binding c++ to lua, part 1

a class based host, so your c++ objects live inside lua scripts like they were always there

what youll have by the end

a small set of c++ classes that sit between your program and lua

- `lua_host_t`, one object that owns a lua state and runs scripts
- `lua_value_t`, any lua value as one c++ type
- `lua_module_t`, a c++ object handed to lua as a table
- `lua_exporter_t<type_t>`, any c++ class turned into a type lua can make with `.new()`

and one c++ class, `inventory_t`, that knows nothing about lua but gets used from a script anyway

where this sits

`lua-c-api-writing-a-library` taught the raw api by hand, the stack, `luaL_check`, metatables, userdata, `__gc`, all in plain c

this is the **binding side**, part 1, the same machinery wrapped up in c++ classes so you write it once and every type after that is cheap

the gaps are on purpose again, each exercise has pseudo steps and a tip pointing at the exact call

the big picture

your game is c++, your gameplay scripts are lua

a **binding** is the glue that lets a lua script reach c++ things, call a function, make an object, call a method on it, and have c++ clean it up when lua is done

the raw c api can do all of it, but every function and every type costs you the same forty lines of stack juggling, so we build classes that do the juggling for us

like a hire shop

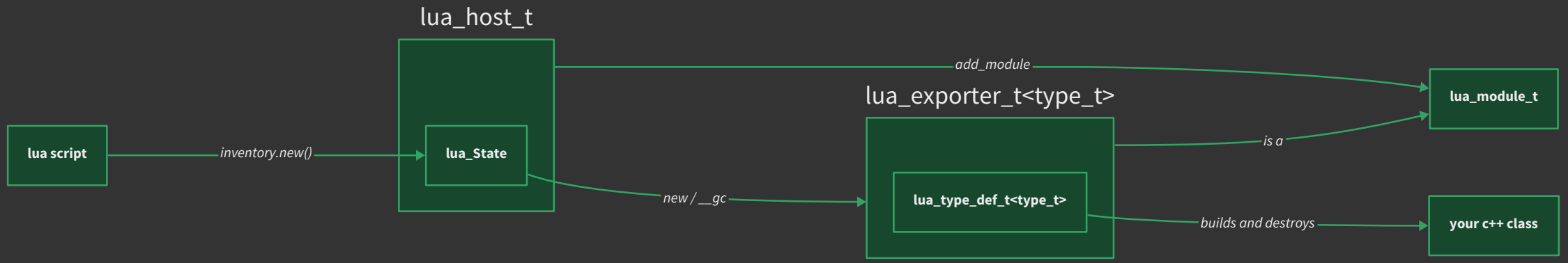
lua walks up to the counter and asks for an inventory, `inventory.new()`

the shop builds one out the back and hands it over with a **tag** on it saying which shop it came from

lua uses it for as long as it wants, and when lua drops it the **returns desk** takes it back and cleans it up

and if lua ever hands the counter something with the wrong tag, the counter refuses it, rather than trying to use a toaster as an inventory

the map


lua_host_t
add_module
lua_module_t
lua_exporter_t<type_t>
lua script
inventory.new()
lua_State
is a
new / __gc
lua_type_def_t<type_t>
builds and destroys
your c++ class

the plan, five layers

we build it bottom up, each layer uses the one under it

1. **the host** a class that owns the lua state and runs scripts
2. **values** one c++ type that can hold any lua value
3. **functions** a c++ function lua can call
4. **modules** a c++ object lua sees as a table
5. **types** a c++ class lua can make as many of as it likes

layer 1, the host

lua_host_t, owning the state

```
class lua_host_t
{
private:
        // the whole lua world, one pointer
        lua_State*                m_state  = nullptr;

        // who hears about errors
        lua_listener_t&   m_listener;

public:
        lua_host_t(lua_listener_t& _listener);
        ~lua_host_t();

        bool run_file(const std::string& _path);
        bool run_string(const std::string& _code);
};
```

a `lua_State` is the whole lua world, every global, every table, every function, one pointer, and every api call takes it first

the ctor and the dtor

```
lua_host_t::lua_host_t(lua_listener_t& _listener) : m_state(luaL_newstate()), m_listener(_listener)
{
        luaL_openlibs(m_state);
}

lua_host_t::~lua_host_t()
{
        lua_close(m_state);
}
```

raii, resource acquisition is initialisation, a fancy name for a plain idea, the object grabs the thing in its ctor and gives it back in its dtor

so there is no `lua_close` to forget, the host going out of scope closes lua for you

the finn-style nod, reading that class

`m_` on `m_state` and `m_listener`, a name tag that says this belongs to the class, not a local

the `_` on `_listener` marks a function argument, the given listener, so it never clashes with a member

the two phase layout every variable first, then every function, so you see what the class *has* before what it *does*

`_t` on every type we write, `lua_host_t`, so a type never looks like a variable

no copies allowed

```
lua_host_t(const lua_host_t&)                    = delete;
lua_host_t& operator=(const lua_host_t&)    = delete;
```

think about what a copy would do, two hosts holding the same pointer, both dtors run, `lua_close` gets called twice on one state, and the second one frees memory that is already gone

deleting the copy turns that runtime crash into a compile error, which is always the better trade

move could be allowed, steal the pointer and null the source, but a host rarely needs to move, so we leave it off until something needs it

running code, load then call

```
//
// runs the given file as one chunk
//
bool lua_host_t::run_file(const std::string& _path)
{
        if (luaL_loadfile(m_state, _path.c_str()) != LUA_OK)
                return report();

        return pcall(0, 0);
}
```

load compiles the file into a **chunk**, the whole file wrapped up as one function, and pushes it

pcall runs whatever function sits on top, protected, so an error comes back as a code instead of crashing

