binding c++ to lua, part 2

the compiler writes the glue, so binding a method is one line and never a wrapper

what youll have by the end

the part 1 classes, plus a small layer that writes every wrapper for you

- `lua_traits_t<T>`, one entry per type saying how to push it, check it and read it
- `fn_traits_t<fn_t>`, pulls a function type apart into its class, its args and its result
- `method<&type_t::fn>("name")`, a whole wrapper in one line
- `prop<&type_t::field>("name")`, a field lua reads and writes with a dot
- `ctor<type_t, args_t...>()`, a constructor that reports the arg number the script actually passed

and a new game type, `item_t`, bound without a single hand written `l_` function

where this sits

part 1, `lua-c-api-binding-lib`, built the host, the value type, modules and `lua_exporter_t`, and it wrote every wrapper by hand

this is part 2 of the **c++ road**, it keeps every class and name from part 1 and grows them, nothing gets renamed

same deal as before, the gaps are on purpose, each exercise has a stub, **pseudo** steps and a **tip**

every snippet and every error message here came out of a real build against the lua 5.5.1 in this repo

the big picture

every wrapper in part 1 did the same four jobs, only the **types** changed

and the compiler already knows the types, `&inventory_t::add` carries its arg types and its result type inside its own type

so instead of writing a wrapper per method, we write **one template** that reads those types and writes the wrapper itself, once per method, at compile time

like a phrasebook

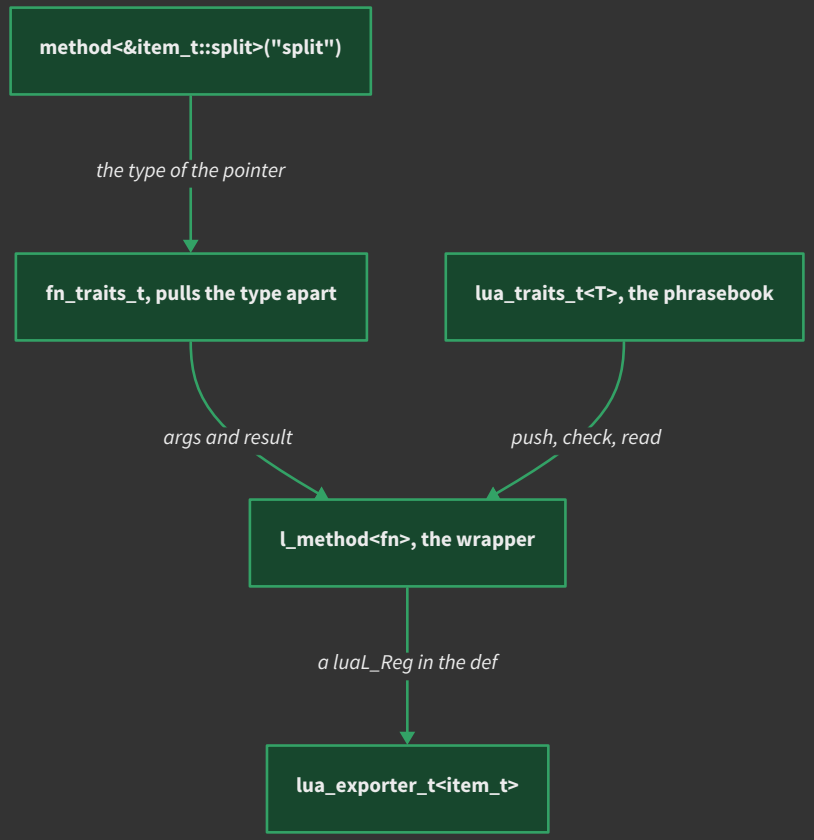
you are in a country where you dont speak the language, so you carry a phrasebook

one page per thing, how to **say** it and how to **understand** it when someone says it back

the translator never improvises, for every word it just looks up the page, and if a page is missing it stops and tells you which one

here the phrasebook is `lua_traits_t`, one page per c++ type, and the generated wrapper is the translator

the map


method<&item_t::split>("split")
the type of the pointer
fn_traits_t, pulls the type apart
lua_traits_t<T>, the phrasebook
args and result
push, check, read
l_method<fn>, the wrapper
a luaL_Reg in the def
lua_exporter_t<item_t>

the plan, seven sections

1. **the pattern** what every part 1 wrapper repeats
2. **the phrasebook** `lua_traits_t`, one entry per type
3. **pulling a function apart** `fn_traits_t` and a pack of numbers
4. **method** the wrapper, generated
5. **constructors** `ctor`, and the arg numbering fixed
6. **properties** fields with a dot, and metamethods of their own
7. **objects in and out** your own types as args and results

section 1, the pattern

three wrappers from part 1

```
int l_add(lua_State* L)
{
        inventory_t& inv  = inventory_lua_t::check(L, 1);
        const char* item  = luaL_checkstring(L, 2);
        const int count           = static_cast<int>(luaL_checkinteger(L, 3));

        inv.add(item, count);
        return 0;
}

int l_count(lua_State* L)
{
        inventory_t& inv  = inventory_lua_t::check(L, 1);
        const char* item  = luaL_checkstring(L, 2);

        lua_pushinteger(L, inv.count(item));
        return 1;
}
```


