes

On LuaJIT, your code may crash at seemingly random points when using Mac OSX. Make sure that your build has these flags, as advised by the LuaJIT website:

```
-pagezero_size 10000 -image_base 100000000
```

These will allow your code to run properly, without crashing arbitrarily. Please read the LuaJIT documentation on compiling and running with LuaJIT for more information.

1.3.5 "compiler out of heap space"

Typical of Visual Studio, the compiler will complain that it is out of heap space because Visual Studio defaults to using the x86 (32-bit) version of itself (it will still compile x86 or x64 or ARM binaries, just the compiler **itself** is a 32-bit executable). In order to get around heap space requirements, add the following statement in your `.vcoxproj` files under the `<Import ... />` statement, as instructed by OrfeasZ in this issue:

```
<PropertyGroup>
        <PreferredToolArchitecture>x64</PreferredToolArchitecture>
</PropertyGroup>
```

This should use the 64-bit tools by default, and increase your maximum heap space to whatever a 64-bit windows machine can handle. If you do not have more than 4 GB of RAM, or you still encounter issues, you should look into using `create_simple_usertype` and adding functions 1 by 1 using `.set( ... )`, as shown in the simple usertype example here.

1.3.6 Linker Errors

There are lots of reasons for compiler linker errors. A common one is not knowing that you've compiled the Lua library as C++: when building with C++, it is important to note that every typical (static or dynamic) library expects the C calling convention to be used and that Sol includes the code using `extern 'C'` where applicable.

However, when the target Lua library is compiled with C++, one must change the calling convention and name mangling scheme by getting rid of the `extern 'C'` block. This can be achieved by adding `#define SOL_USING_CXX_LUA` before including sol2, or by adding it to your compilation's command line. If you build LuaJIT in C++ mode (how you would even, is beyond me), then you need to `#define SOL_USING_CXX_LUAJIT` as well. Typically, there is never a need to use this last one.

Note that you should not be defining these with standard builds of either Lua or LuaJIT. See the *config page* for more details.

1.3.7 "caught (...) exception" errors

Sometimes, you expect properly written errors and instead receive an error about catching a `...` exception instead. This might mean that you either built Lua as C++ or are using a framework like LuaJIT that has full interopability support for exceptions on certain system types (x64 for LuaJIT 2.0.5, x86 and x64 on LuaJIT 2.1.x-beta and later).

Please make sure to use the `SOL_EXCEPTIONS_SAFE_PROPAGATION` define before including sol2 to make this work out. You can read more *at the exception page here*.

1.3.8 Catch and CRASH!

By default, Sol will add a `default_at_panic` handler to states opened by Sol (see *sol::state automatic handlers* for more details). If exceptions are not turned off, this handler will throw to allow the user a chance to recover. However, in almost all cases, when Lua calls `lua_atpanic` and hits this function, it means that something *irreversibly wrong* occured in your code or the Lua code and the VM is in an unpredictable or dead state. Catching an error thrown from the default handler and then proceeding as if things are cleaned up or okay is NOT the best idea. Unexpected bugs in optimized and release mode builds can result, among other serious issues.

It is preferred if you catch an error that you log what happened, terminate the Lua VM as soon as possible, and then crash if your application cannot handle spinning up a new Lua state. Catching can be done, but you should understand the risks of what you're doing when you do it. For more information about catching exceptions, the potentials, not turning off exceptions and other tricks and caveats, read about *exceptions in Sol here*.

Lua is a C API first and foremost: exceptions bubbling out of it is essentially last-ditch, terminal behavior that the VM does not expect. You can see an example of handling a panic on the exceptions page *here*. This means that setting up a `try { ... } catch (...) {}` around an unprotected sol2 function or script call is **NOT** enough to keep the VM in a clean state. Lua does not understand exceptions and throwing them results in undefined behavior if they bubble through the C API once and then the state is used again. Please catch, and crash.

Furthermore, it would be a great idea for you to use the safety features talked about *safety section*, especially for those related to functions.

1.3.9 Destructors and Safety

Another issue is that Lua is a C API. It uses `setjmp` and `longjmp` to jump out of code when an error occurs. This means it will ignore destructors in your code if you use the library or the underlying Lua VM improperly. To solve this issue, build Lua as C++. When a Lua VM error occurs and `lua_error` is triggered, it raises it as an exception which will provoke proper unwinding semantics.

Building Lua as C++ gets around this issue, and allows lua-thrown errors to properly stack unwind.

1.3.10 Protected Functions and Access

By default, *sol::function* assumes the code ran just fine and there are no problems. *sol::state(_view)::script(_file)* also assumes that code ran just fine. Use *sol::protected_function* to have function access where you can check if things worked out. Use *sol::optional* to get a value safely from Lua. Use *sol::state(_view)::do_string/do_file/load/load_file* to safely load and get results from a script. The defaults are provided to be simple and fast with thrown exceptions to violently crash the VM in case things go wrong.

1.3.11 Protected Functions Are Not Catch All

Sometimes, some scripts load poorly. Even if you protect the function call, the actual file loading or file execution will be bad, in which case *sol::protected_function* will not save you. Make sure you register your own panic handler so you can catch errors, or follow the advice of the catch + crash behavior above. Remember that you can also bind your own functions and forego sol2's built-in protections for you own by binding a *raw lua_CFunction function*

1.3.12 Iteration

Tables may have other junk on them that makes iterating through their numeric part difficult when using a bland `for-each` loop, or when calling sol's `for_each` function. Use a numeric look to iterate through a table. Iteration does not iterate in any defined order also: see *this note in the table documentation for more explanation*.

1.4 supported compilers, binary size, compile time

1.4.1 getting good final product out of sol

1.4.2 supported compilers

GCC 7.x is now out alongside Visual Studio 2018. This means that sol release v2.20.1 is the current version of the code targeted at the older compilers not listed below. Newer code will be targeted at working with the following compilers and leveraging their features, possibly taking advantage of whatever C++17 features are made available by the compilers and standard libraries bundled by-default with them.

`v2.20.1` supports:

- **VC++**
  - Visual Studio 2018
  - Visual Studio 2015 (Latest updates)
- **GCC (includes MinGW)**
  - v7.x
  - v6.x
  - v5.x
  - v4.8+
- **Clang**
  - v4.x
  - v3.9.x
  - v3.8.x
  - v3.7.x
  - v3.6.x
  - Note: this applies to XCode's Apple Clang as well, but that compiler packs its own deficiencies and problems as well

This does not mean we are immediately abandoning older compilers. We will update this page as relevant bugfixes are backported to the v2.x.x releases. Remember that sol2 is feature-complete: there is nothing more we can add to the library at this time with C++11/C++14 compiler support, so your code will be covered for a long time to come.

Newer features will be targeted at the following compilers:

- **VC++**
  - Visual Studio vNext
  - Visual Studio 2018
- **GCC (includes MinGW)**
  - v8.x
  - v7.x
- **Clang**
  - v7.x
  - v6.x
  - v5.x
  - v4.x
  - v3.9.x

Note that Visual Studio's 2018 Community Edition is absolutely free now, and installs faster and easier than ever before. It also removes a lot of hacky work arounds and formally supports decltype SFINAE.

MinGW's GCC version 7.x of the compiler fixes a long-standing derp in the <codecvt> header that swapped the endianness of utf16 and utf32 strings.

Clang 3.4, 3.5 and 3.6 have many bugs we have run into when developing sol2 and that have negatively impacted users for a long time now.

We encourage all users to upgrade immediately. If you need old code for some reason, use sol release v2.20.1: otherwise, always grab sol2's latest.

1.4.3 feature support

track future compiler and feature support in this issue here.

1.4.4 supported Lua version

We support:

- Lua 5.3+
- Lua 5.2
- Lua 5.1
- LuaJIT 2.0.x+
- LuaJIT 2.1.x-beta3+

1.4.5 binary sizes

For individiauls who use *usertypes* a lot, they can find their compilation times increase. This is due to C++11 and C++14 not having very good facilities for handling template parameters and variadic template parameters. There are a few things in cutting-edge C++17 and C++Next that sol can use, but the problem is many people cannot work with the latest and greatest: therefore, we have to use older techniques that result in a fair amount of redundant function specializations that can be subject to the pickiness of the compiler's inlining and other such techniques.

1.4.6 compile speed improvemements

Here are some notes on achieving better compile times without sacrificing too much performance:

- **When you bind lots of usertypes, put them all in a *single* translation unit (one C++ file) so that it is not recompiled multipl**
  - Remember that the usertype binding ends up being serialized into the Lua state, so you never need them to appear in a header and cause that same compilation overhead for every compiled unit in your project.
- **Consider placing groups of bindings in multiple different translation units (multiple C++ source files) so that only part of**
  - Avoid putting your bindings into headers: it *will* slow down your compilation
- If you are developing a shared library, restrict your overall surface area by specifically and explicitly marking functions as visible and exported and leaving everything else as hidden or invisible by default
- For people who already have a tool that retrieves function signatures and arguments, it might be in your best interest to hook into that tool or generator and dump out the information once using sol2's lower-level abstractions. An issue describing preliminary steps can be found here.

1.4.7 next steps

The next step for Sol from a developer standpoint is to formally make the library a C++17 one. This would mean using Fold Expressions and several other things which will reduce compilation time drastically. Unfortunately, that means also boosting compiler requirements. While most wouldn't care, others are very slow to upgrade: finding the balance is difficult, and often we have to opt for backwards compatibility and fixes for bad / older compilers (of which there are many in the codebase already).

Hopefully, as things progress, we move things forward.

1.5 features

what does Sol (and other libraries) support?

The goal of Sol is to provide an incredibly clean API that provides high performance (comparable or better than the C it was written on) and extreme ease of use. That is, users should be able to say: "this works pretty much how I expected it to."

For the hard technical components of Lua and its ecosystem we support, here is the full rundown:

1.5.1 what Sol supports

- Support for Lua 5.1, 5.2, and 5.3+ and LuaJIT 2.0.4 + 2.1.x-beta3+. We achieve this through our *compatibility* header.
- ***Table*** **support: setting values, getting values of multiple (different) types**
  - ***Lazy evaluation*** **for nested/chained queries** `table["a"]["b"]["c"] = 24;`
  - **Implicit conversion to the types you want** `double b = table["computed_value"];`
- *yielding* support: tag a function as whose return is meant to yield into a coroutine
- ***Optional*** **support: setting values, getting values of multiple (different) types**
  - ***Lazy evaluation*** **for nested/chained queries** `optional<int> maybe_number = table["a"]["b"]["invalid_key"];`
  - Turns on safety when you want it: speed when you don't
- **Support for callables (functions, lambdas, member functions)**
  - Pull out any Lua function with *sol::function*: `sol::function fx = table["socket_send"];`
  - Can also set callables into *operator[] proxies*: `table["move_dude"] = &engine::move_dude;`
  - **Safety: use *sol::protected_function* to catch any kind of error**
    - ANY kind: C++ exception or Lua erors are trapped and run through the optional `error_handler` variable
  - *Advanced: overloading of a single function name* so you don't need to do boring typechecks
  - *Advanced: efficient handling and well-documented* way of dealing with arguments
- **User-Defined Type (*sol::usertype* in the API) support:**
  - Set member functions to be called

  - Set member variables
  - Set variables on a class that are based on setter/getter functions using properties
  - Use free-functions that take the Type as a first argument (pointer or reference)
  - Support for "Factory" classes that do not expose constructor or destructor
  - Modifying memory of userdata in C++ directly affects Lua without copying, and
  - **Modifying userdata in Lua directly affects C++ references/pointers** `my_class& a = table["a"]; my_class* a_ptr = table["a"];`
  - **If you want a copy, just use value semantics and get copies:** `my_class a = table["a"];`
- **Thread/Coroutine support**
  - Use, resume, and play with *coroutines* like regular functions
  - Get and use them even on a separate Lua *thread*
  - Monitor status and get check errors
- *Advanced:* Customizable and extensible to your own types if you override *getter/pusher/checker* template struct definitions.

1.5.2 The Feature Matrix™

The below feature table checks for the presence of something. It, however, does not actually account for any kind of laborious syntax.

✓ full support: works as you'd expect (operator[] on tables, etc…)

~ partial support / wonky support: this means its either supported through some other fashion (not with the desired syntax, serious caveats, etc.). Sometimes means dropping down to use the plain C API (at which point, what was the point of the abstraction?).

no support: feature doesn't work or, if it's there, it REALLY sucks to use

Implementation notes from using the libraries are below the tables.

1.5.3 category explanations

Explanations for a few categories are below (rest are self-explanatory).

- optional: Support for getting an element, or potentially not (and not forcing the default construction of what amounts to a bogus/dead object). Usually comes with `std(::experimental)::optional`. It's a fairly new class, so a hand-rolled class internal to the library with similar semantics is also acceptable
- tables: Some sort of abstraction for dealing with tables. Ideal support is `mytable["some_key"] = value`, and everything that the syntax implies.
- table chaining: In conjunction with tables, having the ability to query deeply into tables `mytable["key1"]["key2"]["key3"]`. Note that this becomes a tripping point for some libraries: crashing if `"key1"` doesn't exist while trying to access `"key2"` (Sol avoids this specifically when you use `sol::optional`), and sometimes it's also a heavy performance bottleneck as expressions are not lazy-evaluated by a library.
- arbitrary keys: Letting C++ code use userdata, other tables, integers, etc. as keys for into a table.
- user-defined types (udts): C++ types given form and function in Lua code.

- udts - member functions: C++ member functions on a type, usually callable with `my_object:foo(1)` or similar in Lua.
- udts - table variables: C++ member variables/properties, manipulated by `my_object.var = 24` and in Lua
- function binding: Support for binding all types of functions. Lambdas, member functions, free functions, in different contexts, etc...
- protected function: Use of `lua_pcall` to call a function, which offers error-handling and trampolining (as well as the ability to opt-in / opt-out of this behavior)
- multi-return: returning multiple values from and to Lua (generally through `std::tuple<...>` or in some other way)
- variadic/variant argument: being able to accept "anything" from Lua, and even return "anything" to Lua (`object` abstraction, variadic arguments, etc...)
- inheritance: allowing some degree of subtyping or inheritance on classes / userdata from Lua - this generally means that you can retrieve a base pointer from Lua even if you hand the library a derived pointer
- overloading: the ability to call overloaded functions, matched based on arity or type (`foo( 1 )` from lua calls a different function then `foo( "bark" )`).
- Lua thread: basic wrapping of the lua thread API; ties in with coroutine.
- coroutines: allowing a function to be called multiple times, resuming the execution of a Lua coroutine each time
- yielding C++ functions: allowing a function from C++ to be called multiple times, and yield any results it has back through the C API or into Lua
- environments: an abstraction for getting, setting and manipulating an environment, using table techniques, functions or otherwise. Typically for the purposes of sandboxing

	plain C	lu-awrap-per	lua-intf	lu-abind	Se-lene	Sol2	oolua	lua-api-pp	kaguya	SLB3	SWIG	luacp-pinter-face	luwra
optional	~		✓			✓			✓				
tables	~	~	~	✓	✓	✓	~	✓	✓			~	✓
table chain-ing	~	~	~	✓	✓	✓		✓	✓			~	✓
arbitrary keys	~	✓	✓	✓	✓	✓		~	✓				
user-defined types (udts)	~	✓	✓	✓	✓	✓	~	✓	✓	✓	✓	✓	✓
udts: mem-ber functions	~	✓	✓	✓	✓	✓	~	✓	✓	✓	✓	✓	✓
udts: table variables	~	~	~	~	~	✓	~	~	~		✓		~
stack ab-stractions	~	✓	✓	✓	✓	✓	✓	✓	✓	~		~	✓
lua callables from C(++)	~	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	~
function binding	~	✓	✓	✓	✓	✓	~	~	✓	~	~	~	✓
protected call	~		~	~	~	✓	~	✓	~	~	~	~	~
multi-return	~		✓	✓	✓	✓	~	✓	✓	~	✓	~	
vari-adic/variant argument	~	✓	✓	✓	✓	✓	~	✓	✓	~	~	~	
inheritance	~	✓	✓	✓	✓	✓	~	~	✓	~	✓	~	
overloading	~		✓			✓			✓	✓	✓		
Lua thread	~		~			✓	✓		✓			✓	
environ-ments						✓							
coroutines	~		~	✓	✓	✓			✓			✓	
yielding C++ func-tions	~	✓	✓	✓	~	✓	~		✓		~	✓	~
no-rtti support	✓		✓			✓	✓		✓	✓	~	✓	✓
no-exception support	✓		✓	~		✓	✓		✓	✓	~	✓	✓
Lua 5.1	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓		✓
Lua 5.2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Lua 5.3	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
luajit	✓	✓	✓	✓	~	✓	✓	✓	✓	✓	✓		✓
distribution	com-pile	header	both	com-pile	header	header	com-pile	com-pile	header	com-pile	gen-er-ated	compile	header

1.5.4 notes on implementations

Plain C -

- Obviously you can do anything you want with Plain C, but the effort involved is astronomical in comparison to what other wrappers, libraries and frameworks offer
- Does not scale very well (in terms of developer ease of use)
- Compilation (or package manager use) is obviously required for your platform and required to use ANY of these libraries, but that's okay because all libraries need some version of Lua anyways, so you always have this!

kaguya -

- Table variables / member variables are automatically turned into `obj:x( value )` to set and `obj:x()` to get
- Has optional support
- Inspired coroutine support for Sol
- Library author (satoren) is a nice guy!
- C++11/14, or boostified (which makes it C++03 compatible)
- Class registration is a bit verbose, but not as offensive as OOLua or lua-intf or others
- Constructor setting syntax is snazzy and good

Sol -

- One of the few libraries with optional support!
- Basically the fastest in almomst all respects: http://sol2.readthedocs.io/en/latest/benchmarks.html
- Overloading support can get messy with inheritance, see *here*
- C++14/"C++1y" (-std=c++14, -std=c++1y, =std=c++1z) flags are used (available since GCC 4.9 and Clang 3.5)
- Active issues, active individuals
- Deserves lots of love!

lua-intf -

- Can be both header-only or compiled
- Has optional support
- C++11
- Macro-based registration (strange pseudo-language)
- Fairly fast in most regards
- Registering classes/"modules" in using C++ code is extremely verbose
- In order to chain lookups, one has to glue the keys together (e.g. `"mykey.mykey2"`) on the `operator[]` lookup (e.g., you can't nest them arbitrarily, you have to pre-compose the proper lookup string) (fails miserably for non-string lookups!).
- Not too shabby!

Selene -

- Table variables / member variables are automatically turned into `obj:set_x( value )` to set and `obj:x()` to get
- Registering classes/"modules" using C++ code is extremely verbose, similar to lua-intf's style

- Eats crap when it comes to performance, most of the time (see *benchmarks*)
- Lots of users (blogpost etc. made it popular), but the Repository is kinda stagnant...

luawrapper -

- Takes the approach of writing and reading tables using `readVariable` and `writeVariable` functions
- C++11, no macros!
- The interface can be clunky (no table-like data structures: most things go though `readVariable` / `writeVariable`)
- Internal Compiler errors in Visual Studio 2015 - submitted a PR to fix it, hopefully it'll get picked up

SWIG (3.0) -

- Very comprehensive for binding concepts of C++ (classes, variables, etc.) to Lua
- Helps with literally nothing else (tables, threads, stack abstractions, etc.)
- Not really a good, full-featured Library...
- Requires preprocessing step (but it's not a... TERRIBLY complicated preprocessing step); some boilerplate in writing additional classes that you've already declared

luacppinterface -

- The branch that fixes VC++ warnings and introduces some new work has type checker issues, so use the stable branch only
- No table variable support
- Actually has tables (but no operator[])
- Does not support arbitrary keys

luabind -

- One of the older frameworks, but has many people updating it and providing "deboostified" versions
- **Strange in-lua keywords and parsing to allow for classes to be written in lua**
  - not sure if good feature; vendor lock-in to that library to depend on this specific class syntax?
- Comprehensive lua bindings (can even bind "properties")
- There's some code that produces an ICE in Visual C++: I submitted a fix to the library in the hopes that it'll get accepted
- Wonky table support: no basic conversion functions on `luabind::object`; have to push object then use lua API to get what you want

lua-api-pp -

- Compiled, but the recommendation is to add the source files directly to your project
- **Userdata registration with thick setup-macros: LUAPP_USERDATA(...) plus a bunch of free functions that take a `T&` s**
  - You can bind member functions directly but only if you override metatable entries
  - Otherwise, COMPLICATED self-registration that makes you wonder why you're using the framework
- **You have to create a context and then call it to start accessing the lua state (adding more boilerplate... thanks)**
  - Thankfully, unlike many libraries, it actually has a Table type that can be used semi-easily. FINALLY

- C++11-ish in some regards
- Sad face, thanks to the way userdata registration is handled

SLB3 -

- Old code exported to github from dying google code
- ".NET Style" - to override functionality, derive from class – boilerplate (isn't that what we're trying to get rid of?)
- Pointers everywhere: ownership semantics unclear
- Piss-poor documentation, ugh!
- Least favorite to work with, for sure!

oolua -

- The syntax for this library is not my favorite... go read the docs, decide for yourself!
- **The worst in terms of how to use it: may have docs, but the DSL is extraordinarily crappy with thick, hard-to-debug/hard**
  - Same problem as lua-api-pp: cannot have the declaration macros anywhere but the toplevel namespace because of template declaration macro
- Supports not having exceptions or rtti turned on (shiny!)
- **Poor RAII support: default-construct-and-get style (requires some form of initalization to perform a `get` of an object, an**
  - The library author has informed me that he does personally advises individuals do not use the `Table` abstraction in OOLua... Do I likewise tell people to consider its table abstractions defunct?
- Table variables / member variables from C++ are turned into function calls (`get_x` and `set_x` by default)

luwra -

- How do you store stateful functors / lambas? So far, no support for such.
- Cannot pull functions without first leaving them on the stack: manual cleanup becomes a thing
- Doesn't understand `std::function` conversions and the like (but with some extra code can get it to work)
- Recently improved by a lot: can chain tables and such, even if performance is a bit sad for that use case
- When you do manage to set function calls with the macros they are fast (can a template solution do just as good? Sol is going to find out!)
- No table variable support - get turned into getter/setter functions, similar to kaguya
- Table variables become class statics (surprising)
- Tanks in later MSVCs

1.6 functions

working with functions in sol2

There are a number of examples dealing with functions and how they can be bound to sol2:

- For a quicker walkthrough that demonstrates almost everything, see the examples and the *the quick and dirty tutorial*
- For a full explanation, *read the tutorial* and consult the subjects below

- If you have bindings and set-ups that want to leverage the C API without sol2's interference, you can push a raw function, which has certain implications (noted *below*)
- **Return multiple values into Lua by:**
  - returning a `std::tuple`
  - using *sol::variadic_results*
- ***Overload function calls with different argument types and count on a single name*** **(first-bound, first-serve overloading)**
  - Note: because of this feature, automatic number to string conversion from Lua is not permitted for overloads and does not work when safeties are turned on
  - int/float overloads must have `SOL_SAFE_NUMERICS` turned on to differentiate between the two
  - Use C++ captures and lambdas to bind member functions tied to a single object /
- **You can work with transparent arguments that provide you with special information, such as**
  - *sol::variadic_args*, for handling variable number of arguments at runtime
  - *sol::this_state*, for getting the current Lua state
  - *sol::this_environment*, for potentially retrieving the current Lua environment
- Control serialization of arguments and return types with *sol::nested*, *sol::as_table*, *sol::as_args* and *sol::as_function*
- Set environments for Lua functions and scripts with *sol::environment*
- You can use *filters* to control dependencies and streamline return values, as well as apply custom behavior to a functions return

1.6.1 working with callables/lambdas

To be explicit about wanting a struct to be interpreted as a function, use `my_table.set_function( key, func_value );`. You can also use the *sol::as_function* call, which will wrap and identify your type as a function.

Note: When you set lambdas/callables through `my_table.set( ... )` using the same function signature, you can suffer from `const static` data (like string literals) from not "behaving properly". This is because some compilers do not provide unique type names that we can get at in C++ with RTTI disabled, and thusly it will register the first lambda of the specific signature as the one that will be called. The result is that string literals and other data stored in an compiler implementation-defined manner might be folded and the wrong routine run, even if other observable side effects are nice.

To avoid this problem, register all your lambdas with *my_table.set_function* and avoid the nightmare altogether.

Furthermore, it is important to know that lambdas without a specified return type (and a non-const, non-reference-qualified `auto`) will decay return values. To capture or return references explicitly, use `decltype(auto)` or specify the return type **exactly** as desired:

```
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

#include "../assert.hpp"

int main(int, char*[]) {
```

(continues on next page)

(continued from previous page)

```
7
8           struct test {
9                   int blah = 0;
10          };
11
12          test t;
13          sol::state lua;
14          lua.set_function("f", [&t]() {
15                  return t;
16          });
17          lua.set_function("g", [&t]() -> test& {
18                  return t;
19          });
20
21          lua.script("t1 = f()");
22          lua.script("t2 = g()");
23
24          test& from_lua_t1 = lua["t1"];
25          test& from_lua_t2 = lua["t2"];
26
27          // not the same: 'f' lambda copied
28          c_assert(&from_lua_t1 != &t);
29          // the same: 'g' lambda returned reference
30          c_assert(&from_lua_t2 == &t);
31
32          return 0;
33  }
```

1.6.2 exception safety/handling

All functions bound to sol2 set up an exception trampoline around the function (unless you are working with a *raw lua_CFunction you pushed yourself*). *protected_function* also has an error handler member and an exception trampoline around its internals, but it is not guaranteed safe if an exception bubbles outside of it. Catching that exception is not safe either: if an exception has exploded out from the sol2 API somehow, you must assume the VM is in some indeterminate and/or busted state.

Please read the *error page* and *exception page* for more details about what to do with exceptions that explode out from the API.

1.6.3 functions and argument passing

All arguments are forwarded. Unlike *get/set/operator[] on sol::state* or *sol::table*, value semantics are not used here. It is forwarding reference semantics, which do not copy/move unless it is specifically done by the receiving functions / specifically done by the user.

Note: This also means that you should pass and receive arguments in certain ways to maximize efficiency. For example, `sol::table`, `sol::object`, `sol::userdata` and friends are cheap to copy, and should simply by taken as values. This includes primitive types like `int` and `double`. However, C++ types – if you do not want copies – should be taken as `const type&` or `type&`, to save on copies if it's important. Note that taking references from Lua also means you can modify the data inside of Lua directly, so be careful. Lua by default deals with things mostly by reference (save for primitive types).

When you bind a function to Lua, please take any pointer arguments as `T*`, unless you specifically know you are going to match the exact type of the unique/shared pointer and the class it wraps. sol2 cannot support "implicit wrapped pointer casting", such as taking a `std::shared_ptr<MySecondBaseClass>` when the function is passed a `std::shared_ptr<MyDerivedClass>`. Sometimes it may work because the compiler might be able to line up your classes in such a way that raw casts work, but this is undefined behavior through and through and sol2 has no mechanisms by which it can make this safe and not blow up in the user's face.

Note: Please avoid taking special unique_usertype arguments, by either reference or value. In many cases, by-value does not work (e.g., with `std::unique_ptr`) because many types are move-only and Lua has no concept of "move" semantics. By-reference is dangerous because sol2 will hand you a reference to the original data: but, any pointers stored in Lua can be invalidated if you call `.reset()` or similar on the core pointer. Please take a pointer (`T*`) if you anticipate `nil`/`nullptr` being passed to your function, or a reference (`const T&` or `T&`) if you do not. As a side note, if you write a small wrapping class that holds a base pointer type, and interact using the wrapper, then when you get the wrapper as an argument in a C++-function bound to Lua you can cast the internal object freely. It is simply a direct cast as an argument to a function that is the problem.

Note: You can get even more speed out of `sol::object` style of types by taking a `sol::stack_object` (or `sol::stack_...`, where `...` is `userdata`, `reference`, `table`, etc.). These reference a stack position directly rather than cheaply/safely the internal Lua reference to make sure it can't be swept out from under you. Note that if you manipulate the stack out from under these objects, they may misbehave, so please do not blow up your Lua stack when working with these types.

`std::string` (and `std::wstring`) are special. Lua stores strings as `const char*` null-terminated strings. `std::string` will copy, so taking a `std::string` by value or by const reference still invokes a copy operation. You can take a `const char*`, but that will mean you're exposed to what happens on the Lua stack (if you change it and start chopping off function arguments from it in your function calls and such, as warned about previously).

1.6.4 function call safety

You can have functions here and on usertypes check to definitely make sure that the types passed to C++ functions are what they're supposed to be by adding a `#define SOL_CHECK_ARGUMENTS` before including Sol, or passing it on the command line. Otherwise, for speed reasons, these checks are only used where absolutely necessary (like discriminating between *overloads*). See *safety* for more information.

1.6.5 raw functions (`lua_CFunction`)

When you push a function into Lua using Sol using any methods and that function exactly matches the signature `int(lua_State*);`, it will be treated as a *raw C function* (a `lua_CFunction`). This means that the usual exception trampoline Sol wraps your other function calls in will not be present. You will be responsible for catching exceptions and handling them before they explode into the C API (and potentially destroy your code). Sol in all other cases adds an exception-handling trampoline that turns exceptions into Lua errors that can be caught by the above-mentioned protected functions and accessors.

Note that stateless lambdas can be converted to a function pointer, so stateless lambdas similar to the form `[](lua_State*) -> int { ... }` will also be pushed as raw functions. If you need to get the Lua state that is calling a function, use *sol::this_state*.

Warning: Do NOT assume that building Lua as C++ will allow you to throw directly from a raw function. If an exception is raised and it bubbles into the Lua framework, even if you compile as C++, Lua does not recognize

exceptions other than the ones that it uses with `lua_error`. In other words, it will return some completely bogus result, potentially leave your Lua stack thrashed, and the rest of your VM *can* be in a semi-trashed state. Please avoid this!

1.7 usertypes

Perhaps the most powerful feature of sol2, `usertypes` are the way sol2 and C++ communicate your classes to the Lua runtime and bind things between both tables and to specific blocks of C++ memory, allowing you to treat Lua userdata and other things like classes.

To learn more about usertypes, visit:

- *the basic tutorial*
- *customization point tutorial*
- *api documentation*
- *memory documentation*

The examples folder also has a number of really great examples for you to see. There are also some notes about guarantees you can find about usertypes, and their associated userdata, below:

- Containers get pushed as special usertypes, but can be disabled if problems arise as detailed *here*.
- Certain operators are detected and bound automatically for usertypes
- You can use bitfields but it requires some finesse on your part. We have an example to help you get started here, that uses a few tricks.
- **All usertypes are runtime extensible in both Lua and C++**
  - If you need dynamic callbacks or runtime overridable functions, have a `std::function` member variable and get/set it on the usertype object
  - `std::function` works as a member variable or in passing as an argument / returning as a value (you can even use it with `sol::property`)
  - You can also create an entirely dynamic object: see the dynamic_object example for more details
- You can use *filters* to control dependencies and streamline return values, as well as apply custom behavior to a functions return
- **You can work with special wrapper types such as `std::unique_ptr<T>`, `std::shared_ptr<T>`, and others by defa**
  - Extend them using the *sol::unique_usertype<T> traits*
  - This allows for custom smart pointers, special pointers, custom handles and others to be given certain handling semantics to ensure proper RAII with Lua's garbage collection
- (Advanced) You can override the iteration function for Lua 5.2 and above (LuaJIT does not have the capability) as shown in the pairs example
- **(Advanced) Interop with `toLua`, `kaguya`, `OOLua`, `LuaBind`, `luwra`, and all other existing libraries by using the stack A**
  - Must turn on `SOL_ENABLE_INTEROP`, as defined in the *configuration and safety documentation*, to use

Note: Note that to use many of sol2's features, such as automatic constructor creation, `sol::property`, and similar, one must pass these things to the usertype as part of its initial creation and grouping of arguments. Attempting to do so afterwards will result in unexpected and wrong behavior, as the system will be missing information it needs. This is because many of these features rely on `__index` and `__newindex` Lua metamethods being overridden and handled in a special way!

Here are some other general advice and tips for understanding and dealing with usertypes:

- **Please note that the colon is necessary to "automatically" pass the `this`/`self` argument to Lua methods**
  - `obj:method_name()` is how you call "member" methods in Lua
  - It is purely syntactic sugar that passes the object name as the first argument to the `method_name` function
  - `my_obj:foo(bar, baz)` is the same as `my_obj.foo(my_obj, bar, baz)`
  - **Please note** that one uses a colon, and the other uses a dot, and forgetting to do this properly will crash your code
  - There are safety defines outlined in the *safety page here*
- **You can push types classified as userdata before you register a usertype.**
  - You can register a usertype with the Lua runtime at any time
  - You can retrieve a usertype from the Lua runtime at any time
  - Methods and properties will be added to the type only after you register the usertype with the Lua runtime
  - All methods and properties will appear on all userdata, even if that object was pushed before the usertype (all userdata will be updated)
- **Types either copy once or move once into the memory location, if it is a value type. If it is a pointer, we store only the refer**
  - This means retrieval of class types (not primitive types like strings or integers) by `T&` or `T*` allow you to modify the data in Lua directly
  - Retrieve a plain `T` to get a copy
  - Return types and passing arguments to `sol::function`-types use perfect forwarding and reference semantics, which means no copies happen unless you specify a value explicitly. See *this note for details*
- **You can set `index` and `new_index` freely on any usertype you like to override the default "if a key is missing, find it / set**
  - `new_index` and `index` will not be called if you try to manipulate the named usertype table directly. sol2's will be called to add that function to the usertype's function/variable lookup table
  - `new_index` and `index` will be called if you attempt to call a key that does not exist on an actual userdata object (the C++ object) itself
  - If you made a usertype named `test`, this means `t = test()`, with `t.hi = 54` will call your function, but `test.hi = function () print ("hi"); end` will instead set the key `hi` to to lookup that function for all `test` types

- The first `sizeof( void* )` bytes is always a pointer to the typed C++ memory. What comes after is based on what you've pushed into the system according to *the memory specification for usertypes*. This is compatible with a number of systems other than just sol2, making it easy to interop with select other Lua systems.
- **Member methods, properties, variables and functions taking `self&` arguments modify data directly**
  - Work on a copy by taking arguments or returning by value.
  - Do not use r-value references: they do not mean anything in Lua code.
  - Move-only types can only be taken by reference: sol2 cannot know if/when to move a value (except when serializing with perfect forwarding *into* Lua, but not calling a C++ function from Lua)
- The actual metatable associated with the usertype has a long name and is defined to be opaque by the Sol implementation.
- The actual metatable inner workings is opaque and defined by the Sol implementation, and there are no internal docs because optimizations on the operations are applied based on heuristics we discover from performance testing the system.

1.8 containers

working with containers in sol2

Containers are objects that are meant to be inspected and iterated and whose job is to typically provide storage to a collection of items. The standard library has several containers of varying types, and all of them have `begin()` and `end()` methods which return iterators. C-style arrays are also containers, and sol2 will detect all of them for use and bestow upon them special properties and functions.

- **Containers from C++ are stored as `userdata` with special `usertype` metatables with *special operations***
  - **In Lua 5.1, this means containers pushed without wrappers like *as_table* and *nested* will not work with `pairs` o**
    - Lua 5.2+ will behave just fine (does not include LuaJIT 2.0.x)
  - You must push containers into C++ by returning them directly and getting/setting them directly, and they will have a type of `sol::type::userdata` and treated like a usertype
- Containers can be manipulated from both C++ and Lua, and, like userdata, will reflect changes if you use a reference to the data.
- **This means containers do not automatically serialize as Lua tables**
  - If you need tables, consider using `sol::as_table` and `sol::nested`
  - See this table serialization example for more details
- Lua 5.1 has different semantics for `pairs` and `ipairs`: be wary. See *examples down below* for more details
- You can override container behavior by overriding *the detection trait* and *specializing the container_traits template*
- You can bind typical C-style arrays, but must follow *the rules*

Note: Please note that c-style arrays must be added to Lua using `lua["my_arr"] = &my_c_array;` or `lua["my_arr"] = std::ref(my_c_array);` to be bestowed these properties. No, a plain `T*` pointer is **not** considered an array. This is important because `lua["my_string"] = "some string";` is also typed as

an array (`const char[n]`) and thusly we can only use `std::reference_wrapper`s or pointers to the actual array types to work for this purpose.

1.8.1 container detection

containers are detected by the type trait `sol::is_container<T>`. If that turns out to be true, sol2 will attempt to push a userdata into Lua for the specified type `T`, and bestow it with some of the functions and properties listed below. These functions and properties are provided by a template struct `sol::container_traits<T>`, which has a number of static Lua C functions bound to a safety metatable. If you want to override the behavior for a specific container, you must first specialize `sol::is_container<T>` to drive from `std::true_type`, then override the functions you want to change. Any function you do not override will call the default implementation or equivalent. The default implementation for unrecognized containers is simply errors.