`run_string` is the same with `luaL_loadbuffer` in place of `luaL_loadfile`

the two failures are different

```
print("hi"          -- load fails, a syntax error, never runs
print(a / 2)        -- load is fine, pcall fails, a is nil
```

a **load** error means lua could not understand the code at all

a **run** error means it understood it fine and something went wrong while running

keeping them as two steps is what lets you tell those apart, which matters the first time a modder sends you a broken script

pcall and report, the private helpers

```
//
// calls the function on top, reports any error
//
bool lua_host_t::pcall(int _args, int _results)
{
        if (lua_pcall(m_state, _args, _results, 0) == LUA_OK)
                return true;

        return report();
}

//
// hands the error on top to the listener, then pops it
//
bool lua_host_t::report()
{
        // error(some_table) is legal lua, so the message might not be a string
        const char* msg = lua_tostring(m_state, -1);

        m_listener.on_error(msg ? msg : "error object is not a string");
        lua_pop(m_state, 1);
        return false;
}
```

the listener, errors go somewhere

```
class lua_listener_t
{
public:
        virtual ~lua_listener_t() = default;

        // called with lua error text, load or run
        virtual void on_error(const std::string& _msg) = 0;
};

class log_listener_t final : public lua_listener_t
{
public:
        void on_error(const std::string& _msg) override { std::cerr << "[lua] " << _msg << "\n"; }
};
```

the host never decides *where* errors go, it just tells whoever was handed in

your game passes its own listener, a log file, an in game console, a popup, and the host never changes

exercise 1, a traceback

right now an error says `script.lua:12: attempt to index a nil value`, it does not say how you got there

pseudo

1. write a `lua_CFunction` that takes the message at slot 1 and calls `luaL_traceback` on it
2. in `pcall`, push that function and move it **under** the chunk and its args
3. pass its stack index as the last argument to `lua_pcall`
4. remove it again afterwards, both paths, ok or error

tip the fourth arg of `lua_pcall` is the **message handler**, lua calls it with the error before the stack unwinds, which is the only moment the call chain still exists to print, `lua_insert(L, base)` is how you tuck it underneath

layer 2, values

the type gap

lua has eight types, nil, boolean, number, string, table, function, userdata, thread

c++ has thousands, so a binding has to pick a **mapping**, which c++ type stands in for which lua type

for plain data we only need five

lua	c++
nil	`std::nullptr_t`
boolean	`bool`
number, integer	`lua_Integer`, a 64 bit int
number, float	`lua_Number`, a `double`
string	`std::string`

lua 5.3 and up really does keep integers and floats apart inside one number type, so we keep them apart too

lua_value_t, one type for all of them

```
using lua_value_t = std::variant<std::nullptr_t, bool, lua_Integer, lua_Number, std::string>;
```

a `std::variant` is a box that holds exactly one of a fixed list of types, and knows which one it is holding

so a function can take or return a `lua_value_t` and it might be a bool, a number or a string, decided at runtime, with the compiler still checking every use

no base class, no `new`, no inheritance, the variant does the job a class hierarchy would have

push, visiting the variant

```
//
// pushes the given value, whichever type it holds
//
void lua_host_t::push(const lua_value_t& _value)
{
        auto push_one = [this](const auto& _v)
        {
                using value_t = std::decay_t<decltype(_v)>;

                if constexpr (std::is_same_v<value_t, std::nullptr_t>)
                        lua_pushnil(m_state);
                else if constexpr (std::is_same_v<value_t, bool>)
                        lua_pushboolean(m_state, _v);
                else if constexpr (std::is_same_v<value_t, lua_Integer>)
                        lua_pushinteger(m_state, _v);
                else if constexpr (std::is_same_v<value_t, lua_Number>)
                        lua_pushnumber(m_state, _v);
                else
                        lua_pushlstring(m_state, _v.data(), _v.size());
        };

        std::visit(push_one, _value);
}
```

what just happened there

`std::visit` opens the box and calls your lambda with whatever is inside, typed properly

`auto _v` makes the lambda a template, so the compiler writes one copy per type in the variant

`if constexpr` picks a branch at compile time, each copy only keeps the one line that fits its type

so the string copy never even sees `lua_pushboolean`, and a type you forgot to handle is a compile error, not a silent nil at runtime

read, the other direction

```
//
// reads the value at the given index as a lua_value_t
//
lua_value_t lua_host_t::read(int _idx)
{
        switch (lua_type(m_state, _idx))
        {
                case LUA_TBOOLEAN:
                        return lua_toboolean(m_state, _idx) != 0;

                case LUA_TNUMBER:
                        if (lua_isinteger(m_state, _idx))
                                return lua_tointeger(m_state, _idx);
                        return lua_tonumber(m_state, _idx);

                case LUA_TSTRING:
                {
                        size_t size               = 0;
                        const char* str   = lua_tolstring(m_state, _idx, &size);
                        return std::string(str, size);
                }

                default:
                        return nullptr;
        }
}
```

two quiet details in read

the string is copied on purpose, the `const char*` lua hands you points into a lua string the garbage collector can free the moment it leaves the stack

`std::string(str, size)` not `std::string(str)`, lua strings can hold a zero byte in the middle, the length form keeps all of it

and tables, functions and userdata come back as nil for now, the mapping only covers plain data, bigger things stay on the lua side where they belong

calling lua from c++

```
template <typename... args_t>
std::vector<lua_value_t> lua_host_t::call(const std::string& _name, const args_t&... _args)
{
        const int base = lua_gettop(m_state);

        if (lua_getglobal(m_state, _name.c_str()) != LUA_TFUNCTION)
        {
                lua_settop(m_state, base);
                m_listener.on_error("no function called " + _name);
                return {};
        }

        // push each arg left to right
        (push(_args), ...);

        if (!pcall(sizeof...(_args), LUA_MULTRET))
                return {};

        // whatever sits above base now is the results
        std::vector<lua_value_t> results;
        for (int idx = base + 1; idx <= lua_gettop(m_state); idx++)
                results.push_back(read(idx));

        lua_settop(m_state, base);
        return results;
}
```

the three tricks in call

`(push(_args), ...)` is a **fold expression**, it expands to `push(a), push(b), push(c)`, and the comma guarantees left to right, so arg 1 really is pushed first

`LUA_MULTRET` says give me every result, no matter how many the lua function returned, rather than a fixed count