`l_remove` is the same again with a `bool` coming back, squint and they are one function

the four steps, and who knows them

check self slot 1 must be the right type

read the args slot 2 is the first real arg, each one checked for its type

call the real c++ method

push the results and return how many

every step depends only on **types**, the class, each arg type, the result type, and `&inventory_t::add` has all of them baked into its own type

```
void (inventory_t::*)(const std::string&, int)
```

a member function pointer, plainly

`&inventory_t::add` is a **member function pointer**, it says which method, not which object

you need an object to call it on, and the syntax is odd looking

```
auto fn = &inventory_t::add;
(inv.*fn)("potion", 3);          // the same as inv.add("potion", 3)
```

read its type right to left, a pointer to a member of `inventory_t`, taking a `const std::string&` and an `int`, returning `void`

that type is what the rest of this lecture takes apart

section 2, the phrasebook

lua_traits_t, a template with no body

```
// one specialisation per type, no body here on purpose
template <typename type_t>
struct lua_traits_t;
```

this is the empty phrasebook, it is **declared** but never **defined**, so `lua_traits_t<anything>` is an error until someone writes that page

writing a page is **explicit specialisation**, `template <> struct lua_traits_t<bool> { ... }`, a separate version of the template just for `bool`

that is the extension point, a user binds a new type by adding a page, **never** by editing the library

the first page, bool

```
template <>
struct lua_traits_t<bool>
{
        static void push(lua_State* L, bool _value) { lua_pushboolean(L, _value); }

        // pass 1, may raise
        static void check(lua_State* L, int _idx) { luaL_checktype(L, _idx, LUA_TBOOLEAN); }

        // pass 2, never raises
        static bool read(lua_State* L, int _idx) { return lua_toboolean(L, _idx) != 0; }
};
```

push c++ to lua, **check** is this slot the right type, **read** hand me the value, three functions, every page

`static` bc there is never a `lua_traits_t` object, it is just a named bundle of functions, and `L` stays `L`, the one name finn-style lets through uncased bc the whole lua world calls it that

why check and read are two functions

part 1 ended on one rule, **every `luaL_check` before any c++ object with a dtor exists**

this lua is plain c, so a failed check is a `longjmp`, it skips every dtor on its way out

a single `check` that returned a `std::string` would build arg 1s string, then fail on arg 2, and jump straight over the string

so the page splits in two, the part that can fail and the part that builds

check can raise, builds nothing

read builds the value, and is only ever called once every check has passed, so it can use the non raising `lua_to*` calls

the two passes



integers, one page for every width

```
template <typename type_t>
concept lua_integer = std::integral<type_t> && !std::same_as<type_t, bool>;

template <lua_integer int_t>
struct lua_traits_t<int_t>
{
        static void push(lua_State* L, int_t _value) { lua_pushinteger(L, static_cast<lua_Integer>(_value)); }

        static void check(lua_State* L, int _idx);
        static int_t read(lua_State* L, int _idx);
};
```

`int`, `uint8_t`, `int64_t`, `size_t`, one page covers the lot, a **partial specialisation**, still a template, just narrowed to types that pass `lua_integer`

a **concept** is a named yes or no question about a type, `std::integral` asks is this an integer type, and c++ counts `bool` as one, so we carve it out, `bool` has its own page

why the range check

a lua integer is always **64 bits**, an `int` is usually **32**

so a script can pass a number an `int` cannot hold, and `static_cast<int>` quietly chops the top off, `2^40` becomes `0`

the check has to ask **does this fit** before read ever runs, and c++20 has the exact question, `std::in_range<int_t>(value)`, true when the value fits the type, signed or unsigned

what the script gets back

```
script:1: bad argument #2 to 'add' (integer out of range)
```

and lua itself already refuses a fraction, `inv:add("potion", 2.5)` gives `number has no integer representation`

exercise 1, the integer check

write `check` and `read` for the integer page

pseudo, check

1. `luaL_checkinteger` the slot, it raises on a non number and on a fraction
2. `std::in_range<int_t>` the result
3. if it does not fit, `luaL_argerror(L, _idx, "integer out of range")`

pseudo, read `lua_tointeger`, then `static_cast<int_t>`, nothing in here may raise

tip `std::in_range` refuses `bool` and plain `char` at compile time, `bool` is already carved out, a `char` param will fail to compile inside your check, which is fair, a character is not a number in lua anyway

floats

```
template <std::floating_point float_t>
struct lua_traits_t<float_t>
{
        static void push(lua_State* L, float_t _value) { lua_pushnumber(L, static_cast<lua_Number>(_value)); }
        static void check(lua_State* L, int _idx) { luaL_checknumber(L, _idx); }
        static float_t read(lua_State* L, int _idx) { return static_cast<float_t>(lua_tonumber(L, _idx)); }
};
```

`float` and `double` in one page, a `lua_Number` is a `double` so a `float` loses a little precision on the way in, the same as any `double` to `float` in plain c++

`template <std::floating_point float_t>` is the short way to write a constrained template, the concept sits where `typename` would

std::string

```
template <>
struct lua_traits_t<std::string>
{
        static void push(lua_State* L, const std::string& _value) { lua_pushlstring(L, _value.data(), _value.size()); }
        static void check(lua_State* L, int _idx) { luaL_checkstring(L, _idx); }

        // a copy, lua can free its own string once the call ends
        static std::string read(lua_State* L, int _idx)
        {
                size_t size                = 0;
                const char* str   = lua_tolstring(L, _idx, &size);
                return std::string(str, size);
        }
};
```

read copies, the lua string belongs to the garbage collector, and a copy is the only thing the c++ side can safely keep

the length form keeps a zero byte in the middle, same as part 1s `read`

a number passes `luaL_checkstring` turns `5` into `"5"`, that is luas own rule for strings

string_view, push only

```
// push only, see why
template <>
struct lua_traits_t<std::string_view>
{
        static void push(lua_State* L, std::string_view _value) { lua_pushlstring(L, _value.data(), _value.size()); }
};
```

a `std::string_view` is a pointer and a length into someone elses string, no copy

pushing one is fine, lua copies the bytes as it pushes

reading one would hand c++ a view into a **lua** string, and the method might keep it, then lua collects the string and the view points at freed memory

so the page has no `check` and no `read`, a method taking a `string_view` will not bind, part 6 comes back to where a view is safe