You can also specialize `sol::is_container<T>` to turn off container detection, if you find it too eager for a type that just happens to have `begin` and `end` functions, like so:

Listing 19: not_container.hpp

```
struct not_container {
        void begin() {

        }

        void end() {

        }
};

namespace sol {
        template <>
        struct is_container<not_container> : std::false_type {};
}
```

This will let the type be pushed as a regular userdata.

Note: Pushing a new *usertype* will prevent a qualifying C++ container type from being treated like a container. To force a type that you've registered/bound as a usertype using `new_usertype` or `new_simple_usertype` to be treated like a container, use *sol::as_container*.

1.8.2 container overriding

If you **want** it to participate as a table, use `std::true_type` instead of `std::false_type` from the *containter detection* example. and provide the appropriate `iterator` and `value_type` definitions on the type. Failure to do so will result in a container whose operations fail by default (or compilation will fail).

If you need a type whose declaration and definition you do not have control over to be a container, then you must override the default behavior by specializing container traits, like so:

Listing 20: specializing.hpp

```
struct not_my_type { ... };
```

(continues on next page)

(continued from previous page)

```
namespace sol {
        template <>
        struct is_container<not_my_type> : std::true_type {};

        template <>
        struct container_traits<not_my_type> {

                ...
                // see below for implemetation details
        };
}
```

The various operations provided by `container_traits<T>` are expected to be like so, below. Ability to override them requires familiarity with the Lua stack and how it operates, as well as knowledge of Lua's *raw C functions*. You can read up on raw C functions by looking at the "Programming in Lua" book. The online version's information about the stack and how to return information is still relevant, and you can combine that by also using sol's low-level *stack API* to achieve whatever behavior you need.

Warning: Exception handling **WILL** be provided around these particular raw C functions, so you do not need to worry about exceptions or errors bubbling through and handling that part. It is specifically handled for you in this specific instance, and **ONLY** in this specific instance. The raw note still applies to every other raw C function you make manually.

1.8.3 container operations

Below are the many container operations and their override points for `container_traits<T>`. Please use these to understand how to use any part of the implementation.

operation	lua syntax	container_traits<T> extension point	stack argument order	notes/caveats
set	`c:set(key, value)`	`static int set(lua_State*);`	1 self 2 key 3 value	• if `value` is nil, it performs an erase in default implementation • if this is a sequence container and it support insertion and `key`,is an index equal to the size of the container,+ 1, it will insert at,the end of the container (this is a Lua idiom)
index_set	`c[key] = value`	`static int index_set(lua_State*);`	1 self 2 key 3 value	• default implementation calls "set" • if this is a sequence container and it support insertion and `key` is an index equal to the size of the container + 1, it will insert at the end of the container (this is a Lua idiom)
at	`v = c:at(key)`	`static int at(lua_State*);`	1 self 2 index	• can return multiple values • default implementation increments iterators linearly for non-random-access • if the type does not have random-access iterators, **do not use this in a for loop** !

Note: If your type does not adequately support `begin()` and `end()` and you cannot override it, use the `sol::is_container` trait override along with a custom implementation of `pairs` on your usertype to get it to work as you want it to. Note that a type not having proper `begin()` and `end()` will not work if you try to forcefully serialize it as a table (this means avoid using *sol::as_table* and *sol::nested*, otherwise you will have compiler errors). Just set it or get it directly, as shown in the examples, to work with the C++ containers.

Note: Overriding the detection traits and operation traits listed above and then trying to use `sol::as_table` or similar can result in compilation failures if you do not have a proper `begin()` or `end()` function on the type. If you want things to behave with special usertype considerations, please do not wrap the container in one of the special table-converting/forcing abstractions.

1.8.4 container classifications

When you serialize a container into sol2, the default container handler deals with the containers by inspecting various properties, functions, and typedefs on them. Here are the broad implications of containers sol2's defaults will recognize, and which already-known containers fall into their categories:

container type	requirements	known containers	notes/caveats
sequence	`erase(iterator)` `push_back/insert(value_type)`	std::vector std::deque std::list std::forward_list	• `find` operation is linear in size of list (searches all elements) • std::forward_list has forward-only iterators: set/find is a linear operation • std::forward_list uses "insert_after" idiom, requires special handling internally
fixed	lacking `push_back/insert` lacking `erase`	std::array<T, n> T[n] (fixed arrays)	• regular c-style arrays must be set with `std::ref( arr )` or `&arr` to be usable
ordered	`key_type` typedef `erase(key)` `find(key)` `insert(key)`	std::set std::multi_set	• `container[key] = stuff` operation erases when `stuff` is nil, inserts/sets when not • `container.get(key)` returns the key itself
associative, ordered	`key_type`, `mapped_type` typedefs `erase(key)` `find(key)` `insert({ key, value })`	std::map std::multi_map	
unordered	same as ordered	std::unordered_set std::unordered_multiset	• `container[key] = stuff` operation erases when `stuff` is nil, inserts/sets when not • `container.get(key)` returns the key itself • iteration not guaranteed to be in order of insertion, just like in C++ container
unordered, associative	same as ordered, associative	std::unordered_map std::unordered_multimap	• iteration not guaranteed to be in order of insertion, just like in C++ con-

1.8.5 a complete example

Here's a complete working example of it working for Lua 5.3 and Lua 5.2, and how you can retrieve out the container in all versions:

```
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

#include <vector>
#include <iostream>

int main(int, char**) {
        std::cout << "=== containers ===" << std::endl;

        sol::state lua;
        lua.open_libraries();

        lua.script(R"(
function f (x)
        print("container has:")
        for k=1,#x do
                v = x[k]
                print("\t", k, v)
        end
        print()
end
        )");

        // Have the function we
        // just defined in Lua
        sol::function f = lua["f"];

        // Set a global variable called
        // "arr" to be a vector of 5 lements
        lua["arr"] = std::vector<int>{ 2, 4, 6, 8, 10 };

        // Call it, see 5 elements
        // printed out
        f(lua["arr"]);

        // Mess with it in C++
        // Containers are stored as userdata, unless you
        // use `sol::as_table()` and `sol::as_table_t`.
        std::vector<int>& reference_to_arr = lua["arr"];
        reference_to_arr.push_back(12);

        // Call it, see *6* elements
        // printed out
        f(lua["arr"]);

        lua.script(R"(
arr:add(28)
        )");

        // Call it, see *7* elements
        // printed out
        f(lua["arr"]);

```

(continues on next page)

(continued from previous page)

```
        lua.script(R"(
arr:clear()
        )");

        // Now it's empty
        f(lua["arr"]);

        std::cout << std::endl;

        return 0;
}
```

Note that this will not work well in Lua 5.1, as it has explicit table checks and does not check metamethods, even when `pairs` or `ipairs` is passed a table. In that case, you will need to use a more manual iteration scheme or you will have to convert it to a table. In C++, you can use *sol::as_table* when passing something to the library to get a table out of it: `lua["arr"] = as_table( std::vector<int>{ ... });`. For manual iteration in Lua code without using `as_table` for something with indices, try:

Listing 21: iteration.lua

```
for i = 1, #vec do
        print(i, vec[i])
end
```

There are also other ways to iterate over key/values, but they can be difficult AND cost your performance due to not having proper support in Lua 5.1. We recommend that you upgrade to Lua 5.2 or 5.3 if this is integral to your infrastructure.

If you can't upgrade, use the "member" function `my_container:pairs()` in Lua to perform iteration:

```
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

#include "assert.hpp"

#include <unordered_set>
#include <iostream>

int main() {
        struct hasher {
                typedef std::pair<std::string, std::string> argument_type;
                typedef std::size_t result_type;

                result_type operator()(const argument_type& p) const {
                        return std::hash<std::string>()(p.first);
                }
        };

        using my_set = std::unordered_set<std::pair<std::string, std::string>, hasher>
↪;

        std::cout << "=== containers with std::pair<> ===" << std::endl;

        sol::state lua;
        lua.open_libraries(sol::lib::base);

```

(continues on next page)

(continued from previous page)

```
        lua.set_function("f", []() {
                return my_set{ { "key1", "value1" },{ "key2", "value2" },{ "key3",
↪"value3" } };
        });

        lua.safe_script("v = f()");
        lua.safe_script("print('v:', v)");
        lua.safe_script("print('#v:', #v)");
        // note that using my_obj:pairs() is a
        // way around pairs(my_obj) not working in Lua 5.1/LuaJIT: try it!
        lua.safe_script("for k,v1,v2 in v:pairs() do print(k, v1, v2) end");

        std::cout << std::endl;

        return 0;
}
```

1.9 threading

Lua has no thread safety. sol does not force thread safety bottlenecks anywhere. Treat access and object handling like you were dealing with a raw `int` reference (`int&`) (no safety or order guarantees whatsoever).

Assume any access or any call on Lua affects the whole `sol::state`/`lua_State*` (because it does, in a fair bit of cases). Therefore, every call to a state should be blocked off in C++ with some kind of access control (when you're working with multiple C++ threads). When you start hitting the same state from multiple threads, race conditions (data or instruction) can happen.

Individual Lua coroutines might be able to run on separate C++-created threads without tanking the state utterly, since each Lua coroutine has the capability to run on an independent Lua execution stack (Lua confusingly calls it a `thread` in the C API, but it really just means a separate execution stack) as well as some other associated bits and pieces that won't quite interfere with the global state.

To handle multithreaded environments, it is encouraged to either spawn a Lua state (`sol::state`) for each thread you are working with and keep inter-state communication to synchronized serialization points. This means that 3 C++ threads should each have their own Lua state, and access between them should be controlled using some kind of synchronized C++ mechanism (actual transfer between states must be done by serializing the value into C++ and then re-pushing it into the other state).

Using coroutines and Lua's threads might also buy you some concurrency and parallelism (**unconfirmed and likely untrue, do not gamble on this**), but remember that Lua's 'threading' technique is ultimately cooperative and requires explicit yielding and resuming (simplified as function calls for *sol::coroutine*).

1.9.1 getting the main thread

Lua 5.1 does not keep a reference to the main thread, therefore the user has to store it themselves. If you create a `sol::state` or follow the *steps for opening up compatibility and default handlers here*, you can work with `sol::main_thread` to retrieve you the main thread, given a `lua_State*` that is either a full state or a thread: `lua_state* Lmain = sol::main_thread( Lcoroutine );` This function will always work in Lua 5.2 and above: in Lua 5.1, if you do not follow the `sol::state` instructions and do not pass a fallback `lua_State*` to the function, this function may not work properly and return `nullptr`.

1.9.2 working with multiple Lua threads

You can mitigate some of the pressure of using coroutines and threading by using the `lua_xmove` constructors that sol implements. Simply keep a reference to your `sol::state_view` or `sol::state` or the target `lua_State*` pointer, and pass it into the constructor along with the object you want to copy. Note that there is also some implicit `lua_xmove` checks that are done for copy and move assignment operators as well, as noted *at the reference constructor explanations*.

Note: Advanced used: Furthermore, for every single `sol::reference` derived type, there exists a version prefixed with the word `main_`, such as `sol::main_table`, `sol::main_function`, `sol::main_object` and similar. These classes, on construction, assignment and other operations, forcibly obtain the `lua_State*` associated with the main thread, if possible. Using these classes will allow your code to be immune when a wrapped coroutine or a lua thread is set to `nil` and then garbage-collected.

Note: This does **not** provide immunity from typical multithreading issues in C++, such as synchronized access and the like. Lua's coroutines are cooperative in nature and concurrent execution with things like `std::thread` and similar still need to follow good C++ practices for multi threading.

Here's an example of explicit state transferring below:

```
#define SOL_CHECK_ARGUMENTS 1

#include <sol.hpp>

#include "../assert.hpp"
#include <iostream>

int main (int, char*[]) {

        sol::state lua;
        lua.open_libraries();
        sol::function transferred_into;
        lua["f"] = [&lua, &transferred_into](sol::object t, sol::this_state this_L) {
                std::cout << "state of main     : "  << (void*)lua.lua_state() <<␣
↪std::endl;
                std::cout << "state of function : "  << (void*)this_L.lua_state() <<␣
↪std::endl;
                // pass original lua_State* (or sol::state/sol::state_view)
                // transfers ownership from the state of "t",
                // to the "lua" sol::state
                transferred_into = sol::function(lua, t);
        };

        lua.script(R"(
        i = 0
        function test()
        co = coroutine.create(
                function()
                        local g = function() i = i + 1 end
                        f(g)
                        g = nil
                        collectgarbage()
                end
        )
```

(continues on next page)

(continued from previous page)

```
        coroutine.resume(co)
        co = nil
        collectgarbage()
        end
        )");

        // give it a try
        lua.safe_script("test()");
        // should call 'g' from main thread, increment i by 1
        transferred_into();
        // check
        int i = lua["i"];
        c_assert(i == 1);

        return 0;
}
```

1.10 customization traits

These are customization points within the library to help you make sol2 work for the types in your framework and types.

To learn more about various customizable traits, visit:

- ***containers customization traits***
  - This is how to work with containers in their entirety and what operations you're afforded on them
  - when you have an compiler error when serializing a type that has `begin` and `end` functions but isn't exactly a container
- ***unique usertype (custom pointer) traits***
  - This is how to deal with unique usertypes, e.g. `boost::shared_ptr`, reference-counted pointers, etc
  - Useful for custom pointers from all sorts of frameworks or handle types that employ very specific kinds of destruction semantics and access
- ***customization points***
  - This is how to customize a type to work with sol2
  - Can be used for specializations to push strings and other class types that are not natively `std::string` or `const char*`, like a wxString, for example

1.11 api reference manual

Browse the various function and classes *Sol* utilizes to make your life easier when working with Lua.

1.11.1 state

owning and non-owning state holders for registry and globals

```
class state_view;

class state : state_view, std::unique_ptr<lua_State*, deleter>;
```

The most important class here is `state_view`. This structure takes a `lua_State*` that was already created and gives you simple, easy access to Lua's interfaces without taking ownership. `state` derives from `state_view`, inheriting all of this functionality, but has the additional purpose of creating a fresh `lua_State*` and managing its lifetime for you in its constructors.

The majority of the members between `state_view` and *sol::table* are identical, with a few added for this higher-level type. Therefore, all of the examples and notes in *sol::table* apply here as well.

`state_view` is cheap to construct and creates 2 references to things in the `lua_State*` while it is alive: the global Lua table, and the Lua C Registry.

`sol::state` automatic handlers

One last thing you should understand: constructing a `sol::state` does a few things behind-the-scenes for you, mostly to ensure compatibility and good error handler/error handling. The function it uses to do this is `set_default_state`. They are as follows:

- set a default panic handler with `state_view::set_panic`/`lua_atpnic`
- set a default `sol::protected_function` handler with `sol::protected_function::set_default_handler`, using a `sol::reference` to `&sol::detail::default_traceback_error_handler` as the default handler function
- set a default exception handler to `&sol::detail::default_exception_handler`
- register the state as the main thread (only does something for Lua 5.1, which does not have a way to get the main thread) using `sol::stack::register_main_thread(L)`
- register the LuaJIT C function exception handler with `stack::luajit_exception_handler(L)`

You can read up on the various panic and exception handlers on the *exceptions page*.

sol::state_view does none of these things for you. If you want to make sure your self-created or self-managed state has the same properties, please apply this function once to the state. Please note that it will override your panic handler and, if using LuaJIT, your LuaJIT C function handler.

Warning: It is your responsibility to make sure `sol::state_view` goes out of scope before you call `lua_close` on a pre-existing state, or before `sol::state` goes out of scope and its destructor gets called. Failure to do so can result in intermittent crashes because the `sol::state_view` has outstanding references to an already-dead `lua_State*`, and thusly will try to decrement the reference counts for the Lua Registry and the Global Table on a dead state. Please use `{` and `}` to create a new scope, or other lifetime techniques, when you know you are going to call `lua_close` so that you have a chance to specifically control the lifetime of a `sol::state_view` object.

enumerations

Listing 22: in-lua libraries

```
enum class lib : char {
    base,
    package,
```

(continues on next page)

(continued from previous page)

```
    coroutine,
    string,
    os,
    math,
    table,
    debug,
    bit32,
    io,
    ffi,
    jit,
    count // do not use
};
```

This enumeration details the various base libraries that come with Lua. See the standard lua libraries for details about the various standard libraries.

members

Listing 23: function: open standard libraries/modules

```
template<typename... Args>
void open_libraries(Args&&... args);
```

This function takes a number of *sol::lib* as arguments and opens up the associated Lua core libraries.

Listing 24: function: script / safe_script / script_file / safe_script_file / unsafe_script / unsafe_script_file

```
function_result script(const string_view& code, const std::string& chunk_name =
↪"[string]", load_mode mode = load_mode::any);
protected_function_result script(const string_view& code, const environment& env,␣
↪const std::string& chunk_name = "[string]", load_mode mode = load_mode::any);
template <typename ErrorFunc>
protected_function_result script(const string_view& code, ErrorFunc&& on_error, const␣
↪std::string& chunk_name = "[string]", load_mode mode = load_mode::any);
template <typename ErrorFunc>
protected_function_result script(const string_view& code, const environment& env,␣
↪ErrorFunc&& on_error, const std::string& chunk_name = "[string]", load_mode mode =␣
↪load_mode::any);

function_result script_file(const std::string& filename, load_mode mode = load_
↪mode::any);
protected_function_result script_file(const std::string& filename, const environment&␣
↪env, load_mode mode = load_mode::any);
template <typename ErrorFunc>
protected_function_result script_file(const std::string& filename, ErrorFunc&& on_
↪error, load_mode mode = load_mode::any);
template <typename ErrorFunc>
protected_function_result script_file(const std::string& filename, const environment&␣
↪env, ErrorFunc&& on_error, load_mode mode = load_mode::any);
```

If you need safety, please use the version of these functions with `safe` (such as `safe_script(_file)`) appended in front of them. They will always check for errors and always return a `sol::protected_function_result`. If you explicitly do not want to check for errors and want to simply invoke `lua_error` in the case of errors (which will call `panic`), use `unsafe_script(_file)` versions.

These functions run the desired blob of either code that is in a string, or code that comes from a filename, on the `lua_State*`. It will not run isolated: any scripts or code run will affect code in the `lua_State*` the object uses as well (unless `local` is applied to a variable declaration, as specified by the Lua language). Code ran in this fashion is not isolated. If you need isolation, consider creating a new state or traditional Lua sandboxing techniques.

If your script returns a value, you can capture it from the returned *sol::unsafe_function_result*/*sol::protected_function_result*. Note that the plain versions that do not take an environment or a callback function assume that the contents internally not only loaded properly but ran to completion without errors, for the sake of simplicity and performance.

To handle errors when using the second overload, provide a callable function/object that takes a `lua_State*` as its first argument and a `sol::protected_function_result` as its second argument. `sol::script_default_on_error` and `sol::script_pass_on_error` are 2 functions provided by sol that will either generate a traceback error to return / throw (if throwing is allowed); or, pass the error on through and return it to the user (respectively). An example of having your:

```
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

#include <iostream>

int main () {

        std::cout << "=== safe_script usage ===" << std::endl;

        sol::state lua;
        // uses sol::script_default_on_error, which either panics or throws,
        // depending on your configuration and compiler settings
        try {
                auto result1 = lua.safe_script("bad.code");
        }
        catch( const sol::error& e ) {
                std::cout << "an expected error has occurred: " << e.what() <<␣
↪std::endl;
        }

        // a custom handler that you write yourself
        // is only called when an error happens with loading or running the script
        auto result2 = lua.safe_script("123 bad.code", [](lua_State*, sol::protected_
↪function_result pfr) {
                // pfr will contain things that went wrong, for either loading or␣
↪executing the script
                // the user can do whatever they like here, including throw.␣
↪Otherwise...
                sol::error err = pfr;
                std::cout << "An error (an expected one) occurred: " << err.what() <<␣
↪std::endl;

                // ... they need to return the protected_function_result
                return pfr;
        });

        std::cout << std::endl;

        return 0;
}
```

You can also pass a *sol::environment* to `script`/`script_file` to have the script have sandboxed / contained in a

way inside of a state. This is useful for runnig multiple different "perspectives" or "views" on the same state, and even has fallback support. See the *sol::environment* documentation for more details.

Listing 25: function: require / require_file

```
sol::object require(const std::string& key, lua_CFunction open_function, bool create_
↪global = true);
sol::object require_script(const std::string& key, const std::string& code, bool␣
↪create_global = true);
sol::object require_file(const std::string& key, const std::string& file, bool create_
↪global = true);
```

These functions play a role similar to luaL_requiref except that they make this functionality available for loading a one-time script or a single file. The code here checks if a module has already been loaded, and if it has not, will either load / execute the file or execute the string of code passed in. If `create_global` is set to true, it will also link the name `key` to the result returned from the open function, the code or the file. Regardless or whether a fresh load happens or not, the returned module is given as a single *sol::object* for you to use as you see fit.

Thanks to Eric (EToreo) for the suggestion on this one!

Listing 26: function: load / load_file

```
sol::load_result load(lua_Reader reader, void* data, const std::string& chunk_name =
↪"[string]", load_mode mode = load_mode::any);
sol::load_result load(const string_view& code, const std::string& chunk_name =
↪"[string]", load_mode mode = load_mode::any);
sol::load_result load_buffer(const char* buff, std::size_t buffsize, const␣
↪std::string& chunk_name = "[string]", load_mode mode = load_mode::any);
sol::load_result load_file(const std::string& filename, load_mode mode = load_
↪mode::any);
```

These functions *load* the desired blob of either code that is in a string, or code that comes from a filename, on the `lua_State*`. That blob will be turned into a Lua Function. It will not be run: it returns a `load_result` proxy that can be called to actually run the code, when you are ready. It can also be turned into a `sol::function`, a `sol::protected_function`, or some other abstraction that can serve to call the function. If it is called, it will run on the object's current `lua_State*`: it is not isolated. If you need isolation, consider using *sol::environment*, creating a new state, or other Lua sandboxing techniques.

Finally, if you have a custom source of data, you can use the `lua_Reader` overloaded function alongside passing in a `void*` pointing to a single type that has everything you need to run it. Use that callback to provide data to the underlying Lua implementation to read data, as explained in the Lua manual.

This is a low-level function and if you do not understand the difference between loading a piece of code versus running that code, you should be using *state_view::script*.

Listing 27: function: do_string / do_file

```
sol::protected_function_result do_string(const string_view& code);
sol::protected_function_result do_file(const std::string& filename);
sol::protected_function_result do_string(const string_view& code, sol::environment␣
↪env);
sol::protected_function_result do_file(const std::string& filename, sol::environment␣
↪env);
```

These functions *loads and performs* the desired blob of either code that is in a string, or code that comes from a filename, on the `lua_State*`. It *will* run, and then return a `protected_function_result` proxy that can be examined for either an error or the return value. This function does not provide a callback like *state_view::script* does. It is a lower-level function that performs less checking and directly calls `load(_file)` before running the result,

with the optional environment.

It is advised that, unless you have specific needs or the callback function is not to your liking, that you work directly with *state_view::script*.

Listing 28: function: global table / registry table

```
sol::global_table globals() const;
sol::table registry() const;
```

Get either the global table or the Lua registry as a *sol::table*, which allows you to modify either of them directly. Note that getting the global table from a `state`/`state_view` is usually unnecessary as it has all the exact same functions as a *sol::table* anyhow.

Listing 29: function: set_panic

```
void set_panic(lua_CFunction panic);
```

Overrides the panic function Lua calls when something unrecoverable or unexpected happens in the Lua VM. Must be a function of the that matches the `int(lua_State*)` function signature.

Listing 30: function: memory_used

```
std::size_t memory_used() const;
```

Returns the amount of memory used *in bytes* by the Lua State.

Listing 31: function: collect_garbage

```
void collect_garbage();
```

Attempts to run the garbage collector. Note that this is subject to the same rules as the Lua API's collect_garbage function: memory may or may not be freed, depending on dangling references or other things, so make sure you don't have tables or other stack-referencing items currently alive or referenced that you want to be collected.

Listing 32: function: make a table

```
sol::table create_table(int narr = 0, int nrec = 0);
template <typename Key, typename Value, typename... Args>
sol::table create_table(int narr, int nrec, Key&& key, Value&& value, Args&&... args);


template <typename... Args>
sol::table create_table_with(Args&&... args);

static sol::table create_table(lua_State* L, int narr = 0, int nrec = 0);
template <typename Key, typename Value, typename... Args>
static sol::table create_table(lua_State* L, int narr, int nrec, Key&& key, Value&&␣
↪value, Args&&... args);
```

Creates a table. Forwards its arguments to *table::create*. Applies the same rules as *table.set* when putting the argument values into the table, including how it handles callable objects.

1.11.2 this_state

transparent state argument for the current state

```
struct this_state;
```

This class is a transparent type that is meant to be gotten in functions to get the current lua state a bound function or usertype method is being called from. It does not actually retrieve anything from lua nor does it increment the argument count, making it "invisible" to function calls in lua and calls through `std::function<...>` and *sol::function* on this type. It can be put in any position in the argument list of a function:

```
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

#include "assert.hpp"

int main () {
        sol::state lua;

        lua.set_function("bark", []( sol::this_state s, int a, int b ){
                lua_State* L = s; // current state
                return a + b + lua_gettop(L);
        });

        lua.script("first = bark(2, 2)"); // only takes 2 arguments, NOT 3

        // Can be at the end, too, or in the middle: doesn't matter
        lua.set_function("bark", []( int a, int b, sol::this_state s ){
                lua_State* L = s; // current state
                return a + b + lua_gettop(L);
        });

        lua.script("second = bark(2, 2)"); // only takes 2 arguments
        int first = lua["first"];
        c_assert(first == 6);
        int second = lua["second"];
        c_assert(second == 6);

        return 0;
}
```

1.11.3 reference

general purpose reference to Lua object in registry

Listing 33: reference

```
class reference;
```

This type keeps around a reference to something inside of Lua, whether that object was on the stack or already present as an object in the Lua Runtime. It places the object Lua registry and will keep it alive.

It is the backbone for all things that reference items on the stack that need to be kept around beyond their appearance and lifetime on said Lua stack or need to be kept alive outside of a script beyond garbage collection times. Its progeny include *sol::coroutine*, *sol::function*, *sol::protected_function*, *sol::object*, *sol::table*/*sol::global_table*, *sol::thread*, and *sol::(light_)userdata*, which are type-specific versions of `sol::reference`.

Note that if you need to keep a reference to something inside of Lua, it is better to use `sol::reference` or *sol::object* to keep a reference to it and then use the `obj.as<T>()` member function to retrieve what you need than to take a direct dependency on the memory by retrieving a pointer or reference to the userdata itself. This will ensure

that if a script or the Lua Runtime is finished with an object, it will not be garbage collected. Do this only if you need long-term storage.

For all of these types, there's also a `sol::stack_{x}` version of them, such as `sol::stack_table`. They are useful for a small performance boost at the cost of not having a strong reference, which has implications for what happens when the item is moved off of the stack. See *sol::stack_reference* for more details.

members

Listing 34: constructor: reference

```
reference(lua_State* L, int index = -1);
reference(lua_State* L, lua_nil_t);
reference(lua_State* L, absolute_index index);
reference(lua_State* L, raw_index index);
reference(lua_State* L, ref_index index);
template <typename Object>
reference(Object&& o);
template <typename Object>
reference(lua_State* L, Object&& o);
```

The first constructor creates a reference from the Lua stack at the specified index, saving it into the metatable registry. The second attemtps to register something that already exists in the registry. The third attempts to reference a pre-existing object and create a reference to it. These constructors are exposed on all types that derive from `sol::reference`, meaning that you can grab tables, functions, and coroutines from the registry, stack, or from other objects easily.

Note: Note that the last constructor has `lua_xmove` safety built into it. You can pin an object to a certain thread (or the main thread) by initializing it with `sol::reference pinned(state, other_reference_object);` . This ensures that `other_reference_object` will exist in the state/thread of `state`. Also note that copy/move assignment operations will also use pinning semantics if it detects that the state of the object on the right is `lua_xmove` compatible. (But, the `reference` object on the left must have a valid state as well. You can have a nil `reference` with a valid state by using the `sol::reference pinned(state, sol::lua_nil)` constructor as well.) This applies for any `sol::reference` derived type.

You can un-pin and null the state by doing `ref = sol::lua_nil;`. This applies to **all derived types**, including `sol::(protected_)function`, `sol::thread`, `sol::object`, `sol::table`, and similar.

Listing 35: function: push referred-to element from the stack

```
int push() const noexcept;
```

This function pushes the referred-to data onto the stack and returns how many things were pushed. Typically, it returns 1.

Listing 36: function: reference value

```
int registry_index() const noexcept;
```

The value of the reference in the registry.

Listing 37: functions: non-nil, non-null check

```
bool valid () const noexcept;
explicit operator bool () const noexcept;
```

These functions check if the reference at `T` is valid: that is, if it is not *nil* and if it is not non-existing (doesn't refer to anything, including nil) reference. The explicit operator bool allows you to use it in the context of an `if ( my_obj )` context.

Listing 38: function: retrieves the type

```
type get_type() const noexcept;
```

Gets the *sol::type* of the reference; that is, the Lua reference.

Listing 39: function: lua_State* of the reference

```
lua_State* lua_state() const noexcept;
```

Gets the `lua_State*` this reference exists in.

non-members

Listing 40: operators: reference comparators

```
bool operator==(const reference&, const reference&);
bool operator!=(const reference&, const reference&);
```

Compares two references using the Lua API's lua_compare for equality.

1.11.4 stack_reference

zero-overhead object on the stack

When you work with a *sol::reference*, the object gotten from the stack has a reference to it made in the registry, keeping it alive. If you want to work with the Lua stack directly without having any additional references made, `sol::stack_reference` is for you. Its API is identical to `sol::reference` in every way, except it contains a `int stack_index()` member function that allows you to retrieve the stack index.

Note that this will not pin the object since a copy is not made in the registry, meaning things can be pulled out from under it, the stack can shrink under it, things can be added onto the stack before this object's position, and what `sol::stack_reference` will point to will change. Please know what the Lua stack is and have discipline while managing your Lua stack when working at this level.

All of the base types have `stack` versions of themselves, and the APIs are identical to their non-stack forms. This includes *sol::stack_table*, *sol::stack_function*, *sol::stack_protected_function*, *sol::stack_(light_)userdata* and *sol::stack_object*. There is a special case for `sol::stack_function`, which has an extra type called `sol::stack_aligned_function` (and similar `sol::stack_aligned_protected_function`).

stack_aligned_function

This type is particular to working with the stack. It does not push the function object on the stack before pushing the arguments, assuming that the function present is already on the stack before going ahead and invoking the function it is targeted at. It is identical to *sol::function* and has a protected counterpart as well. If you are working with the stack

and know there is a callable object in the right place (i.e., at the top of the Lua stack), use this abstraction to have it call your stack-based function while still having the easy-to-use Lua abstractions.

Furthermore, if you know you have a function in the right place alongside proper arguments on top of it, you can use the `sol::stack_count` structure and give its constructor the number of arguments off the top that you want to call your pre-prepared function with:

Listing 41: stack_aligned_function.cpp

```cpp
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

#include "assert.hpp"

int main(int, char*[]) {
	sol::state lua;
	lua.script("function func (a, b) return (a + b) * 2 end");

	sol::reference func_ref = lua["func"];

	// for some reason, you need to use the low-level API
	func_ref.push(); // function on stack now

	// maybe this is in a lua_CFunction you bind,
	// or maybe you're trying to work with a pre-existing system
	// maybe you've used a custom lua_load call, or you're working
	// with state_view's load(lua_Reader, ...) call...
	// here's a little bit of how you can work with the stack
	lua_State* L = lua.lua_state();
	sol::stack_aligned_function func(L, -1);
	lua_pushinteger(L, 5); // argument 1, using plain API
	lua_pushinteger(L, 6); // argument 2

	// take 2 arguments from the top,
	// and use "stack_aligned_function" to call
	int result = func(sol::stack_count(2));

	// make sure everything is clean
	c_assert(result == 22);
	c_assert(lua.stack_top() == 0); // stack is empty/balanced

	return 0;
}
```

Finally, there is a special abstraction that provides further stack optimizations for `sol::protected_function` variants that are aligned, and it is called `sol::stack_aligned_stack_handler_protected_function`. This typedef expects you to pass a `stack_reference` handler to its constructor, meaning that you have already placed an appropriate error-handling function somewhere on the stack before the aligned function. You can use `sol::stack_count` with this type as well.

Warning: Do not use `sol::stack_count` with a `sol::stack_aligned_protected_function`. The default behavior checks if the `error_handler` member variable is valid, and attempts to push the handler onto the stack in preparation for calling the function. This inevitably changes the stack. Only use `sol::stack_aligned_protected_function` with `sol::stack_count` if you know that the handler is not valid (it is `nil` or its `error_handler.valid()` function returns `false`), or if you use `sol::stack_aligned_stack_handler_protected_function`, which references an existing stack

index that can be before the precise placement of the function and its arguments.

1.11.5 make_object/make_reference

create a value in the lua registry / on the Lua stack and return it

Listing 42: function: make_reference

```
template <typename R = reference, bool should_pop = (R is not base of sol::stack_
↪index), typename T>
R make_reference(lua_State* L, T&& value);
template <typename T, typename R = reference, bool should_pop = (R is base of␣
↪sol::stack_index), typename... Args>
R make_reference(lua_State* L, Args&&... args);
```

Makes an `R` out of the value. The first overload deduces the type from the passed in argument, the second allows you to specify a template parameter and forward any relevant arguments to `sol::stack::push`. The type figured out for `R` is what is referenced from the stack. This allows you to request arbitrary pop-able types from Sol and have it constructed from `R(lua_State* L, int stack_index)`. If the template boolean `should_pop` is `true`, the value that was pushed will be popped off the stack. It defaults to popping, but if it encounters a type such as *sol::stack_reference* (or any of its typically derived types in Sol), it will leave the pushed values on the stack.

Listing 43: function: make_object

```
template <typename T>
object make_object(lua_State* L, T&& value);
template <typename T, typename... Args>
object make_object(lua_State* L, Args&&... args);
```

Makes an object out of the value. It pushes it onto the stack, then pops it into the returned `sol::object`. The first overload deduces the type from the passed in argument, the second allows you to specify a template parameter and forward any relevant arguments to `sol::stack::push`. The implementation essentially defers to *sol::make_reference* with the specified arguments, `R == object` and `should_pop == true`. It is preferred that one uses the *in_place object constructor instead*, since it's probably easier to deal with, but both versions will be supported for forever, since there's really no reason not to and people already have dependencies on `sol::make_object`.

1.11.6 table

a representation of a Lua (meta)table

```
template <bool global>
class table_core;

typedef table_core<false> table;
typedef table_core<true> global_table;
```

`sol::table` is an extremely efficient manipulator of state that brings most of the magic of the Sol abstraction. Capable of doing multiple sets at once, multiple gets into a `std::tuple`, being indexed into using `[key]` syntax and setting keys with a similar syntax (see: *here*), `sol::table` is the corner of the interaction between Lua and C++.

There are two kinds of tables: the global table and non-global tables: however, both have the exact same interface and all `sol::global_table` s are convertible to regular `sol::table` s.

Tables are the core of Lua, and they are very much the core of Sol.

members

Listing 44: constructor: table

```
table(lua_State* L, int index = -1);
table(lua_State* L, sol::new_table nt)
```

The first takes a table from the Lua stack at the specified index and allows a person to use all of the abstractions therein. The second creates a new table using the capacity hints specified in `sol::new_table`'s structure (`sequence_hint` and `map_hint`). If you don't care exactly about the capacity, create a table using `sol::table my_table(my_lua_state, sol::create);`. Otherwise, specify the table creation size hints by initializing it manually through *sol::new_table's simple constructor*.

Listing 45: function: get / traversing get

```
template<typename... Args, typename... Keys>
decltype(auto) get(Keys&&... keys) const;

template<typename T, typename... Keys>
decltype(auto) traverse_get(Keys&&... keys) const;

template<typename T, typename Key>
decltype(auto) get_or(Key&& key, T&& otherwise) const;

template<typename T, typename Key, typename D>
decltype(auto) get_or(Key&& key, D&& otherwise) const;
```

These functions retrieve items from the table. The first one (`get`) can pull out *multiple* values, 1 for each key value passed into the function. In the case of multiple return values, it is returned in a `std::tuple<Args...>`. It is similar to doing `return table["a"], table["b"], table["c"]`. Because it returns a `std::tuple`, you can use `std::tie`/`std::make_tuple` on a multi-get to retrieve all of the necessary variables. The second one (`traverse_get`) pulls out a *single* value, using each successive key provided to do another lookup into the last. It is similar to doing `x = table["a"]["b"]["c"][...]`.