`base` is the stack top before we started, after the call everything above it is a result, and `lua_settop(base)` throws the lot away in one go

that last one is the habit worth stealing, **note the top, do the work, reset the top**

a note on what c++20 bought us

`host.call("greet", "finn", 3)` has to turn a `const char*` and an `int` into `lua_value_t`s

before c++20 the variant picked the **closest** conversion, and a pointer converts to `bool`, so `"finn"` quietly became `true`

c++20 changed the rule so a variant never picks a **narrowing** conversion, pointer to bool counts as narrowing now, so `"finn"` becomes a `std::string` and `3` becomes a `lua_Integer`

this is a real reason to be on c++20 for a binding, the old behaviour is the kind of bug you stare at for an hour

the stack guard, raii for balance

```
class lua_stack_guard_t
{
private:
        // the state we watch
        lua_State*        m_state  = nullptr;

        // where the top sat when we started
        int                       m_top    = 0;

public:
        lua_stack_guard_t(lua_State* L) : m_state(L), m_top(lua_gettop(L)) {}

        // ensure every push got a matching pop
        ~lua_stack_guard_t() { assert(lua_gettop(m_state) == m_top); }
};
```

drop one at the top of any host function, `lua_stack_guard_t guard(m_state);`, and a debug build tells you the moment a function leaks a value onto the stack

a leaked value is not a crash, it is a slow creep until something far away breaks, so catching it at the

exercise 2, globals

add `get_global` and `set_global` to the host, both in terms of `lua_value_t`

```
lua_value_t get_global(const std::string& _name);
void set_global(const std::string& _name, const lua_value_t& _value);
```

pseudo, get `lua_getglobal` pushes it, `read(-1)`, pop it, return

pseudo, set `push` the value, `lua_setglobal` pops it into the name

tip put a `lua_stack_guard_t` in both, then break one on purpose by removing the pop, and watch the assert fire, that is the guard earning its place

layer 3, functions lua can call

the only shape lua will call

```
typedef int (*lua_CFunction)(lua_State* L);
```

every c++ function lua calls has exactly this shape, one `lua_State*` in, one `int` out

the args are on the stack, slot 1 is the first, `lua_gettop(L)` is how many

the results you push, then return how many you pushed

each call gets its own **private stack**, so slot 1 is always your first arg, whatever else is going on

it is a plain function, not a method, lua is c and has no idea what `this` is, which is the whole problem the rest of this lecture solves

a first host function

```
namespace
{
        //
        // hands lua the hosts version number
        //
        int l_host_version(lua_State* L)
        {
                lua_pushinteger(L, lua_host_t::m_version);
                return 1;
        }
}
```

```
lua_register(m_state, "host_version", l_host_version);
```

`m_version` is a `static constexpr int` on the host, `lua_register` is a macro for `lua_pushcfunction` then `lua_setglobal`, a function is just a value, so setting a global to it is all registering means

the **anonymous namespace** keeps `l_host_version` private to this `.cpp`, the finn-style way to say file

replacing print

scripts love `print`, and the stock one writes straight to stdout, which your game might not even have

so we swap it for ours, and it is just a table write, `print` is a field on the global table like any other

```
const luaL_Reg overrides[] =
{
        { "print",        l_print  },
        { nullptr,        nullptr  },
};

lua_pushglobaltable(m_state);
luaL_setfuncs(m_state, overrides, 0);
lua_pop(m_state, 1);
```

the same trick locks a sandbox down, set `io` or `os.execute` to nil and a modded script cannot touch the disk

l_print, built on the lua side

```
int l_print(lua_State* L)
{
        const int count = lua_gettop(L);

        luaL_Buffer buf;
        luaL_buffinit(L, &buf);
        luaL_addstring(&buf, "[lua]");

        for (int idx = 1; idx <= count; idx++)
        {
                luaL_addchar(&buf, ' ');
                luaL_tolstring(L, idx, nullptr);
                luaL_addvalue(&buf);
        }

        luaL_pushresult(&buf);
        std::cout << lua_tostring(L, -1) << "\n";
        return 0;
}
```

exercise 3, print into your logger

`l_print` writes to `std::cout`, but it should go to the **listener**, so the game decides where script output lands

the problem, `l_print` is a free function, it has no `this`, so how does it find the listener ?

pseudo

1. push the listener pointer with `lua_pushlightuserdata` before calling `luaL_setfuncs`
2. pass `1` as the last arg instead of `0`
3. in `l_print`, fetch it back with `lua_touserdata(L, lua_upvalueindex(1))`

tip you just used an **upvalue**, the next layer is built entirely on that one idea, so get this working before moving on, and add an `on_print` to the listener while youre there

layer 4, modules

the class that knows nothing

```
class inventory_t
{
private:
        // item name to how many we hold
        std::map<std::string, int>          m_items;

public:
        void add(const std::string& _item, int _count);
        bool remove(const std::string& _item);

        int count(const std::string& _item) const;
        std::vector<std::string> list() const;
};
```

no `lua.h`, no stack, no `lua_State`, and it stays that way the whole lecture

that is the point, the **game** class is written like any other class, and a separate layer does the lua side, so you can test it, reuse it, and change your scripting language without touching it

what lua actually sees

lua cannot see a c++ object, so we hand it a **table of functions** instead

```
inventory.add("sword", 1)
inventory.add("potion", 3)
print(inventory.count("potion"))   -- 3
print(inventory.list())            -- potion  sword
```

that table is a **module**, any c++ thing that wants to appear in lua as a table of functions is one, so we give that idea a class

lua_module_t, the interface

```
class lua_module_t
{
public:
        virtual ~lua_module_t() = default;

        // the global name lua sees
        virtual const std::string& name() const = 0;

        // the functions for the table, ending in a null row
        virtual std::span<const luaL_Reg> funcs() const = 0;

        // pushes what every function carries, returns how many
        virtual int push_upvalues(lua_State* L)
        {
                lua_pushlightuserdata(L, this);
                return 1;
        }

        void open(lua_State* L);
};
```

open, the table in four calls

```
//
// leaves this modules table on top of the stack
//
void lua_module_t::open(lua_State* L)
{
        const auto list = funcs();

        // sized for the functions, minus the null row
        lua_createtable(L, 0, static_cast<int>(list.size() - 1));

        // every function gets its own copy of these
        const int ups = push_upvalues(L);
        luaL_setfuncs(L, list.data(), ups);
}
```

```
//
// sets the given module as a global
//
void lua_host_t::add_module(lua_module_t& _module)
{
```

upvalues, a pocket on a function

an **upvalue** is a value stuck to a function, a private pocket it carries around, every call can reach into it with `lua_upvalueindex(n)`

push them onto the stack before `luaL_setfuncs`, then say how many

each function gets a copy, `luaL_setfuncs` copies the same values into every function in the list

a function plus its upvalues is called a **c closure**, same `lua_CFunction`, different pockets