two concepts, can it go, can it come back

```
template <typename type_t>
concept lua_pushable = requires(lua_State* L, const type_t& _value)
{
        lua_traits_t<type_t>::push(L, _value);
};

template <typename type_t>
concept lua_checkable = requires(lua_State* L)
{
        lua_traits_t<type_t>::check(L, 1);
        lua_traits_t<type_t>::read(L, 1);
};
```

a **requires expression** lists some code and asks would this compile, it never runs, the answer is just true or false

a missing page makes `lua_traits_t<T>` incomplete, so the code would not compile, so the concept answers **no** instead of blowing up, section 4 cashes this in

section 3, pulling a function apart

fn_traits_t, pattern matching on a type

```
template <typename fn_t>
struct fn_traits_t;

// a free function
template <typename ret_t, typename... args_t>
struct fn_traits_t<ret_t (*)(args_t...)>
{
        using result_t            = ret_t;
        using arg_list_t  = std::tuple<std::remove_cvref_t<args_t>...>;
};
```

hand it `bool (*)(const item_t&, float)` and the compiler tries the **shape** `ret_t (*)(args_t...)` against it

it fits, so `ret_t` is `bool` and `args_t...` is `const item_t&, float`, the pieces come out **named**

a partial specialisation is a `switch` on types, the case that fits wins and hands you the parts

member functions, one more piece

```
// a member function
template <typename ret_t, typename owner_t, typename... args_t>
struct fn_traits_t<ret_t (owner_t::*)(args_t...)>
{
        using class_t              = owner_t;
        using result_t             = ret_t;
        using arg_list_t  = std::tuple<std::remove_cvref_t<args_t>...>;
};
```

`void (inventory_t::*)(const std::string&, int)` matches, `class_t` is `inventory_t`, `result_t` is `void`

`arg_list_t` is a `std::tuple` type, never an object, just a box to carry a list of types around, bc you cannot store a bare pack in a `using`

`std::remove_cvref_t` strips `const` and `&`, `const std::string&` becomes `std::string`, bc that is the name of the phrasebook page

const is part of the type

`inventory_t::count` is `const`, and that makes it a **different type**

```
int (inventory_t::*)(const std::string&) const
```

the member pattern does not match it, so it needs its own, and it is the same answer, so it inherits

```
// a const member, the same as a member
template <typename ret_t, typename owner_t, typename... args_t>
struct fn_traits_t<ret_t (owner_t::*)(args_t...) const> : fn_traits_t<ret_t (owner_t::*)(args_t...)> {};
```

one line, matching on the const shape and borrowing every answer from the non const one

exercise 2, noexcept

`noexcept` is part of a function type too, since c++17, so `float length() const noexcept` matches **none** of our patterns

add the missing shapes

pseudo

1. a member `noexcept`
2. a const member `noexcept`
3. a free function `noexcept`
4. each one inherits from the plain version, same as the const line

tip a missing shape shows up as `implicit instantiation of undefined template 'fn_traits_t<...>'` with the full type spelled out, read the end of that type and you will see which word it is missing

a pack of numbers

we have a list of **types**, `std::tuple<std::string, int>`, but to read the args we need a list of **numbers** to go with it, 0 for the first, 1 for the second

```
std::make_index_sequence<2>          // is the type std::index_sequence<0, 1>
```

a function taking one can pull the numbers back out as a pack

```
template <auto fn, size_t... I>
int call_method(lua_State* L, std::index_sequence<I...>)
```

`I...` is now `0, 1`, and every `...` below it expands once per number, in step with the types

I + 2, numbers to slots

```
template <size_t I, typename list_t>
using arg_traits_t = lua_traits_t<std::tuple_element_t<I, list_t>>;
```

`std::tuple_element_t<1, args_t>` is the type of arg 1, so `arg_traits_t<1, args_t>` is its phrasebook page

and the slot is `I + 2`, bc a method call `inv:add("potion", 3)` lands like this

slot 1 `inv`, self

slot 2 `"potion"`, arg 0, so `I + 2`

slot 3 `3`, arg 1

lua knows about that hidden self too, a colon call reports slot 3 as `#2`, the number the script actually wrote

the evaluation order trap

```
// pass 1, a comma fold, strictly left to right
(arg_traits_t<I, args_t>::check(L, I + 2), ...);

// pass 2, the args of a call, any order the compiler likes
(self.*fn)(arg_traits_t<I, args_t>::read(L, I + 2)...);
```

the comma fold is guaranteed in order, so the first bad arg is the one reported, like a hand written wrapper

the call args have no order, clang happened to go left to right when tested, the standard promises nothing and other compilers do go the other way

so `read` reads a **fixed slot** and never pops, a reader that popped the top would get the args backwards on some compilers, fixed slots make the order not matter

section 4, method

method, a whole wrapper in one line

```
template <auto fn>
         requires lua_bindable<decltype(fn)>
constexpr luaL_Reg method(const char* _name)
{
         return { _name, l_method<fn> };
}
```

`template <auto fn>` the pointer itself is the template argument, not a type, so `l_method<&inventory_t::add>` and `l_method<&inventory_t::count>` are two separate functions the compiler writes

why not an upvalue a member function pointer is not a plain pointer, it can be twice the size, so it cannot ride in a light userdata, baked into the template it costs nothing at runtime

why not a macro a template is type checked and debuggable, a macro is text pasted in