If the keys within nested queries try to traverse into a table that doesn't exist, it will first pull out a `nil` value. If there are further lookups past a key that do not exist, the additional lookups into the nil-returned variable will cause a panic to be fired by the lua C API. If you need to check for keys, check with `auto x = table.get<sol::optional<int>>( std::tie("a", "b", "c" ) );`, and then use the *optional* interface to check for errors. As a short-hand, easy method for returning a default if a value doesn't exist, you can use `get_or` instead.

This function does not create tables where they do not exist.

Listing 46: function: raw get / traversing raw get

```
template<typename... Args, typename... Keys>
decltype(auto) raw_get(Keys&&... keys) const;

template<typename T, typename... Keys>
decltype(auto) traverse_raw_get(Keys&&... keys) const;

template<typename T, typename Key>
decltype(auto) raw_get_or(Key&& key, T&& otherwise) const;
```

(continues on next page)

(continued from previous page)

```
template<typename T, typename Key, typename D>
decltype(auto) raw_get_or(Key&& key, D&& otherwise) const;
```

Similar to *get*, but it does so "raw" (ignoring metamethods on the table's metatable).

Listing 47: function: set / traversing set

```
template<typename... Args>
table& set(Args&&... args);

template<typename... Args>
table& traverse_set(Args&&... args);
```

These functions set items into the table. The first one (`set`) can set *multiple* values, in the form `key_a, value_a, key_b, value_b, ...`. It is similar to `table[key_a] = value_a; table[key_b] = value_b, ...`. The second one (`traverse_set`) sets a *single* value, using all but the last argument as keys to do another lookup into the value retrieved prior to it. It is equivalent to `table[key_a][key_b][...] = value;`.

If the keys within nested queries try to traverse into a table that doesn't exist, it will first pull out a `nil` value. If there are further lookups past a key that do not exist, the additional lookups into the nil-returned variable will cause a panic to be fired by the lua C API.

Please note how callables and lambdas are serialized, as there may be issues on GCC-based implementations. See this *note here*.

This function does not create tables where they do not exist.

Listing 48: function: raw set / traversing raw set

```
template<typename... Args>
table& raw_set(Args&&... args);

template<typename... Args>
table& traverse_raw_set(Args&&... args);
```

Similar to *set*, but it does so "raw" (ignoring metamethods on the table's metatable).

Please note how callables and lambdas are serialized, as there may be issues on GCC-based implementations. See this *note here*.

Note: Value semantics are applied to all set operations. If you do not `std::ref( obj )` or specifically make a pointer with `std::addressof( obj )` or `&obj`, it will copy / move. This is different from how *sol::function* behaves with its call operator. Also note that this does not detect callables by default: see the *note here*.

Listing 49: function: set a function with the specified key into lua

```
template<typename Key, typename Fx>
state_view& set_function(Key&& key, Fx&& fx, [...]);
```

Sets the desired function to the specified key value. Note that it also allows for passing a member function plus a member object or just a single member function: however, using a lambda is almost always better when you want to bind a member function + class instance to a single function call in Lua. Also note that this will allow Lua to understand that a callable object (such as a lambda) should be serialized as a function and not as a userdata: see the *note here* for more details.

Listing 50: function: add

```
template<typename... Args>
table& add(Args&&... args);
```

This function appends a value to a table. The definition of appends here is only well-defined for a table which has a perfectly sequential (and integral) ordering of numeric keys with associated non-null values (the same requirement for the *size* function). Otherwise, this falls to the implementation-defined behavior of your Lua VM, whereupon is may add keys into empty 'holes' in the array (e.g., the first empty non-sequential integer key it gets to from `size`) or perhaps at the very "end" of the "array". Do yourself the favor of making sure your keys are sequential.

Each argument is appended to the list one at a time.

Listing 51: function: size

```
std::size_t size() const;
```

This function returns the size of a table. It is only well-defined in the case of a Lua table which has a perfectly sequential (and integral) ordering of numeric keys with associated non-null values.

Listing 52: function: setting a usertype

```
template<typename Class, typename... Args>
table& new_usertype(const std::string& name, Args&&... args);
template<typename Class, typename CTor0, typename... CTor, typename... Args>
table& new_usertype(const std::string& name, Args&&... args);
template<typename Class, typename... CArgs, typename... Args>
table& new_usertype(const std::string& name, constructors<CArgs...> ctor, Args&&...␣
↪args);
```

This class of functions creates a new *usertype* with the specified arguments, providing a few extra details for constructors. After creating a usertype with the specified argument, it passes it to *set_usertype*.

Listing 53: function: creating an enum

```
template<bool read_only = true, typename... Args>
basic_table_core& new_enum(const std::string& name, Args&&... args);
template<typename T, bool read_only = true>
basic_table_core& new_enum(const std::string& name, std::initializer_list<std::pair
↪<string_view, T>> items);
```

Use this function to create an enumeration type in Lua. By default, the enum will be made read-only, which creates a tiny performance hit to make the values stored in this table behave exactly like a read-only enumeration in C++. If you plan on changing the enum values in Lua, set the `read_only` template parameter in your `new_enum` call to false. The arguments are expected to come in `key, value, key, value, ...` list.

If you use the second overload, you will create a (runtime) `std::initializer_list`. This will avoid compiler overhead for excessively large enumerations. For this overload, hoever, you must pass the enumeration name as a template parameter first, and then the `read_only` parameter, like `lua.new_enum<my_enum>( "my_enum", { {"a", my_enum::  a}, { "b", my_enum::b } } );`.

Listing 54: function: setting a pre-created usertype

```
template<typename T>
table& set_usertype(usertype<T>& user);
template<typename Key, typename T>
table& set_usertype(Key&& key, usertype<T>& user);
```

Sets a previously created usertype with the specified `key` into the table. Note that if you do not specify a key, the implementation falls back to setting the usertype with a `key` of `usertype_traits<T>::name`, which is an implementation-defined name that tends to be of the form `{namespace_name 1}_[{namespace_name 2 ...}_{class name}`.

Listing 55: function: begin / end for iteration

```
table_iterator begin () const;
table_iterator end() const;
table_iterator cbegin() const;
table_iterator cend() const;
```

Provides (what can barely be called) input iterators for a table. This allows tables to work with single-pass, input-only algorithms (like `std::for_each`). Note that manually getting an iterator from `.begin()` without a `.end()` or using postfix incrementation (`++mytable.begin()`) will lead to poor results. The Lua stack is manipulated by an iterator and thusly not performing the full iteration once you start is liable to ruin either the next iteration or break other things subtly. Use a C++11 ranged for loop, `std::for_each`, or other algorithims which pass over the entire collection at least once and let the iterators fall out of scope.

Warning: The iterators you use to walk through a `sol::table` are NOT guaranteed to iterate in numeric order, or ANY kind of order. They may iterate backwards, forwards, in the style of cuckoo-hashing, by accumulating a visited list while calling `rand()` to find the next target, or some other crazy scheme. Now, no implementation would be crazy, but it is behavior specifically left undefined because there are many ways that your Lua package can implement the table type.

Iteration order is NOT something you should rely on. If you want to figure out the length of a table, call the length operation (`int count = mytable.size();` using the sol API) and then iterate from `1` to `count` (inclusive of the value of count, because Lua expects iteration to work in the range of `[1, count]`). This will save you some headaches in the future when the implementation decides not to iterate in numeric order.

Listing 56: function: iteration with a function

```
template <typename Fx>
void for_each(Fx&& fx);
```

A functional `for_each` loop that calls the desired function. The passed in function must take either `sol::object key, sol::object value` or take a `std::pair<sol::object, sol::object> key_value_pair`. This version can be a bit safer as allows the implementation to definitively pop the key/value off the Lua stack after each call of the function.

Listing 57: function: operator[] access

```
template<typename T>
proxy<table&, T> operator[](T&& key);
template<typename T>
proxy<const table&, T> operator[](T&& key) const;
```

Generates a *proxy* that is templated on the table type and the key type. Enables lookup of items and their implicit conversion to a desired type. Lookup is done lazily.

Please note how callables and lambdas are serialized, as there may be issues on GCC-based implementations. See this *note here*.

Listing 58: function: create a table with defaults

```
table create(int narr = 0, int nrec = 0);
template <typename Key, typename Value, typename... Args>
table create(int narr, int nrec, Key&& key, Value&& value, Args&&... args);

static table create(lua_State* L, int narr = 0, int nrec = 0);
template <typename Key, typename Value, typename... Args>
static table create(lua_State* L, int narr, int nrec, Key&& key, Value&& value, Args&&
↪... args);
```

Creates a table, optionally with the specified values pre-set into the table. If `narr` or `nrec` are 0, then compile-time shenanigans are used to guess the amount of array entries (e.g., integer keys) and the amount of hashable entries (e.g., all other entries).

Listing 59: function: create a table with compile-time defaults assumed

```
template <typename... Args>
table create_with(Args&&... args);
template <typename... Args>
static table create_with(lua_State* L, Args&&... args);
```

Creates a table, optionally with the specified values pre-set into the table. It checks every 2nd argument (the keys) and generates hints for how many array or map-style entries will be placed into the table. Applies the same rules as *table.set* when putting the argument values into the table, including how it handles callable objects.

Listing 60: function: create a named table with compile-time defaults assumed

```
template <typename Name, typename... Args>
table create_named(Name&& name, Args&&... args);
```

Creates a table, optionally with the specified values pre-set into the table, and sets it as the key `name` in the table. Applies the same rules as *table.set* when putting the argument values into the table, including how it handles callable objects.

1.11.7 userdata

reference to a userdata

Listing 61: (light_)userdata reference

```
class userdata : public table;

class light_userdata : public table;
```

These types are meant to hold a reference to a (light) userdata from Lua and make it easy to push an existing userdata onto the stack. It is essentially identical to *table* in every way, just with a definitive C++ type that ensures the type is some form of userdata (helpful for trapping type errors with *safety features turned on*). You can also use its `.is<T>()` and `.as<T>()` methods to check if its of a specific type and retrieve that type, respectively.

1.11.8 environment

encapsulation table for script sandboxing

Listing 62: environment

```
class environment : public table;

template <typename T>
void set_environment( const environment& env, const T& target );
template <typename E = reference, typename T>
basic_environment<E> get_environment( const T& target );
```

This type is passed to *sol::state(_view)::script/do_x* to provide an environment where local variables that are set and get retrieve. It is just a plain table, and all the same operations *from table still apply*. This is important because it allows you to do things like set the table's metatable (using *sol::metatable_key* for instance) and having its `__index` entry point to the global table, meaning you can get – but not set – variables from a Global environment, for example.

There are many more uses, including storing state or special dependent variables in an environment that you pre-create using regular table opertions, and then changing at-will:

```
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

int main (int, char*[]) {
        sol::state lua;
        lua.open_libraries();
        sol::environment my_env(lua, sol::create);
        // set value, and we need to explicitly allow for
        // access to "print", since a new environment hides
        // everything that's not defined inside of it
        // NOTE: hiding also hides library functions (!!)
        // BE WARNED
        my_env["var"] = 50;
        my_env["print"] = lua["print"];

        sol::environment my_other_env(lua, sol::create, lua.globals());
        // do not need to explicitly allow access to "print",
        // since we used the "Set a fallback" version
        // of the sol::environment constructor
        my_other_env["var"] = 443;

        // output: 50
        lua.script("print(var)", my_env);

        // output: 443
        lua.script("print(var)", my_other_env);

        return 0;
}
```

Also note that `sol::environment` derives from `sol::table`, which also derives from `sol::reference`: in other words, copying one `sol::environment` value to another `sol::environment` value **does not** deep-copy the table, just creates a new reference pointing to the same lua object.

`sol::environment` objects can be used with script calls, and it can also be set on functions. It can even be applied to *threads*.

You can set the environment using `sol::set_environment( my_env, some_reference );` or `my_env.set_on( some_reference );`.

free functions

Listing 63: function: set_environment

```
template <typename T>
void set_environment( const environment& env, const T& target );
```

See *environment::set_on*.

Listing 64: function: get_environment

```
template <typename E = reference, typename T>
basic_environment<E> get_environment( const T& target );
```

This function retrieves the environment from the target object. If it does not have a valid environment, then the environment's valid function will return false after creation. Every function (regular Lua function, executable script, and similar) has an environment, as well as userdata in certain versions of the Lua runtime.

members

Listing 65: constructor: environment

```
environment(lua_State* L, sol::new_table nt);
environment(lua_State* L, sol::new_table nt, const sol::reference& fallback);
environment(sol::env_t, const sol::reference& object_that_has_environment);
environment(sol::env_t, const sol::stack_reference& object_that_has_environment);
```

The ones from table are used here (of particular note is the ability to use `sol::environment(my_lua_state, sol::create);` to make a fresh, unnamed environment), plus the three unique constructors shown above.

The first constructor is generally used as `sol::environment my_env(my_lua_state, sol::create, my_fallback_table);`. The fallback table serves as the backup to lookup attempts on the environment table being created. It is achieved by simply creating a metatable for the `sol::environment` being created, and then doing `env_metatable["__index"] = fallback;`. You can achieve fancier effects by changing the metatable of the environment to your liking, by creating it in some fashion and then setting the metatable explicitly and populating it with data, particularly with *sol::metatable_key*.

The second and third unique constructors take a special empty type that serves as a key to trigger this constructor and serves no other purpose, `sol::env_t`. The shortcut value so you don't have to create one is called `sol::env_key`. It is used like `sol::environment my_env(sol::env_key, some_object);`. It will extract the environment out of whatever the second argument is that may or may not have an environment. If it does not have an environment, the constructor will complete but the object will have `env.valid() == false`, since it will reference Lua's `nil`.

Listing 66: function: set_on

```
template <typename T>
void set_on(const T& target);
```

This function applies the environment to the desired target. Not that lua 5.1 only tolerates the application of environments to userdata, threads and functions, while 5.2+ has different (more relaxed) rules. It is called by the free function `sol::set_environment( env, target );`.

1.11.9 this_environment

retrieving the environment of the calling function

Sometimes in C++ it's useful to know where a Lua call is coming from and what *environment* it is from. The former is covered by Lua's Debug API, which is extensive and is not fully wrapped up by sol2. But, sol2 covers the latter in letting you get the environment of the calling script / function, if it has one. `sol::this_environment` is a *transparent argument* and does not need to be passed in Lua scripts or provided when using *sol::function*, similar to *sol::this_state*:

```
#define SOL_CHECK_ARGUMENTS
#include <sol.hpp>

#include <iostream>


void env_check(sol::this_state ts, int x, sol::this_environment te) {
        std::cout << "In C++, 'int x' is in the second position, and its value is: " <
↪< x << std::endl;
        if (!te) {
                std::cout << "function does not have an environment: exiting function␣
↪early" << std::endl;
                return;
        }
        sol::environment& env = te;
        sol::state_view lua = ts;
        sol::environment freshenv = lua["freshenv"];
        bool is_same_env = freshenv == env;
        std::cout << "env == freshenv : " << is_same_env << std::endl;
}

int main() {
        sol::state lua;
        sol::environment freshenv(lua, sol::create, lua.globals());
        lua["freshenv"] = freshenv;

        lua.set_function("f", env_check);

        // note that "f" only takes 1 argument and is okay here
        lua.script("f(25)", freshenv);

        return 0;
}
```

Also see this example for more details.

1.11.10 proxy, (protectedunsafe)_function_result - proxy_base derivatives

‘‘table[x]‘‘ and ‘‘function(...)‘‘ conversion struct

```
template <typename Recurring>
struct proxy_base;

template <typename Table, typename Key>
struct proxy : proxy_base<...>;

struct stack_proxy: proxy_base<...>;

struct unsafe_function_result : proxy_base<...>;

struct protected_function_result: proxy_base<...>;
```

These classes provide implicit assignment operator `operator=` (for `set`) and an implicit conversion operator `operator T` (for `get`) to support items retrieved from the underlying Lua implementation, specifically *sol::table* and the results of function calls on *sol::function* and *sol::protected_function*.

proxy

`proxy` is returned by lookups into *sol::table* and table-like entities. Because it is templated on key and table type, it would be hard to spell: you can capture it using the word `auto` if you feel like you need to carry it around for some reason before using it. `proxy` evaluates its arguments lazily, when you finally call `get` or `set` on it. Here are some examples given the following lua script:

```
        bark = {
                woof = {
                        [2] = "arf!"
                }
        }
```

After loading that file in or putting it in a string and reading the string directly in lua (see *state*), you can start kicking around with it in C++ like so:

```
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

#include "assert.hpp"

#include <iostream>

int main () {
        sol::state lua;
        lua.open_libraries(sol::lib::base);
        lua.script(code);

        // produces proxy, implicitly converts to std::string, quietly destroys proxy
        std::string arf_string = lua["bark"]["woof"][2];

        // lazy-evaluation of tables
        auto x = lua["bark"];
        auto y = x["woof"];
        auto z = y[2];

```

(continues on next page)

(continued from previous page)

```
        // retrivies value inside of lua table above
        std::string value = z;
        c_assert(value == "arf!");

        // Can change the value later...
        z = 20;

        // Yay, lazy-evaluation!
        int changed_value = z; // now it's 20!
        c_assert(changed_value == 20);
        lua.script("assert(bark.woof[2] == 20)");
```

We don't recommend using `proxy` lazy evaluation the above to be used across classes or between function: it's more of something you can do to save a reference to a value you like, call a script or run a lua function, and then get it afterwards. You can also set functions (and function objects) this way, and retrieve them as well:

```
        lua["a_new_value"] = 24;
        lua["chase_tail"] = [](int chasing) {
                int r = 2;
                for (int i = 0; i < chasing; ++i) {
                        r *= r;
                }
                return r;
        };

        lua.script("assert(a_new_value == 24)");
        lua.script("assert(chase_tail(2) == 16)");

        return 0;
}
```

members

Listing 67: functions: [overloaded] implicit conversion get

```
requires( sol::is_primitive_type<T>::value == true )
template <typename T>
operator T() const;

requires( sol::is_primitive_type<T>::value == false )
template <typename T>
operator T&() const;
```

Gets the value associated with the keys the proxy was generated and convers it to the type `T`. Note that this function will always return `T&`, a non-const reference, to types which are not based on *sol::reference* and not a *primitive lua type*

Listing 68: function: get a value

```
template <typename T>
decltype(auto) get( ) const;
```

Gets the value associated with the keys and converts it to the type `T`.

Listing 69: function: optionally get a value

```
template <typename T, typename Otherwise>
optional<T> get_or( Otherwise&& otherise ) const;
```

Gets the value associated with the keys and converts it to the type `T`. If it is not of the proper type, it will return a `sol::nullopt` instead.

Listing 70: function: [overloaded] optionally get or create a value

```
template <typename T>
decltype(auto) get_or_create();
template <typename T, typename Otherwise>
decltype(auto) get_or_create( Otherwise&& other );
```

Gets the value associated with the keys if it exists. If it does not, it will set it with the value and return the result.

`operator[]` proxy-only members

Listing 71: function: valid

```
bool valid () const;
```

Returns whether this proxy actually refers to a valid object. It uses *sol::stack::probe_get_field* to determine whether or not its valid.

Listing 72: functions: [overloaded] implicit set

```
requires( sol::detail::Function<Fx> == false )
template <typename T>
proxy& operator=( T&& value );

requires( sol::detail::Function<Fx> == true )
template <typename Fx>
proxy& operator=( Fx&& function );
```

Sets the value associated with the keys the proxy was generated with to `value`. If this is a function, calls `set_function`. If it is not, just calls `set`. Does not exist on *unsage_function_result* or *protected_function_result*.

Listing 73: function: set a callable

```
template <typename Fx>
proxy& set_function( Fx&& fx );
```

Sets the value associated with the keys the proxy was generated with to a function `fx`. Does not exist on *unsafe_function_result* or *protected_function_result*.

Listing 74: function: set a value

```
template <typename T>
proxy& set( T&& value );
```

Sets the value associated with the keys the proxy was generated with to `value`. Does not exist on *unsafe_function_result* or *protected_function_result*.

stack_proxy

`sol::stack_proxy` is what gets returned by *sol::variadic_args* and other parts of the framework. It is similar to proxy, but is meant to alias a stack index and not a named variable.

unsafe_function_result

`unsafe_function_result` is a temporary-only, intermediate-only implicit conversion worker for when *function* is called. It is *NOT* meant to be stored or captured with `auto`. It provides fast access to the desired underlying value. It does not implement `set` / `set_function` / templated `operator=`, as is present on *proxy*.

This type does, however, allow access to multiple underlying values. Use `result.get<Type>(index_offset)` to retrieve an object of `Type` at an offset of `index_offset` in the results. Offset is 0 based. Not specifying an argument defaults the value to 0.

`unsafe_function_result` also has `begin()` and `end()` functions that return (almost) "random-acess" iterators. These return a proxy type that can be implicitly converted to *stack_proxy*.

protected_function_result

`protected_function_result` is a nicer version of `unsafe_function_result` that can be used to detect errors. Its gives safe access to the desired underlying value. It does not implement `set` / `set_function` / templated `operator=` as is present on *proxy*.

This type does, however, allow access to multiple underlying values. Use `result.get<Type>(index_offset)` to retrieve an object of `Type` at an offset of `index_offset` in the results. Offset is 0 based. Not specifying an argument defaults the value to 0.

`unsafe_function_result` also has `begin()` and `end()` functions that return (almost) "random-acess" iterators. These return a proxy type that can be implicitly converted to *stack_proxy*.

on function objects and proxies

Note: As of recent versions of sol2 (2.18.2 and above), this is no longer an issue, as even bound classes will have any detectable function call operator automatically bound to the object, to allow this to work without having to use `.set` or `.set_function`. The note here is kept for posterity and information for older versions. There are only some small caveats, see: *this note here*.

1.11.11 as_container

force a type to be viewed as a container-type when serialized to Lua

```
template <typename T>
struct as_returns_t { ... };

template <typename T>
as_returns_t<T> as_returns( T&& );
```

Sometimes, you have a type whose metatable you claim with a usertype metatable via *usertype semantics*. But, it still has parts of it that make it behave like a container in C++: A `value_type` typedef, an `iterator` typedef, a `begin`, an `end`, and other things that satisfy the Container requirements or the Sequence Container requirements or behaves like a forward_list.

Whatever the case is, you need it to be returned to Lua and have many of the traits and functionality described in the *containers documentation*. Wrap a return type or a setter in `sol::as_container( value );` to allow for a type to be treated like a container, regardless of whether `sol::is_container` triggers or not.

See this container example to see how it works.

1.11.12 nested

```
template <typename T>
struct nested {
        T source;
};
```

`sol::nested<...>` is a template class similar to *sol::as_table*, but with the caveat that every *container type* within the `sol::nested` type will be retrieved as a table from lua. This is helpful when you need to receive C++-style vectors, lists, and maps nested within each other: all of them will be deserialized from lua using table properties rather than anything else.

Note that any caveats with Lua tables apply the moment it is serialized, and the data cannot be gotten out back out in C++ as a C++ type. You can deserialize the Lua table into something explicitly using the `sol::as_table_t` marker for your get and conversion operations using Sol. At that point, the returned type is deserialized **from** a table, meaning you cannot reference any kind of C++ data directly as you do with regular userdata/usertypes. *All C++ type information is lost upon serialization into Lua.*

The example provides a very in-depth look at both `sol::as_table<T>` and `sol::nested<T>`, and how the two are equivalent.

1.11.13 as_table

make sure an object is pushed as a table

```
template <typename T>
as_table_t { ... };

template <typename T>
as_table_t<T> as_function ( T&& container );
```

This function serves the purpose of ensuring that an object is pushed – if possible – like a table into Lua. The container passed here can be a pointer, a reference, a `std::reference_wrapper` around a container, or just a plain container value. It must have a begin/end function, and if it has a `std::pair<Key, Value>` as its `value_type`, it will be pushed as a dictionary. Otherwise, it's pushed as a sequence.

```
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

#include <vector>

int main (int, char*[]) {

        sol::state lua;
        lua.open_libraries();
        lua.set("my_table", sol::as_table(std::vector<int>{ 1, 2, 3, 4, 5 }));
        lua.script("for k, v in ipairs(my_table) do print(k, v) assert(k == v) end");

```

(continues on next page)

(continued from previous page)

```
        return 0;
}
```

Note that any caveats with Lua tables apply the moment it is serialized, and the data cannot be gotten out back out in C++ as a C++ type. You can deserialize the Lua table into something explicitly using the `sol::as_table_t` marker for your get and conversion operations using Sol. At that point, the returned type is deserialized **from** a table, meaning you cannot reference any kind of C++ data directly as you do with regular userdata/usertypes. *All C++ type information is lost upon serialization into Lua.*

If you need this functionality with a member variable, use a *property on a getter function* that returns the result of `sol::as_table`.

This marker does NOT apply to *usertypes*.

You can also use this to nest types and retrieve tables within tables as shown by this example.

1.11.14 usertype<T>

structures and classes from C++ made available to Lua code

Note: `T` refers to the type being turned into a usertype.

While other frameworks extend lua's syntax or create Data Structure Languages (DSLs) to create classes in Lua, *Sol* instead offers the ability to generate easy bindings that pile on performance. You can see a small starter example here. These use metatables and userdata in Lua for their implementation. Usertypes are also runtime extensible.

There are more advanced use cases for how to create and use a usertype, which are all based on how to use its constructor (see below).

enumerations

Listing 75: meta_function enumeration for names

```
enum class meta_function {
        construct,
        index,
        new_index,
        mode,
        call,
        call_function = call,
        metatable,
        to_string,
        length,
        unary_minus,
        addition,
        subtraction,
        multiplication,
        division,
        modulus,
        power_of,
        involution = power_of,
        concatenation,
        equal_to,
```

(continues on next page)

(continued from previous page)

```
        less_than,
        less_than_or_equal_to,
        garbage_collect,
        floor_division,
        bitwise_left_shift,
        bitwise_right_shift,
        bitwise_not,
        bitwise_and,
        bitwise_or,
        bitwise_xor,
        pairs,
        ipairs,
        next,
        type,
        type_info,
};

typedef meta_function meta_method;
```

Use this enumeration to specify names in a manner friendlier than memorizing the special lua metamethod names for each of these. Each binds to a specific operation indicated by the descriptive name of the enum. You can read more about the metamethods in the Lua manual and learn about how they work and are supposed to be implemented there. Each of the names here (except for the ones used as shortcuts to other names like `meta_function::call_function` and `meta_function::involution` and not including `construct`, which just maps to the name `new`) link directly to the Lua name for the operation. `meta_function::pairs` is only available in Lua 5.2 and above (does not include LuaJIT or Lua 5.1) and `meta_function::ipairs` is only available in Lua 5.2 exactly (disregarding compatibiltiy flags).

members

Listing 76: function: usertype<T> constructor

```
template<typename... Args>
usertype<T>(Args&&... args);
```

The constructor of usertype takes a variable number of arguments. It takes an even number of arguments (except in the case where the very first argument is passed as the *constructor list type*). Names can either be strings, *special meta_function enumerations*, or one of the special indicators for initializers.

usertype constructor options

If you don't specify any constructor options at all and the type is default_constructible, Sol will generate a `new` for you. Otherwise, the following are special ways to handle the construction of a usertype:

- **`"{name}", constructors<T(), T(arg-1-0), T(arg-2-0, arg-2-1), ...>`**
  - Specify the constructors to be bound under `name`: list constructors by specifying their function signature with `class_type(arg0, arg1, ... argN)`
  - If you pass the `constructors<...>` argument first when constructing the usertype, then it will automatically be given a `"{name}"` of `"new"`
- **`"{name}", constructors<Type-List-0, Type-List-1, ...>`**

  - This syntax is longer and provided for backwards-compatibility: the above argument syntax is shorter and cleaner
  - `Type-List-N` must be a `sol::types<Args...>`, where `Args...` is a list of types that a constructor takes. Supports overloading by default
  - If you pass the `constructors<...>` argument first when constructing the usertype, then it will automatically be given a `"{name}"` of `"new"`

- **`"{name}", sol::initializers( func1, func2, ... )`**
  - Used to handle *initializer functions* that need to initialize the memory itself (but not actually allocate the memory, since that comes as a userdata block from Lua)
  - **Given one or more functions, provides an overloaded Lua function for creating the specified type**
    - The function must have the argument signature `func( T*, Arguments... )` or `func( T&, Arguments... )`, where the pointer or reference will point to a place of allocated memory that has an uninitialized `T`. Note that Lua controls the memory, so performing a `new` and setting it to the `T*` or `T&` is a bad idea: instead, use `placement new` to invoke a constructor, or deal with the memory exactly as you see fit

- **`{anything}, sol::factories( func1, func2, ... )`**
  - Used to indicate that a factory function (e.g., something that produces a `std::unique_ptr<T, ...>`, `std::shared_ptr<T>`, `T`, or similar) will be creating the object type
  - **Given one or more functions, provides an overloaded function for invoking**
    - The functions can take any form and return anything, since they're just considered to be some plain function and no placement new or otherwise needs to be done. Results from this function will be pushed into Lua according to the same rules as everything else.
    - Can be used to stop the generation of a `.new()` default constructor since a `sol::factories` entry will be recognized as a constructor for the usertype
    - If this is not sufficient, see next 2 entries on how to specifically block a constructor

- **`{anything}, {some_factory_function}`**
  - Essentially binds whatever the function is to name `{anything}`
  - When used WITH the `sol::no_constructor` option below (e.g. `"new", sol::no_constructor` and after that having `"create", &my_creation_func`), one can remove typical constructor avenues and then only provide specific factory functions. Note that this combination is similar to using the `sol::factories` method mentioned earlier in this list. To control the destructor as well, see further below

- **`sol::call_constructor, {valid constructor / initializer / factory}`**
  - The purpose of this is to enable the syntax `local v = my_class( 24 )` and have that call a constructor; it has no other purpose
  - This is compatible with luabind, kaguya and other Lua library syntaxes and looks similar to C++ syntax, but the general consensus in Programming with Lua and other places is to use a function named `new`
  - Note that with the `sol::call_constructor` key, a construct type above must be specified. A free function without it will pass in the metatable describing this object as the first argument without that distinction, which can cause strange runtime errors.

- **`{anything}, sol::no_constructor`**

  - Specifically tells Sol not to create a `.new()` if one is not specified and the type is default-constructible
  - When the key `{anything}` is called on the table, it will result in an error. The error might be that the type is not-constructible.
  - *Use this plus some of the above to allow a factory function for your function type but prevent other types of constructor idioms in Lua*

usertype destructor options

If you don't specify anything at all and the type is destructible, then a destructor will be bound to the garbage collection metamethod. Otherwise, the following are special ways to handle the destruction of a usertype:

- **`"__gc", sol::destructor( func )` or `sol::meta_function::garbage_collect, sol::destructor(`**
  - Creates a custom destructor that takes an argument `T*` or `T&` and expects it to be destructed/destroyed. Note that lua controls the memory and thusly will deallocate the necessary space AFTER this function returns (e.g., do not call `delete` as that will attempt to deallocate memory you did not `new`)
  - If you just want the default constructor, you can replace the second argument with `sol::default_destructor`
  - The usertype will error / throw if you specify a destructor specifically but do not map it to `sol::meta_function::gc` or a string equivalent to `"__gc"`

Note: You MUST specify `sol::destructor` around your destruction function, otherwise it will be ignored.

usertype automatic meta functions

If you don't specify a `sol::meta_function` name (or equivalent string metamethod name) and the type `T` supports certain operations, sol2 will generate the following operations provided it can find a good default implementation:

- **for `to_string` operations where `std::ostream& operator<<( std::ostream&, const T& ),obj.to_str`**
  - a `sol::meta_function::to_string` operator will be generated
  - **writing is done into either**
    - a `std::ostringstream` before the underlying string is serialized into Lua
    - directly serializing the return value of `obj.to_string()` or `to_string( const T& )`
  - order of preference is the `std::ostream& operator<<`, then the member function `obj.to_string()`, then the ADL-lookup based `to_string( const T& )`
  - if you need to turn this behavior off for a type (for example, to avoid compiler errors for ADL conflicts), specialize `sol::is_to_stringable<T>` for your type to be `std::false_type`, like so:

```
namespace sol {
        template <>
        struct is_to_stringable<my_type> : std::false_type {};
}
```

- **for call operations where `operator()( parameters ... )` exists on the C++ type**
  - a `sol::meta_function::call` operator will be generated
  - the function call operator in C++ must not be overloaded, otherwise sol will be unable to bind it automagically
  - the function call operator in C++ must not be templated, otherwise sol will be unable to bind it automagically
  - if it is overloaded or templated, it is your reponsibility to bind it properly
- **for automatic iteration where `begin()` and `end()` exist on the C++ type**
  - a `sol::meta_function::pairs` operator is generated for you
  - Allows you to iterate using `for k, v in pairs( obj ) do ... end` in Lua
  - **Lua 5.2 and better only: LuaJIT does not allow this, Lua 5.1 does NOT allow this**
- **for cases where `.size()` exists on the C++ type**
  - the length operator of Lua (`#my_obj`) operator is generated for you
- **for comparison operations where `operator <` and `operator <=` exist on the C++ type**
  - These two `sol::meta_function::less_than(_or_equal_to)` are generated for you
  - > and >= operators are generated in Lua based on < and <= operators
- **for `operator==`**
  - An equality operator will always be generated, doing pointer comparison if `operator==` on the two value types is not supported or doing a reference comparison and a value comparison if `operator==` is supported
- heterogenous operators cannot be supported for equality, as Lua specifically checks if they use the same function to do the comparison: if they do not, then the equality method is not invoked; one way around this would be to write one `int super_equality_function(lua_State* L) { ... }`, pull out arguments 1 and 2 from the stack for your type, and check all the types and then invoke `operator==` yourself after getting the types out of Lua (possibly using *sol::stack::get* and *sol::stack::check_get*)

usertype regular function options

Otherwise, the following is used to specify functions to bind on the specific usertype for `T`.

- **`"{name}", &free_function`**
  - Binds a free function / static class function / function object (lambda) to `"{name}"`. If the first argument is `T*` or `T&`, then it will bind it as a member function. If it is not, it will be bound as a "static" function on the lua table
- **`"{name}", &type::function_name` or `"{name}", &type::member_variable`**
  - Binds a typical member function or variable to `"{name}"`. In the case of a member variable or member function, `type` must be `T` or a base of `T`
- **`"{name}", sol::readonly( &type::member_variable )`**
  - Binds a typical variable to `"{name}"`. Similar to the above, but the variable will be read-only, meaning an error will be generated if anything attemps to write to this variable
- **`"{name}", sol::as_function( &type::member_variable )`**

  - Binds a typical variable to `"{name}"` *but forces the syntax to be callable like a function*. This produces a getter and a setter accessible by `obj:name()` to get and `obj:name(value)` to set.

- **`"{name}", sol::property( getter_func, setter_func )`**
  - Binds a typical variable to `"{name}"`, but gets and sets using the specified setter and getter functions. Not that if you do not pass a setter function, the variable will be read-only. Also not that if you do not pass a getter function, it will be write-only

- **`"{name}", sol::var( some_value )` or `"{name}", sol::var( std::ref( some_value ) )`**
  - Binds a typical variable to `"{name}"`, optionally by reference (e.g., refers to the same memory in C++). This is useful for global variables / static class variables and the like

- **`"{name}", sol::overload( Func1, Func2, ... )`**
  - Creates an oveloaded member function that discriminates on number of arguments and types
  - Dumping multiple functions out with the same name **does not make an overload**: you must use **this syntax** in order for it to work

- **`sol::base_classes, sol::bases<Bases...>`**
  - Tells a usertype what its base classes are. You need this to have derived-to-base conversions work properly. See *inheritance*

runtime functions

You can add functions at runtime **to the whole class** (not to individual objects). Set a name under the metatable name you bound using `new_usertype` to an object. For example:

Listing 77: runtime_extension.cpp

```
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

#include <iostream>

struct object {
        int value = 0;
};

int main (int, char*[]) {

        std::cout << "==== runtime_extension =====" << std::endl;

        sol::state lua;
        lua.open_libraries(sol::lib::base);

        lua.new_usertype<object>( "object" );

        // runtime additions: through the sol API
        lua["object"]["func"] = [](object& o) { return o.value; };
        // runtime additions: through a lua script
        lua.script("function object:print () print(self:func()) end");

        // see it work
        lua.script("local obj = object.new() \n obj:print()");
```

(continues on next page)

(continued from previous page)

```
26
27          return 0;
28  }
```

Note: You cannot add functions to an individual object. You can only add functions to the whole class / usertype.

overloading

Functions set on a usertype support overloading. See *here* for an example.

inheritance

Sol can adjust pointers from derived classes to base classes at runtime, but it has some caveats based on what you compile with:

If your class has no complicated™ virtual inheritance or multiple inheritance, than you can try to sneak away with a performance boost from not specifying any base classes and doing any casting checks. (What does "complicated™" mean? Ask your compiler's documentation, if you're in that deep.)