so two inventories can share one `l_add`, each copy just has a different inventory in its pocket, and that is how a free function finds its object

light userdata, a bare pointer

what goes in the pocket is a **light userdata**, a raw c++ pointer dressed as a lua value

it is just the address, lua does not own it, will not free it, and cannot check what it points at, so it is only as safe as your promise that the object is still alive

for a module the promise is easy, the module object lives in `main` for the whole program

the module for inventory_t

```
class inventory_module_t final : public lua_module_t
{
private:
        // the inventory we expose
        inventory_t&                                m_inventory;

        // the name lua sees
        std::string                                         m_name   = "inventory";

public:
        inventory_module_t(inventory_t& _inventory) : m_inventory(_inventory) {}

        const std::string& name() const override { return m_name; }

        // the l_ wrappers, defined in the cpp next to them
        std::span<const luaL_Reg> funcs() const override;

        // the pocket holds the inventory, not the module
        int push_upvalues(lua_State* L) override;
};
```

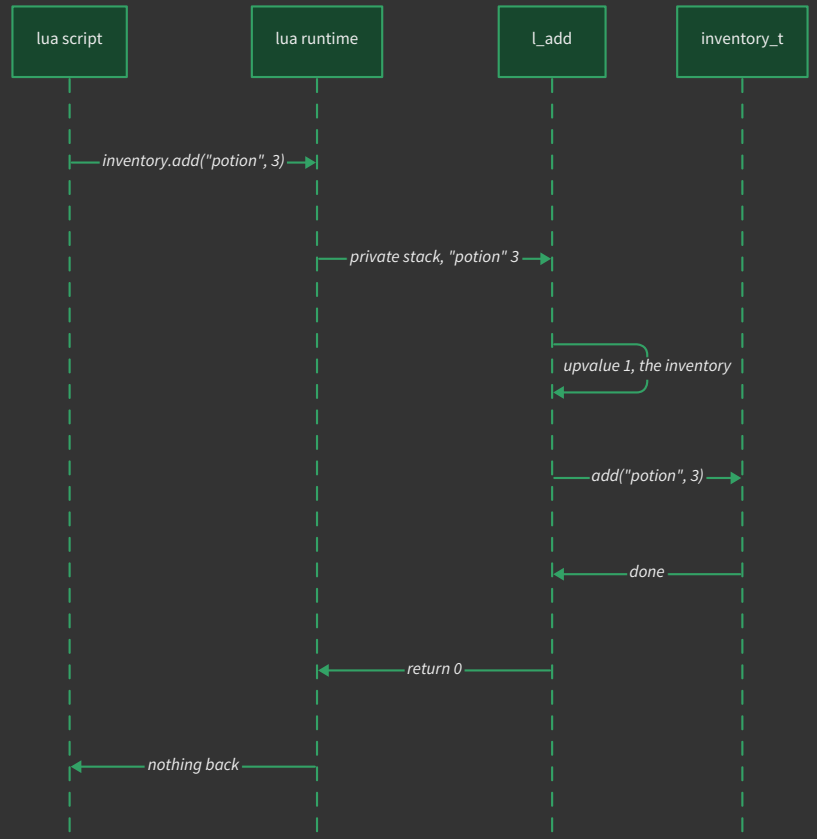
the wrappers

```
namespace
{
	//
	// the inventory this function was registered with
	//
	inventory_t& self(lua_State* L)
	{
		return *static_cast<inventory_t*>(lua_touserdata(L, lua_upvalueindex(1)));
	}

	//
	// adds the given item, an optional count after it
	//
	int l_add(lua_State* L)
	{
		const char* item  = luaL_checkstring(L, 1);
		const int count           = static_cast<int>(luaL_optinteger(L, 2, 1));

		self(L).add(item, count);
		return 0;
	}
}
```

the flow, one call



exercise 4, list

write `l_list`, it returns every item name as a **separate** result, so `print(inventory.list())` prints them all on one line

pseudo

1. `self(L).list()` into a vector
2. make sure the stack has room for all of them
3. push each one with `lua_pushlstring`
4. return the vector size

tip a lua stack only promises about twenty free slots, `luaL_checkstack(L, n, "too many items")` grows it or raises a clean error, skip it and a big inventory writes past the end of the stack, which is a crash on a good day and memory corruption on a bad one

where modules run out

a module is **one** object, made in c++, lent to lua

that suits a service, a logger, the audio system, the one player inventory

but a script cannot say *give me another one*, and it cannot make a chest with its own inventory, a shop with its own stock, a hundred enemies each carrying loot

for that lua has to **make** the objects itself, and then it has to own them, so the last layer is a proper type

layer 5, types lua can make

two kinds of userdata

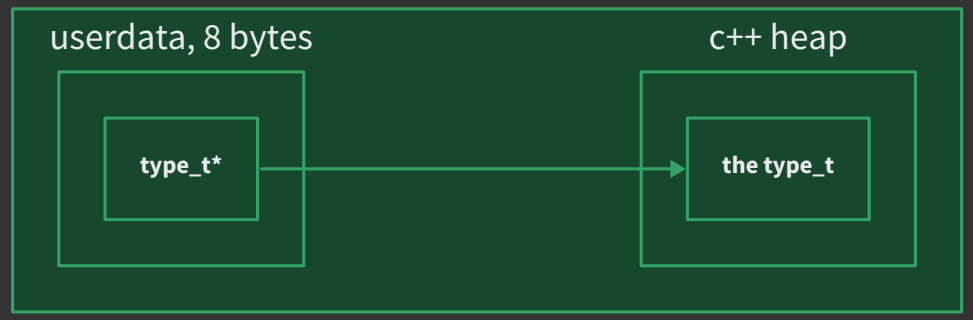
light userdata a bare pointer, lua does not own it, no metatable of its own, what we used for the pocket

full userdata a block of raw memory **lua allocates**, owns, and garbage collects, and it can carry its own metatable