`lua_bindable` is coming, ignore it for one more slide

l_method, making the numbers

```
template <auto fn>
int l_method(lua_State* L)
{
        using args_t = typename fn_traits_t<decltype(fn)>::arg_list_t;
        return call_method<fn>(L, std::make_index_sequence<std::tuple_size_v<args_t>>{});
}
```

this is the actual `lua_CFunction`, the shape part 1 said is the only one lua will call

its one job is turning **how many args** into the pack `0, 1, ...`, `std::tuple_size_v` counts the tuple, `make_index_sequence` makes the numbers, and `{}` makes an empty object of that type, only its type matters

call_method, pass 1

```
template <auto fn, size_t... I>
int call_method(lua_State* L, std::index_sequence<I...>)
{
        using traits_t    = fn_traits_t<decltype(fn)>;
        using class_t     = typename traits_t::class_t;
        using result_t    = typename traits_t::result_t;
        using args_t      = typename traits_t::arg_list_t;

        // pass 1, self then every arg, in order, anything can raise
        class_t& self = lua_exporter_t<class_t>::check(L, 1);
        (arg_traits_t<I, args_t>::check(L, I + 2), ...);

        // pass 2, next slide
}
```

self is part 1s `check`, the tag check, unchanged

a reference is safe here it has no dtor, so a `longjmp` out of the next line skips nothing

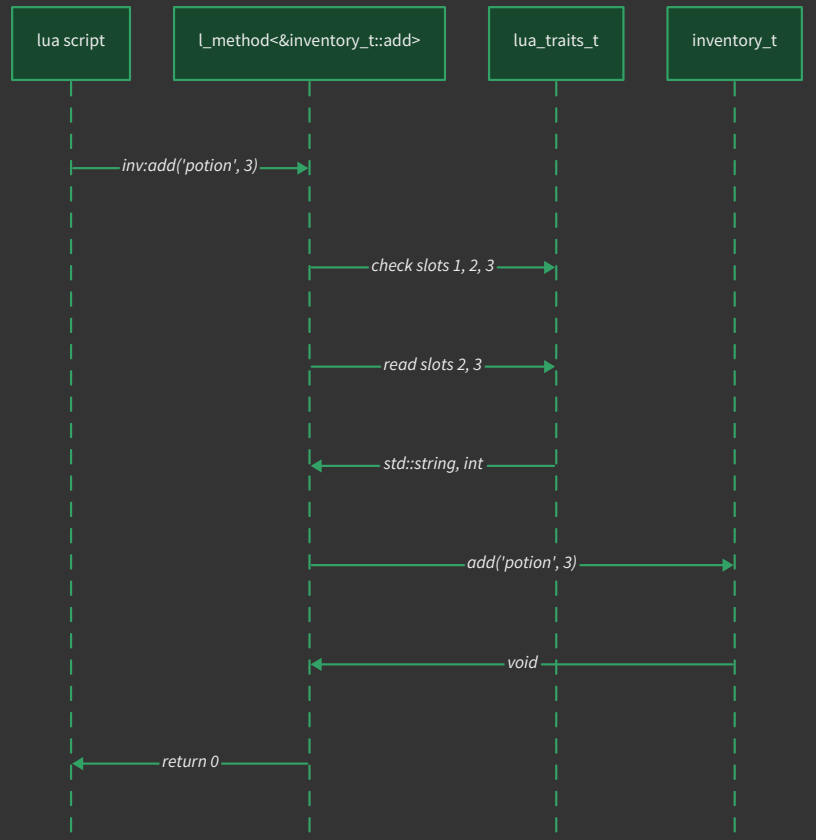
call_method, pass 2

```
// pass 2, nothing in here raises a lua error, it can only throw
return lua_guard(L, [&]
{
        if constexpr (std::is_void_v<result_t>)
        {
                (self.*fn)(arg_traits_t<I, args_t>::read(L, I + 2)...);
                return 0;
        }
        else
                return push_results(L, (self.*fn)(arg_traits_t<I, args_t>::read(L, I + 2)...));
});
```

`lua_guard` is part 1s, a thrown `std::exception` becomes a lua error, and the checks ran **outside** it, so a check failing never jumps over the guards own string

`if constexpr` a `void` method pushes nothing and returns 0, anything else goes to `push_results`, the other branch is never even compiled for that method

the flow, one generated call



the promise, landed

part 1 ended on this exact line, here it is compiling

```
const lua_type_def_t<inventory_t> inventory_def =
{
        .name             = "inventory",
        .methods =
        {
                method<&inventory_t::add>("add"),
                method<&inventory_t::remove>("remove"),
                method<&inventory_t::count>("count"),
        },
};
```

no null row part 1 made you end the list with `{ nullptr, nullptr }`, now the exporter ctor adds it if it is missing, one less thing to forget

```
        if (list->empty() || list->back().name)
                list->push_back({ nullptr, nullptr });
```

what did not survive, default args

part 1s `l_add` had `luaL_optinteger(L, 3, 1)`, so `inv:add("potion")` added one

the generated one says

```
script:1: bad argument #2 to 'add' (number expected, got no value)
```

a **default argument is not part of a function type**, `void add(const std::string&, int _count = 1)` has the same type with or without the `= 1`, so `fn_traits_t` never sees it

two honest fixes, give the method a `std::optional<int>` (a few slides on), or keep a hand written wrapper for that one method, both live in the same list

list, refused at compile time

`list()` returns a `std::vector<std::string>`, and the phrasebook has no page for that yet, part 4 adds one

without a constraint on `method`, clang says this, 2 errors, 29 lines, and **your file is never named**

```
binding.hpp:453:3: error: implicit instantiation of undefined template 'lua_traits_t<std::vector<std::string>>'
  453 |         (arg_traits_t<I, args_t>::check(L, I + 2), ...);
binding.hpp:472:9: note: in instantiation of function template specialization 'call_method<&inventory_t::add_all, 0UL>' requested here
...
```

the error points inside the library, at a line you did not write, and you have to hunt for which of your methods caused it

the constraint, lua_bindable

```
// true when every type in the tuple has a check and a read
template <typename list_t>
constexpr bool lua_all_checkable_v = false;

template <typename... args_t>
constexpr bool lua_all_checkable_v<std::tuple<args_t...>> = (lua_checkable<args_t> && ...);

template <typename fn_t>
concept lua_bindable = lua_all_checkable_v<typename fn_traits_t<fn_t>::arg_list_t>
        && lua_returnable<typename fn_traits_t<fn_t>::result_t>;
```