For the rest of us safe individuals out there: You must specify the `sol::base_classes` tag with the `sol::bases<Types...>()` argument, where `Types...` are all the base classes of the single type `T` that you are making a usertype out of.

Register the base classes explicitly.

Note: Always specify your bases if you plan to retrieve a base class using the Sol abstraction directly and not casting yourself.

Listing 78: inheritance.cpp

```
1   #define SOL_CHECK_ARGUMENTS 1
2   #include <sol.hpp>
3
4   struct A {
5           int a = 10;
6           virtual int call() { return 0; }
7   };
8   struct B : A {
9           int b = 11;
10          virtual int call() override { return 20; }
11  };
12
13  int main (int, char*[]) {
14
15          sol::state lua;
16
17          lua.new_usertype<B>( "A",
18                  "call", &A::call
19          );
20
21          lua.new_usertype<B>( "B",
```

(continues on next page)

(continued from previous page)

```
                "call", &B::call,
                sol::base_classes, sol::bases<A>()
        );

        return 0;
}
```

Note: You must list ALL base classes, including (if there were any) the base classes of A, and the base classes of those base classes, etc. if you want Sol/Lua to handle them automagically.

Note: Sol does not support down-casting from a base class to a derived class at runtime.

Warning: Specify all base class member variables and member functions to avoid current implementation caveats regarding automatic base member lookup. Sol currently attempts to link base class methods and variables with their derived classes with an undocumented, unsupported feature, provided you specify `sol::bases<...>`. Unfortunately, this can come at the cost of performance, depending on how "far" the base is from the derived class in the bases lookup list. If you do not want to suffer the performance degradation while we iron out the kinks in the implementation (and want it to stay performant forever), please specify all the base methods on the derived class in the method listing you write. In the future, we hope that with reflection we will not have to worry about this.

automagical usertypes

Usertypes automatically register special functions, whether or not they're bound using *new_usertype*. You can turn this off by specializing the `sol::is_automagical<T>` template trait:

```
struct my_strange_nonconfirming_type { /* ... */ };

namespace sol {
        template <>
        struct is_automagical<my_strange_nonconforming_type> : std::false_type {};
}
```

inheritance + overloading

While overloading is supported regardless of inheritance caveats or not, the current version of Sol has a first-match, first-call style of overloading when it comes to inheritance. Put the functions with the most derived arguments first to get the kind of matching you expect or cast inside of an intermediary C++ function and call the function you desire.

compilation speed

Note: MSVC and clang/gcc may need additional compiler flags to handle compiling extensive use of usertypes. See: *the error documentation* for more details.

performance note

Note: Note that performance for member function calls goes down by a fixed overhead if you also bind variables as well as member functions. This is purely a limitation of the Lua implementation and there is, unfortunately, nothing that can be done about it. If you bind only functions and no variables, however, Sol will automatically optimize the Lua runtime and give you the maximum performance possible. *Please consider ease of use and maintenance of code before you make everything into functions.*

1.11.15 usertype memory

memory layout of usertypes

Note: Sol does not take ownership of raw pointers, returned from functions or set through the `set` functions. Return a value, a `std::unique_ptr`, a `std::shared_ptr` of some kind, or hook up the *unique usertypes traits* to work for some specific handle structure you use (AKA, for `boost::shared_ptr`).

The userdata generated by Sol has a specific layout, depending on how Sol recognizes userdata passed into it. All of the referred to metatable names are generated from the name of the class itself. Note that we use 1 metatable per the 3 styles listed below, plus 1 additional metatable that is used for the actual table that you bind with the name when calling `table::new/set_(simple_)usertype`.

In general, we always insert a `T*` in the first `sizeof(T*)` bytes, so the any framework that pulls out those first bytes expecting a pointer will work. The rest of the data has some different alignments and contents based on what it's used for and how it's used.

Warning: The layout of memory described below does **not** take into account alignment. sol2 now takes alignment into account and aligns memory, which is important for misbehaving allocators and types that do not align well to the size of a pointer on their system. If you need to obtain proper alignments for usertypes stored in userdata pointers, **please** use the detail functions named `sol::detail::align_usertype_pointer`, `sol::detail::align_usertype`, and `sol::detail::align_usertype_unique`. This will shift a `void*` pointer by the appropriate amount to reach a certain section in memory. For almost all other use cases, please use `void* memory = lua_touserdata(L, index);`, followed by `memory = sol::detail::align_usertype_pointer( memory );` to adjust the pointer to be at the right place.

Warning: The diagrams and explanations from below is only guaranteed to work 100% of the time if you define *SOL_NO_MEMORY_ALIGNMENT*. Be aware that this may result in unaligned reads/writes, which can crash some older processors and trigger static analyzer/instrumentation tool warnings, like Clang's Address Sanitizer (ASan).

To retrieve a `T`

If you want to retrieve a `T*` pointer to the data managed by a sol2 userdata and are not using sol2's abstractions to do it (e.g., messing with the plain Lua C API), simply use `lua_touserdata` to get the `void*` pointer. Then, execute a `T* object_pointer = *static_cast<T**>(the_void_pointer);`. Every type pushed into C++ that is classified as a userdata (e.g., all user-defined objects that are not covered by the stack abstraction's basic types) can be retrieved in this format, whether they are values or pointers or `unique_ptr`. The reasons for why this works is below.

For `T`

These are classified with a metatable name generally derived from the class name itself.

The data layout for references is as follows:

```
|        T*        |               T               |
^-sizeof(T*) bytes-^-sizeof(T) bytes, actual data-^
```

Lua will clean up the memory itself but does not know about any destruction semantics T may have imposed, so when we destroy this data we simply call the destructor to destroy the object and leave the memory changes to for lua to handle after the "__gc" method exits.

For `T*`

These are classified as a separate `T*` metatable, essentially the "reference" table. Things passed to Sol as a pointer or as a `std::reference<T>` are considered to be references, and thusly do not have a `__gc` (garbage collection) method by default. All raw pointers are non-owning pointers in C++. If you're working with a C API, provide a wrapper around pointers that are supposed to own data and use the constructor/destructor idioms (e.g., with an internal `std::unique_ptr`) to keep things clean.

The data layout for data that only refers is as follows:

```
|        T*        |
^-sizeof(T*) bytes-^
```

That is it. No destruction semantics need to be called.

For `std::unique_ptr<T, D>` and `std::shared_ptr<T>`

These are classified as *"unique usertypes"*, and have a special metatable for them as well. The special metatable is either generated when you add the usertype to Lua using *set_usertype* or when you first push one of these special types. In addition to the data, a deleter function that understands the following layout is injected into the userdata layout.

The data layout for these kinds of types is as follows:

```
|        T*        |    void(*)(void*) function_pointer    |               T               |
↪      |
^-sizeof(T*) bytes-^-sizeof(void(*)(void*)) bytes, deleter-^- sizeof(T) bytes, actal␣
↪data -^
```

Note that we put a special deleter function before the actual data. This is because the custom deleter must know where the offset to the data is and where the special deleter is. In other words, fixed-size-fields come before any variably-sized data (T can be known at compile time, but when serialized into Lua in this manner it becomes a runtime entity). Sol just needs to know about `T*` and the userdata (and userdata metatable) to work, everything else is for preserving construction / destruction semantics.

1.11.16 unique_usertype_traits<T>

trait for hooking special handles / pointers

Listing 79: unique_usertype

```
template <typename T>
struct unique_usertype_traits {
        typedef T type;
        typedef T actual_type;
        static const bool value = false;

        static bool is_null(const actual_type&) {...}

        static type* get (const actual_type&) {...}
};
```

This is a customization point for users who need to *work with special kinds of pointers/handles*. The traits type alerts the library that a certain type is to be pushed as a special userdata with special deletion / destruction semantics, like many smart pointers / custom smart pointers / handles. It is already defined for `std::unique_ptr<T, D>` and `std::shared_ptr<T>` and works properly with those types (see shared_ptr here and unique_ptr here for examples). You can specialize this to get `unique_usertype_traits` semantics with your code. For example, here is how `boost::shared_ptr<T>` would look:

```
namespace sol {
        template <typename T>
        struct unique_usertype_traits<boost::shared_ptr<T>> {
                typedef T type;
                typedef boost::shared_ptr<T> actual_type;
                static const bool value = true;

                static bool is_null(const actual_type& value) {
                        return value == nullptr;
                }

                static type* get (const actual_type& p) {
                        return p.get();
                }
        }
}
```

This will allow the library to properly handle `boost::shared_ptr<T>`, with ref-counting and all. The `type` is the type that lua and sol will interact with, and will allow you to pull out a non-owning reference / pointer to the data when you just ask for a plain `T*` or `T&` or `T` using the getter functions and properties of Sol. The `actual_type` is just the "real type" that controls the semantics (shared, unique, `CComPtr`, `ComPtr`, OpenGL handles, DirectX objects, the list goes on).

Note: If `is_null` triggers (returns `true`), a `nil` value will be pushed into Lua rather than an empty structure.

1.11.17 tie

improved version of std::tie

std::tie() does not work well with *sol::function*'s `sol::function_result` returns. Use `sol::tie` instead. Because they're both named *tie*, you'll need to be explicit when you use Sol's by naming it with the namespace (`sol::tie`), even with a `using namespace sol;`. Here's an example:

```cpp
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

#include "assert.hpp"

int main (int, char*[]) {

	const auto& code = R"(
	bark_power = 11;

	function woof ( bark_energy )
		return (bark_energy * (bark_power / 4))
	end
)";

	sol::state lua;

	lua.script(code);

	sol::function woof = lua["woof"];
	double numwoof = woof(20);
	c_assert(numwoof == 55.0);

	lua.script( "function f () return 10, 11, 12 end" );

	sol::function f = lua["f"];
	std::tuple<int, int, int> abc = f();
	c_assert(std::get<0>(abc) == 10);
	c_assert(std::get<1>(abc) == 11);
	c_assert(std::get<2>(abc) == 12);
	// or
	int a, b, c;
	sol::tie(a, b, c) = f();
	c_assert(a == 10);
	c_assert(b == 11);
	c_assert(c == 12);

	return 0;
}
```

1.11.18 function

calling functions bound to Lua

Note: This abstraction assumes the function runs safely. If you expect your code to have errors (e.g., you don't always have explicit control over it or are trying to debug errors), please use *sol::protected_function* explicitly. You can also make `sol::function` default to `sol::protected_function` by turning on *the safety features*.

```cpp
class unsafe_function : public reference;
typedef unsafe_function function;
```

Function is a correct-assuming version of *protected_function*, omitting the need for typechecks and error handling (thus marginally increasing speed in some cases). It is the default function type of Sol. Grab a function directly off the stack using the constructor:

Listing 80: constructor: unsafe_function

```
unsafe_function(lua_State* L, int index = -1);
```

Calls the constructor and creates this type, straight from the stack. For example:

Listing 81: funcs.lua

```
        bark_power = 11;

        function woof ( bark_energy )
                return (bark_energy * (bark_power / 4))
        end
```

The following C++ code will call this function from this file and retrieve the return value:

```
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

#include "assert.hpp"

int main (int, char*[]) {

        sol::state lua;

        lua.script(code);

        sol::function woof = lua["woof"];
        double numwoof = woof(20);
        c_assert(numwoof == 55.0);
```

The call `woof(20)` generates a *unsafe_function_result*, which is then implicitly converted to an `double` after being called. The intermediate temporary `function_result` is then destructed, popping the Lua function call results off the Lua stack.

You can also return multiple values by using `std::tuple`, or if you need to bind them to pre-existing variables use `sol::tie`:

```
        lua.script( "function f () return 10, 11, 12 end" );

        sol::function f = lua["f"];
        std::tuple<int, int, int> abc = f();
        c_assert(std::get<0>(abc) == 10);
        c_assert(std::get<1>(abc) == 11);
        c_assert(std::get<2>(abc) == 12);
        // or
        int a, b, c;
        sol::tie(a, b, c) = f();
        c_assert(a == 10);
        c_assert(b == 11);
        c_assert(c == 12);

        return 0;
}
```

This makes it much easier to work with multiple return values. Using `std::tie` from the C++ standard will result in dangling references or bad behavior because of the very poor way in which C++ tuples/`std::tie` were specified and implemented: please use `sol::tie( ... )` instead to satisfy any multi-return needs.

Warning: Do NOT save the return type of a *unsafe_function_result* (`function_result` when *safety configurations are not turned on*) with `auto`, as in `auto numwoof = woof(20);`, and do NOT store it anywhere. Unlike its counterpart *protected_function_result*, `function_result` is NOT safe to store as it assumes that its return types are still at the top of the stack and when its destructor is called will pop the number of results the function was supposed to return off the top of the stack. If you mess with the Lua stack between saving `function_result` and it being destructed, you will be subject to an incredible number of surprising and hard-to-track bugs. Don't do it.

Listing 82: function: call operator / function call

```
template<typename... Args>
unsafe_function_result operator()( Args&&... args );

template<typename... Ret, typename... Args>
decltype(auto) call( Args&&... args );

template<typename... Ret, typename... Args>
decltype(auto) operator()( types<Ret...>, Args&&... args );
```

Calls the function. The second `operator()` lets you specify the templated return types using the `my_func(sol::types<int, std::string>, ...)` syntax. Function assumes there are no runtime errors, and thusly will call the `atpanic` function if a detectable error does occur, and otherwise can return garbage / bogus values if the user is not careful.

To know more about how function arguments are handled, see *this note*

1.11.19 protected_function

Lua function calls that trap errors and provide error handling

```
class protected_function : public reference;
typedef protected_function safe_function;
```

Inspired by a request from starwing in the *old sol repository*, this class provides the same interface as *function* but with heavy protection and a potential error handler for any Lua errors and C++ exceptions. You can grab a function directly off the stack using the constructor, or pass to it 2 valid functions, which we'll demonstrate a little later.

When called without the return types being specified by either a `sol::types<...>` list or a `call<Ret...>( ... )` template type list, it generates a *protected_function_result* class that gets implicitly converted to the requested return type. For example:

```
1   function got_problems( error_msg )
2           return "got_problems handler: " .. error_msg
3   end
4
5   function woof ( bark_energy )
6           if bark_energy < 20 then
7                   error("*whine*")
8           end
9           return (bark_energy * (bark_power / 4))
10  end
11
12  function woofers ( bark_energy )
13          if bark_energy < 10 then
```

(continues on next page)

(continued from previous page)

```
                        error("*whine*")
                end
                return (bark_energy * (bark_power / 4))
        end
        )";
```

The following C++ code will call this function from this file and retrieve the return value, unless an error occurs, in which case you can bind an error handling function like so:

```
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

#include <iostream>

int main () {
        sol::state lua;
        lua.open_libraries(sol::lib::base);

        lua.script(code);

        sol::protected_function problematic_woof = lua["woof"];
        problematic_woof.error_handler = lua["got_problems"];

        auto firstwoof = problematic_woof(20);
        if ( firstwoof.valid() ) {
                // Can work with contents
                double numwoof = firstwoof;
                std::cout << "Got value: " << numwoof << std::endl;
        }
        else{
                // An error has occured
                sol::error err = firstwoof;
                std::string what = err.what();
                std::cout << what << std::endl;
        }

        // errors, calls handler and then returns a string error from Lua at the top
↪of the stack
        auto secondwoof = problematic_woof(19);
        if (secondwoof.valid()) {
                // Call succeeded
                double numwoof = secondwoof;
                std::cout << "Got value: " << numwoof << std::endl;
        }
        else {
                // Call failed
                // Note that if the handler was successfully called, this will include
                // the additional appended error message information of
                // "got_problems handler: " ...
                sol::error err = secondwoof;
                std::string what = err.what();
                std::cout << what << std::endl;
        }
```

This code is much more long-winded than its *function* counterpart but allows a person to check for errors. The type here for `auto` are `sol::protected_function_result`. They are implicitly convertible to result types, like all *proxy-style* types are.

Alternatively, with a bad or good function call, you can use `sol::optional` to check if the call succeeded or failed:

```
        // can also use optional to tell things
        sol::optional<double> maybevalue = problematic_woof(19);
        if (maybevalue) {
                // Have a value, use it
                double numwoof = maybevalue.value();
                std::cout << "Got value: " << numwoof << std::endl;
        }
        else {
                std::cout << "No value!" << std::endl;
        }

        std::cout << std::endl;

        return 0;
}
```

That makes the code a bit more concise and easy to reason about if you don't want to bother with reading the error. Thankfully, unlike `sol::unsafe_function_result`, you can save `sol::protected_function_result` in a variable and push/pop things above it on the stack where its returned values are. This makes it a bit more flexible than the rigid, performant `sol::unsafe_function_result` type that comes from calling *sol::unsafe_function*.

If you're confident the result succeeded, you can also just put the type you want (like `double` or `std::string`) right there and it will get it. But, if it doesn't work out, sol can throw and/or panic if you have the *safety* features turned on:

```
// construct with function + error handler
// shorter than old syntax
sol::protected_function problematicwoof(lua["woof"], lua["got_problems"]);

// dangerous if things go wrong!
double value = problematicwoof(19);
```

Finally, it is *important* to note you can set a default handler. The function is described below: please use it to avoid having to constantly set error handlers:

```
// sets got_problems as the default
// handler for all protected_function errors
sol::protected_function::set_default_handler(lua["got_problems"]);

sol::protected_function problematicwoof = lua["woof"];
sol::protected_function problematicwoofers = lua["woofers"];

double value = problematicwoof(19);
double value2 = problematicwoof(9);
```

members

Listing 83: constructor: protected_function

```
template <typename T>
protected_function( T&& func, reference handler = sol::protected_function::get_
↪default_handler() );
protected_function( lua_State* L, int index = -1, reference handler = sol::protected_
↪function::get_default_handler() );
```

(continues on next page)

(continued from previous page)

Constructs a `protected_function`. Use the 2-argument version to pass a custom error handling function more easily. You can also set the *member variable error_handler* after construction later. `protected_function` will always use the latest error handler set on the variable, which is either what you passed to it or the default *at the time of construction*.

Listing 84: function: call operator / protected function call

```
template<typename... Args>
protected_function_result operator()( Args&&... args );

template<typename... Ret, typename... Args>
decltype(auto) call( Args&&... args );

template<typename... Ret, typename... Args>
decltype(auto) operator()( types<Ret...>, Args&&... args );
```

Calls the function. The second `operator()` lets you specify the templated return types using the `my_func(sol::types<int, std::string>, ...)` syntax. If you specify no return type in any way, it produces s `protected_function_result`.

Note: All arguments are forwarded. Unlike *get/set/operator[] on sol::state* or *sol::table*, value semantics are not used here. It is forwarding reference semantics, which do not copy/move unless it is specifically done by the receiving functions / specifically done by the user.

Listing 85: default handlers

```
static const reference& get_default_handler ();
static void set_default_handler( reference& ref );
```

Get and set the Lua entity that is used as the default error handler. The default is a no-ref error handler. You can change that by calling `protected_function::set_default_handler( lua["my_handler"] );` or similar: anything that produces a reference should be fine.

Listing 86: variable: handler

```
reference error_handler;
```

The error-handler that is called should a runtime error that Lua can detect occurs. The error handler function needs to take a single string argument (use type std::string if you want to use a C++ function bound to lua as the error handler) and return a single string argument (again, return a std::string or string-alike argument from the C++ function if you're using one as the error handler). If *exceptions* are enabled, Sol will attempt to convert the `.what()` argument of the exception into a string and then call the error handling function. It is a *reference*, as it must refer to something that exists in the lua registry or on the Lua stack. This is automatically set to the default error handler when `protected_function` is constructed.

Note: `protected_function_result` safely pops its values off the stack when its destructor is called, keeping track of the index and number of arguments that were supposed to be returned. If you remove items below it using `lua_remove`, for example, it will not behave as expected. Please do not perform fundamentally stack-rearranging operations until the destructor is called (pushing/popping above it is just fine).

To know more about how function arguments are handled, see *this note*.

1.11.20 coroutine

resumable/yielding functions from Lua

A `coroutine` is a *reference* to a function in Lua that can be called multiple times to yield a specific result. It is run on the *lua_State* that was used to create it (see *thread* for an example on how to get a coroutine that runs on a thread separate from your usual "main" *lua_State*).

The `coroutine` object is entirely similar to the *protected_function* object, with additional member functions to check if a coroutine has yielded (*call_status::yielded*) and is thus runnable again, whether it has completed (*call_status::ok*) and thus cannot yield anymore values, or whether it has suffered an error (see *status()* and *call_status*'s error codes).

For example, you can work with a coroutine like this:

Listing 87: co.lua

```
    function loop()
        while counter ~= 30
        do
            coroutine.yield(counter);
            counter = counter + 1;
        end
        return counter
    end
```

This is a function that yields:

Listing 88: main.cpp

```
sol::state lua;
lua.open_libraries(sol::lib::base, sol::lib::coroutine);
lua.script_file("co.lua");
sol::coroutine cr = lua["loop"];

for (int counter = 0; // start from 0
    counter < 10 && cr; // we want 10 values, and we only want to run if the␣
↪coroutine "cr" is valid
    // Alternative: counter < 10 && cr.valid()
    ++counter) {
        // Call the coroutine, does the computation and then suspends
        int value = cr();
}
```

Note that this code doesn't check for errors: to do so, you can call the function and assign it as `auto result = cr();`, then check `result.valid()` as is the case with *protected_function*. Finally, you can run this coroutine on another thread by doing the following:

Listing 89: main_with_thread.cpp

```
sol::state lua;
lua.open_libraries(sol::lib::base, sol::lib::coroutine);
lua.script_file("co.lua");
sol::thread runner = sol::thread::create(lua.lua_state());
sol::state_view runnerstate = runner.state();
sol::coroutine cr = runnerstate["loop"];

for (int counter = 0; counter < 10 && cr; ++counter) {
    // Call the coroutine, does the computation and then suspends
```

(continues on next page)

(continued from previous page)

```
    int value = cr();
}
```

The following are the members of `sol::coroutine`:

members

Listing 90: function: constructor

```
coroutine(lua_State* L, int index = -1);
```

Grabs the coroutine at the specified index given a `lua_State*`.

Listing 91: returning the coroutine's status

```
call_status status() const noexcept;
```

Returns the status of a coroutine.

Listing 92: checks for an error

```
bool error() const noexcept;
```

Checks if an error occured when the coroutine was run.

Listing 93: runnable and explicit operator bool

```
bool runnable () const noexcept;
explicit operator bool() const noexcept;
```

These functions allow you to check if a coroutine can still be called (has more values to yield and has not errored). If you have a coroutine object `coroutine my_co = /*...*/`, you can either check `runnable()` or do `if ( my_co ) { /* use coroutine */ }`.

Listing 94: calling a coroutine

```
template<typename... Args>
protected_function_result operator()( Args&&... args );

template<typename... Ret, typename... Args>
decltype(auto) call( Args&&... args );

template<typename... Ret, typename... Args>
decltype(auto) operator()( types<Ret...>, Args&&... args );
```

Calls the coroutine. The second `operator()` lets you specify the templated return types using the `my_co(sol::types<int, std::string>, ...)` syntax. Check `status()` afterwards for more information about the success of the run or just check the coroutine object in an ifs tatement, as shown *above*.

1.11.21 yielding

telling a C++ function to yield its results into Lua

```
template <typename F>
yield_wrapper<F> yielding( F&& f )
```

`sol::yielding` is useful for calling C++ functions which need to yield into a Lua coroutine. It is a wrapper around a single argument which is expected to be bound as a function. You can pass it anywhere a regular function can be bound, **except for in usertype definitions**.

1.11.22 error

the single error/exception type

```
class error : public std::runtime_error {
public:
        error(const std::string& str): std::runtime_error("lua: error: " + str) {}
};
```

Note: Please do not throw this error type yourself. It belongs to the library and we do some information appending at the front.

If an eror is thrown by Sol, it is going to be of this type. We use this in a single place: the default `at_panic` function we bind on construction of a *sol::state*. If you turn *off exceptions*, the chances of you seeing this error are nil unless you specifically use it to pull errors out of things such as *sol::protected_function*.

As it derives from `std::runtime_error`, which derives from `std::exception`, you can catch it with a `catch (const std::exception& )` clause in your try/catch blocks. You can retrieve a string error from Lua (Lua pushes all its errors as string returns) by using this type with any of the get or lookup functions in Sol.

1.11.23 object

general-purpose safety reference to an existing object

```
class object : reference;
```

`object`'s goal is to allow someone to pass around the most generic form of a reference to something in Lua (or propogate a `nil`). It is the logical extension of *sol::reference*, and is used in *sol::table's iterators*.

members

Listing 95: overloaded constructor: object

```
template <typename T>
object(T&&);
object(lua_State* L, int index = -1);
template <typename T, typename... Args>
object(lua_State* L, in_place_t, T&& arg, Args&&... args);
template <typename T, typename... Args>
object(lua_State* L, in_place_type_t<T>, Args&&... args);
```

There are 4 kinds of constructors here. One allows construction of an object from other reference types such as *sol::table* and *sol::stack_reference*. The second creates an object which references the specific element at the given index in the specified `lua_State*`. The more advanced `in_place...` constructors create a single object by

pushing the specified type `T` onto the stack and then setting it as the object. It gets popped from the stack afterwards (unless this is an instance of `sol::stack_object`, in which case it is left on the stack). An example of using this and *sol::make_object* can be found in the any_return example.

Listing 96: function: type conversion

```
template<typename T>
decltype(auto) as() const;
```

Performs a cast of the item this reference refers to into the type `T` and returns it. It obeys the same rules as *sol::stack::get<T>*.

Listing 97: function: type check

```
template<typename T>
bool is() const;
```

Performs a type check using the *sol::stack::check* api, after checking if the reference is valid.

non-members

Listing 98: functions: nil comparators

```
bool operator==(const object& lhs, const nil_t&);
bool operator==(const nil_t&, const object& rhs);
bool operator!=(const object& lhs, const nil_t&);
bool operator!=(const nil_t&, const object& rhs);
```

These allow a person to compare an `sol::object` against *nil*, which essentially checks if an object references a non-nil value, like so:

```
if (myobj == sol::nil) {
        // doesn't have anything...
}
```

Use this to check objects.

1.11.24 thread

a separate state that can contain and run functions

```
class thread : public reference { /* ... */ };
```

`sol::thread` is a separate runnable part of the Lua VM that can be used to execute work separately from the main thread, such as with *coroutines*. To take a table or a coroutine and run it specifically on the `sol::thread` you either pulled out of lua or created, just get that function through the *state of the thread*

Note: A CPU thread is not always equivalent to a new thread in Lua: `std::this_thread::get_id()` can be the same for 2 callbacks that have 2 distinct Lua threads. In order to know which thread a callback was called in, hook into *sol::this_state* from your Lua callback and then construct a `sol::thread`, passing in the `sol::this_state` for both the first and last arguments. Then examine the results of the status and `is_...` calls below.

---

free function

Listing 99: function: main_thread

```
main_thread(lua_State* current, lua_State* backup_if_bad_platform = nullptr);
```

The function `sol::main_thread( ... )` retrieves the main thread of the application on Lua 5.2 and above *only*. It is designed for code that needs to be multithreading-aware (e.g., uses multiple *threads* and *coroutines*).

Warning: This code function will be present in Lua 5.1/LuaJIT, but only have proper behavior when given a single argument on Lua 5.2 and beyond. Lua 5.1 does not support retrieving the main thread from its registry, and therefore it is entirely suggested if you are writing cross-platform Lua code that you must store the main thread of your application in some global storage accessible somewhere. Then, pass this item into the `sol::main_thread( possibly_thread_state, my_actual_main_state )` and it will select that `my_actual_main_state` every time. If you are not going to use Lua 5.1 / LuaJIT, you can ignore the last parameter.

members

Listing 100: constructor: thread

```
thread(stack_reference r);
thread(lua_State* L, int index = -1);
thread(lua_State* L, lua_State* actual_thread);
```

Takes a thread from the Lua stack at the specified index and allows a person to use all of the abstractions therein. It can also take an actual thread state to make a thread from that as well.

Listing 101: function: view into thread_state()'s state

```
state_view state() const;
```

This retrieves the current state of the thread, producing a *state_view* that can be manipulated like any other. *Coroutines* pulled from Lua using the thread's state will be run on that thread specifically.

Listing 102: function: retrieve thread state object

```
lua_State* thread_state () const;
```

This function retrieves the `lua_State*` that represents the thread.

Listing 103: current thread status

```
thread_status status () const;
```

Retrieves the *thread status* that describes the current state of the thread.

Listing 104: main thread status

```
bool is_main_thread () const;
```

Checks to see if the thread is the main Lua thread.

Listing 105: function: thread creation

```
thread create();
static thread create (lua_State* L);
```

Creates a new thread from the given a `lua_State*`.

1.11.25 optional<T>

This is an implemention of optional from the standard library. If it detects that a proper optional exists, it will attempt to use it. This is mostly an implementation detail, used in the *sol::stack::check_get* and *sol::stack::get<optional<T>>* and `sol::optional<T> maybe_value = table["arf"];` implementations for additional safety reasons.

See this example here for a demonstration on how to use it and other features!

1.11.26 variadic_args

transparent argument to deal with multiple parameters to a function

```
struct variadic_args;
```

This class is meant to represent every single argument at its current index and beyond in a function list. It does not increment the argument count and is thus transparent. You can place it anywhere in the argument list, and it will represent all of the objects in a function call that come after it, whether they are listed explicitly or not.

`variadic_args` also has `begin()` and `end()` functions that return (almost) random-acess iterators. These return a proxy type that can be implicitly converted to a type you want, much like the *table proxy type*.

```
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

#include <iostream>

int main() {
        std::cout << "=== variadic_args ===" << std::endl;

        sol::state lua;
        lua.open_libraries(sol::lib::base);

        // Function requires 2 arguments
        // rest can be variadic, but:
        // va will include everything after "a" argument,
        // which means "b" will be part of the varaidic_args list too
        // at position 0
        lua.set_function("v", [](int a, sol::variadic_args va, int /*b*/) {
                int r = 0;
                for (auto v : va) {
                        int value = v; // get argument out (implicit conversion)
                                       // can also do int v = v.as<int>();
                                       // can also do int v = va.get<int>(i);
↪ with index i
                        r += value;
                }
                // Only have to add a, b was included from variadic_args and beyond
                return r + a;
```

(continues on next page)

(continued from previous page)

```
        });

        lua.script("x = v(25, 25)");
        lua.script("x2 = v(25, 25, 100, 50, 250, 150)");
        lua.script("x3 = v(1, 2, 3, 4, 5, 6)");
        // will error: not enough arguments
        //lua.script("x4 = v(1)");

        lua.script("assert(x == 50)");
        lua.script("assert(x2 == 600)");
        lua.script("assert(x3 == 21)");
        lua.script("print(x)"); // 50
        lua.script("print(x2)"); // 600
        lua.script("print(x3)"); // 21

        std::cout << std::endl;

        return 0;
}
```

You can also "save" arguments and the like later, by stuffing them into a `std::vector<sol::object>` or something similar that serializes them into the registry. Below is an example of saving all of the arguments provided by `sol::variadic_args` in a lambda capture variable called `args`.

```
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

#include <iostream>
#include <functional>

int main() {

        std::cout << "=== variadic_args serialization/storage ===" << std::endl;

        sol::state lua;
        lua.open_libraries(sol::lib::base);

        std::function<void()> function_storage;

        auto store_routine = [&function_storage] (sol::function f, sol::variadic_args va) {
                function_storage = [f, args = std::vector<sol::object>(va.begin(), va.end())]() {
                        f(sol::as_args(args));
                };
        };

        lua.set_function("store_routine", store_routine);

        lua.script(R"(
function a(name)
        print(name)
end
store_routine(a, "some name")
)");
        function_storage();
```

(continues on next page)

(continued from previous page)

```

        lua.script(R"(
function b(number, text)
        print(number, "of", text)
end
store_routine(b, 20, "these apples")
)");
        function_storage();

        std::cout << std::endl;

        return 0;
}
```

Finally, note that you can use `sol::variadic_args` constructor to "offset"/"shift over" the arguments being viewed:

```
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

#include <iostream>

int main () {

        std::cout << "=== variadic_args shifting constructor ===" << std::endl;

        sol::state lua;
        lua.open_libraries(sol::lib::base);

        lua.set_function("f", [](sol::variadic_args va) {
                int r = 0;
                sol::variadic_args shifted_va(va.lua_state(), 3);
                for (auto v : shifted_va) {
                        int value = v;
                        r += value;
                }
                return r;
        });

        lua.script("x = f(1, 2, 3, 4)");
        lua.script("x2 = f(8, 200, 3, 4)");
        lua.script("x3 = f(1, 2, 3, 4, 5, 6)");

        lua.script("print(x)"); // 7
        lua.script("print(x2)"); // 7
        lua.script("print(x3)"); // 18

        std::cout << std::endl;

        return 0;
}
```

1.11.27 variadic_results

push multiple disparate arguments into lua

```
struct variadic_results : std::vector<object> { ... };
```

This type allows someone to prepare multiple returns before returning them into Lua. It derives from `std::vector`, so it can be used exactly like that, and objects can be added using the various constructors and functions relating to *sol::object*. You can see it and other return-type helpers in action here.

1.11.28 as_args

turn an iterable argument into multiple arguments

```
template <typename T>
struct as_args_t { ... };

template <typename T>
as_args_t<T> as_args( T&& );
```

`sol::as_args` is a function that that takes an iterable and turns it into multiple arguments to a function call. It forwards its arguments, and is meant to be used as shown below:

Listing 106: args_from_container.cpp

```
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

#include <iostream>
#include <vector>
#include <set>

int main(int , const char*[]) {

        std::cout << "=== args_from_container ===" << std::endl;

        sol::state lua;
        lua.open_libraries();

        lua.script("function f (a, b, c, d) print(a, b, c, d) end");

        sol::function f = lua["f"];

        std::vector<int> v2{ 3, 4 };
        f(1, 2, sol::as_args(v2));

        std::set<int> v4{ 3, 1, 2, 4 };
        f(sol::as_args(v4));

        int v3[] = { 2, 3, 4 };
        f(1, sol::as_args(v3));

        std::cout << std::endl;

        return 0;
}
```

It is basically implemented as a one-way customization point. For more information about customization points, see the *tutorial on how to customize Sol to work with your types*.

1.11.29 as_returns

turn an iterable argument into a multiple-return type

```
template <typename T>
struct as_returns_t { ... };

template <typename T>
as_returns_t<T> as_returns( T&& );
```

This allows you to wrap up a source that has `begin` and `end` iterator-returning functions on it and return it as multiple results into Lua. To have more control over the returns, use *sol::variadic_results*.

```
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

#include "assert.hpp"

#include <string>
#include <set>

int main () {
        sol::state lua;

        lua.set_function("f", []() {
                std::set<std::string> results{ "arf", "bark", "woof" };
                return sol::as_returns(std::move(results));
        });

        lua.script("v1, v2, v3 = f()");

        std::string v1 = lua["v1"];
        std::string v2 = lua["v2"];
        std::string v3 = lua["v3"];

        c_assert(v1 == "arf");
        c_assert(v2 == "bark");
        c_assert(v3 == "woof");

        return 0;
}
```

1.11.30 overload

calling different functions based on argument number/type

Listing 107: function: create overloaded set

```
template <typename... Args>
struct overloaded_set : std::tuple<Args...> { /* ... */ };

template <typename... Args>
overloaded_set<Args...> overload( Args&&... args );
```

The actual class produced by `sol::overload` is essentially a type-wrapper around `std::tuple` that signals to the library that an overload is being created. The function helps users make overloaded functions that can be called from Lua using 1 name but multiple arguments. It is meant to replace the spaghetti of code where

users mock this up by doing strange if statements and switches on what version of a function to call based on luaL_check{number/udata/string}.

Note: Please note that default parameters in a function (e.g., `int func(int a = 20)`) do not exist beyond C++'s compile-time fun. When that function gets bound or serialized into Lua's framework, it is bound as a function taking 1 argument, not 2 functions taking either 0 or 1 argument. If you want to achieve the same effect, then you need to use overloading and explicitly call the version of the function you want. There is no magic in C++ that allows me to retrieve default parameters and set this up automatically.