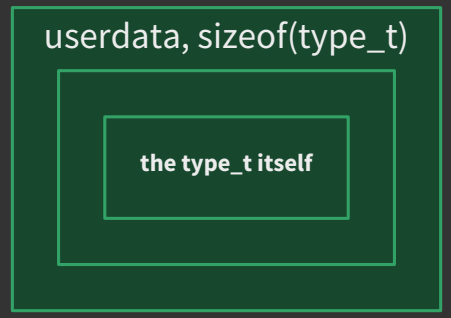
full userdata is the one that becomes a type, lua owns the memory, the metatable says what it is and what it can do

where does the object live

pointer in userdata



object in userdata



the two layouts, weighed

pointer in userdata c++ does `new`, lua only holds the pointer, two allocations and a hop on every call, but the object can be anything, even a derived class behind a base pointer

object in userdata lua allocates `sizeof(type_t)` bytes and the object is built **right there**, one allocation, no hop, and lua owns it outright

we take the second as our default, it is less code, less memory, and the ownership is obvious, lua made it so lua ends it

placement new

a normal `new` does two jobs, **allocate** the memory then **construct** the object in it

lua already did the first job, so we only want the second

```
void* mem                          = lua_newuserdatauv(L, sizeof(inventory_t), 0);
inventory_t* inv  = std::construct_at(static_cast<inventory_t*>(mem));
```

`std::construct_at` is the c++20 spelling of **placement new**, `new (mem) inventory_t()`, build it at this address, do not allocate

the `0` is how many **user values** to attach, extra lua values riding on the userdata, we do not need any

alignment, the fine print

lua lines userdata up for the usual types, doubles, pointers, 64 bit ints

a type marked `alignas(16)` or more, some simd maths types, can land at an address it is not allowed to live at

```
static_assert(alignof(type_t) <= alignof(std::max_align_t), "over aligned, use a pointer in userdata");
```

a `static_assert` costs nothing at runtime and turns a mystery crash into a compile error with your own message, so it goes in once and stays

the tag, a metatable

a fresh userdata is just bytes, lua cannot index it, cannot call a method on it, does not know what it is

so every one gets the same **metatable**, a table lua checks when you do something to a value

`__index` where to look up `inv:add`, we point it at the metatable itself, the methods live there

`__gc` what to run when lua frees it, the returns desk

`__name` the tag, what the type is called, `luaL_newmetatable` sets it for you

the registry, where the tag is kept

one metatable is shared by every object of a type, so it needs a home everyone can find

the **registry** is a hidden table only c and c++ can reach, a script cannot see it or change it

```
luaL_newmetatable(L, "inventory");     // makes it, files it in the registry under the name
luaL_setmetatable(L, "inventory");     // looks it up, sticks it on the value on top
```

and since only c++ can touch the registry, a script can never forge the tag, which is what makes the tag trustworthy

check, refusing the wrong tag

```
//
// the inventory at the given index, or a lua error
//
inventory_t& check(lua_State* L, int _idx)
{
        return *static_cast<inventory_t*>(luaL_checkudata(L, _idx, "inventory"));
}
```

`luaL_checkudata` looks at the metatable of that value and errors unless it is **the** inventory metatable

pass it a number and the script gets `bad argument #1 to 'add' (inventory expected, got number)`, built from `__name`, for free

skip it and a script can hand `add` a different userdata, and your code reads a map out of memory that holds something else, this line is the counter refusing a toaster

methods, self in slot 1

```lua
local chest = inventory.new()
chest:add("gold", 50)       -- the same as inventory.add(chest, "gold", 50)
```

the colon is sugar, it passes the object as the hidden first arg, so every method now finds **self in slot 1** and the real args **from slot 2**

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

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

no upvalue any more, the object arrives with the call, which is exactly how c++ passes `this` behind your back

__gc, the returns desk

```
int l_gc(lua_State* L)
{
        std::destroy_at(static_cast<inventory_t*>(lua_touserdata(L, 1)));
        return 0;
}
```

we built the object by hand, so we destroy it by hand, `std::destroy_at` runs the dtor and nothing else

lua frees the **bytes** after `__gc` returns, you only clean up what is **inside** them, the map, its strings, whatever the object owns

forget `__gc` and lua frees the bytes but the dtor never runs, the map inside never frees its memory, and every inventory a script ever made leaks

the leak you cannot see

without `__gc`, add a print to the ctor and the dtor and run a script that makes three inventories

```
inventory made    0x7f8a
inventory made    0x7f9c
inventory made    0x7fb0
```

three made, **none** destroyed, and nothing warns you, the script runs perfectly and the memory just goes

that print in the ctor and dtor is the cheapest test there is, do it for every type you bind, once

__close, cleaning up on time

`__gc` runs **whenever** the collector gets round to it, maybe seconds later

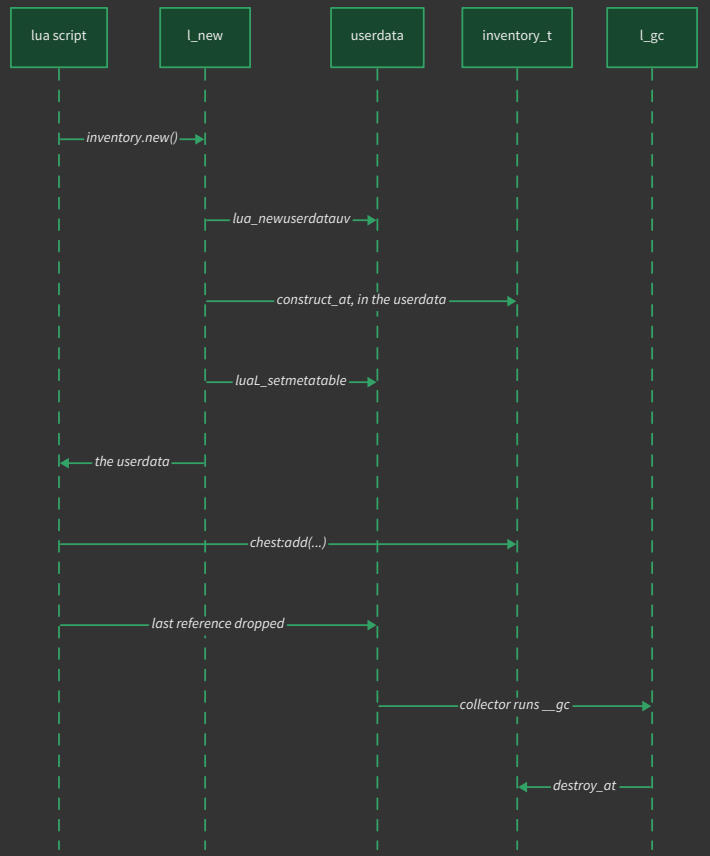
fine for memory, bad for a file, a lock, a network socket, anything that should close **now**