`lua_returnable` is `void`, or something `push_results` can take, it gets one more rule in section 7

now `method` is only a candidate for bindable pointers, so nothing inside the library is ever built for a bad one

the same mistake, with the constraint

```
host.cpp:98:3: error: no matching function for call to 'method'
   98 |                 method<&inventory_t::list>("list"),
note: because 'decltype(&inventory_t::list)' (aka 'std::vector<std::string> (inventory_t::*)() const') does not satisfy 'lua_bindable'
note: because 'typename fn_traits_t<...>::result_t' (aka 'std::vector<std::string>') does not satisfy 'lua_returnable'
note: and 'push_results(L, _result)' would be invalid: no matching function for call to 'push_results'
```

one error, on your line, naming the type with no page

clang is already fairly kind here, other compilers print a lot more for the unconstrained version, the constrained one reads the same everywhere

and the fix for today, `list` keeps its hand written `l_list` from part 1, `{ "list", l_list }` sits in the same list as the generated ones

exercise 3, free functions

`method` is for members, a module or a type table also wants free functions, `util.can_carry(sword, 25)`

```
bool can_carry(const item_t& _item, float _limit);
```

write `function<&fn>("name")`

pseudo

1. copy `method`, `l_method` and `call_method`, rename them
2. no self, so no `check(L, 1)`
3. args start at slot 1, so `I + 1`
4. call it as `fn(...)` not `(self.*fn)(...)`

tip the free function shape of `fn_traits_t` has no `class_t`, so if your copy still says `typename traits_t::class_t` anywhere it will not compile, that is the compiler finding the self you forgot to remove

multiple results

a lua function can return more than one value, so a c++ method returning a pair or a tuple should too

```lua
local name, count = inv:most()        -- "gold"  50
```

```cpp
template <lua_pushable result_t>
int push_results(lua_State* L, const result_t& _result)
{
        lua_traits_t<result_t>::push(L, _result);
        return 1;
}
```

that is the one value case, a **pair** or **tuple** overload is more specific, so the compiler picks it over this one whenever the result is a tuple

exercise 4, tuple results

write the tuple overload

```
template <lua_pushable... results_t>
int push_results(lua_State* L, const std::tuple<results_t...>& _results);
```

pseudo

1. push each part of the tuple, in order, through its page
2. return how many, `sizeof...(results_t)`
3. then a `std::pair` overload, two pushes, return 2

tip `std::apply(fn, tuple)` calls `fn` with the tuple unpacked as separate args, hand it a generic lambda `[L](const auto&... _each)` and do a comma fold of pushes inside, the part type is `std::remove_cvref_t<decltype(_each)>`

std::optional, nil means empty

lua lets a script leave args off the end, c++ has a type for maybe there

```cpp
void add_some(const std::string& _item, std::optional<int> _count);
```

```lua
inv:add_some("gold")          -- _count is empty
inv:add_some("gold", 4)       -- _count is 4
inv:add_some("gold", nil)     -- empty again
inv:add_some("gold", "x")     -- bad argument #2 to 'add_some' (number expected, got string)
```

a missing arg and an explicit `nil` both read as **empty**, anything else has to pass the inner types check

one page, `lua_traits_t<std::optional<type_t>>`, works for every type that has a page

exercise 5, the optional page

pseudo, check if the slot is none or nil, pass, otherwise run the inner pages `check`

pseudo, read none or nil gives `std::nullopt`, otherwise the inner pages `read`

pseudo, push empty pushes `nil`, otherwise the inner push

tip `lua_isnoneornil(L, idx)` is true for both a slot past the top and a real `nil`, and it is a partial specialisation on a template, `template <typename type_t> struct lua_traits_t<std::optional<type_t>>`, the same move as `fn_traits_t` matching a shape

section 5, constructors

the wart from part 1

part 1s `l_new` slid the userdata **under** the args with `lua_insert(L, 1)`, so `create` saw self at 1 like a method

which pushed every arg up one slot, and a bad **third** arg said

```
bad argument #4 to 'new' (string expected, got table)
```

`new` is called with a dot, `item.new(...)`, so lua has no self to subtract, and it reports the raw slot

the fix is to leave the args where the script put them

l_new, args stay put

```
// makes the userdata on top of the stack, hands back where the object goes
using lua_alloc_t = void* (*)(lua_State*);

// only once every arg is checked, the userdata goes on top
static void* alloc(lua_State* L) { return lua_newuserdatauv(L, sizeof(type_t), 0); }

static int l_new(lua_State* L)
{
        auto& exporter = from_upvalue(L);

        // create checks the args from slot 1, then calls alloc, the userdata is left on top
        exporter.m_def.create(L, alloc);
        luaL_setmetatable(L, m_tag.c_str());
        return 1;
}
```

`create` is no longer handed the memory, it is handed **a way to get it**, so it checks every arg first and only then asks, the userdata lands **on top**, and the **tag still goes on last**

`create` now reads its args **from slot 1**, if you wrote a custom one in part 1 its loop starts at 1 now, not 2.