Note: Overload resolution can be affected by configuration defines in the *safety pages*. For example, it is impossible to differentiate between integers (uint8_t, in32_t, etc.) versus floating-point types (float, double, half) when `SOL_SAFE_NUMERICS` is not turned on.

Its use is simple: wherever you can pass a function type to Lua, whether its on a *usertype* or if you are just setting any kind of function with `set` or `set_function` (for *table* or *state(_view)*), simply wrap up the functions you wish to be considered for overload resolution on one function like so:

```
sol::overload( func1, func2, ... funcN );
```

The functions can be any kind of function / function object (lambda). Given these functions and struct:

```
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

#include "assert.hpp"

#include <iostream>

struct pup {
        int barks = 0;

        void bark () {
                ++barks; // bark!
        }

        bool is_cute () const {
                return true;
        }
};

void ultra_bark( pup& p, int barks) {
        for (; barks --> 0;) p.bark();
}

void picky_bark( pup& p, std::string s) {
        if ( s == "bark" )
            p.bark();
}
```

You then use it just like you would for any other part of the api:

```
int main () {
        std::cout << "=== overloading with members ===" << std::endl;

```

(continues on next page)

(continued from previous page)

```
        sol::state lua;
        lua.open_libraries(sol::lib::base);

        lua.set_function( "bark", sol::overload(
                ultra_bark,
                []() { return "the bark from nowhere"; }
        ) );

        lua.new_usertype<pup>( "pup",
                // regular function
                "is_cute", &pup::is_cute,
                // overloaded function
                "bark", sol::overload( &pup::bark, &picky_bark )
        );
```

Doing the following in Lua will call the specific overloads chosen, and their associated functions:

```
        const auto& code = R"(
        barker = pup.new()
        print(barker:is_cute())
        barker:bark() -- calls member function pup::bark
        barker:bark("meow") -- picky_bark, no bark
        barker:bark("bark") -- picky_bark, bark

        bark(barker, 20) -- calls ultra_bark
        print(bark()) -- calls lambda which returns that string
        )";

        lua.script(code);

        pup& barker = lua["barker"];
        std::cout << barker.barks << std::endl;
        c_assert(barker.barks == 22);

        std::cout << std::endl;
        return 0;
}
```

Note: Overloading is done on a first-come, first-serve system. This means if two overloads are compatible, workable overloads, it will choose the first one in the list.

Note that because of this system, you can use *sol::variadic_args* to make a function that serves as a "fallback". Be sure that it is the last specified function in the listed functions for `sol::overload( ... )`. This example shows how.

Note: Please keep in mind that doing this bears a runtime cost to find the proper overload. The cost scales directly not exactly with the number of overloads, but the number of functions that have the same argument count as each other (Sol will early-eliminate any functions that do not match the argument count).

1.11.31 property

wrapper to specify read and write variable functionality using functions

```
template <typename Read, typename Write>
decltype(auto) property ( Read&& read_function, Write&& write_function );
template <typename Read>
decltype(auto) property ( Read&& read_function );
template <typename Write>
decltype(auto) property ( Write&& write_function );
```

These set of functions create a type which allows a setter and getter pair (or a single getter, or a single setter) to be used to create a variable that is either read-write, read-only, or write-only. When used during *usertype* construction, it will create a variable that uses the setter/getter member function specified.

```
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

#include "assert.hpp"

#include <iostream>

class Player {
public:
        int get_hp() const {
                return hp;
        }

        void set_hp( int value ) {
                hp = value;
        }

        int get_max_hp() const {
                return hp;
        }

        void set_max_hp( int value ) {
                maxhp = value;
        }

private:
        int hp = 50;
        int maxhp = 50;
};

int main (int, char*[]) {

        std::cout << "=== properties from C++ functions ===" << std::endl;

        sol::state lua;
        lua.open_libraries(sol::lib::base);

        lua.set("theplayer", Player());

        // Yes, you can register after you set a value and it will
        // connect up the usertype automatically
        lua.new_usertype<Player>( "Player",
                "hp", sol::property(&Player::get_hp, &Player::set_hp),
                "maxHp", sol::property(&Player::get_max_hp, &Player::set_max_hp)
        );

```

(continues on next page)

(continued from previous page)

```
        const auto& code = R"(
        -- variable syntax, calls functions
        theplayer.hp = 20
        print('hp:', theplayer.hp)
        print('max hp:', theplayer.maxHp)
        )";

        lua.script(code);

        return 0;
}
```

1.11.32 var

For hooking up static / global variables to Lua usertypes

The sole purpose of this tagging type is to work with *usertypes* to provide `my_class.my_static_var` access, and to also provide reference-based access as well.

```
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

#include "assert.hpp"
#include <iostream>

struct test {
        static int number;
};
int test::number = 25;


int main() {
        sol::state lua;
        lua.open_libraries();
        lua.new_usertype<test>("test",
                "direct", sol::var(2),
                "number", sol::var(test::number),
                "ref_number", sol::var(std::ref(test::number))
        );

        int direct_value = lua["test"]["direct"];
        c_assert(direct_value == 2);

        int number = lua["test"]["number"];
        c_assert(number == 25);
        int ref_number = lua["test"]["ref_number"];
        c_assert(ref_number == 25);

        test::number = 542;

        // number is its own memory: was passed by value
        // So does not change
        int number_again = lua["test"]["number"];
        c_assert(number_again == 25);

```

(continues on next page)

(continued from previous page)

```
        // ref_number is just test::number
        // passed through std::ref
        // so, it holds a reference
        // which can be updated
        int ref_number_again = lua["test"]["ref_number"];
        c_assert(ref_number_again == 542);
        // be careful about referencing local variables,
        // if they go out of scope but are still reference
        // you'll suffer dangling reference bugs!

        return 0;
}
```

1.11.33 protect

routine to mark a function / variable as requiring safety

```
template <typename T>
auto protect( T&& value );
```

`sol::protect( my_func )` allows you to protect a function call or member variable call when it is being set to Lua. It can be used with usertypes or when just setting a function into Sol. Below is an example that demonstrates that a call that would normally not error without *Safety features turned on* that instead errors and makes the Lua safety-call wrapper `pcall` fail:

```
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

#include "assert.hpp"

int main () {
        struct protect_me {
                int gen(int x) {
                        return x;
                }
        };

        sol::state lua;
        lua.open_libraries(sol::lib::base);
        lua.new_usertype<protect_me>("protect_me",
                "gen", sol::protect( &protect_me::gen )
        );

        lua.script(R"__(
        pm = protect_me.new()
        value = pcall(pm.gen,"wrong argument")
        )__");
        bool value = lua["value"];
        c_assert(!value);

        return 0;
}
```

1.11.34 filters

stack modification right before lua call returns

`sol::filters` is an advanced, low-level modification feature allowing you to take advantage of sol2's abstractions before applying your own stack-based modifications at the last moment. They cover the same functionality as luabind's "return reference to" and "dependency" types. A few pre-rolled filters are defined for your use:

filter	usage	modification
`sol::returns_self`	`sol::filters( some_function, sol::returns_self() )`	• takes the argument at stack index 1 (`self` in member function calls and lambdas that take a specific userdata first) and makes that to be the return value • rather than creating a new userdata that references the same C++ memory, it copies the userdata, similar to writing `obj2 = obj1` just increases the reference count • saves memory space on top of keeping original memory alive
`sol::returns_self_with<int..>`	`sol::filters( some_function, sol::returns_self_with<2, 3>() )`	• same as above, with the caveat that the `self` is returned while also putting dependencies into the `self` • can keep external dependencies alive
`sol::self_dependency`	`sol::filters( some_function, sol::self_dependency() );`	• this makes the value returned by the bindable take a dependency on the `self` argument • useful for returning a reference to a member variable and keeping the parent class of that member variable alive
`sol::stack_dependencies`	`sol::filters( some_function, sol::stack_dependencies( target_index, 2, 1, ... ) );`	• whatever is at `target_index` on the stack is given a special "keep alive" table with the elements on the stack specified by the integer indices after `target_index` • allows you to keep arguments and other things alive for the duration of the existence of the class
custom	`sol::filters( some_function, [](lua_State* L, int current_stack_return_count) -> int { ... } )`	• whatever you want, so long as it has the form `int (lua_State*, int )` • works with callables (such as lambdas), so long as it has the correct form • expected to return the number of things on the stack to return to Lua

- "some_function" can be any callable function, member variable, or similar
- dependency additions only work on userdata

You can specify multiple filters on the same `sol::filters` call, and can also specify custom filters as long as the signature is correct.

1.11.35 readonly

routine to mark a member variable as read-only

```
template <typename T>
auto readonly( T&& value );
```

The goal of read-only is to protect a variable set on a usertype or a function. Simply wrap it around a member variable, e.g. `sol::readonly( &my_class::my_member_variable )` in the appropriate place to use it. If someone tries to set it, it will error their code.

`sol::readonly` is especially important when you're working with types that do not have a copy constructor. Lua does not understand move semantics, and therefore setters to user-defined-types require a C++ copy constructor. Containers as member variables that contain types that are not copyable but movable – e.g. `std::vector<my_move_only_type>` amongst others – also can erroneously state they are copyable but fail with compiler errors. If your type does not fit a container's definition of being copyable or is just not copyable in general and it is a member variable, please use `sol::readonly`.

If you are looking to make a read-only table, you need to go through a bit of a complicated song and dance by overriding the `__index` metamethod. Here's a complete example on the way to do that using `sol`:

Listing 108: read_only.cpp

```
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

#include <iostream>

struct object {
    void my_func() {
        std::cout << "hello\n";
    }
};

int deny(lua_State* L) {
    return luaL_error(L, "HAH! Deniiiiied!");
}

int main() {
    sol::state lua;
    lua.open_libraries(sol::lib::base);

    object my_obj;

    sol::table obj_table = lua.create_named_table("object");

    sol::table obj_metatable = lua.create_table_with();
    obj_metatable.set_function("my_func", &object::my_func, &my_obj);
    // Set whatever else you need to
    // on the obj_metatable,
    // not on the obj_table itself!

    // Properly self-index metatable to block things
    obj_metatable[sol::meta_function::new_index] = deny;
```

(continues on next page)

(continued from previous page)

```
    obj_metatable[sol::meta_function::index] = obj_metatable;

    // Set it on the actual table
    obj_table[sol::metatable_key] = obj_metatable;

    try {
        lua.script(R"(
print(object.my_func)
object["my_func"] = 24
print(object.my_func)
        )");
    }
    catch (const std::exception& e) {
        std::cout << "an expected error occurred: " << e.what() << std::endl;
    }
    return 0;
}
```

It is a verbose example, but it explains everything. Because the process is a bit involved and can have unexpected consequences for users that make their own tables, making read-only tables is something that we ask the users to do themselves with the above code, as getting the semantics right for the dozens of use cases would be tremendously difficult.

1.11.36 as_function

make sure an object is pushed as a function

```
template <typename Sig = sol::function_sig<>, typename... Args>
function_argumants<Sig, Args...> as_function ( Args&& ... );
```

This function serves the purpose of ensuring that a callable struct (like a lambda) can be passed to the `set( key, value )` calls on *sol::table* and be treated like a function binding instead of a userdata. It is recommended that one uses the *sol::table::set_function* call instead, but if for some reason one must use `set`, then `as_function` can help ensure a callable struct is handled like a lambda / callable, and not as just a userdata structure.

This class can also make it so usertypes bind variable types as functions to for usertype bindings.

```
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

int main () {
        struct callable {
                int operator()( int a, bool b ) {
                        return a + b ? 10 : 20;
                }
        };


        sol::state lua;
        // Binds struct as userdata
        // can still be callable, but beware
        // caveats
        lua.set( "not_func", callable() );
        // Binds struct as function
        lua.set( "func", sol::as_function( callable() ) );
```

(continues on next page)

(continued from previous page)

```
        // equivalent: lua.set_function( "func", callable() );
        // equivalent: lua["func"] = callable();
}
```

Note that if you actually want a userdata, but you want it to be callable, you simply need to create a *sol::table::new_usertype* and then bind the `"__call"` metamethod (or just use `sol::meta_function::call` *enumeration*). This may or may not be done automatically for you, depending on whether or not the call operator is overloaded and such.

Here's an example of binding a variable as a function to a usertype:

```
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

int main () {
        class B {
        public:
                int bvar = 24;
        };

        sol::state lua;
        lua.open_libraries(sol::lib::base);
        lua.new_usertype<B>("B",
                // bind as variable
                "b", &B::bvar,
                // bind as function
                "f", sol::as_function(&B::bvar)
        );

        B b;
        lua.set("b", &b);
        lua.script(R"(x = b:f()
                y = b.b
                assert(x == 24)
                assert(y == 24)
        )");

        return 0;
}
```

1.11.37 c_call

templated type to transport functions through templates

```
template <typename Function, Function f>
int c_call (lua_State* L);

template <typename... Functions>
int c_call (lua_State* L);
```

The goal of `sol::c_call<...>` is to provide a way to wrap a function and transport it through a compile-time context. This enables faster speed at the cost of a much harder to read / poorer interface, and can alleviate some template compilation speed issues. `sol::c_call` expects a type for its first template argument, and a value of the previously provided type for the second template argument. To make a compile-time transported overloaded function, specify multiple functions in the same `type, value` pairing, but put it inside of a `sol::wrap`.

Note: This can also be placed into the argument list for a *usertype* as well.

This pushes a raw `lua_CFunction` into whatever you pass the resulting `c_call` function pointer into, whether it be a table or a userdata or whatever else using sol2's API. The resulting `lua_CFunction` can also be used directly with the lua API, just like many of sol2's types can be intermingled with Lua's API if you know what you're doing.

It is advisable for the user to consider making a macro to do the necessary `decltype( &function_name, ), function_name`. Sol does not provide one because many codebases already have one similar to this.

Here's an example below of various ways to use `sol::c_call`:

```
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

#include "assert.hpp"

int f1(int) { return 32; }

int f2(int, int) { return 1; }

struct fer {
        double f3(int, int) {
                return 2.5;
        }
};


int main() {

        sol::state lua;
        // overloaded function f
        lua.set("f", sol::c_call<sol::wrap<decltype(&f1), &f1>, sol::wrap<decltype(&
↪f2), &f2>, sol::wrap<decltype(&fer::f3), &fer::f3>>);
        // singly-wrapped function
        lua.set("g", sol::c_call<sol::wrap<decltype(&f1), &f1>>);
        // without the 'sol::wrap' boilerplate
        lua.set("h", sol::c_call<decltype(&f2), &f2>);
        // object used for the 'fer' member function call
        lua.set("obj", fer());

        // call them like any other bound function
        lua.script("r1 = f(1)");
        lua.script("r2 = f(1, 2)");
        lua.script("r3 = f(obj, 1, 2)");
        lua.script("r4 = g(1)");
        lua.script("r5 = h(1, 2)");

        // get the results and see
        // if it worked out
        int r1 = lua["r1"];
        c_assert(r1 == 32);
        int r2 = lua["r2"];
        c_assert(r2 == 1);
        double r3 = lua["r3"];
        c_assert(r3 == 2.5);
        int r4 = lua["r4"];
        c_assert(r4 == 32);
```

(continues on next page)

(continued from previous page)

```
        int r5 = lua["r5"];
        c_assert(r5 == 1);

        return 0;
}
```

1.11.38 resolve

utility to pick overloaded C++ function calls

Listing 109: function: resolve C++ overload

```
template <typename... Args, typename F>
constexpr auto resolve( F f );
```

`resolve` is a function that is meant to help users pick a single function out of a group of overloaded functions in C++. It works for *both member and free functions* You can use it to pick overloads by specifying the signature as the first template argument. Given a collection of overloaded functions:

```
int overloaded(int x);
int overloaded(int x, int y);
int overloaded(int x, int y, int z);

struct thing {
        int overloaded() const;
        int overloaded(int x);
        int overloaded(int x, int y);
        int overloaded(int x, int y, int z);
};
```

You can disambiguate them using `resolve`:

```
auto one_argument_func = resolve<int(int)>( overloaded );
auto two_argument_func = resolve<int(int, int)>( overloaded );
auto three_argument_func = resolve<int(int, int, int)>( overloaded );
auto member_three_argument_func = resolve<int(int, int, int)>( &thing::overloaded );
auto member_zero_argument_const_func = resolve<int() const>( &thing::overloaded );
```

It is *important* to note that `const` is placed at the end for when you desire const overloads. You will get compiler errors if you are not specific and do not properly disambiguate for const member functions. This resolution also becomes useful when setting functions on a *table* or *state_view*:

```
sol::state lua;

lua.set_function("a", resolve<int(int)>( overloaded ) );
lua.set_function("b", resolve<int(int, int)>( overloaded ));
lua.set_function("c", resolve<int(int, int, int)>( overloaded ));
```

It can also be used with *sol::c_call*:

```
sol::state lua;

auto f = sol::c_call<
        decltype(sol::resolve<int(int, int)>(&overloaded)),
```

(continues on next page)

(continued from previous page)

```
        sol::resolve<int(int, int)>(&overloaded)
>;
lua.set_function("f", f);

lua.script("f(1, 2)");
```

Note: You cannot use `sol::resolve<...>(...)` when one function is templated and it has a non-templated overload: it will always fail in this case. To resolve this, please use a manual `static_cast<R(Args...)>( &func )` or `static_cast<R (T::*)(Args...)>( &T::overloaded_member_func )` (with the right const-ness and volatile-ness and r-value/l-value qualifiers if necessary).

1.11.39 stack namespace

the nitty-gritty core abstraction layer over Lua

```
namespace stack
```

If you find that the higher level abstractions are not meeting your needs, you may want to delve into the `stack` namespace to try and get more out of Sol. `stack.hpp` and the `stack` namespace define several utilities to work with Lua, including pushing / popping utilities, getters, type checkers, Lua call helpers and more. This namespace is not thoroughly documented as the majority of its interface is mercurial and subject to change between releases to either heavily boost performance or improve the Sol *api*.

Working at this level of the stack can be enhanced by understanding how the Lua stack works in general and then supplementing it with the objects and items here.

There are, however, a few *template customization points* that you may use for your purposes and a handful of potentially handy functions. These may help if you're trying to slim down the code you have to write, or if you want to make your types behave differently throughout the Sol stack. Note that overriding the defaults **can** throw out many of the safety guarantees Sol provides: therefore, modify the *extension points* at your own discretion.

structures

Listing 110: struct: record

```
struct record {
        int last;
        int used;

        void use(int count);
};
```

This structure is for advanced usage with *stack::get* and *stack::check_get*. When overriding the customization points, it is important to call the `use` member function on this class with the amount of things you are pulling from the stack. `used` contains the total accumulation of items produced. `last` is the number of items gotten from the stack with the last operation (not necessarily popped from the stack). In all trivial cases for types, `last == 1` and `used == 1` after an operation; structures such as `std::pair` and `std::tuple` may pull more depending on the classes it contains.

When overriding the *customization points*, please note that this structure should enable you to push multiple return values and get multiple return values to the stack, and thus be able to seamlessly pack/unpack return values from Lua into a single C++ struct, and vice-versa. This functionality is only recommended for people who need to customize

the library further than the basics. It is also a good way to add support for the type and propose it back to the original library so that others may benefit from your work.

Note that customizations can also be put up on a separate page here, if individuals decide to make in-depth custom ones for their framework or other places.

Listing 111: struct: probe

```
struct probe {
        bool success;
        int levels;

        probe(bool s, int l);
        operator bool() const;
};
```

This struct is used for showing whether or not a *probing get_field* was successful or not.

members

Listing 112: function: call_lua

```
template<bool check_args = stack_detail::default_check_arguments, bool clean_stack =␣
↪true, typename Fx, typename... FxArgs>
inline int call_lua(lua_State* L, int start, Fx&& fx, FxArgs&&... fxargs);
```

This function is helpful for when you bind to a raw C function but need sol's abstractions to save you the agony of setting up arguments and know how calling C functions works. The `start` parameter tells the function where to start pulling arguments from. The parameter `fx` is what's supposed to be called. Extra arguments are passed to the function directly. There are intermediate versions of this (`sol::stack::call_into_lua` and similar) for more advanced users, but they are not documented as they are subject to change to improve performance or adjust the API accordingly in later iterations of sol2. Use the more advanced versions at your own peril.

Listing 113: function: get

```
template <typename T>
auto get( lua_State* L, int index = -1 )
template <typename T>
auto get( lua_State* L, int index, record& tracking )
```

Retrieves the value of the object at `index` in the stack. The return type varies based on `T`: with primitive types, it is usually `T`: for all unrecognized `T`, it is generally a `T&` or whatever the extension point *stack::getter<T>* implementation returns. The type `T` has top-level `const` qualifiers and reference modifiers removed before being forwarded to the extension point *stack::getter<T>* struct. `stack::get` will default to forwarding all arguments to the *stack::check_get* function with a handler of `type_panic` to strongly alert for errors, if you ask for the *safety*.

You may also retrieve an *sol::optional<T>* from this as well, to have it attempt to not throw errors when performing the get and the type is not correct.

Listing 114: function: check

```
template <typename T>
bool check( lua_State* L, int index = -1 )

template <typename T, typename Handler>
bool check( lua_State* L, int index, Handler&& handler )
```

(continues on next page)

(continued from previous page)

```
template <typename T, typename Handler>
bool check( lua_State* L, int index, Handler&& handler, record& tracking )
```

Checks if the object at `index` is of type `T`. If it is not, it will call the `handler` function with `lua_State* L`, `int index`, `sol::type expected`, and `sol::type actual` as arguments (and optionally with a 5th string argument `sol::string_view message`. If you do not pass your own handler, a `no_panic` handler will be passed.

Listing 115: function: get_usertype

```
template <typename T>
auto get_usertype( lua_State* L, int index = -1 )
template <typename T>
auto get_usertype( lua_State* L, int index, record& tracking )
```

Directly attempts to rertieve the type `T` using sol2's usertype mechanisms. Similar to a regular `get` for a user-defined type. Useful when you need to access sol2's usertype getter mechanism while at the same time providing your own customization.

Listing 116: function: check_usertype

```
template <typename T>
bool check_usertype( lua_State* L, int index = -1 )

template <typename T, typename Handler>
bool check_usertype( lua_State* L, int index, Handler&& handler )

template <typename T, typename Handler>
bool check_usertype( lua_State* L, int index, Handler&& handler, record& tracking )
```

Checks if the object at `index` is of type `T` and stored as a sol2 usertype. Useful when you need to access sol2's usertype checker mechanism while at the same time providing your own customization.

Listing 117: function: check_get

```
template <typename T>
auto check_get( lua_State* L, int index = -1 )
template <typename T, typename Handler>
auto check_get( lua_State* L, int index, Handler&& handler, record& tracking )
```

Retrieves the value of the object at `index` in the stack, but does so safely. It returns an `optional<U>`, where `U` in this case is the return type deduced from `stack::get<T>`. This allows a person to properly check if the type they're getting is what they actually want, and gracefully handle errors when working with the stack if they so choose to. You can define `SOL_CHECK_ARGUMENTS` to turn on additional *safety*, in which `stack::get` will default to calling this version of the function with some variant on a handler of `sol::type_panic_string` to strongly alert for errors and help you track bugs if you suspect something might be going wrong in your system.

Listing 118: function: push

```
// push T inferred from call site, pass args... through to extension point
template <typename T, typename... Args>
int push( lua_State* L, T&& item, Args&&... args )

// push T that is explicitly specified, pass args... through to extension point
```

(continues on next page)

(continued from previous page)

```
template <typename T, typename Arg, typename... Args>
int push( lua_State* L, Arg&& arg, Args&&... args )

// recursively call the the above "push" with T inferred, one for each argument
template <typename... Args>
int multi_push( lua_State* L, Args&&... args )
```

Based on how it is called, pushes a variable amount of objects onto the stack. in 99% of cases, returns for 1 object pushed onto the stack. For the case of a `std::tuple<...>`, it recursively pushes each object contained inside the tuple, from left to right, resulting in a variable number of things pushed onto the stack (this enables multi-valued returns when binding a C++ function to a Lua). Can be called with `sol::stack::push<T>( L, args... )` to have arguments different from the type that wants to be pushed, or `sol::stack::push( L, arg, args... )` where `T` will be inferred from `arg`. The final form of this function is `sol::stack::multi_push`, which will call one `sol::stack::push` for each argument. The `T` that describes what to push is first sanitized by removing top-level `const` qualifiers and reference qualifiers before being forwarded to the extension point *stack::pusher<T>* struct.

Listing 119: function: push_reference

```
// push T inferred from call site, pass args... through to extension point
template <typename T, typename... Args>
int push_reference( lua_State* L, T&& item, Args&&... args )

// push T that is explicitly specified, pass args... through to extension point
template <typename T, typename Arg, typename... Args>
int push_reference( lua_State* L, Arg&& arg, Args&&... args )

// recursively call the the above "push" with T inferred, one for each argument
template <typename... Args>
int multi_push_reference( lua_State* L, Args&&... args )
```

These functinos behave similarly to the ones above, but they check for specific criteria and instead attempt to push a reference rather than forcing a copy if appropriate. Use cautiously as sol2 uses this mainly as a return from usertype functions and variables to preserve chaining/variable semantics from that a class object. Its internals are updated to fit the needs of sol2 and while it generally does the "right thing" and has not needed to be changed for a while, sol2 reserves the right to change its internal detection mechanisms to suit its users needs at any time, generally without breaking backwards compatibility and expectations but not exactly guaranteed.

Listing 120: function: pop

```
template <typename... Args>
auto pop( lua_State* L );
```

Pops an object off the stack. Will remove a fixed number of objects off the stack, generally determined by the `sol::lua_size<T>` traits of the arguments supplied. Generally a simplicity function, used for convenience.

Listing 121: function: top

```
int top( lua_State* L );
```

Returns the number of values on the stack.

Listing 122: function: set_field

```
template <bool global = false, typename Key, typename Value>
void set_field( lua_State* L, Key&& k, Value&& v );

template <bool global = false, typename Key, typename Value>
void set_field( lua_State* L, Key&& k, Value&& v, int objectindex);
```

Sets the field referenced by the key `k` to the given value `v`, by pushing the key onto the stack, pushing the value onto the stack, and then doing the equivalent of `lua_setfield` for the object at the given `objectindex`. Performs optimizations and calls faster verions of the function if the type of `Key` is considered a c-style string and/or if its also marked by the templated `global` argument to be a global.

Listing 123: function: get_field

```
template <bool global = false, typename Key>
void get_field( lua_State* L, Key&& k [, int objectindex] );
```

Gets the field referenced by the key `k`, by pushing the key onto the stack, and then doing the equivalent of `lua_getfield`. Performs optimizations and calls faster verions of the function if the type of `Key` is considered a c-style string and/or if its also marked by the templated `global` argument to be a global.

This function leaves the retrieved value on the stack.

Listing 124: function: probe_get_field

```
template <bool global = false, typename Key>
probe probe_get_field( lua_State* L, Key&& k [, int objectindex] );
```

Gets the field referenced by the key `k`, by pushing the key onto the stack, and then doing the equivalent of `lua_getfield`. Performs optimizations and calls faster verions of the function if the type of `Key` is considered a c-style string and/or if its also marked by the templated `global` argument to be a global. Furthermore, it does this safely by only going in as many levels deep as is possible: if the returned value is not something that can be indexed into, then traversal queries with `std::tuple`/`std::pair` will stop early and return probing information with the *probe struct*.

This function leaves the retrieved value on the stack.

objects (extension points)

You can customize the way Sol handles different structures and classes by following the information provided in the *adding your own types*.

Below is more extensive information for the curious.

The structs below are already overriden for a handful of types. If you try to mess with them for the types `sol` has already overriden them for, you're in for a world of thick template error traces and headaches. Overriding them for your own user defined types should be just fine, however.

Listing 125: struct: getter

```
template <typename T, typename = void>
struct getter {
        static T get (lua_State* L, int index, record& tracking) {
                // ...
                return // T, or something related to T.
```

(continues on next page)

(continued from previous page)

```
        }
};
```

This is an SFINAE-friendly struct that is meant to expose static function `get` that returns a `T`, or something convertible to it. The default implementation assumes `T` is a usertype and pulls out a userdata from Lua before attempting to cast it to the desired `T`. There are implementations for getting numbers (`std::is_floating`, `std::is_integral`-matching types), getting `std::string` and `const char*`, getting raw userdata with *userdata_value* and anything as upvalues with *upvalue_index*, getting raw lua_CFunction s, and finally pulling out Lua functions into `std::function<R(Args...)>`. It is also defined for anything that derives from *sol::reference*. It also has a special implementation for the 2 standard library smart pointers (see *usertype memory*).

Listing 126: struct: pusher

```
template <typename X, typename = void>
struct pusher {
        template <typename T>
        static int push ( lua_State* L, T&&, ... ) {
                // can optionally take more than just 1 argument
                // ...
                return // number of things pushed onto the stack
        }
};
```

This is an SFINAE-friendly struct that is meant to expose static function `push` that returns the number of things pushed onto the stack. The default implementation assumes `T` is a usertype and pushes a userdata into Lua with a class-specific, state-wide metatable associated with it. There are implementations for pushing numbers (`std::is_floating`, `std::is_integral`-matching types), getting `std::string` and `const char*`, getting raw userdata with *userdata* and raw upvalues with *upvalue*, getting raw lua_CFunction s, and finally pulling out Lua functions into `sol::function`. It is also defined for anything that derives from *sol::reference*. It also has a special implementation for the 2 standard library smart pointers (see *usertype memory*).

Listing 127: struct: checker

```
template <typename T, type expected = lua_type_of<T>, typename = void>
struct checker {
        template <typename Handler>
        static bool check ( lua_State* L, int index, Handler&& handler, record&␣
↪tracking ) {
                // if the object in the Lua stack at index is a T, return true
                if ( ... ) {
                        tracking.use(1); // or however many you use
                        return true;
                }
                // otherwise, call the handler function,
                // with the required 4/5 arguments, then return false
                //
                handler(L, index, expected, indextype, "optional message");
                return false;
        }
};
```

This is an SFINAE-friendly struct that is meant to expose static function `check` that returns whether or not a type at a given index is what its supposed to be. The default implementation simply checks whether the expected type passed in through the template is equal to the type of the object at the specified index in the Lua stack. The default implementation for types which are considered `userdata` go through a myriad of checks to support checking if a type is *actually* of type `T` or if its the base class of what it actually stored as a userdata in that index. Down-casting

from a base class to a more derived type is, unfortunately, impossible to do.

Listing 128: struct: userdata_checker

```
template <typename T, typename = void>
struct userdata_checker {
        template <typename Handler>
        static bool check ( lua_State* L, int index, type indextype, Handler&&␣
↪handler, record& tracking ) {
                // implement custom checking here for a userdata:
                // if it doesn't match, return "false" and regular
                // sol userdata checks will kick in
                return false;
                // returning true will skip sol's
                // default checks
        }
};
```

This is an SFINAE-friendly struct that is meant to expose a function `check=` that returns `true` if a type meets some custom userdata specifiction, and `false` if it does not. The default implementation just returns `false` to let the original sol2 handlers take care of everything. If you want to implement your own usertype checking; e.g., for messing with `toLua` or `OOLua` or `kaguya` or some other libraries. Note that the library must have a with a *memory compatible layout* if you **want to specialize this checker method but not the subsequent getter method**. You can specialize it as shown in the interop examples.

Note: You must turn this feature on with `SOL_ENABLE_INTEROP`, as described in the *config and safety section*.

Listing 129: struct: userdata_getter

```
template <typename T, typename = void>
struct userdata_getter {
        static std::pair<bool, T*> get ( lua_State* L, int index, void* unadjusted_
↪pointer, record& tracking ) {
                // implement custom getting here for non-sol2 userdatas:
                // if it doesn't match, return "false" and regular
                // sol userdata checks will kick in
                return { false, nullptr };
        }
};
```

This is an SFINAE-friendly struct that is meant to expose a function `get` that returns `true` and an adjusted pointer if a type meets some custom userdata specifiction (from, say, another library or an internal framework). The default implementation just returns `{ false, nullptr }` to let the original sol2 getter take care of everything. If you want to implement your own usertype getter; e.g., for messing with `kaguya` or some other libraries. You can specialize it as shown in the interop examples.

Note: You do NOT need to use this method in particular if the *memory layout* is compatible. (For example, `toLua` stores userdata in a sol2-compatible way.)

Note: You must turn it on with `SOL_ENABLE_INTEROP`, as described in the *config and safety section*.

1.11.40 light<T>/user<T>

utility class for the cheapest form of (light) userdata

```
template <typename T>
struct user;

template <typename T>
struct light;
```

`sol::user<T>` and `sol::light<T>` are two utility classes that do not participate in the full *sol::usertype<T>* system. The goal of these classes is to provide the most minimal memory footprint and overhead for putting a single item and getting a single item out of Lua. `sol::user<T>`, when pushed into Lua, will create a thin, unnamed metatable for that instance specifically which will be for calling its destructor. `sol::light<T>` specifically pushes a reference / pointer into Lua as a `sol::type::lightuserdata`.

If you feel that you do not need to have something participate in the full *usertype<T>* system, use the utility functions `sol::make_user( ... )` and `sol::make_light( ... )` to create these types and store them into Lua. You can get them off the Lua stack / out of the Lua system by using the same retrieval techniques on `get` and `operator[]` on tables and with stack operations.

Both have implicit conversion operators to `T*` and `T&`, so you can set them immediately to their respective pointer and reference types if you need them.

1.11.41 compatibility.hpp

Lua 5.3/5.2 compatibility for Lua 5.1/LuaJIT

This is a detail header used to maintain compatability with the 5.2 and 5.3+ APIs. It contains code from the MIT-Licensed Lua code in some places and also from the lua-compat repository by KeplerProject.

It is not fully documented as this header's only purpose is for internal use to make sure Sol compiles across all platforms / distributions with no errors or missing Lua functionality. If you think there's some compatibility features we are missing or if you are running into redefinition errors, please make an issue in the issue tracker.

If you have this already in your project or you have your own compatibility layer, then please `#define SOL_NO_COMPAT 1` before including `sol.hpp` or pass this flag on the command line to turn off the compatibility wrapper.

For the licenses, see *here*

1.11.42 types

nil, lua_primitive type traits, and other fundamentals

The `types.hpp` header contains various fundamentals and utilities of Sol.

enumerations

Listing 130: syntax of a function called by Lua

```
enum class call_syntax {
dot = 0,
colon = 1
};
```

This enumeration indicates the syntax a function was called with in a specific scenario. There are two ways to call a function: with `obj:func_name( ... )` or `obj.func_name( ... );` The first one passes "obj" as the first argument: the second one does not. In the case of usertypes, this is used to determine whether the call to a *constructor/initializer* was called with a `:` or a `.`, and not misalign the arguments.

Listing 131: status of a Lua function call

```
enum class call_status : int {
    ok       = LUA_OK,
    yielded  = LUA_YIELD,
    runtime  = LUA_ERRRUN,
    memory   = LUA_ERRMEM,
    handler  = LUA_ERRERR,
    gc       = LUA_ERRGCMM
};
```

This strongly-typed enumeration contains the errors potentially generated by a call to a *protected function* or a *coroutine*.

Listing 132: status of a Lua thread

```
enum class thread_status : int {
    ok   = LUA_OK,
    yielded = LUA_YIELD,
    runtime = LUA_ERRRUN,
    memory  = LUA_ERRMEM,
    gc      = LUA_ERRGCMM,
    handler = LUA_ERRERR,
    dead,
};
```

This enumeration contains the status of a thread. The `thread_status::dead` state is generated when the thread has nothing on its stack and it is not running anything.

Listing 133: status of a Lua load operation

```
enum class load_status : int {
    ok       = LUA_OK,
    runtime  = LUA_ERRSYNTAX,
    memory   = LUA_ERRMEM,
    gc       = LUA_ERRGCMM,
    file     = LUA_ERRFILE,
};
```

This enumeration contains the status of a load operation from *state::load(_file)*.

Listing 134: type enumeration

```
enum class type : int {
    none          = LUA_TNONE,
    nil           = LUA_TNIL,
    string        = LUA_TSTRING,
    number        = LUA_TNUMBER,
    thread        = LUA_TTHREAD,
    boolean       = LUA_TBOOLEAN,
    function      = LUA_TFUNCTION,
    userdata      = LUA_TUSERDATA,
```

(continues on next page)

(continued from previous page)

```
    lightuserdata = LUA_TLIGHTUSERDATA,
    table         = LUA_TTABLE,
    poly          = none    | nil      | string    | number    | thread          |
                    table   | boolean  | function  | userdata  | lightuserdata
};
```

The base types that Lua natively communicates in and understands. Note that "poly" isn't really a true type, it's just a symbol used in Sol for something whose type hasn't been checked (and you should almost never see it).

type traits

Listing 135: lua_type_of trait

```
template <typename T>
struct lua_type_of;
```

This type trait maps a C++ type to a *type enumeration* value. The default value is `type::userdata`.