lua 5.4 added **to be closed** locals

```
do
        local save <close> = savefile.open("slot1")
        save:write(data)
end   -- __close runs right here, on the way out, even on an error
```

it is raii for lua, the scope ending is what triggers the cleanup, the same idea as our `lua_host_t` dtor

the lifecycle



exercise 5, __tostring

`print(chest)` shows `inventory: 0x600001234` right now, make it show `inventory (3 items)`

pseudo

1. write `l_tostring`, `check` slot 1
2. count the items with `list().size()`
3. `lua_pushfstring(L, "inventory (%d items)", n)`, return 1
4. set it as `__tostring` on the metatable, next to `__gc`

tip `lua_pushfstring` takes `%d` as an `int`, not a `size_t`, so cast the size first, passing the wrong width to a varargs function compiles clean and prints garbage

layer 5, but for every type

the repetition, again

look at what a type needed

- a `new` that allocates, constructs, tags
- a `__gc` that destroys
- a metatable with `__index` and the methods
- a `check` that knows the tag

swap `inventory_t` for `enemy_t` and **all four are identical** except the type name, and the methods themselves

identical except for a type is exactly what a template is for

split what changes from what does not

what changes per type the name, the method list, how to build one, how to destroy one

what never changes the userdata, the metatable plumbing, the tag check, the `__gc` wiring

so we make two things, a plain **struct** holding the changing bits, and a **template class** holding the plumbing, and the struct gets handed to the class

lua_type_def_t, what changes

```
template <typename type_t>
struct lua_type_def_t
{
        // the global name, and the metatable tag
        std::string name;

        // the methods, ending in a null row
        std::vector<luaL_Reg> methods;

        // builds a type_t in the given memory, ctor args from slot 2
        std::function<void(lua_State*, void*)> create = [](lua_State*, void* _mem) { std::construct_at(static_cast<type_t*>(_mem)); };

        // tears one down, lua frees the bytes after
        std::function<void(type_t*)> destroy = [](type_t* _obj) { std::destroy_at(_obj); };
};
```

struct fields are **bare**, no `m_`, finn-style keeps `m_` for class members only, a struct is plain data and reads that way

why std::function and not virtual

the obvious c++ way is a base class with a `virtual create()`, and every type overrides it

that means a **new derived class** per type just to change one function

a `std::function` member is a slot you fill with any callable, a lambda, a free function, so you change behaviour by **handing it something**, not by inheriting

and the defaults mean most types never touch `create` or `destroy` at all, only the ones that need ctor args or a special cleanup

that is **composition over inheritance**, build behaviour out of parts rather than out of a family tree

lua_exporter_t, the plumbing

```
template <typename type_t>
class lua_exporter_t final : public lua_module_t
{
private:
        // what the type looks like to lua
        lua_type_def_t<type_t>              m_def;

        // the table lua sees, just new
        std::array<luaL_Reg, 2>             m_funcs;

        // the tag, static so check can reach it
        inline static std::string m_tag;

private:
        // what lua calls, static bc lua has no this
        static int l_new(lua_State* L);
        static int l_gc(lua_State* L);
        static lua_exporter_t& from_upvalue(lua_State* L);

public:
        lua_exporter_t(lua_type_def_t<type_t> _def);

        const std::string& name() const override                    { return m_def.name;      }
        std::span<const luaL_Reg> funcs() const override    { return m_funcs;                 }

        void setup(lua_State* L) override;
        static type_t& check(lua_State* L, int _idx);
};
```

a new hook, setup

the metatable has to exist before the first `new`, so modules get one more virtual, with an empty default

```
        // one time work before the table is made
        virtual void setup(lua_State* L) {}
```

and `open` calls it first

```
void lua_module_t::open(lua_State* L)
{
        setup(L);
        // ... the table, the upvalues, luaL_setfuncs, as before
}
```

an empty default means the inventory module does not change at all, only the classes that need it override it

setup, the metatable built once

```
template <typename type_t>
void lua_exporter_t<type_t>::setup(lua_State* L)
{
        // the tag, filed in the registry
        luaL_newmetatable(L, m_tag.c_str());

        // methods live on the metatable, so it is its own __index
        lua_pushvalue(L, -1);
        lua_setfield(L, -2, "__index");
        luaL_setfuncs(L, m_def.methods.data(), 0);

        // the returns desk carries the exporter, same as new does
        lua_pushlightuserdata(L, static_cast<lua_module_t*>(this));
        lua_pushcclosure(L, l_gc, 1);
        lua_setfield(L, -2, "__gc");

        lua_pop(L, 1);
}
```


`lua_pushcclosure(L, fn, 1)` is how you give one function an upvalue, `luaL_setfuncs` is for a whole

finding the exporter again

`new` and `__gc` are `static`, lua calls them with no `this`, so they dig the exporter out of upvalue 1

```
// upvalue 1 is the exporter, pushed as its lua_module_t base
static lua_exporter_t& from_upvalue(lua_State* L)
{
        auto* base = static_cast<lua_module_t*>(lua_touserdata(L, lua_upvalueindex(1)));
        return static_cast<lua_exporter_t&>(*base);
}
```

why the two step cast ? the default `push_upvalues` pushed `this` from inside `lua_module_t`, so the address stored is a `lua_module_t*`

casting a `void*` straight to the derived type only works while those two addresses happen to match, going **back through the base** is correct whatever the layout, so we always cast out the same way we cast in

l_new, built then tagged

```
template <typename type_t>
int lua_exporter_t<type_t>::l_new(lua_State* L)
{
        auto& exporter = from_upvalue(L);

        // the bytes, moved to slot 1 so ctor args start at 2 like a method
        void* mem = lua_newuserdatauv(L, sizeof(type_t), 0);
        lua_insert(L, 1);

        exporter.m_def.create(L, mem);

        // tag it only once it is fully built
        lua_settop(L, 1);
        luaL_setmetatable(L, m_tag.c_str());
        return 1;
}
```