item_t, the new game type

```
struct item_t
{
        // what it is
        std::string       name;

        // how many in this stack
        int                       stack    = 1;

        // the weight of one
        float             weight   = 0.0f;

        item_t(std::string _name, int _stack, float _weight);

        float total_weight() const;
        item_t split(int _count);
};
```

a **struct**, so bare field names, no `m_`, finn-style keeps `m_` for classes, a struct is plain data you are meant to reach into

ctor, generated

```
template <typename type_t, typename... args_t>
auto ctor()
{
	return [](lua_State* L, lua_alloc_t _alloc)
	{
		[&]<size_t... I>(std::index_sequence<I...>)
		{
			// pass 1, the scripts own arg numbers, from slot 1, nothing made yet
			(lua_traits_t<args_t>::check(L, I + 1), ...);

			// exactly our args, then the userdata above them
			lua_settop(L, sizeof...(args_t));
			void* mem = _alloc(L);

			// pass 2, built straight into the userdata
			lua_guard(L, [&]
			{
				std::construct_at(static_cast<type_t*>(mem), lua_traits_t<args_t>::read(L, I + 1)...);
				return 0;
			});
		}(std::index_sequence_for<args_t...>{});
	};
}
```

reading that ctor

`.create = ctor<item_t, std::string, int, float>()` is how the def uses it, `item_t` has to be named bc `create` only gets back a `void*`, it cannot learn the type

`[&]<size_t... I>(...) { ... }(...)` a c++20 **template lambda**, called on the spot, it is how you get an index pack inside a function without writing a second one

`I + 1` no self in a ctor, so arg 0 is slot 1

`lua_settop(L, sizeof...(args_t))` trims extra args and pads missing ones with `nil`, so the userdata always lands **above** the last arg

the guard a throwing ctor leaves nothing built and no tag, so no `__gc` ever runs on it

the ctor errors, for real

```
item.new("sword", 1, {})
item.new("sword")
item.new("sword", 0, 4.5)
```

```
script:1: bad argument #3 to 'new' (number expected, got table)
script:1: bad argument #2 to 'new' (number expected, got no value)
an item stack needs at least 1
```

#3, the arg the script actually wrote, the part 1 wart is gone

the second is the `std::invalid_argument` from the ctor, caught by the guard and raised as a normal lua error, so `pcall(item.new, "sword", 0, 4.5)` gives `false` and that message

no line number on that one though, exercise 7 fixes it

section 6, properties

fields, not methods

an item has data you want to just read and write

```lua
local sword = item.new("sword", 1, 4.5)
print(sword.weight)        -- 4.5
sword.stack = 3
print(sword)               -- sword x3
```

`sword:weight()` and `sword:set_weight(5)` would work, but it is not how anyone wants to write it

a **property** is a field lua reads with a dot while the value really lives in the c++ object

metamethods, a list of their own

part 1 keeps the methods in their own table, apart from the metatable, bc a script that can reach `__gc` like a method can destroy an object twice, or crash the game with `c.__gc(42)`

that leaves a question for part 2, where do **generated** metamethods go, `__eq` and `__tostring` are just wrappers with self in slot 1, same as any method

not in `.methods` that table is `__index`, a `__tostring` in there would be callable as `sword.__tostring` and lua would never treat it as a metamethod

in `.metas` a new list in the def, set straight onto the metatable, `method<&item_t::to_string>("__tostring")`, the same one line wrapper

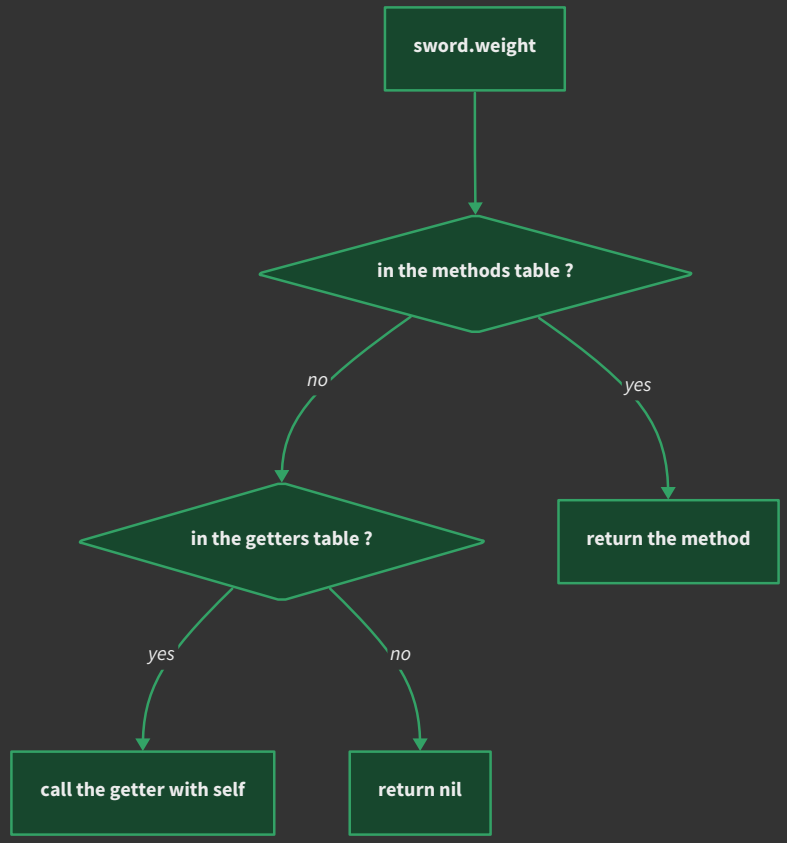
`__metatable` still set, as in part 1, so `getmetatable(sword)` hands back `"item"`, not the table

```
print(sword.nothing, sword.__gc, getmetatable(sword))
nil      nil      item
```