Listing 136: primitive checking traits

```
template <typename T>
struct is_lua_primitive;

template <typename T>
struct is_proxy_primitive;
```

This trait is used by *proxy* to know which types should be returned as references to internal Lua memory (e.g., `userdata` types) and which ones to return as values (strings, numbers, *references*). `std::reference_wrapper`, `std::tuple<...>` are returned as values, but their contents can be references. The default value is false.

special types

Listing 137: nil

```
strunil_t {};
const nil_t nil {};
bool operator==(nil_t, nil_t);
bool operator!=(nil_t, nil_t);
```

`nil` is a constant used to signify Lua's `nil`, which is a type and object that something does not exist. It is comparable to itself, *sol::object* and *proxy values*.

Listing 138: non_null

```
template <typename T>
struct non_null {};
```

A tag type that, when used with *stack::get<non_null<T*>>*, does not perform a `nil` check when attempting to retrieve the userdata pointer.

Listing 139: type list

```
template <typename... Args>
struct types;
```

A type list that, unlike `std::tuple<Args...>`, does not actually contain anything. Used to indicate types and groups of types all over Sol.

functions

Listing 140: type_of

```
template<typename T>
type type_of();

type type_of(lua_State* L, int index);
```

These functions get the type of a C++ type `T`; or the type at the specified index on the Lua stack.

Listing 141: type checking convenience functions

```
int type_panic_string(lua_State* L, int index, type expected, type actual, const␣
↪std::string& message);

int type_panic_c_str(lua_State* L, int index, type expected, type actual, const char*␣
↪message);

int no_panic(lua_State*, int, type, type, const char*) noexcept;

void type_error(lua_State* L, int expected, int actual);

void type_error(lua_State* L, type expected, type actual);

void type_assert(lua_State* L, int index, type expected, type actual);

void type_assert(lua_State* L, int index, type expected);
```

The purpose of these functions is to assert / throw / crash / error (or do nothing, as is the case with `no_panic`). They're mostly used internally in the framework, but they're provided here if you should need them.

Listing 142: type name retrieval

```
std::string type_name(lua_State*L, type t);
```

Gets the Lua-specified name of the *type*.

structs

```
struct userdata_value {
        void* value;
};

struct light_userdata_value {
        void* value;
```

(continues on next page)

(continued from previous page)

```
};

struct upvalue_index {
        int index;
};

struct raw_index {
        int index;
};

struct absolute_index {
        int index;
};

struct ref_index {
        int index;
};
```

Types that differentiate between the two kinds of `void*` Lua hands back from its API: full userdata and light userdata, as well as a type that modifies the index passed to `get` to refer to up values These types can be used to trigger different underlying API calls to Lua when working with *stack* namespace and the `push`/`get`/`pop`/`check` functions.

The `raw_index` type is used to tell a *sol::reference* type or similar that the desired index – negative or not – should be passed through directly to the API.

The `absolute_index` type is used to tell a *sol::reference* type or similar that the desired index – negative or not – should be passed through Lua's lua_absindex function first to adjust where it is, and then given to the underlying API.

The `ref_index` type is used to tell a *sol::reference* type or similar that it should look into the Lua C Registry for its type.

1.11.43 metatable_key

a key for setting and getting an object's metatable

```
struct metatable_key_t {};
const metatable_key_t metatable_key;
```

You can use this in conjunction with *sol::table* to set/get a metatable. Lua metatables are powerful ways to override default behavior of objects for various kinds of operators, among other things. Here is an entirely complete example, showing getting and working with a *usertype*'s metatable defined by Sol:

Listing 143: messing with metatables

```
1  #define SOL_CHECK_ARGUMENTS 1
2  #include <sol.hpp>
3
4  #include "assert.hpp"
5
6  int main () {
7
8          struct bark {
9                  int operator()(int x) {
10                         return x;
11                 }
12         };
```

(continues on next page)

(continued from previous page)

```

        sol::state lua;
        lua.open_libraries(sol::lib::base);

        lua.new_usertype<bark>("bark",
                sol::meta_function::call_function, &bark::operator()
        );

        bark b;
        lua.set("b", &b);

        sol::table b_as_table = lua["b"];
        sol::table b_metatable = b_as_table[sol::metatable_key];
        sol::function b_call = b_metatable["__call"];
        sol::function b_as_function = lua["b"];

        int result1 = b_as_function(1);
        // pass 'self' directly to argument
        int result2 = b_call(b, 1);
        c_assert(result1 == result2);
        c_assert(result1 == 1);
        c_assert(result2 == 1);
}
```

1.11.44 new_table

a table creation hint to environment/table

```
struct new_table;

constexpr const new_table create = new_table{};
```

`sol::new_table` serves the purpose of letting you create tables using the constructor of *sol::table* and *sol::environment*. It also disambiguates the other kinds of constructors, so is **necessary** to be specified. Leaving it off will result in the wrong constructor to be called, for either `sol::table` or `sol::environment`.

members

Listing 144: constructor: new_table

```
new_table(int sequence_hint = 0, int map_hint = 0);
```

The constructor's sole purpose is to either let you default-constructor the type, in which case it uses the values of "0" for its two hints, or letting you specify either `sequence_hint` or both the `sequence_hint` and `map_hint`. Each hint is a heuristic helper for Lua to allocate an appropriately sized and structured table for what you intend to do. In 99% of cases, you will most likely not care about it and thusly will just use the constant `sol::create` as the second argument to object-creators like `sol::table`'s constructor.

1.12 mentions

so does anyone cool use this thing... ?

First off, feel free to tell me about your uses!

Okay, so the features don't convince you, the documentation doesn't convince you, you want to see what *other* people think about Sol? Well, aside from the well-wishes that come through in the issue tracker, here's a few things floating around about sol2 that I occasionally get pinged about:

eevee demonstrating the sheer code reduction by using sol2:

```
// TODO this appears to also iterate over actors that are in another actor's inventory...
static int zlua_allactors_iter(lua_State *L);
static int zlua_allactors(lua_State *L) {
    // Create space for the iterator
    // TODO note must be first etc
    TThinkerIterator<AActor>** iter = (TThinkerIterator<AActor>**)lua_newuserdata(L,
↳ sizeof(TThinkerIterator<AActor>*));

    // Set its metatable BEFORE opening it
    luaL_getmetatable(L, "zdoom.TThinkerIterator.AActor");
    lua_setmetatable(L, -2);

    // Create the iterator
    *iter = new TThinkerIterator<AActor>();
    // TODO check for null??
    // Create and return the iterator function, whose upvalue (closed-over
    // value) is the iterator we just created
    lua_pushcclosure(L, zlua_allactors_iter, 1);
    return 1;
}

static int zlua_allactors_iter(lua_State *L) {
    // TODO luaL_checkudata
    TThinkerIterator<AActor>** iter = (TThinkerIterator<AActor>**)lua_touserdata(L, lua_upvalueindex(1));
    AActor *actor = (*iter)->Next();
    if (actor) {
        luaZ_pushactor(L, actor);
        return 1;
    }
    else {
        return 0;
    }
}

static int zlua_tthinkeraactor_gc(lua_State *L) {
    TThinkerIterator<AActor>** iter = (TThinkerIterator<AActor>**)lua_touserdata(L, 1);
    delete *iter;
    return 0;
}
```

```
lua["allactors"] = [] {
    TThinkerIterator<AActor> iter;
    // TODO upgrade sol, rename to as_function
    return sol::function_args([iter]() mutable {
        return iter.Next();
    });
};
```

- In High Performance Computing research
- **The Multiple Arcade Machine Emulator (MAME) project switched from using LuaBridge to sol2!**
  - The pull request in which it was introduced to the master branch.
- For scripting, in OpenMPT
- **(CppNow) sol2 was mentioned in a comparison to other scripting languages by ChaiScript developer, Jason Turner (@left**
  - Jason Turner's presentation
- **(CppCast) Showed up in CppCast with Elias Daler!**
  - Elias Daler's blog
  - CppCast
- **(Eevee) A really nice and neat developer/artist/howaretheysotalented person is attempting to use it for zdoom!**
  - eevee's blog
- **(Twitter) Twitter has some people that link it:**
  - The image above, tweeted out by eevee
  - Eevee: "I heartily recommend sol2"
  - Elias Daler: "sol2 saved my life."
  - Racod's Lair: "from outdated LuaBridge to superior #sol2"
- **(Reddit) Posts on reddit about it!**

  - sol2's initial reddit release
  - Benchmarking Discussing

- **Somehow landed on a Torque3D thread...**
  - http://forums.torque3d.org/viewtopic.php?f=32&t=629&p=5246&sid=8e759990ab1ce38a48e896fc9fd62653#p5241

Are you using sol2 for something neat? Want it to be featured here or think it's unfair that ThePhD hasn't found it yet? Well, drop an issue in the repo or send an e-mail!

1.13 benchmarks

1.13.1 because somebody is going to ask eventually...

Here are measurements of the *overhead that libraries impose around the Lua C API*: that is, the cost of abstracting / wrapping the plain Lua C API. Measurements are (at the time of writing) done with all libraries compiled against a DLL version of Lua 5.3.3 to make sure each Lua call has the same overhead between libraries (no Link Time Optimizations or other static library optimizations).

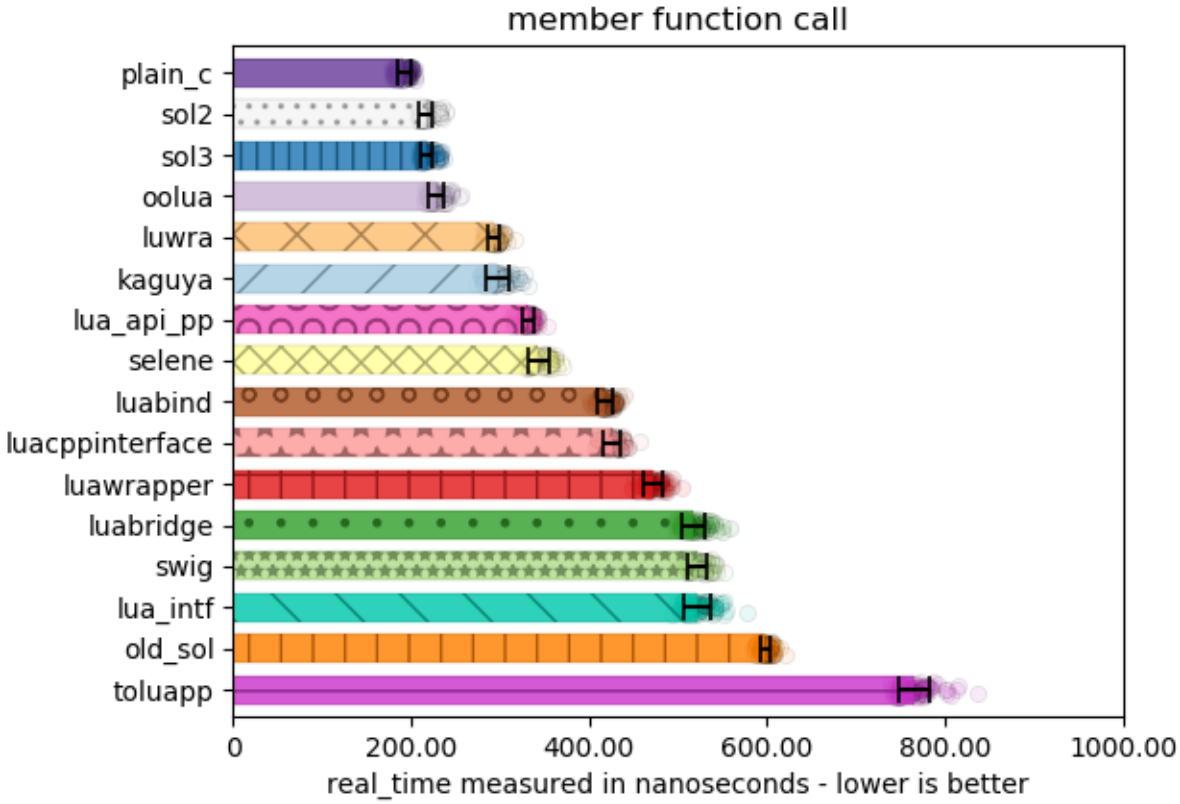
These are some informal and formal benchmarks done by both the developers of sol and other library developers / users. We leave you to interpret the data as you see fit.

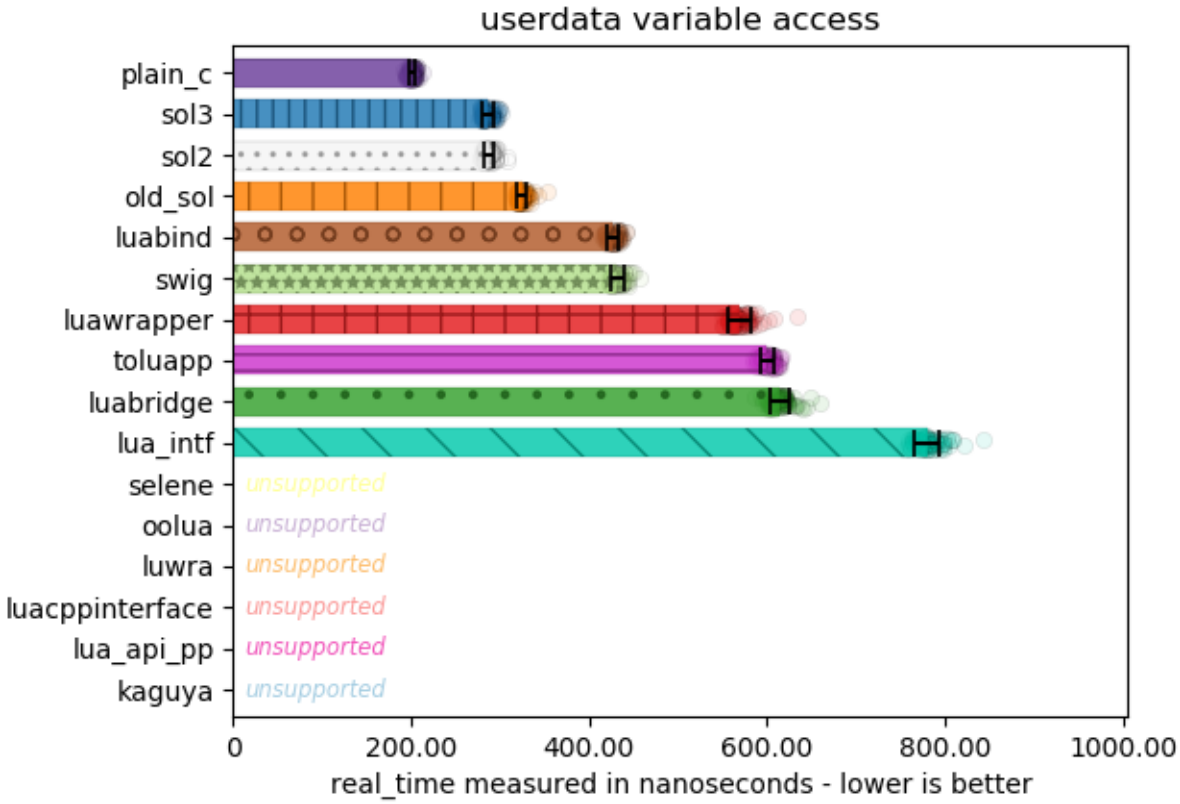
- lua_binding_benchmarks by satoren (developer of kaguya) (sol is the "sol2" entry)
- lua-bindings-shootout by ThePhD (developer of sol)

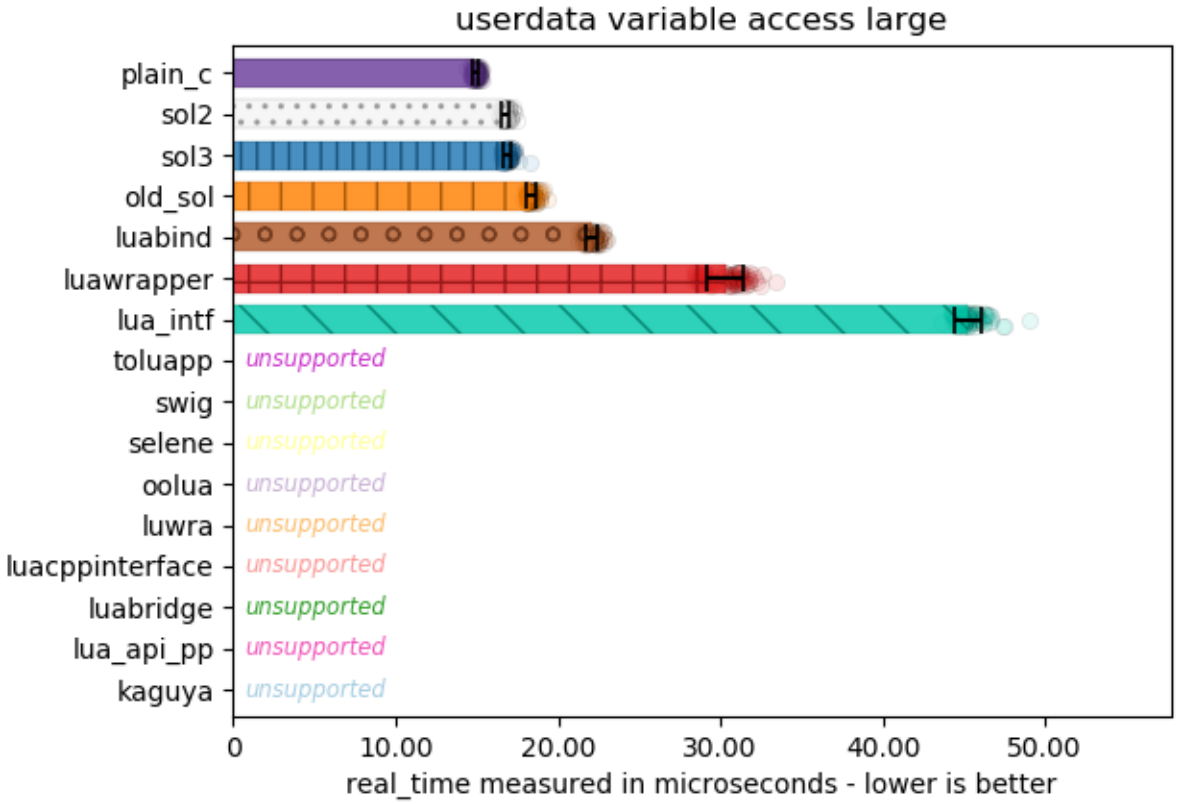
As of the writing of this documentation (May 17th, 2018), *sol* seems to take the cake in most categories for speed! Below are some graphs from lua-bindings-shootout. You can read the benchmarking code there if you think something was done wrong, and submit a pull requests or comment on something to make sure that ThePhD is being honest about his work. All categories are the performance of things described at the top of the *feature table*.

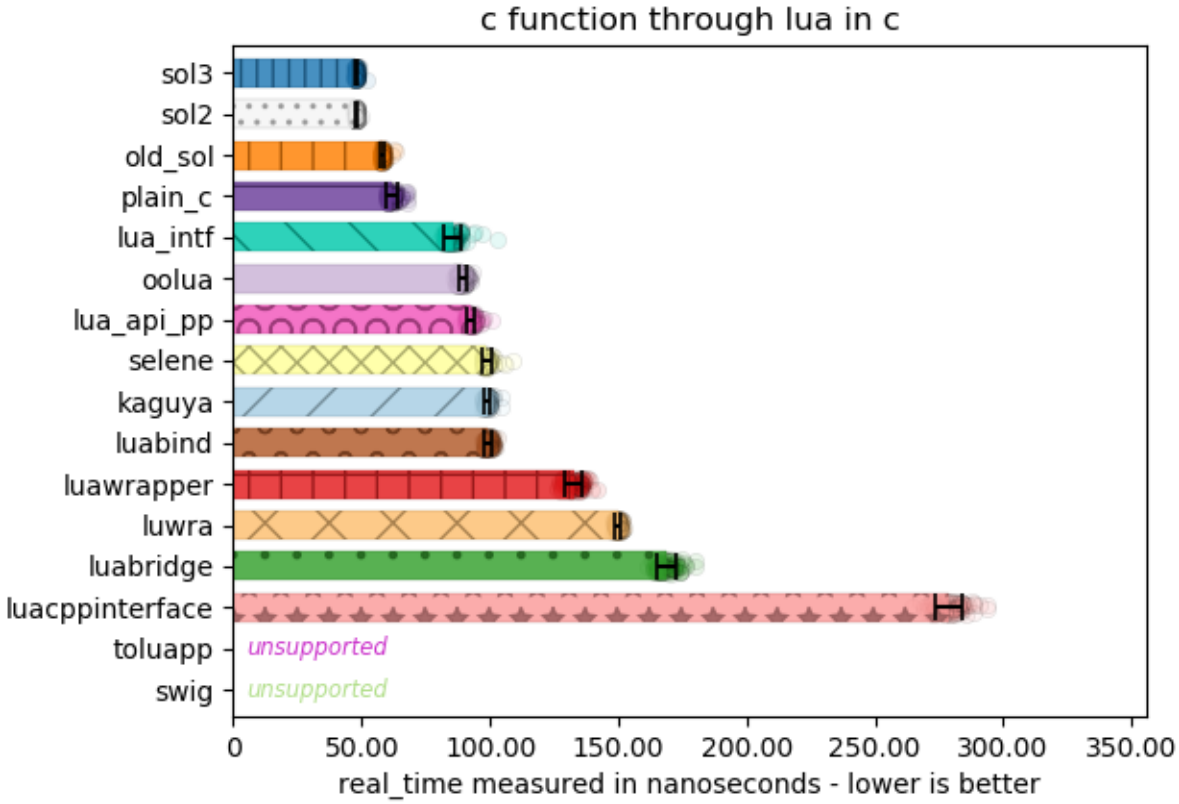
Note that sol here makes use of its more performant variants (see *c_call* and others), and ThePhD also does his best to make use of the most performant variants for other frameworks by disabling type checks where possible as well (Thanks to Liam Devine of OOLua for explaining how to turn off type checks in OOLua).

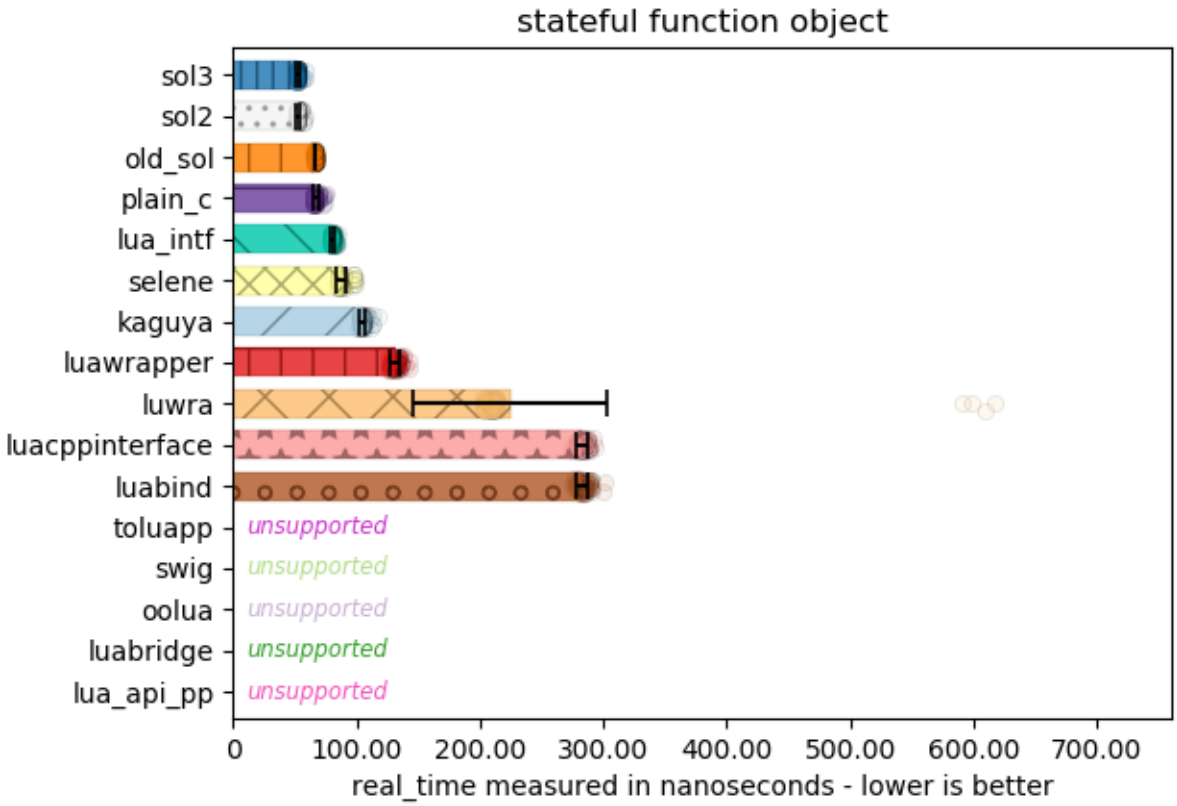
Bars go up to the average execution time. Lower is better. Reported times are for the desired operation run through nonius. Results are sorted from top to bottom by best to worst. Note that there are error bars to show potential variance in performance: generally, same-sized errors bars plus very close average execution time implies no significant difference in speed, despite the vastly different abstraction techniques used.

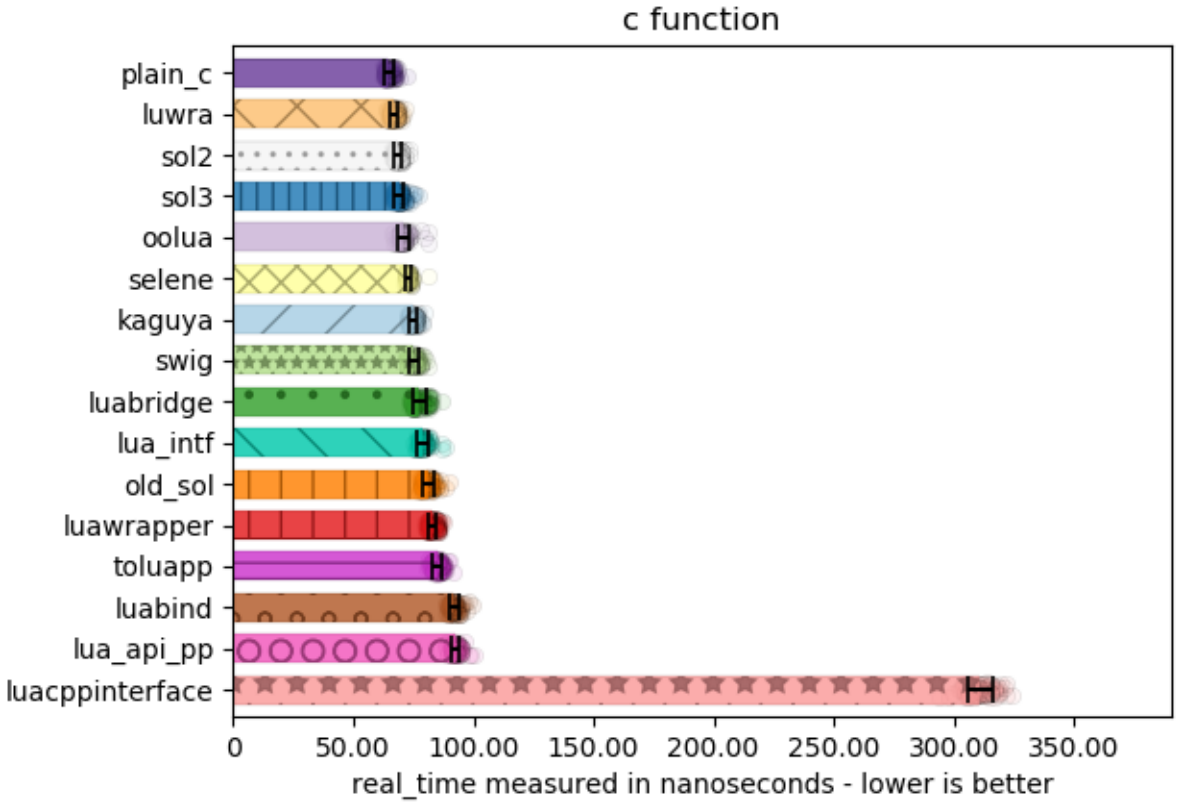

member function call
plain_c
sol2
sol3
oolua
luwra
kaguya
lua_api_pp
selene
luabind
luacppinterface
luawrapper
luabridge
swig
lua_intf
old_sol
toluapp
0
200.00
400.00
600.00
800.00
1000.00
real_time measured in nanoseconds - lower is better

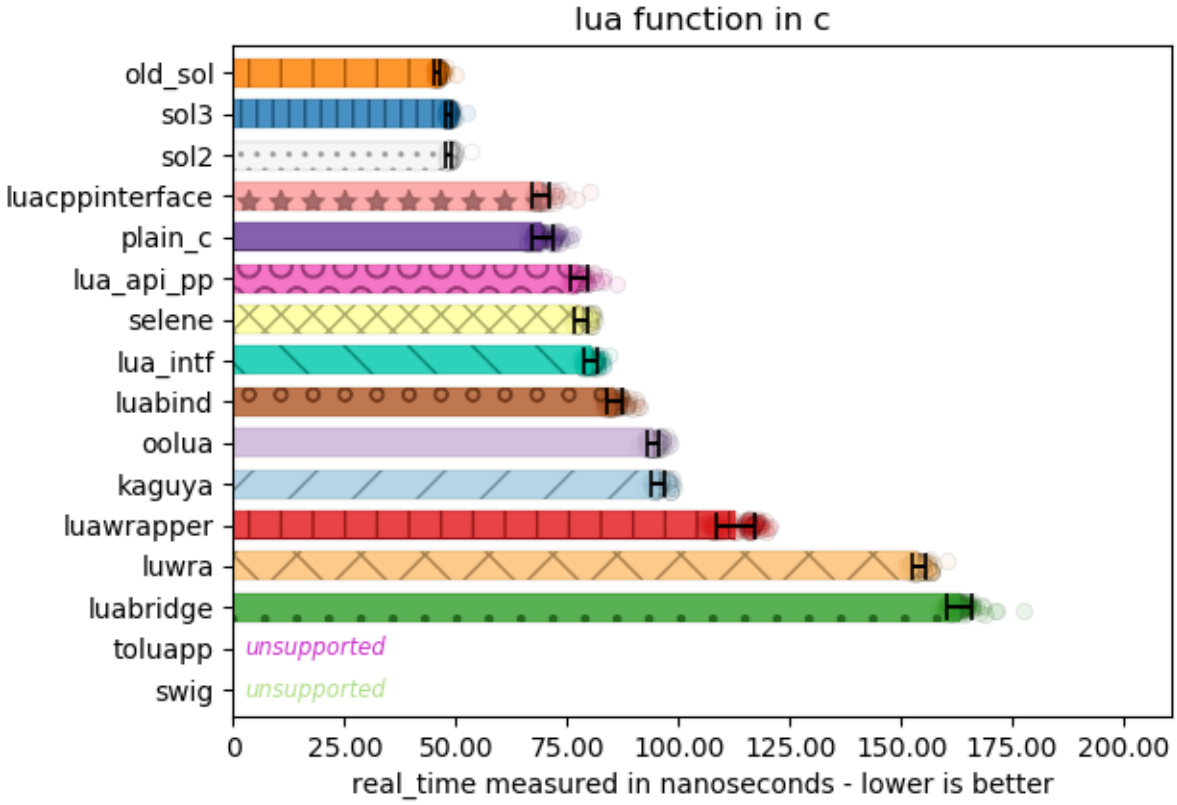

userdata variable access
plain_c
sol3
sol2
old_sol
luabind
swig
luawrapper
toluapp
luabridge
lua_intf
selene
oolua
luwra
luacppinterface
lua_api_pp
kaguya
unsupported
unsupported
unsupported
unsupported
unsupported
unsupported
0
200.00
400.00
600.00
800.00
1000.00
real_time measured in nanoseconds - lower is better

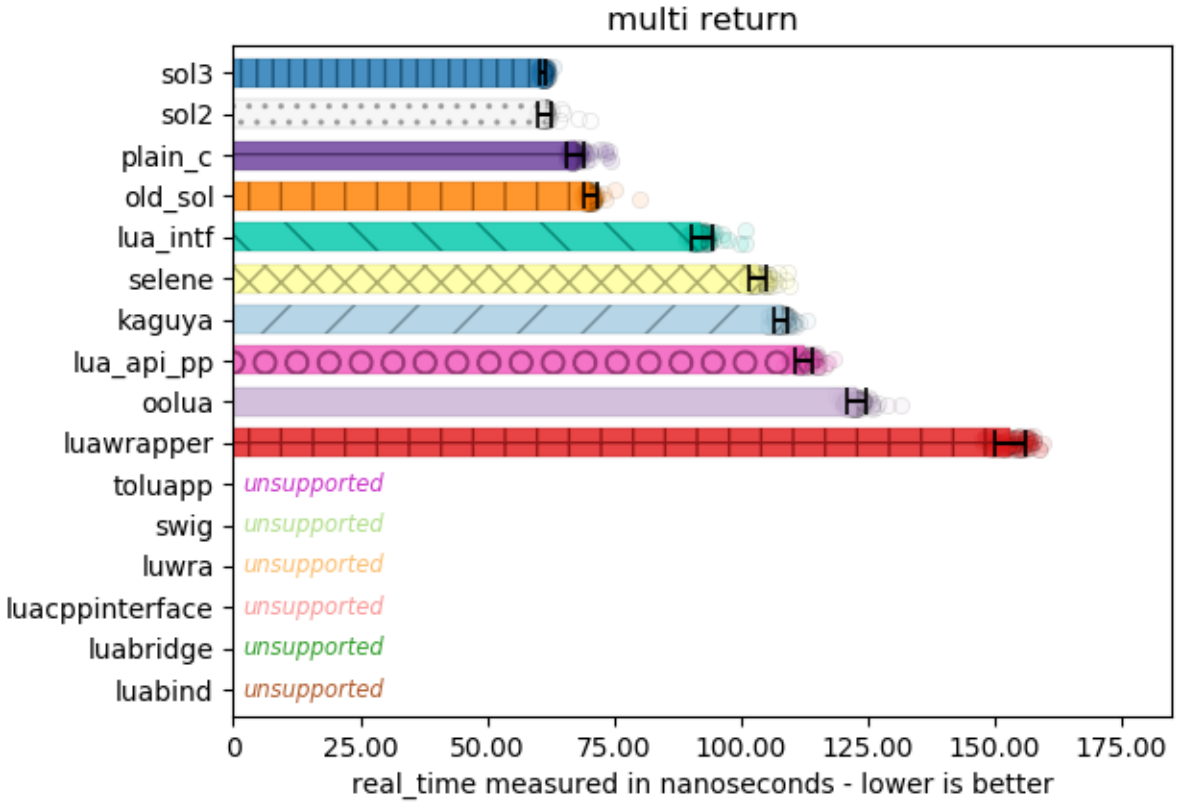

userdata variable access large
plain_c
sol2
sol3
old_sol
luabind
luawrapper
lua_intf
toluapp
swig
selene
oolua
luwra
luacppinterface
luabridge
lua_api_pp
kaguya
unsupported
0
10.00
20.00
30.00
40.00
50.00
real_time measured in microseconds - lower is better