`lua_insert(L, 1)` slides the userdata under the args, so `create` sees the same layout as every method, self at 1, args from 2

l_gc and check

```
template <typename type_t>
int lua_exporter_t<type_t>::l_gc(lua_State* L)
{
        auto& exporter = from_upvalue(L);
        exporter.m_def.destroy(static_cast<type_t*>(lua_touserdata(L, 1)));
        return 0;
}

template <typename type_t>
type_t& lua_exporter_t<type_t>::check(lua_State* L, int _idx)
{
        return *static_cast<type_t*>(luaL_checkudata(L, _idx, m_tag.c_str()));
}
```

`destroy` comes from the def, so whoever decided how to build the object also decides how to end it, create and destroy always live together

using it, the definition

```
namespace
{
        using inventory_lua_t = lua_exporter_t<inventory_t>;

        int l_add(lua_State* L)
        {
                inventory_t& inv = inventory_lua_t::check(L, 1);
                // ... read slot 2 and 3, call inv.add
        }
}

const lua_type_def_t<inventory_t> inventory_def =
{
        .name             = "inventory",
        .methods =
        {
                { "add", l_add             },
                { "remove",       l_remove },
                { "count",        l_count          },
                { "list",         l_list           },
                { nullptr,        nullptr          },
        },
};
```

using it, main

```
int main()
{
        // these outlive the state, lua_close still calls into them
        log_listener_t                      listener;
        lua_exporter_t<inventory_t>         inventory(inventory_def);

        lua_host_t host(listener);
        host.add_module(inventory);
        host.run_file("script.lua");
}
```

declaration order is destruction order, backwards, the host is declared last so it dies first

that matters, `lua_close` runs `__gc` on every object still alive, and `__gc` reaches for the exporter, so the exporter **has** to still exist, swap those lines and the last thing your program does is read freed memory

constructor args from lua

`inventory.new("sword", "shield")` should start with those items, so this type fills in `create`

```
        .create = [](lua_State* L, void* _mem)
        {
                const int top = lua_gettop(L);

                // check every arg first, a bad one jumps out before anything is built
                for (int idx = 2; idx <= top; idx++)
                        luaL_checkstring(L, idx);

                // nothing below can raise, so the object is safe to build
                inventory_t* inv = std::construct_at(static_cast<inventory_t*>(_mem));
                for (int idx = 2; idx <= top; idx++)
                        inv->add(lua_tostring(L, idx), 1);
        },
```

the exporter never learns anything about inventories, it hands over the memory and the stack and lets the type decide, which is the whole point of the def

check first, build second

why two loops ? the one loop version builds the inventory, then checks arg by arg, and `inventory.new("sword", 42, {})` fails on the table

that fail is a `longjmp` in this plain c lua, it jumps out with a live `inventory_t` sat in the userdata, and **no metatable yet**, bc the tag goes on last, so no `__gc` ever runs and the map inside leaks

so the rule for any wrapper, **every `luaL_check` before any c++ object with a dtor exists**

pass 1 checks only, anything can fail here and nothing needs cleaning up

pass 2 builds, and uses `lua_tostring` not `luaL_checkstring`, the checks already passed so nothing in here can raise

the same rule the c api decks teach for a `FILE *`, landing on c++ objects

the real error, and one wart

run the bad call through the fixed version and the script gets

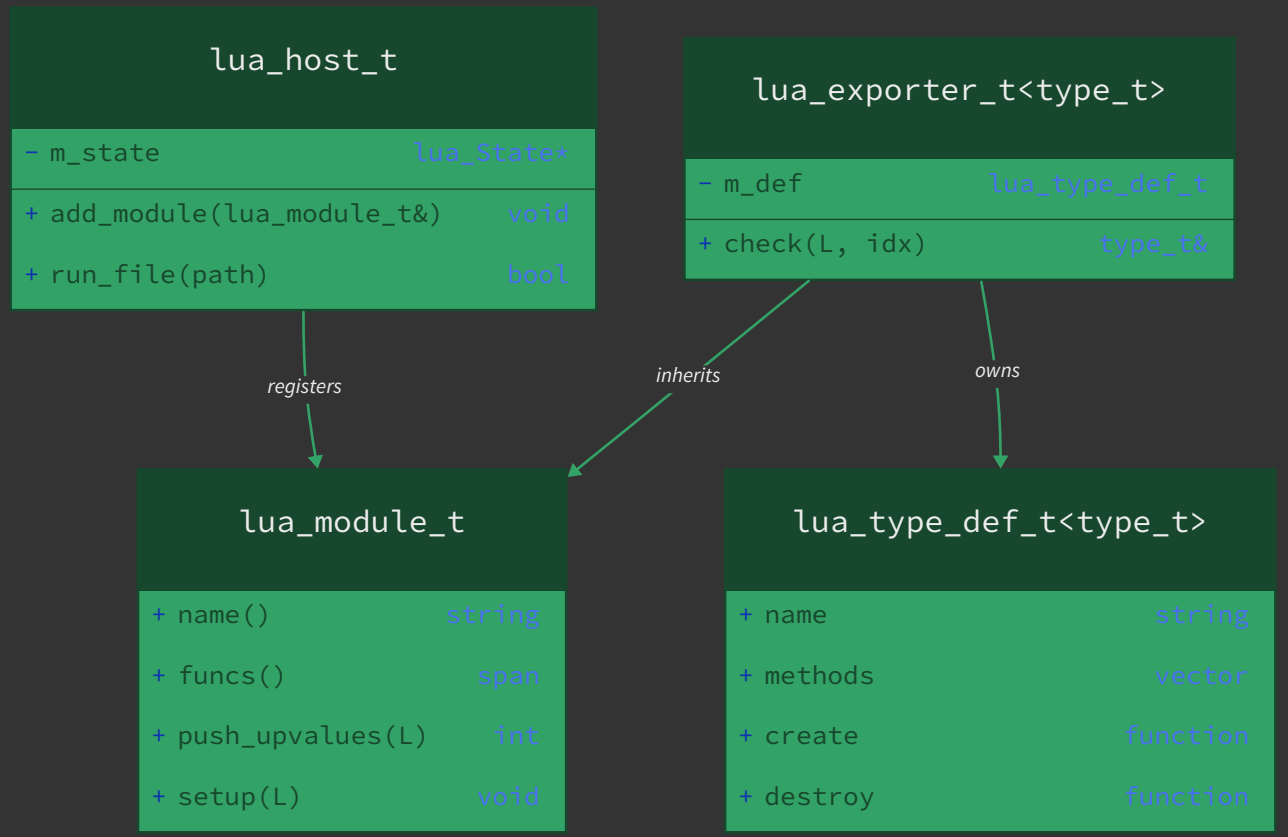
```
bad argument #4 to 'new' (string expected, got table)
```