__index becomes a function

with only methods, `__index` can be a table, lua looks the key up in it

a property needs **code** to run, read the field out of the object, so `__index` becomes a function that looks in two places



a pointer to a field

the same trick as a member function pointer, but for data

```
auto field = &item_t::weight;          // type, float item_t::*
sword.*field = 5.0f;                   // the same as sword.weight = 5.0f
```

and a pattern pulls one apart, the same move as `fn_traits_t`, just matching a **value** this time

```
template <typename owner_t, typename field_t, field_t owner_t::* member>
struct member_traits_t<member>
{
        using class_t     = owner_t;
        using value_t     = field_t;
};
```

prop and the getter

```
template <auto member>
int l_get(lua_State* L)
{
        using traits_t = member_traits_t<member>;

        const auto& self = lua_exporter_t<typename traits_t::class_t>::check(L, 1);
        lua_traits_t<typename traits_t::value_t>::push(L, self.*member);
        return 1;
}

template <auto member>
constexpr lua_prop_t prop(const char* _name)
{
        return { .name = _name, .get = l_get<member>, .set = l_set<member> };
}
```

`lua_prop_t` is a plain struct, `name`, `get`, `set`, and `.props` in the def is a list of them,

`prop<&item_t::weight>("weight")`

l_index

```
static int l_index(lua_State* L)
{
        // 1, a method
        lua_pushvalue(L, 2);
        if (lua_rawget(L, lua_upvalueindex(1)) != LUA_TNIL)
                return 1;

        // 2, a getter, the nil from a miss is already on top
        lua_pushvalue(L, 2);
        if (lua_rawget(L, lua_upvalueindex(2)) == LUA_TNIL)
                return 1;

        // a lua_CFunction is a plain function, call it on our own stack
        return lua_tocfunction(L, -1)(L);
}
```

`__index` gets `(self, key)`, upvalue 1 is the methods table, upvalue 2 the getters

`lua_rawget` looks up without metamethods and returns the type it found, one call answers is it there

setup, wiring it in

```
        if (m_def.props.empty())
        {
                // no props, the methods table is __index, the fast path
                lua_setfield(L, -2, "__index");
        }
        else
        {
                lua_createtable(L, 0, static_cast<int>(m_def.props.size()));
                lua_createtable(L, 0, static_cast<int>(m_def.props.size()));

                // ... each prop, get into the getters table, set into the setters

                // stack, metatable, methods, getters, setters
                lua_pushcclosure(L, l_newindex, 1);
                lua_setfield(L, -4, "__newindex");
                lua_pushcclosure(L, l_index, 2);
                lua_setfield(L, -2, "__index");
        }
```


a type with no props keeps the plain table `__index`, you only pay for the function when you use props

exercise 6, the write side

write `l_set<member>` and `l_newindex`

pseudo, l_newindex `(self, key, value)` arrive, look the key up in the setters table (upvalue 1), a miss is `luaL_error(L, "%s has no field '%s' to set", ...)`, a hit gets called

pseudo, l_set check self, check the value through its page, then inside `lua_guard` assign `self.*member = read(...)`

tip call the setter with `lua_call` and a bad value reports `bad argument #2 to '?'` with **no line number**, a function called from c has no name and no line, call it as a plain function like `l_index` does and it runs inside `__newindex`, lua then says `script:1: bad argument #3 to 'newindex'`, the value is at slot 3 so read it there

section 7, objects in and out

opting a type in

`can_carry(const item_t&, float)` takes an **item**, so `item_t` needs a page, and every bound type wants the same page, so it is written once

```
template <typename type_t>
struct lua_object_traits_t
{
        // a copy, lua owns it
        static void push(lua_State* L, const type_t& _value) { lua_exporter_t<type_t>::push(L, _value); }
        static void check(lua_State* L, int _idx) { lua_exporter_t<type_t>::check(L, _idx); }

        // the one in the userdata, not a copy
        static type_t& read(lua_State* L, int _idx) { return *static_cast<type_t*>(lua_touserdata(L, _idx)); }
};

template <>
struct lua_traits_t<item_t> : lua_object_traits_t<item_t> {};
```

one line per type, and it is opt in on purpose, a catch all page for every class would quietly accept a `std::vector` and fail at runtime instead of compile time

push, a factored out new

part 1s capstone tip said to pull the body of `l_new` out, so a method can return a **new** object

```
// a new object, built from the given args, tagged, left on top
template <typename... args_t>
static type_t& push(lua_State* L, args_t&&... _args)
{
        void* mem                   = lua_newuserdatauv(L, sizeof(type_t), 0);
        type_t* obj                 = std::construct_at(static_cast<type_t*>(mem), std::forward<args_t>(_args)...);

        luaL_setmetatable(L, m_tag.c_str());
        return *obj;
}
```

`args_t&&...` with `std::forward` hands the args to the ctor exactly as they came, a copy stays a copy and a temporary gets moved

a method returning a new item

```
local arrows = item.new("arrow", 20, 0.1)
local bundle = arrows:split(5)
print(arrows, bundle, arrows.stack + bundle.stack)
arrows:split(40)
```

```
arrow x15          arrow x5 20
cannot split 40 off a stack of 15
```

`split` returns an `item_t` by value, `push_results` finds the item page, the page calls `push`, and lua owns a brand new userdata

the error is `split` throwing `std::out_of_range`, through the guard, same as the ctor

and part 1s `vec2` capstone is now `method<&vec2_t::normalised>("normalised")`, one line

a reference return, the quiet bug

a player owns its inventory and hands it out by reference

```
inventory_t& inventory() { return m_inventory; }
```

bind that with what we have and `push_results` takes it as `const inventory_t&`, finds the page, and **pushes a copy**

```
local p = player.new()
p:inventory():add("potion", 3)
print(p:inventory():count("potion"))     -- 0
```

the potion went into a copy that was thrown away, no error, no crash, just wrong, the worst kind of bug