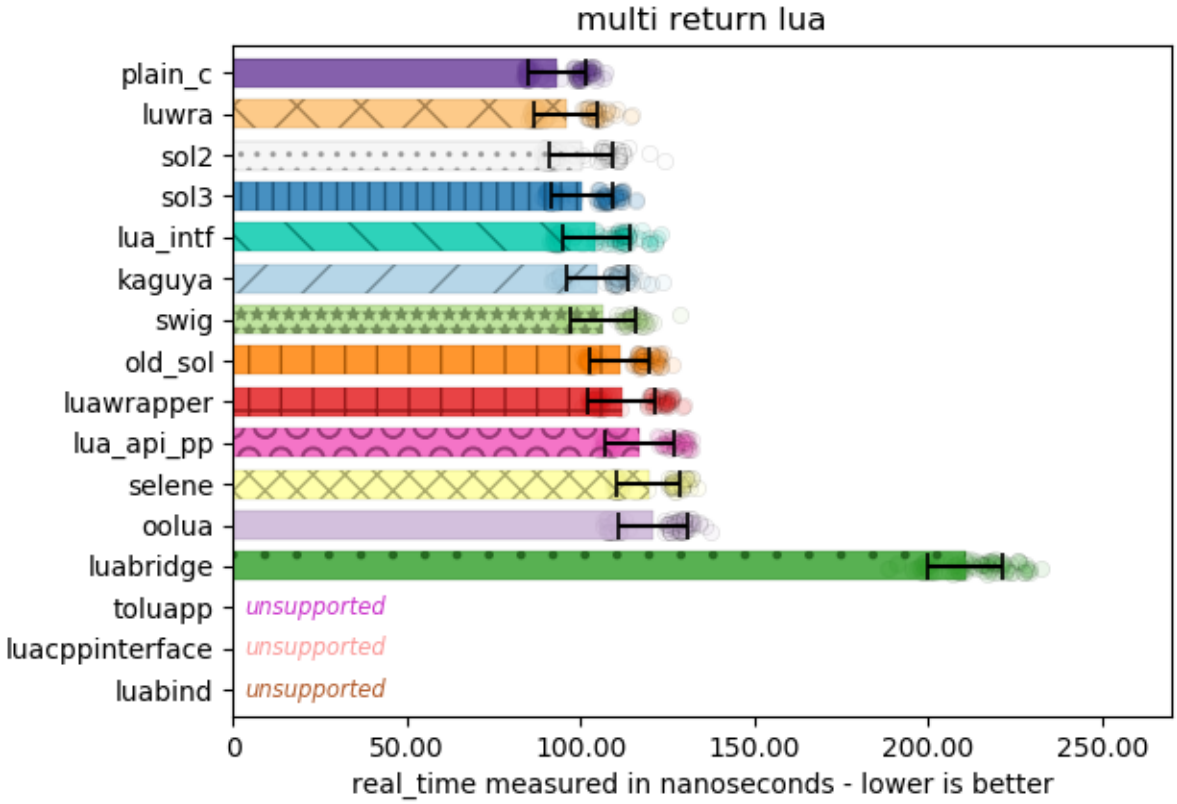

c function through lua in c
sol3
sol2
old_sol
plain_c
lua_intf
oolua
lua_api_pp
selene
kaguya
luabind
luawrapper
luwra
luabridge
luacppinterface
toluapp
unsupported
swig
unsupported
0
50.00
100.00
150.00
200.00
250.00
300.00
350.00
real_time measured in nanoseconds - lower is better


stateful function object
sol3
sol2
old_sol
plain_c
lua_intf
selene
kaguya
luawrapper
luwra
luacppinterface
luabind
toluapp
swig
oolua
luabridge
lua_api_pp
unsupported
unsupported
unsupported
unsupported
unsupported
0
100.00
200.00
300.00
400.00
500.00
600.00
700.00
real_time measured in nanoseconds - lower is better


c function
plain_c
luwra
sol2
sol3
oolua
selene
kaguya
swig
luabridge
lua_intf
old_sol
luawrapper
toluapp
luabind
lua_api_pp
luacppinterface
0
50.00
100.00
150.00
200.00
250.00
300.00
350.00
real_time measured in nanoseconds - lower is better


lua function in c
old_sol
sol3
sol2
luacppinterface
plain_c
lua_api_pp
selene
lua_intf
luabind
oolua
kaguya
luawrapper
luwra
luabridge
toluapp
unsupported
swig
unsupported
0
25.00
50.00
75.00
100.00
125.00
150.00
175.00
200.00
real_time measured in nanoseconds - lower is better


multi return
sol3
sol2
plain_c
old_sol
lua_intf
selene
kaguya
lua_api_pp
oolua
luawrapper
toluapp
swig
luwra
luacppinterface
luabridge
luabind
unsupported
unsupported
unsupported
unsupported
unsupported
unsupported
0
25.00
50.00
75.00
100.00
125.00
150.00
175.00
real_time measured in nanoseconds - lower is better


multi return lua
plain_c
luwra
sol2
sol3
lua_intf
kaguya
swig
old_sol
luawrapper
lua_api_pp
selene
oolua
luabridge
toluapp
luacppinterface
luabind
unsupported
unsupported
unsupported
0
50.00
100.00
150.00
200.00
250.00
real_time measured in nanoseconds - lower is better

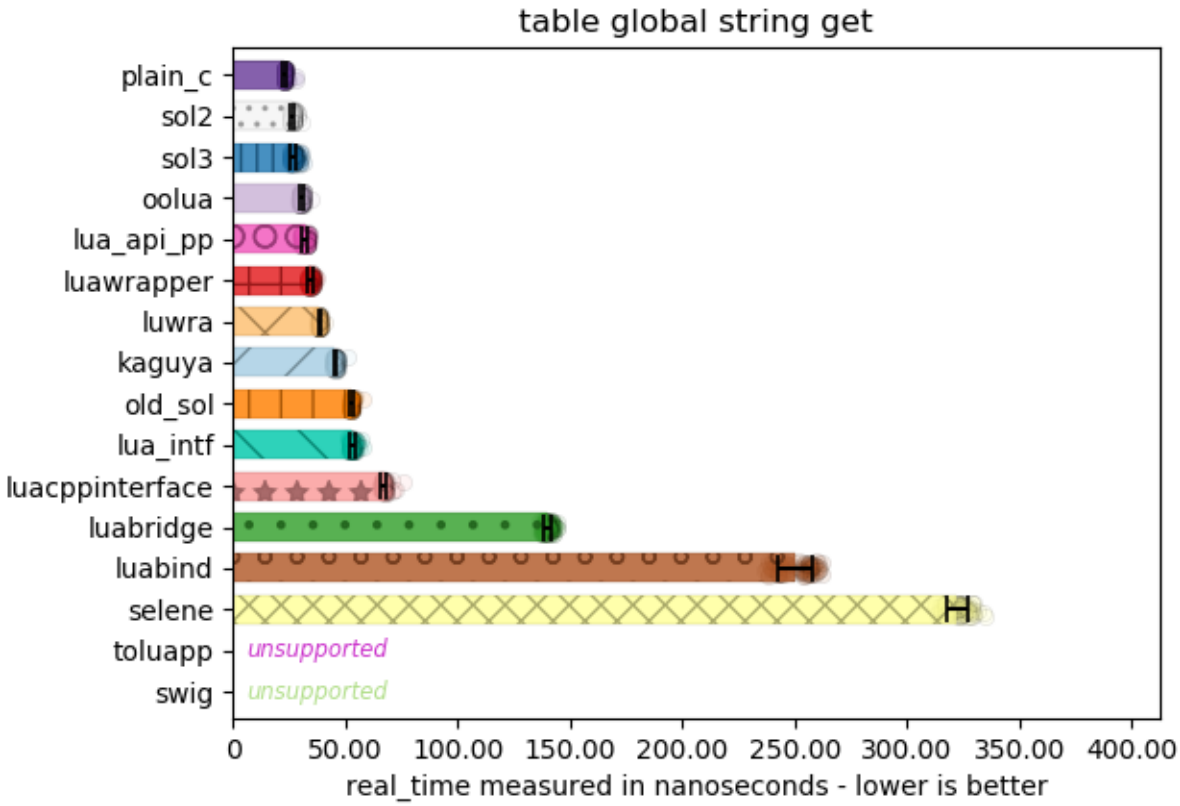


table global string get
plain_c
sol2
sol3
oolua
lua_api_pp
luawrapper
luwra
kaguya
old_sol
lua_intf
luacppinterface
luabridge
luabind
selene
toluapp
unsupported
swig
unsupported
0
50.00
100.00
150.00
200.00
250.00
300.00
350.00
400.00
real_time measured in nanoseconds - lower is better

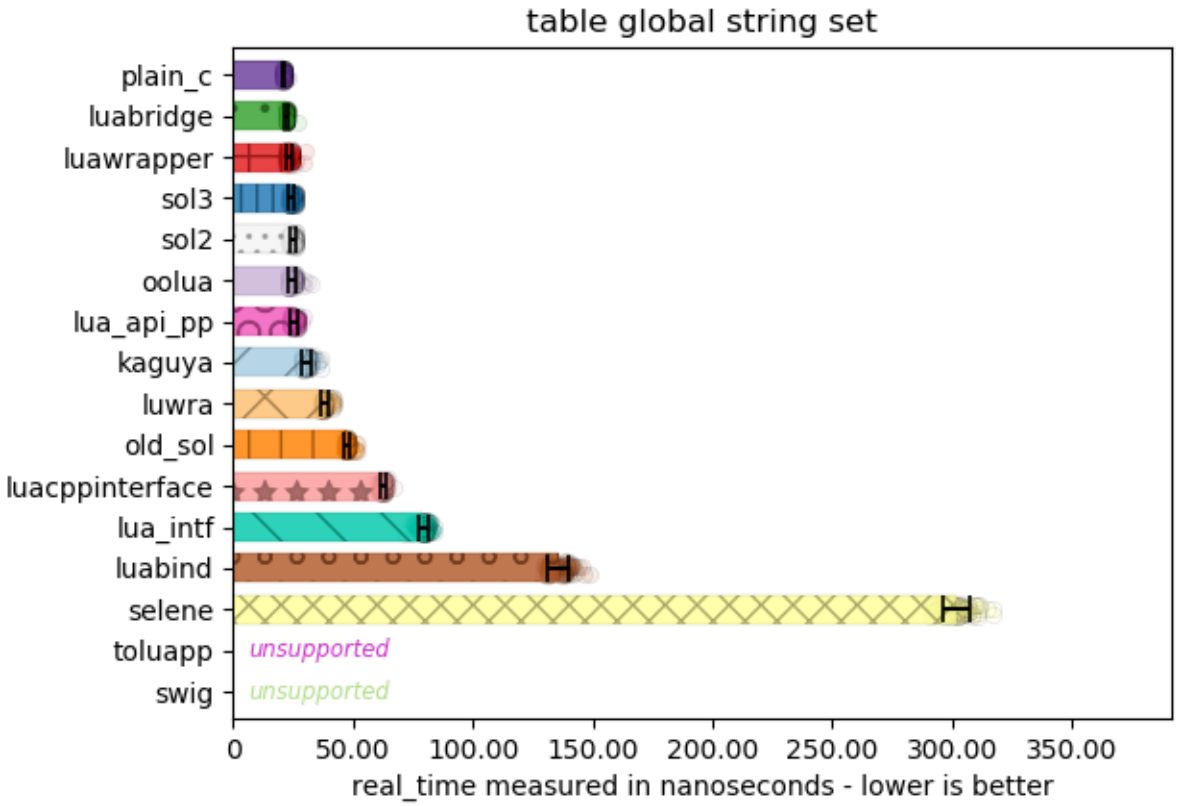


table global string set
plain_c
luabridge
luawrapper
sol3
sol2
oolua
lua_api_pp
kaguya
luwra
old_sol
luacppinterface
lua_intf
luabind
selene
toluapp
unsupported
swig
unsupported
0
50.00
100.00
150.00
200.00
250.00
300.00
350.00
real_time measured in nanoseconds - lower is better

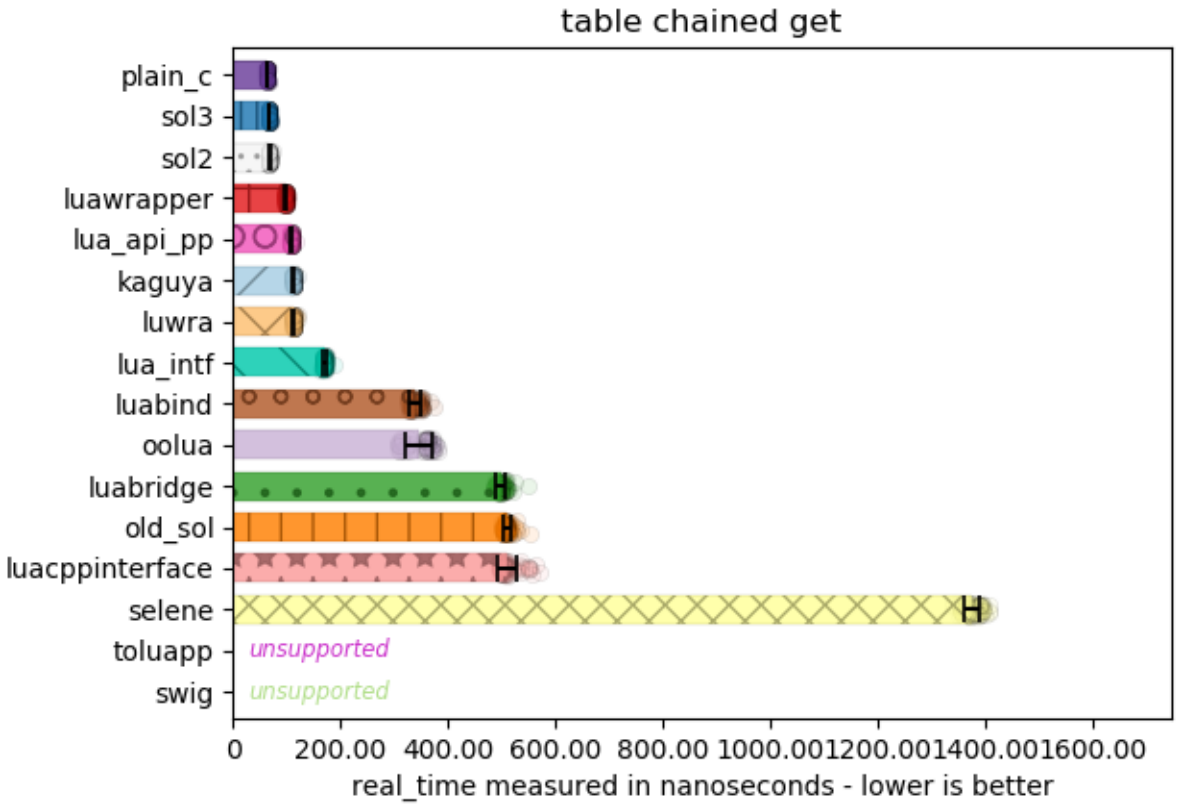


table chained get
plain_c
sol3
sol2
luawrapper
lua_api_pp
kaguya
luwra
lua_intf
luabind
oolua
luabridge
old_sol
luacppinterface
selene
toluapp
unsupported
swig
unsupported
0
200.00
400.00
600.00
800.00
1000.00
1200.00
1400.00
1600.00
real_time measured in nanoseconds - lower is better

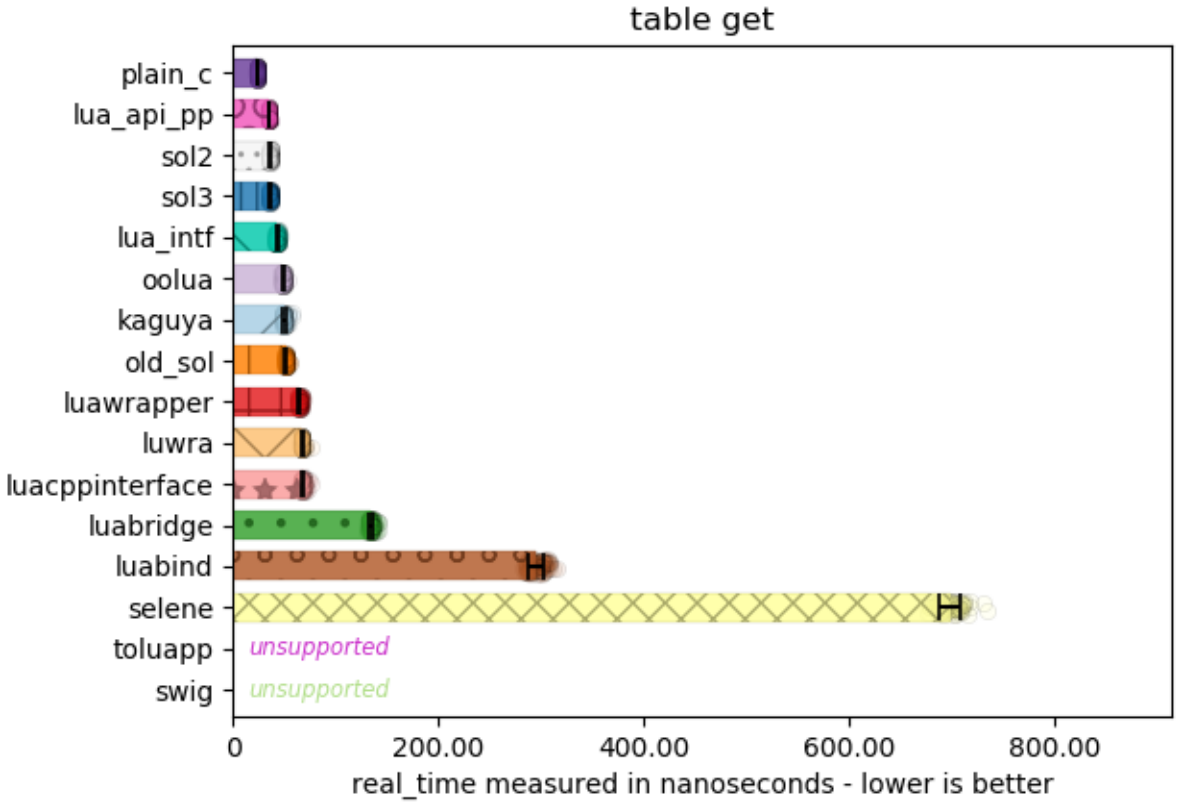


table get
plain_c
lua_api_pp
sol2
sol3
lua_intf
oolua
kaguya
old_sol
luawrapper
luwra
luacppinterface
luabridge
luabind
selene
toluapp
swig
unsupported
unsupported
0
200.00
400.00
600.00
800.00
real_time measured in nanoseconds - lower is better

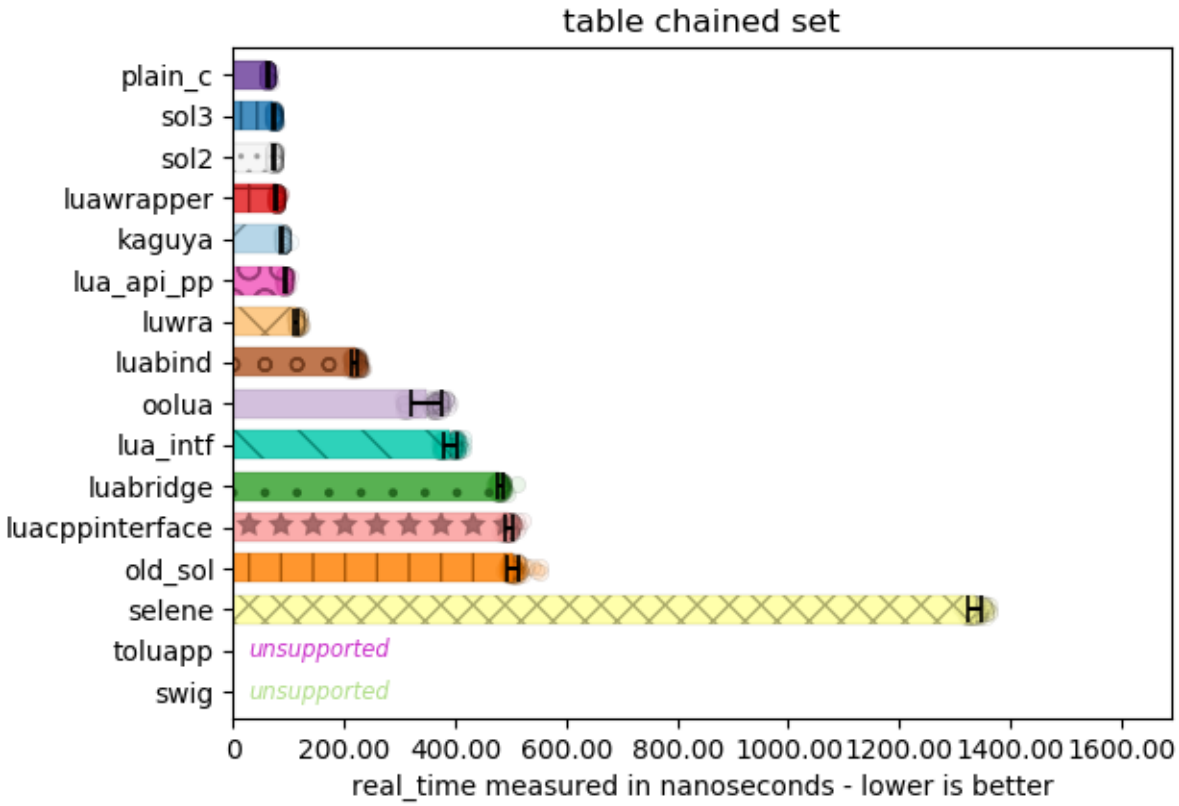


table chained set
plain_c
sol3
sol2
luawrapper
kaguya
lua_api_pp
luwra
luabind
oolua
lua_intf
luabridge
luacppinterface
old_sol
selene
toluapp
swig
unsupported
unsupported
0
200.00
400.00
600.00
800.00
1000.00
1200.00
1400.00
1600.00
real_time measured in nanoseconds - lower is better

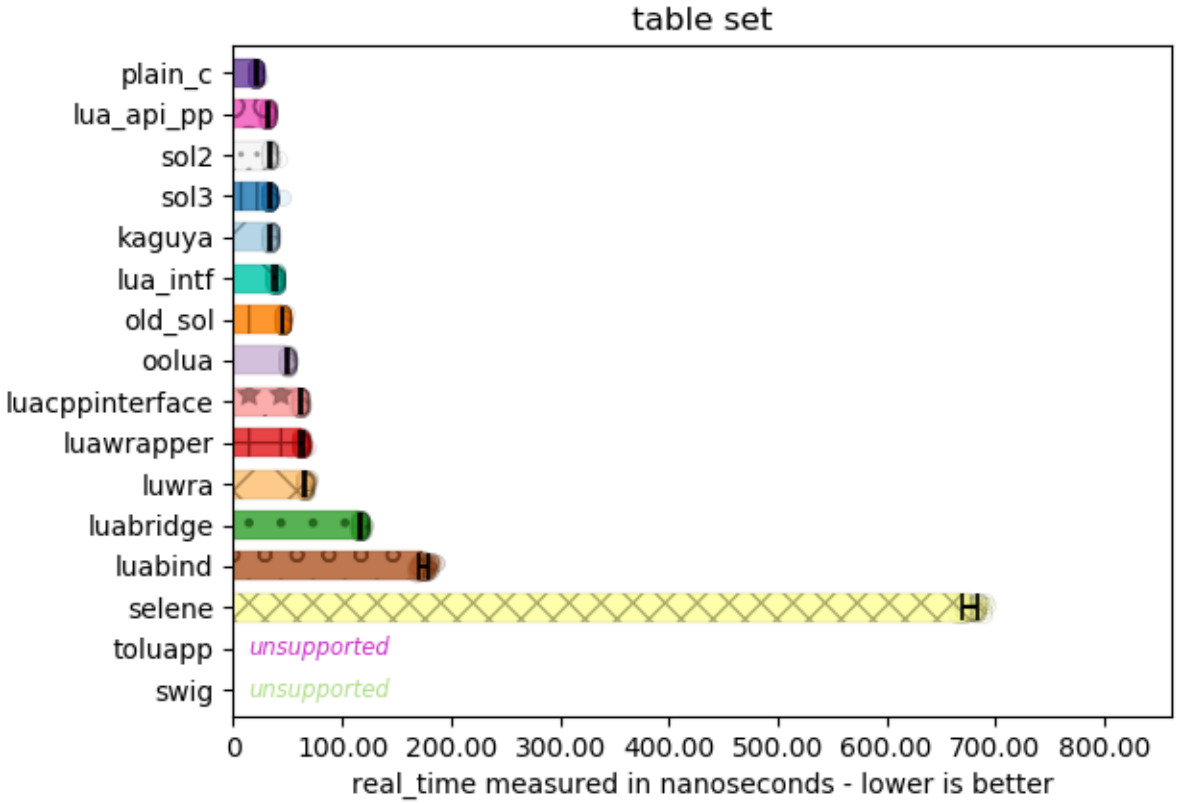
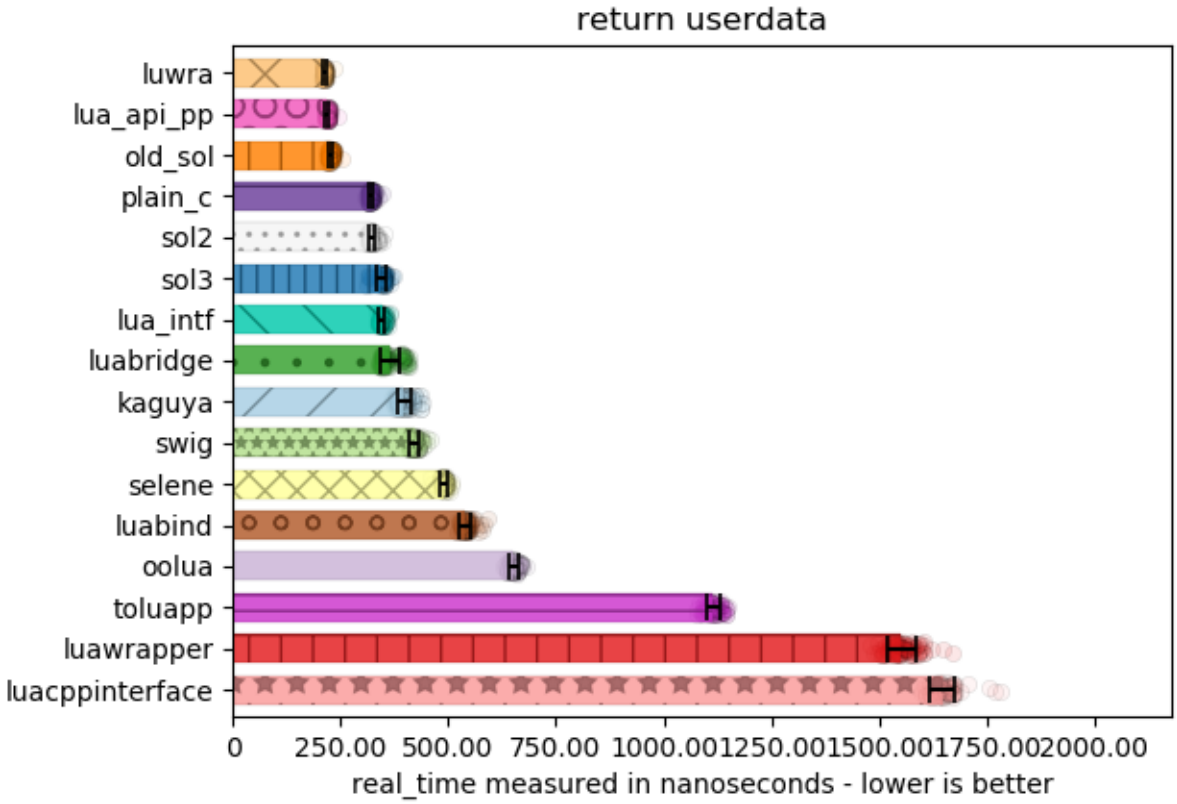
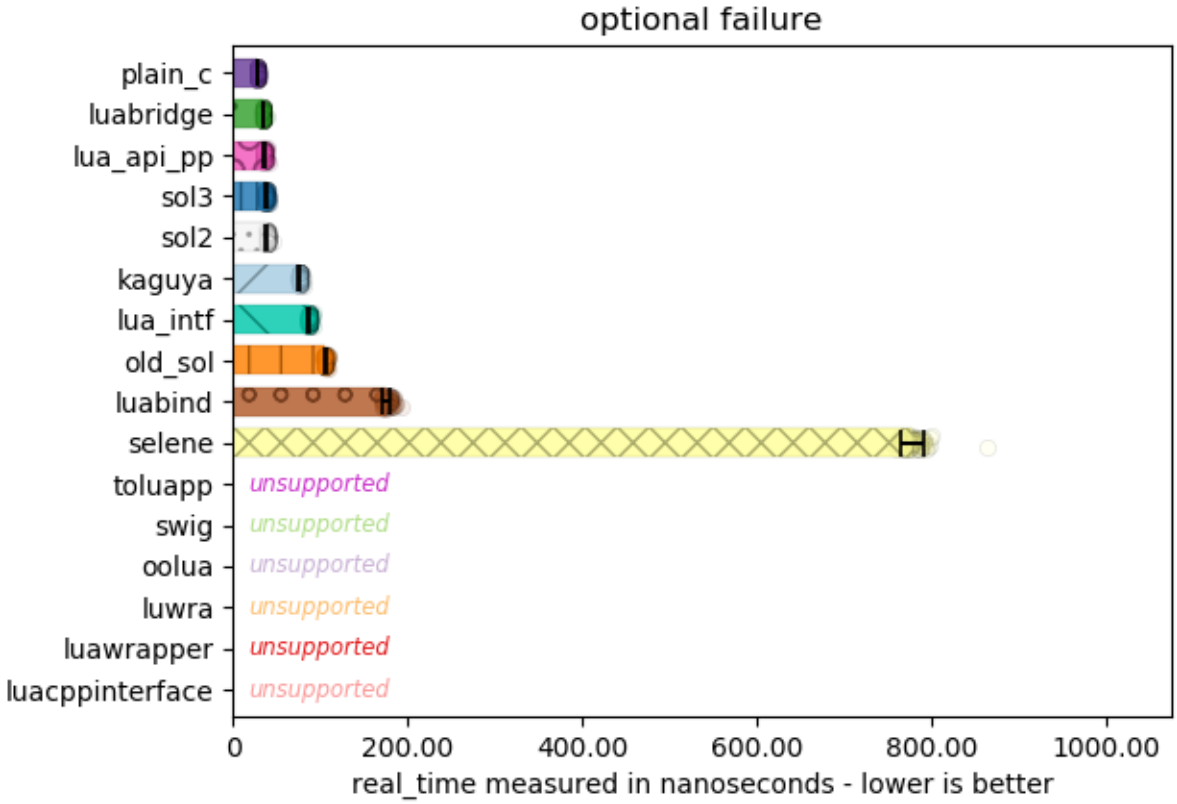
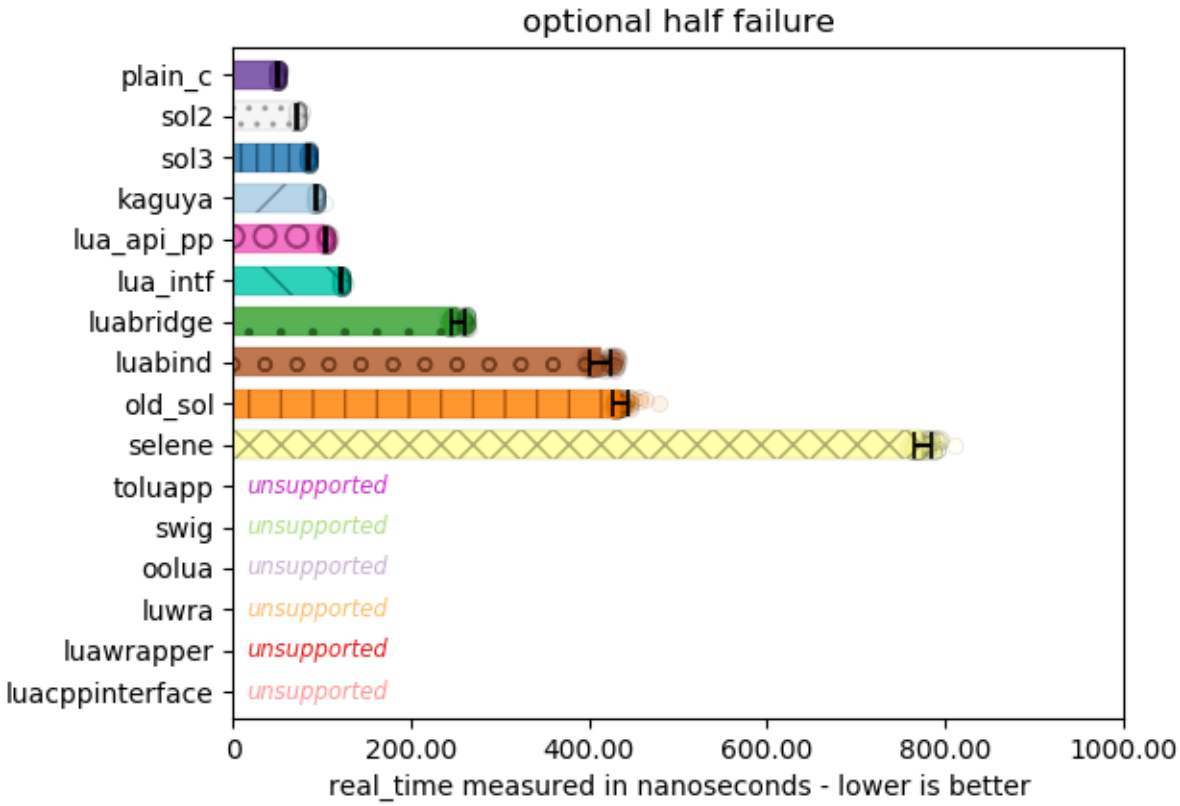
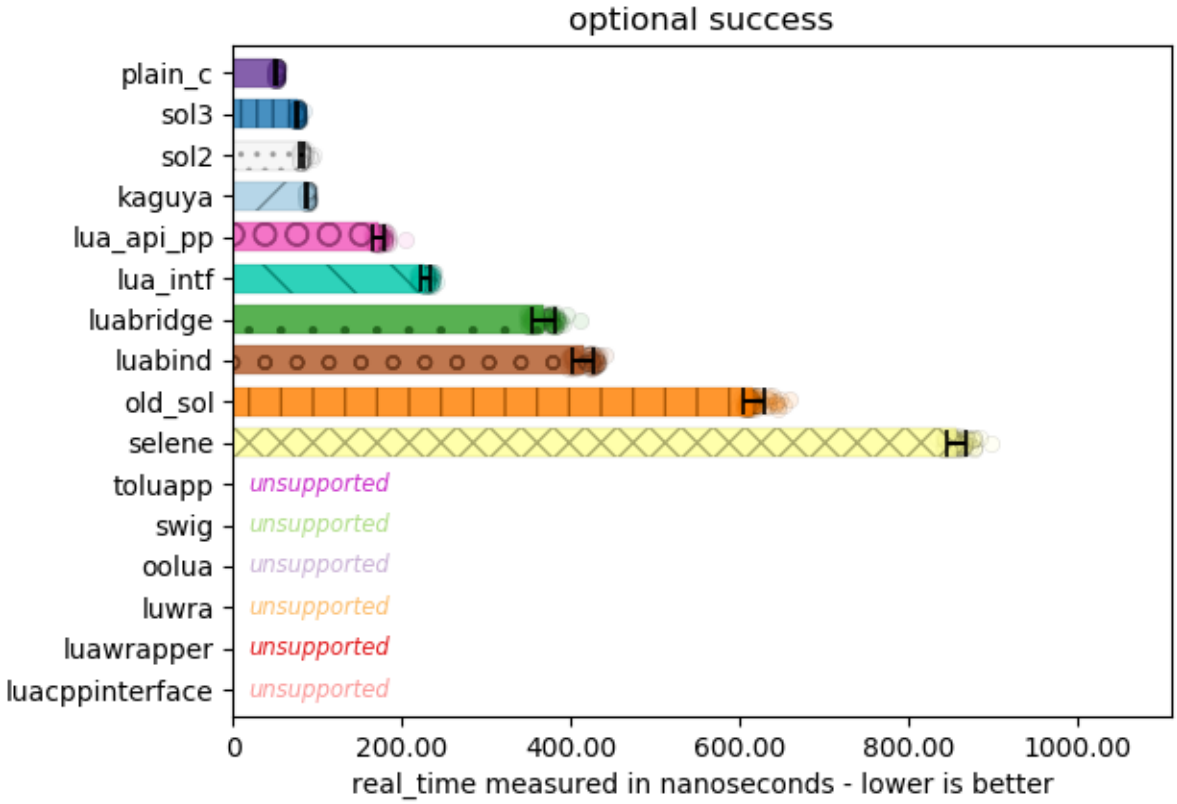


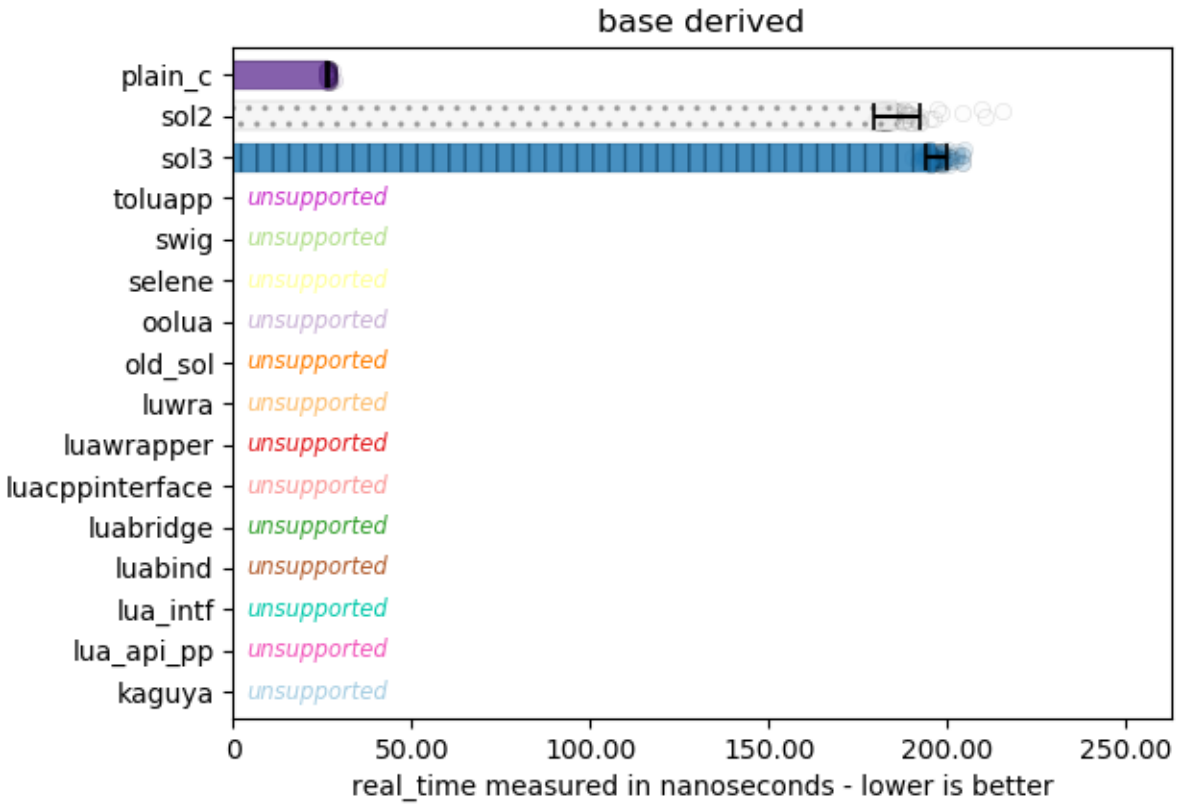
table set
plain_c
lua_api_pp
sol2
sol3
kaguya
lua_intf
old_sol
oolua
luacppinterface
luawrapper
luwra
luabridge
luabind
selene
toluapp
swig
unsupported
unsupported
0
100.00
200.00
300.00
400.00
500.00
600.00
700.00
800.00
real_time measured in nanoseconds - lower is better


return userdata
luwra
lua_api_pp
old_sol
plain_c
sol2
sol3
lua_intf
luabridge
kaguya
swig
selene
luabind
oolua
toluapp
luawrapper
luacppinterface
0
250.00
500.00
750.00
1000.00
1250.00
1500.00
1750.00
2000.00
real_time measured in nanoseconds - lower is better


optional failure
plain_c
luabridge
lua_api_pp
sol3
sol2
kaguya
lua_intf
old_sol
luabind
selene
toluapp
swig
oolua
luwra
luawrapper
luacppinterface
unsupported
unsupported
unsupported
unsupported
unsupported
unsupported
0
200.00
400.00
600.00
800.00
1000.00
real_time measured in nanoseconds - lower is better


optional half failure
plain_c
sol2
sol3
kaguya
lua_api_pp
lua_intf
luabridge
luabind
old_sol
selene
toluapp
swig
oolua
luwra
luawrapper
luacppinterface
unsupported
unsupported
unsupported
unsupported
unsupported
unsupported
0
200.00
400.00
600.00
800.00
1000.00
real_time measured in nanoseconds - lower is better


optional success
plain_c
sol3
sol2
kaguya
lua_api_pp
lua_intf
luabridge
luabind
old_sol
selene
toluapp
swig
oolua
luwra
luawrapper
luacppinterface
unsupported
unsupported
unsupported
unsupported
unsupported
unsupported
0
200.00
400.00
600.00
800.00
1000.00
real_time measured in nanoseconds - lower is better



1.14 getting performance

things to make Sol as fast as possible

As shown by the *benchmarks*, Sol is very performant with its abstractions. However, in the case where you need every last drop of performance from Sol, a number of tips and API usage tricks will be documented here. PLEASE benchmark / profile your code before you start invoking these, as some of them trade in readability / clarity for performance.