nothing leaked and nothing crashed, but look at the number, the table was the **third** arg

that is `lua_insert(L, 1)` showing through, the userdata slid in under the args so every slot moved up one, and `luaL_check*` numbers by slot

we live with it for now, part 2 generates the ctor and fixes the numbering there

the shape of it all



exercise 6, a second type

bind a `timer_t` with `start()`, `elapsed()` returning seconds as a float, and `reset()`

pseudo

1. write `timer_t` with no lua in it at all
2. three wrappers, each starting with `lua_exporter_t<timer_t>::check(L, 1)`
3. a `lua_type_def_t<timer_t>` with the three methods
4. declare the exporter in `main` before the host, `add_module` it

tip count the lines that are not the class and not the three wrappers, it should be about ten, if it is more there is plumbing still leaking out of the exporter, go find it

exceptions meet the boundary

your c++ can **throw**, and a throw must never escape into lua, which is c and has no idea what a c++ exception is

so every wrapper that calls real code goes through a guard

```
template <typename func_t>
int lua_guard(lua_State* L, func_t&& _body)
{
        // scoped so the string is gone before lua_error jumps
        {
                std::string error;

                try
                {
                        return _body();
                }
                catch (const std::exception& _e)
                {
                        error = _e.what();
                }

                lua_pushlstring(L, error.data(), error.size());
```

why the guard looks like that

the extra braces `lua_error` never returns, and this lua is plain c, so it is a `longjmp` that skips every dtor on the way out, the `std::string` has to be dead **before** we jump

push, then raise the message is copied into lua first, so it survives the jump and the script sees it

catch `std::exception` **only** if lua is ever rebuilt as c++, its own errors become real throws too, they are not `std::exception` so they pass straight through, a bare `catch (...)` would swallow lua mid error and leave it broken

what it cannot save a `luaL_check` inside the body still jumps straight past the `try`, so the body follows check first, build second

then `pcall(inventory.new, ...)` in lua gets `false, "out of stock"` rather than your game closing

exercise 7, __close safely

add `__close` to the exporter so `local inv <close> = inventory.new()` works for every type

pseudo

1. write `l_close`, same shape as `l_gc`
2. set it as `__close` in `setup`
3. make sure an object closed early is **not** destroyed again by `__gc` later

tip step 3 is the catch, both metamethods fire on one object, so the userdata needs to remember it is already done, allocate `sizeof(type_t) + 1` and keep a `bool` alive flag in that last byte, set in `new`, checked and cleared by whichever of `__close` or `__gc` runs first

a module lua can require

everything so far registers from **inside** the host, a standalone module flips it, lua loads your c++ as a library

```
namespace
{
        // lives for the whole program, lua keeps a pointer to it
        lua_exporter_t<inventory_t> m_inventory(inventory_def);
}

extern "C" int luaopen_inventory(lua_State* L)
{
        m_inventory.open(L);
        return 1;
}
```

`require "inventory"` looks for a library called `inventory` and calls `luaopen_inventory`, that name is the contract

this is where `open` leaving the table on the stack pays off, one line, no copy of the registration code

the three things that bite here

`extern "C"` c++ mangles function names, `luaopen_inventory` would become `_Z17luaopen_inventoryP9lua_State` and lua would never find it, this turns the mangling off

one lua, not two the module must use the **same** lua as the program loading it, link it in twice and you get two copies of lua fighting over one state, it crashes in strange places

windows exports nothing by default on msvc the function also needs `__declspec(dllexport)`, or the dll loads fine and `require` still says it cannot find the function

who owns the object, decided per type

lua owns it object in userdata, placement new, `__gc` destroys, our default, one allocation, cannot dangle

c++ owns, lua holds a pointer `new` in c++, a `type_t*` in the userdata, `__gc` calls `delete`, for a derived class behind a base type, or an object that must never move

c++ owns, lua borrows a light userdata or a module, no `__gc`, only when the object outlives the state, a service, a singleton, the audio system

most binding bugs are this question answered by accident, so answer it on purpose, once, per type

why it's built this way

a host class raii closes lua, deleted copies stop a double close, and the listener decouples errors from where they end up

`std::variant` **for values** one type for all plain lua data, checked at compile time, no hierarchy

game classes know nothing `inventory_t` never includes `lua.h`, the binding is a layer on top you can swap out

def plus exporter what changes lives in a struct you fill in, what never changes lives in a template you write once

tag last, destroy by hand the metatable only goes on a fully built object, and the dtor runs from `__gc` bc we built the object ourselves

recap

- a `lua_host_t` owns the state, raii closes it, errors go to a listener
- `lua_value_t` is a `std::variant`, visit to push, switch on `lua_type` to read
- lua only calls `int fn(lua_State*)`, an **upvalue** is how a free function finds its object
- a **module** is a table of functions, fine for one object c++ lends to lua
- a **type** is full userdata plus a metatable, placement new in, `destroy_at` out on `__gc`
- `luaL_checkudata` is the tag check, never skip it
- `lua_exporter_t<type_t>` plus a `lua_type_def_t` binds any class in about ten lines
- declare what lua calls into **before** the host, so it outlives `lua_close`

have a go, the capstone

bind something from your engine, a `vec2` is a good first one

- `vec2.new(3, 4)` with a custom `create` reading two numbers
- `v:length()` and `v:normalised()`, the second returning a **new** `vec2`
- `__tostring` so `print(v)` shows `vec2(3, 4)`
- `__add` so `a + b` works, it is just one more field on the metatable, with both args checked

tip `normalised` has to push a new object from inside a method, so pull the body of `l_new` out into a `static type_t& push(lua_State*, ...)` on the exporter, returning a new object is something every maths type needs

part 2, stop writing wrappers

look at every `l_add`, `l_count`, `l_list`, they are all the same four steps, **check self, read args, call, push results**

the only thing that changes is the argument types and the return type, and the compiler already knows those from the member function pointer

part 2 teaches the compiler to write the wrappers, so the definition becomes

```
.methods = { method<&inventory_t::add>("add"), method<&inventory_t::count>("count") }
```

and the hand written glue is gone for good