refused, until part 3

```cpp
// a T& return promises the same object, a push would hand lua a copy
template <typename type_t>
concept lua_mutable_ref = std::is_lvalue_reference_v<type_t> && !std::is_const_v<std::remove_reference_t<type_t>>;

template <typename result_t>
concept lua_returnable = std::is_void_v<result_t>
        || (!lua_mutable_ref<result_t> && requires(lua_State* L, const result_t& _result) { push_results(L, _result); });
```

```
note: and '!lua_mutable_ref<inventory_t &>' evaluated to false
```

a `const T&` return is still fine, `const std::string& name() const` promises nothing about sameness, a copy is what you would get anyway

a `T&` return means lua should see **the same object c++ holds**, and that needs a userdata that borrows rather than owns, part 3 builds exactly that

errors, the whole picture

pass 1 is outside the guard only a reference and plain numbers are alive, so a `longjmp` skips nothing

pass 2 is inside the guard a thrown `std::exception` becomes a lua error with its message, `pcall` gets `false, msg`

what can still slip a push in pass 2 can run out of memory, and that raise is a `longjmp` with the result still alive, rare, and part 5 gives each state a memory budget

the thrown message has no line `lua_error` raises the value as it is, it never adds `script:1:`, so the next exercise does

exercise 7, a line number on a throw

`item.new("sword", 0, 4.5)` says `an item stack needs at least 1` and nothing about where

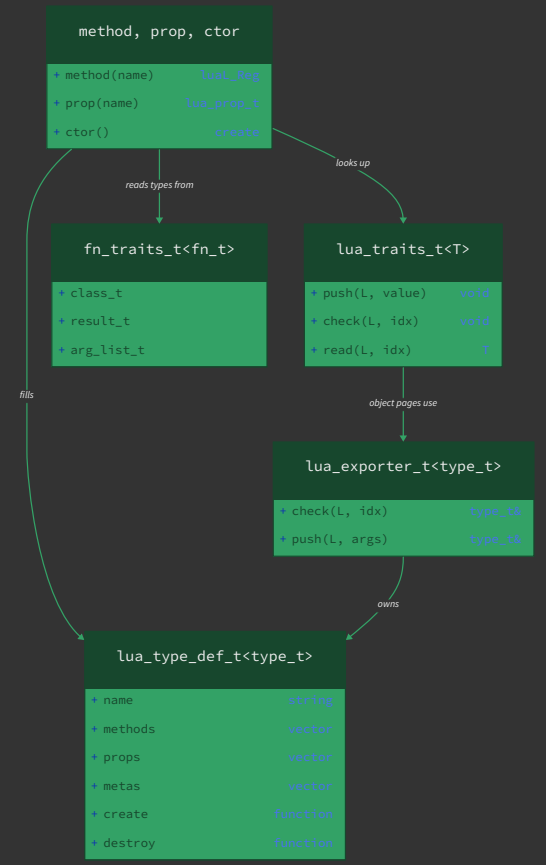
make it say `script:1: an item stack needs at least 1`

pseudo

1. in `lua_guard`, before pushing the message, push where the script is
2. push the message
3. join the two into one string
4. then `lua_error` as before

tip `luaL_where(L, 1)` pushes `"script:1: "` for the function that called us, it is what `luaL_error` does inside, and `lua_concat(L, 2)` joins the top two into one, a multi line script shows the right line too, `script:2:`

the shape of it all



who owns what, part 2 edition

a result by value a new userdata, lua owns it, `__gc` ends it, same as `new`

an object arg by reference `read` hands the method the object **inside** the userdata, no copy

an object arg by value a copy made for that one call, gone when it returns

a `T&` result refused at compile time, part 3

a `std::string` arg a copy per call, part 6 measures what that costs

why its built this way

a phrasebook, not a big switch a new type is a new page, the library never changes, and a missing page is a compile error that names it

check and read split part 1s check first, build second, turned from a habit into the shape of the api, you cannot write a wrapper that gets it wrong

`template <auto fn>` the pointer is baked into the wrapper, no upvalue, no runtime lookup, and the compiler can inline the call

constraints on the entry points errors land on your line, not deep in the library

opt in object pages a wrong type fails at compile time, never at runtime in a script

recap

- `lua_traits_t<T>` is the phrasebook, **push**, **check** that can raise, **read** that cannot
- a **partial specialisation** matches a shape, `fn_traits_t` pulls a method type into class, args and result
- `std::index_sequence` gives a pack of numbers, arg `I` lives in slot `I + 2`
- `method<&type_t::fn>("name")` is a whole wrapper, checks outside `lua_guard`, call and push inside
- `ctor` reads from slot 1, so errors name the arg the script wrote
- `prop` makes a field, `__index` looks in methods then getters, metamethods live apart so `__gc` is out of reach
- by value returns push a copy lua owns, `T&` returns are refused until part 3

have a go, the capstone

bind `item_t` completely, and keep the def **under fifteen lines**, then count them

- `item.new("sword", 1, 4.5)` through `ctor`
- `name`, `stack` and `weight` as properties, readable and writable
- `split` returning a new item, and `total_weight`
- `__eq` so two identical stacks compare equal, and `__tostring` so `print` shows `sword x3`

tip `__eq` is just `method<&item_t::operator==>("__eq")` in `.metas`, but comparing an item with an inventory then raises `bad argument #1 to 'eq' (item expected, got inventory)` instead of giving `false`, a hand written `__eq` using `luaL_testudata`, which returns `nullptr` instead of raising, can say `false` properly

part 3, who owns what, for real

`player:inventory()` would not compile today, and that was the right call

part 3 builds the other owners, a userdata that **borrows** a c++ object, a **handle** that knows when the object is gone, a `shared_ptr` both sides hold

and it fixes identity, so `world.player() == world.player()` is `true`, and `player.inventory` hands back the same inventory every read

plus inheritance, an `enemy_t` passed where an `entity_t&` is expected