- If you have a bound function call / bound member function that you are going to call in a very tight, performance-heavy loop, considering using *sol::c_call*
- Be wary of passing by value / reference, and what it means by reading *this note*.
- It is currently undocumented that usertypes will "inherit" member function / member variables from bound classes, mostly because the semantics are unclear and it is not the most performant (although it is flexible: you can register base classes after / whenever you want in relation to the derived class, provided that derived class has its bases listed). Specifying all member functions / member variables for the usertype constructor / `new_usertype` function call and not relying on base lookup will boost performance of member lookup
- Use the *sol::stack_{}* versions of functions in order to achieve maximum performance benefits when doing things like calling a function from Lua and knowing that certain arguments of certain Lua types will be on the stack. This can save you a very small fraction of performance to not copy to the register (but is also more dangerous and usually not terribly worth it).

- Specifying base classes can make getting the usertype out of Sol a bit slower since we have to check and cast; if you know the exact type wherever you're retrieving it, considering not specifying the bases, retrieving the exact type from Sol, and then casting to a base type yourself
- Member variables can sometimes cost an extra lookup to occur within the Lua system (as mentioned *bottom of the usertype page*); until we find out a safe way around this, member variables will always incur that extra lookup cost

That's it as far as different performance options are avilable to make Sol run faster. Again, please make sure to invoke these only when you know Sol is the bottleneck. If you find some form of the performance unacceptable to you, also feel free to open an issue at the github.

1.15 config and safety

Sol was designed to be correct and fast, and in the pursuit of both uses the regular `lua_to{x}` functions of Lua rather than the checking versions (`lua_check{X}`) functions. The API defaults to paranoidly-safe alternatives if you have a `#define SOL_CHECK_ARGUMENTS` before you include Sol, or if you pass the `SOL_CHECK_ARGUMENTS` define on the build command for your build system. By default, it is off and remains off unless you define this, even in debug mode.

1.15.1 config

Note that you can obtain safety with regards to functions you bind by using the *protect* wrapper around function/variable bindings you set into Lua. Additionally, you can have basic boolean checks when using the API by just converting to a *sol::optional<T>* when necessary for getting things out of Lua and for function arguments.

Also note that you can have your own states use sol2's safety panics and similar to protect your code from crashes. See *sol::state automatic handlers* for more details.

Safety Config

`SOL_SAFE_USERTYPE` triggers the following change:

- If the userdata to a usertype function is nil, will trigger an error instead of letting things go through and letting the system segfault/crash
- Turned on by default with clang++, g++ and VC++ if a basic check for building in debug mode is detected (lack of `_NDEBUG` or similar compiler-specific checks)

`SOL_SAFE_REFERENCES` triggers the following changes:

- Checks the Lua type to ensure it matches what you expect it to be upon using *sol::reference* derived types, such as `sol::thread`, `sol::function`, etc...
- Turned on by default with clang++, g++ and VC++ if a basic check for building in debug mode is detected (lack of `_NDEBUG` or similar compiler-specific checks)

`SOL_SAFE_FUNCTION_CALLS` triggers the following changes:

- `sol::stack::call` and its variants will, if no templated boolean is specified, check all of the arguments for a function call
- All calls from Lua will have their arguments checked
- Turned on by default with clang++, g++ and VC++ if a basic check for building in debug mode is detected (lack of `_NDEBUG` or similar compiler-specific checks)

SOL_SAFE_FUNCTION triggers the following change:

- **All uses of sol::function and sol::stack_function will default to sol::protected_function and so**
  - Note this does not apply to sol::stack_aligned_function: this variant must always be unprotected due to stack positioning requirements, especially in use with sol::stack_count
- Will make any sol::state_view::script calls default to their safe variants if there is no supplied environment or error handler function
- **Not** turned on by default under any detectible compiler settings: *this MUST be turned on manually*

SOL_SAFE_NUMERICS triggers the following changes:

- Numbers will also be checked to see if they fit within a lua_Number if there is no lua_Integer type available that can fit your signed or unsigned number
- You can opt-out of this behavior with SOL_NO_CHECK_NUMBER_PRECISION
- **This option is required to differentiate between floats/ints in overloads**
- **Not** turned on by default under any settings: *this MUST be turned on manually*

SOL_SAFE_GETTER triggers the following changes:

- sol::stack::get (used everywhere) defaults to using sol::stack::check_get and dereferencing the argument. It uses sol::type_panic as the handler if something goes wrong
- Affects nearly the entire library for safety (with some blind spots covered by the other definitions)
- **Not** turned on by default under any settings: *this MUST be turned on manually*

SOL_DEFAULT_PASS_ON_ERROR triggers the following changes:

- The default error handler for sol::state_view::script_ functions is sol::script_pass_on_error rather than sol::script_throw_on_error
- Passes errors on through: **very dangerous** as you can ignore or never be warned about errors if you don't catch the return value of specific functions
- **Not** turned on by default: *this MUST be turned on manually*
- Don't turn this on unless you have an extremely good reason
- *DON'T TURN THIS ON UNLESS YOU HAVE AN EXTREMELY GOOD REASON*

SOL_CHECK_ARGUMENTS triggers the following changes:

- If SOL_SAFE_USERTYPE, SOL_SAFE_REFERENCES, SOL_SAFE_FUNCTION, SOL_SAFE_NUMERICS, SOL_SAFE_GETTER, and SOL_SAFE_FUNCTION_CALLS are not defined, they get defined and the effects described above kick in
- **Not** turned on by default under any settings: *this MUST be turned on manually*

SOL_NO_CHECK_NUMBER_PRECISION triggers the following changes:

- If SOL_SAFE_NUMERICS is defined, turns off number precision and integer precision fitting when pushing numbers into sol2
- **Not** turned on by default under any settings: *this MUST be turned on manually*

SOL_STRINGS_ARE_NUMBERS triggers the following changes:

- **Allows automatic to-string conversions for numbers**
  - lua_tolstring conversions are not permitted on numbers through sol2 by default: only actual strings are allowed

    - This is necessary to allow *sol::overload* to work properly
- `sol::stack::get` and `sol::stack::check_get` will allow anything that Lua thinks is number-worthy to be number-worthy
- This includes: integers, floating-point numbers, and strings
- This **does not** include: booleans, types with `__tostring` enabled, and everything else
- Overrides safety and always applies if it is turned on
- **Not** turned on by default under any settings: *this MUST be turned on manually*

Feature Config

`SOL_USE_BOOST` triggers the following change:

- Attempts to use `boost::optional` instead of sol's own `optional`
- **Not** turned on by default under any settings: *this MUST be turned on manually*

`SOL_PRINT_ERRORS` triggers the following change:

- Includes `<iostream>` and prints all exceptions and errors to `std::cerr`, for you to see
- **Not** turned on by default under any settings: *this MUST be turned on manually*

`SOL_CONTAINERS_START` triggers the following change:

- If defined and **is an integral value**, is used to adjust the container start value
- Applies to C++ containers **only** (not Lua tables or algorithms)
- Defaults to 1 (containers in Lua count from 1)

`SOL_ENABLE_INTEROP` triggers the following change:

- **Allows the use of `extensible<T>` to be used with `userdata_checker` and `userdata_getter` to retrieve non-**
    - Particularly enables non-sol usertypes to be used in overloads
    - See the *stack dcoumentation* for details
- May come with a slight performance penalty: only recommended for those stuck with non-sol libraries that still need to leverage some of sol's power
- **Not** turned on by default under any settings: *this MUST be turned on manually*

Memory Config

`SOL_NO_MEMORY_ALIGNMENT` triggers the following changes:

- Memory is no longer aligned and is instead directly sized and allocated
- If you need to access underlying userdata memory from sol, please see the *usertype memory documentation*
- **Not** turned on by default under any settings: *this MUST be turned on manually*

Linker Config

`SOL_USING_CXX_LUA` triggers the following changes:

- Lua includes are no longer wrapped in `extern "C" {}` blocks
- Turns on `SOL_EXCEPTIONS_SAFE_PROPAGATION` automatically for you
- Only use this if you know you've built your LuaJIT with the C++-specific invocations of your compiler (Lua by default builds as C code and is not distributed as a C++ library, but a C one with C symbols)

`SOL_USING_CXX_LUA_JIT` triggers the following changes:

- LuaJIT includes are no longer wrapped in `extern "C" {}` blocks
- Turns on `SOL_EXCEPTIONS_SAFE_PROPAGATION` automatically for you
- Only use this if you know you've built your LuaJIT with the C++-specific invocations of your compiler
- LuaJIT by default builds as C code, but includes hook to handle C++ code unwinding: this should almost never be necessary for regular builds

`SOL_EXCEPTIONS_ALWAYS_UNSAFE` triggers the following changes:

- If any of the `SOL_USING_CXX_*` defines are in play, it **does NOT** automatically turn on `SOL_EXCEPTIONS_SAFE_PROPAGATION` automatically
- This standardizes some behavior, since throwing exceptions through the C API's interface can still lead to undefined behavior that Lua cannot handle properly

`SOL_EXCEPTIONS_SAFE_PROPAGATION` triggers the following changes:

- try/catch will not be used around C-function trampolines when going from Lua to C++
- try/catch will not be used in `safe_/protected_function` internals
- Should only be used in accordance with compiling vanilla PUC-RIO Lua as C++, using *LuaJIT under the proper conditions*, or in accordance with your Lua distribution's documentation

Tests are compiled with this on to ensure everything is going as expected. Remember that if you want these features, you must explicitly turn them on all of them to be sure you are getting them.

1.15.2 memory

Memory safety can be tricky. Lua is handled by a garbage-collected runtime, meaning object deletion is not cleary defined or deterministic. If you need to keep an object from the Lua Runtime alive, use *sol::reference* or one of its derived types, such as *sol::table*, *sol::object*, or similar. These will pin a reference down to an object controlled in C++, and Lua will not delete an object that you still have a reference to through one of these types. You can then retrieve whatever you need from that Lua slot using object's `obj.as<T>()` member function or other things, and work on the memory from there.

The usertype memory layout for all Lua-instantiated userdata and for all objects pushed/set into the Lua Runtime is also described *here*. Things before or after that specified memory slot is implementation-defined and no assumptions are to be made about it.

Please be wary of alignment issues. sol2 **aligns memory** by default. If you need to access underlying userdata memory from sol, please see the *usertype memory documentation*

1.15.3 functions

The *vast majority* of all users are going to want to work with *sol::safe_function/sol::protected_function*. This version allows for error checking, prunes results, and responds to the defines listed above by throwing errors if you try to use the result of a function without checking. *sol::function/sol::unsafe_function* is unsafe. It assumes that its contents run correctly and throw no errors, which can result in crashes that are hard to debug while offering a very tiny performance boost for not checking error codes or catching exceptions.

If you find yourself crashing inside of `sol::function`, try changing it to a `sol::protected_function` and seeing if the error codes and such help you find out what's going on. You can read more about the API on *the page itself*. You can also define `SOL_SAFE_FUNCTION` as described above, but be warned that the `protected_function` API is a superset of the regular default `function` API: trying to revert back after defining `SOL_SAFE_FUNCTION` may result in some compiler errors if you use things beyond the basic, shared interface of the two types.

As a side note, binding functions with default parameters does not magically bind multiple versions of the function to be called with the default parameters. You must instead use *sol::overload*.

Warning: Do **NOT** save the return type of a *unsafe_function_result* with `auto`, as in `auto numwoof = woof(20);`, and do NOT store it anywhere unless you are exactly aware of the consequences of messing with the stack. See *here* for more information.

1.16 exceptions

1.16.1 since somebody is going to ask about it...

Yes, you can turn off exceptions in Sol with `#define SOL_NO_EXCEPTIONS` before including or by passing the command line argument that defines `SOL_NO_EXCEPTIONS`. We don't recommend it unless you're playing with a Lua distro that also doesn't play nice with exceptions (like non-x64 versions of *LuaJIT*).

If you turn this off, the default at_panic function *state* set for you will not throw (see *sol::state's automatic handlers* for more details). Instead, the default Lua behavior of aborting will take place (and give you no chance of escape unless you implement your own at_panic function and decide to try `longjmp` out).

To make this not be the case, you can set a panic function directly with `lua_atpanic( lua, my_panic_function );` or when you create the `sol::state` with `sol::state lua(my_panic_function);`. Here's an example `my_panic_function` you can have that prints out its errors:

Listing 145: typical panic function

```
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>
#include <iostream>

inline void my_panic(sol::optional<std::string> maybe_msg) {
        std::cerr << "Lua is in a panic state and will now abort() the application" <
↪< std::endl;
        if (maybe_msg) {
                const std::string& msg = maybe_msg.value();
                std::cerr << "\terror message: " << msg << std::endl;
        }
        // When this function exits, Lua will exhibit default behavior and abort()
```

(continues on next page)

(continued from previous page)

```
}

int main (int, char*[]) {
        sol::state lua(sol::c_call<decltype(&my_panic), &my_panic>);
        // or, if you already have a lua_State* L
        // lua_atpanic( L, sol::c_call<decltype(&my_panic)>, &my_panic> );
        // or, with state/state_view:
        // sol::state_view lua(L);
        // lua.set_panic( sol::c_call<decltype(&my_panic)>, &my_panic> );

        // uncomment the below to see
        //lua.script("boom_goes.the_dynamite");

        return 0;
}
```

Note that `SOL_NO_EXCEPTIONS` will also disable *sol::protected_function*'s ability to catch C++ errors you throw from C++ functions bound to Lua that you are calling through that API. So, only turn off exceptions in Sol if you're sure you're never going to use exceptions ever. Of course, if you are ALREADY not using Exceptions, you don't have to particularly worry about this and now you can use Sol!

If there is a place where a throw statement is called or a try/catch is used and it is not hidden behind a `#ifndef SOL_NO_EXCEPTIONS` block, please file an issue at issue or submit your very own pull request so everyone can benefit!

1.16.2 various sol and lua handlers

Lua comes with two kind of built-in handlers that sol provides easy opt-ins for. One is the `panic` function, as *demonstrated above*. Another is the `pcall` error handler, used with *sol::protected_function*. It is any function that takes a single argument. The single argument is the error type being passed around: in Lua, this is a single string message:

Listing 146: regular error handling

```
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

#include <iostream>

int main() {
        std::cout << "=== protected_functions ===" << std::endl;

        sol::state lua;
        lua.open_libraries(sol::lib::base);

        // A complicated function which can error out
        // We define both in terms of Lua code

        lua.script(R"(
                        function handler (message)
                                return "Handled this message: " .. message
                        end

                        function f (a)
                                if a < 0 then
```

(continues on next page)

(continued from previous page)

```
                                        error("negative number detected")
                                end
                                return a + 5
                        end
                )");

        // Get a protected function out of Lua
        sol::protected_function f(lua["f"], lua["handler"]);

        sol::protected_function_result result = f(-500);
        if (result.valid()) {
                // Call succeeded
                int x = result;
                std::cout << "call succeeded, result is " << x << std::endl;
        }
        else {
                // Call failed
                sol::error err = result;
                std::string what = err.what();
                std::cout << "call failed, sol::error::what() is " << what <<␣
↪std::endl;
                // 'what' Should read
                // "Handled this message: negative number detected"
        }

        std::cout << std::endl;
}
```

The other handler is specific to sol2. If you open a `sol::state`, or open the default state handlers for your `lua_State*` (see *sol::state's automatic handlers* for more details), there is a `sol::exception_handler_function` type. It allows you to register a function in the event that an exception happens that bubbles out of your functions. The function requires that you push 1 item onto the stack that will be used with a call to lua_error

Listing 147: exception handling

```
#define SOL_CHECK_ARGUMENTS 1
#include <sol.hpp>

#include "assert.hpp"

#include <iostream>

int my_exception_handler(lua_State* L, sol::optional<const std::exception&> maybe_
↪exception, sol::string_view description) {
        // L is the lua state, which you can wrap in a state_view if necessary
        // maybe_exception will contain exception, if it exists
        // description will either be the what() of the exception or a description␣
↪saying that we hit the general-case catch(...)
        std::cout << "An exception occurred in a function, here's what it says ";
        if (maybe_exception) {
                std::cout << "(straight from the exception): ";
                const std::exception& ex = *maybe_exception;
                std::cout << ex.what() << std::endl;
        }
        else {
```

(continues on next page)

(continued from previous page)

```
                std::cout << "(from the description parameter): ";
                std::cout.write(description.data(), description.size());
                std::cout << std::endl;
        }

        // you must push 1 element onto the stack to be
        // transported through as the error object in Lua
        // note that Lua -- and 99.5% of all Lua users and libraries -- expects a string
        // so we push a single string (in our case, the description of the error)
        return sol::stack::push(L, description);
}

void will_throw() {
        throw std::runtime_error("oh no not an exception!!!");
}

int main() {
        std::cout << "=== exception_handler ===" << std::endl;

        sol::state lua;
        lua.open_libraries(sol::lib::base);
        lua.set_exception_handler(&my_exception_handler);

        lua.set_function("will_throw", &will_throw);

        sol::protected_function_result pfr = lua.safe_script("will_throw()", &sol::script_pass_on_error);

        c_assert(!pfr.valid());

        sol::error err = pfr;
        std::cout << err.what() << std::endl;

        std::cout << std::endl;

        return 0;
}
```

1.16.3 LuaJIT and exceptions

It is important to note that a popular 5.1 distribution of Lua, LuaJIT, has some serious caveats regarding exceptions. LuaJIT's exception promises are flaky at best on x64 (64-bit) platforms, and entirely terrible on non-x64 (32-bit, ARM, etc.) platforms. The trampolines we have in place for all functions bound through conventional means in Sol will catch exceptions and turn them into Lua errors so that LuaJIT remainds unperturbed, but if you link up a C function directly yourself and throw, chances are you might have screwed the pooch.

Testing in this closed issue that it doesn't play nice on 64-bit Linux in many cases either, especially when it hits an error internal to the interpreter (and does not go through Sol). We do have tests, however, that compile for our continuous integration check-ins that check this functionality across several compilers and platforms to keep you protected and given hard, strong guarantees for what happens if you throw in a function bound by Sol. If you stray outside the realm of Sol's protection, however... Good luck.

1.16.4 Lua and LuaJIT C++ Exception Full Interoperability

You can `#define SOL_EXCEPTIONS_SAFE_PROPAGATION` before including Sol or define `SOL_EXCEPTIONS_SAFE_PROPAGATION` on the command line if you know your implmentation of Lua has proper unwinding semantics that can be thrown through the version of the Lua API you have built / are using.

This will prevent sol from catching `(...)` errors in platforms and compilers that have full C++ exception interoperability. This means that Lua errors can be caught with `catch (...)` in the C++ end of your code after it goes through Lua, and exceptions can pass through the Lua API and Stack safely.

Currently, the only known platform to do this is the listed "Full" platforms for LuaJIT and Lua compiled as C++. This define is turned on automatically for compiling Lua as C++ and `SOL_USING_CXX_LUA` (or `SOL_USING_CXX_LUA_JIT`) is defined.

Warning: `SOL_EXCEPTIONS_SAFE_PROPAGATION` is not defined automatically when Sol detects LuaJIT. *It is your job to define it if you know that your platform supports it*!

1.17 run-time type information (rtti)

because somebody's going to want to shut this off, too...

Sol does not use RTTI.

1.18 unicode transformation format handling

1.18.1 because this is surprisingly hard using standard C++

Note: The `<codecvt>` header is no longer used and sol2 now converts utf8, utf16, and utf32 with internal routines. If you have a problem with the transcoding, please file an issue report.

`std::(w)string(u16/u32)` are assumed to be in the platform's native wide (for `wstring`) or unicode format. Lua canonically stores its string literals as utf8 and embraces utf8, albeit its storage is simply a sequence of bytes that are also null-terminated (it is also counted and the size is kept around, so embedded nulls can be used in the string). Therefore, if you need to interact with the unicode or wide alternatives of strings, runtime conversions are performed from the (assumed) utf8 string data into other forms. These conversions check for well-formed UTF, and will replace ill-formed characters with the unicode replacement codepoint, 0xFFFD.

Note that we cannot give you a `string_view` to utf16 or utf32 strings: Lua does not hold them in memory this way. You can perhaps do your own customization to provide for this if need be. Remember that Lua stores a counted sequence of bytes: serializing your string as bytes and pushing a string type into Lua's stack will work, though do not except any complex string routines or printing to behave nicely with your code.

1.19 Build

sol2 comes with a CMake script in the top level. It is primarily made for building and running the examples and tests, but it includes exported and configured targets (`sol2`, `sol2_single`) for your use.

sol2 also comes with a Meson Script. If things stop working, file a bug report.

1.20 licenses

The following licenses cover all of the code in Sol. Spoiler: they're all MIT (and CC0) and it's safe to use in commercial code: feel free to copy/paste the below right into your own attributions / licenses file.

1.20.1 Sol - ThePhD/sol2:

```
The MIT License (MIT)

Copyright (c) 2013-2016 Rapptz, ThePhD, and contributors

Permission is hereby granted, free of charge, to any person obtaining a copy
 of this software and associated documentation files (the "Software"), to
 deal in the Software without restriction, including without limitation the
 rights to use, copy, modify, merge, publish, distribute, sublicense, and/or
 sell copies of the Software, and to permit persons to whom the Software is
 furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED,
INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

1.20.2 Lua-compat-5.3 - keplerproject/Lua-compat-5.3:

```
The MIT License (MIT)

Copyright (c) 2018 Kepler Project.

Permission is hereby granted, free of charge, to any person obtaining a copy
 of this software and associated documentation files (the "Software"), to
 deal in the Software without restriction, including without limitation the
 rights to use, copy, modify, merge, publish, distribute, sublicense, and/or
 sell copies of the Software, and to permit persons to whom the Software is
 furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED,
INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

1.20.3 Lua - Lua.org:

```
The MIT License (MIT)

Copyright © 1994–2018 Lua.org, PUC-Rio.

Permission is hereby granted, free of charge, to any person obtaining a copy
 of this software and associated documentation files (the "Software"), to
 deal in the Software without restriction, including without limitation the
 rights to use, copy, modify, merge, publish, distribute, sublicense, and/or
 sell copies of the Software, and to permit persons to whom the Software is
 furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED,
INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

1.20.4 ogonek - libogonek/ogonek:

```
Creative Commons Legal Code

CC0 1.0 Universal

    CREATIVE COMMONS CORPORATION IS NOT A LAW FIRM AND DOES NOT PROVIDE
    LEGAL SERVICES. DISTRIBUTION OF THIS DOCUMENT DOES NOT CREATE AN
    ATTORNEY-CLIENT RELATIONSHIP. CREATIVE COMMONS PROVIDES THIS
    INFORMATION ON AN "AS-IS" BASIS. CREATIVE COMMONS MAKES NO WARRANTIES
    REGARDING THE USE OF THIS DOCUMENT OR THE INFORMATION OR WORKS
    PROVIDED HEREUNDER, AND DISCLAIMS LIABILITY FOR DAMAGES RESULTING FROM
    THE USE OF THIS DOCUMENT OR THE INFORMATION OR WORKS PROVIDED
    HEREUNDER.

Statement of Purpose

The laws of most jurisdictions throughout the world automatically confer
exclusive Copyright and Related Rights (defined below) upon the creator
and subsequent owner(s) (each and all, an "owner") of an original work of
authorship and/or a database (each, a "Work").

Certain owners wish to permanently relinquish those rights to a Work for
the purpose of contributing to a commons of creative, cultural and
scientific works ("Commons") that the public can reliably and without fear
of later claims of infringement build upon, modify, incorporate in other
works, reuse and redistribute as freely as possible in any form whatsoever
and for any purposes, including without limitation commercial purposes.
These owners may contribute to the Commons to promote the ideal of a free
culture and the further production of creative, cultural and scientific
works, or to gain reputation or greater distribution for their Work in
part through the use and efforts of others.
```

(continues on next page)

(continued from previous page)

```
For these and/or other purposes and motivations, and without any
expectation of additional consideration or compensation, the person
associating CC0 with a Work (the "Affirmer"), to the extent that he or she
is an owner of Copyright and Related Rights in the Work, voluntarily
elects to apply CC0 to the Work and publicly distribute the Work under its
terms, with knowledge of his or her Copyright and Related Rights in the
Work and the meaning and intended legal effect of CC0 on those rights.

1. Copyright and Related Rights. A Work made available under CC0 may be
protected by copyright and related or neighboring rights ("Copyright and
Related Rights"). Copyright and Related Rights include, but are not
limited to, the following:

  i. the right to reproduce, adapt, distribute, perform, display,
        communicate, and translate a Work;
 ii. moral rights retained by the original author(s) and/or performer(s);
iii. publicity and privacy rights pertaining to a person's image or
        likeness depicted in a Work;
 iv. rights protecting against unfair competition in regards to a Work,
        subject to the limitations in paragraph 4(a), below;
  v. rights protecting the extraction, dissemination, use and reuse of data
        in a Work;
 vi. database rights (such as those arising under Directive 96/9/EC of the
        European Parliament and of the Council of 11 March 1996 on the legal
        protection of databases, and under any national implementation
        thereof, including any amended or successor version of such
        directive); and
vii. other similar, equivalent or corresponding rights throughout the
        world based on applicable law or treaty, and any national
        implementations thereof.

2. Waiver. To the greatest extent permitted by, but not in contravention
of, applicable law, Affirmer hereby overtly, fully, permanently,
irrevocably and unconditionally waives, abandons, and surrenders all of
Affirmer's Copyright and Related Rights and associated claims and causes
of action, whether now known or unknown (including existing as well as
future claims and causes of action), in the Work (i) in all territories
worldwide, (ii) for the maximum duration provided by applicable law or
treaty (including future time extensions), (iii) in any current or future
medium and for any number of copies, and (iv) for any purpose whatsoever,
including without limitation commercial, advertising or promotional
purposes (the "Waiver"). Affirmer makes the Waiver for the benefit of each
member of the public at large and to the detriment of Affirmer's heirs and
successors, fully intending that such Waiver shall not be subject to
revocation, rescission, cancellation, termination, or any other legal or
equitable action to disrupt the quiet enjoyment of the Work by the public
as contemplated by Affirmer's express Statement of Purpose.

3. Public License Fallback. Should any part of the Waiver for any reason
be judged legally invalid or ineffective under applicable law, then the
Waiver shall be preserved to the maximum extent permitted taking into
account Affirmer's express Statement of Purpose. In addition, to the
extent the Waiver is so judged Affirmer hereby grants to each affected
person a royalty-free, non transferable, non sublicensable, non exclusive,
irrevocable and unconditional license to exercise Affirmer's Copyright and
Related Rights in the Work (i) in all territories worldwide, (ii) for the
```

(continues on next page)

(continued from previous page)

```
maximum duration provided by applicable law or treaty (including future
time extensions), (iii) in any current or future medium and for any number
of copies, and (iv) for any purpose whatsoever, including without
limitation commercial, advertising or promotional purposes (the
"License"). The License shall be deemed effective as of the date CC0 was
applied by Affirmer to the Work. Should any part of the License for any
reason be judged legally invalid or ineffective under applicable law, such
partial invalidity or ineffectiveness shall not invalidate the remainder
of the License, and in such case Affirmer hereby affirms that he or she
will not (i) exercise any of his or her remaining Copyright and Related
Rights in the Work or (ii) assert any associated claims and causes of
action with respect to the Work, in either case contrary to Affirmer's
express Statement of Purpose.

4. Limitations and Disclaimers.

 a. No trademark or patent rights held by Affirmer are waived, abandoned,
    surrendered, licensed or otherwise affected by this document.
 b. Affirmer offers the Work as-is and makes no representations or
    warranties of any kind concerning the Work, express, implied,
    statutory or otherwise, including without limitation warranties of
    title, merchantability, fitness for a particular purpose, non
    infringement, or the absence of latent or other defects, accuracy, or
    the present or absence of errors, whether or not discoverable, all to
    the greatest extent permissible under applicable law.
 c. Affirmer disclaims responsibility for clearing rights of other persons
    that may apply to the Work or any use thereof, including without
    limitation any person's Copyright and Related Rights in the Work.
    Further, Affirmer disclaims responsibility for obtaining any necessary
    consents, permissions or other rights required for any use of the
    Work.
 d. Affirmer understands and acknowledges that Creative Commons is not a
    party to this document and has no duty or obligation with respect to
    this CC0 or use of the Work.
```

1.21 origin

In the beginning, there was Sir Dennis Ritchie. And Ritchie saw the void, and outstretched his hand, and commanded "Let there be water." And lo, it was so, and there was the C. And with the C, other entities dared to venture on the void given form. Lord Bjarne Stroustrup too did outstretch his hands and say "Let there be an abundance." And lo, into the sea was cast a double portion of surplus of all the things that swam. And for a while, it was good. But other entities were still curious about what yet lay undefined, and one such pantheon, PUC-RIO, saw that it fitting to create. And thusly, they banded together and declared "Let there be a Moon". And thusly, the moon was born into the sky.

And with the waters and sea made and the moon cast in a starry night sky, PUC-RIO and Ritchie and Stroustrup saw that they did good. They oversaw the moon and the sea and all its abundance, and gave sound council and it overflowed wonderfully. But as the time grew, life grew… discontent. No longer were the simple fishing rods and the flowing tides and the dark sky enough lit by a pale moon and stars enough, no matter how miraculously they were made. They sought out more.

They sought out the light.

And lo, Danny Y., Rapptz did stand firm in the sea and cast his hands to the sky and said "Let there be Light!". And in the sky was cast a Sun. It was an early sun, a growing sun, and many gathered to its warmth, marveling at a life they never knew. And he saw that it was good…

1.21.1 seriously

Sol was originally started by many moon cycles ago to interop with Lua and C++, by Rapptz. It was very successful and many rejoiced at having an easy to use abstraction on top of the Lua API. Rapptz continued to make a number of great projects and has been busy with other things, so upon seeing the repository grow stagnant and tired in the last very long while (over a year), ThePhD forked it into Sol2 and rebooted the code with the hopes of reaching the Milestone and the documentation you have today.

To get to the old repo, head over here.

1.21.2 the name

Sol means sun. The moon (Lua) needs a sun, because without it only the bleak night of copy-paste programming and off-by-one errors would prevail. . . . Or something.

CHAPTER 2

“I need feature X, maybe you have it?”

Take a look at the *Features* page: it links to much of the API. You can also just straight up browse the *api* or ease in with the *tutorials*. To know more about the implementation for usertypes, see *here* To know how function arguments are handled, see *this note*. Don’t see a feature you want? Send inquiries for support for a particular abstraction to the issues tracker.

CHAPTER 3

the basics:

Note: The code below *and* more examples can be found in the examples directory

```cpp
#define SOL_CHECK_ARGUMENTS 1

#include <sol.hpp>
#include "../assert.hpp"

int main() {
	sol::state lua;
	int x = 0;
	lua.set_function("beep", [&x]{ ++x; });
	lua.script("beep()");
	c_assert(x == 1);

	sol::function beep = lua["beep"];
	beep();
	c_assert(x == 2);

	return 0;
}
```

```cpp
#define SOL_CHECK_ARGUMENTS 1

#include <sol.hpp>
#include "../assert.hpp"

struct vars {
	int boop = 0;

	int bop () const {
		return boop + 1;
	}
```

(continues on next page)

(continued from previous page)

```
};

int main() {
        sol::state lua;
        lua.new_usertype<vars>("vars",
                "boop", &vars::boop,
                "bop", &vars::bop);
        lua.script("beep = vars.new()\n"
                "beep.boop = 1\n"
                "bopvalue = beep:bop()");

        vars& beep = lua["beep"];
        int bopvalue = lua["bopvalue"];

        c_assert(beep.boop == 1);
        c_assert(lua.get<vars>("beep").boop == 1);
        c_assert(beep.bop() == 2);
        c_assert(bopvalue == 2);

        return 0;
}
```

CHAPTER 4

helping out

You can support sol2 development by donating here. This is a time-consuming effort, so individuals who donate get to:

- steer the direction and time spent on sol
- get a role on the Discord server
- get their name put up in the CONTRIBUTORS list
- put something of their choice on sol2's README or the documentation's front page

You can also help out the library by submitting pull requests to fix anything or add anything you think would be helpful! This includes making small, useful examples of something you haven't seen, or fixing typos and bad code in the documentation.

Finally, come join in Discord!

CHAPTER 5

Indices and tables

- genindex
- search