# binding c++ to lua, part 3

ownership and identity, who ends an object, and making one object always one userdata

## what youll have by the end

the part 2 classes, plus every kind of owner a game actually needs

- a **borrowed** object, c++ owns it and lua just points at it
- a **handle**, a borrowed object that knows when it is gone, `enemy is gone` instead of a crash
- a **shared** object, a `std::shared_ptr` both sides hold
- `lua_box_t`, one small header in front of all of them, so `check`, `__gc` and `__close` go down one path
- **identity**, `world.player() == world.player()` is `true`
- **member objects**, `player.inventory` is the real inventory, and it can never outlive its player
- **inheritance**, an `enemy_t` passed where an `entity_t&` is wanted, with its `virtual` overrides

## where this sits

part 1, `lua-c-api-binding-lib`, built the host, modules and `lua_exporter_t`

part 2, `lua-binding-part2`, taught the compiler to write every wrapper, and it **refused** one thing, a method returning `T&`, bc lua would only ever have got a copy

this part lifts that refusal, and to do it properly it has to answer the question part 1 left on its last slide, **who owns the object**

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

so far lua has owned everything it touched, `item.new` made an item, lua kept it, `__gc` ended it

a real game does not work like that, the **world** owns the player and the enemies, c++ decides when they die, and lua only gets to look at them

so each userdata has to say whose object it is, and a script holding one has to be safe even after c++ lets it go

and since c++ hands the same player out over and over, lua has to recognise it as **the same player**, not a new stranger every time

## like a library card system

a library lends books, and it has three kinds

**on loan** the book belongs to the library, you borrow it, and the library can recall it any time

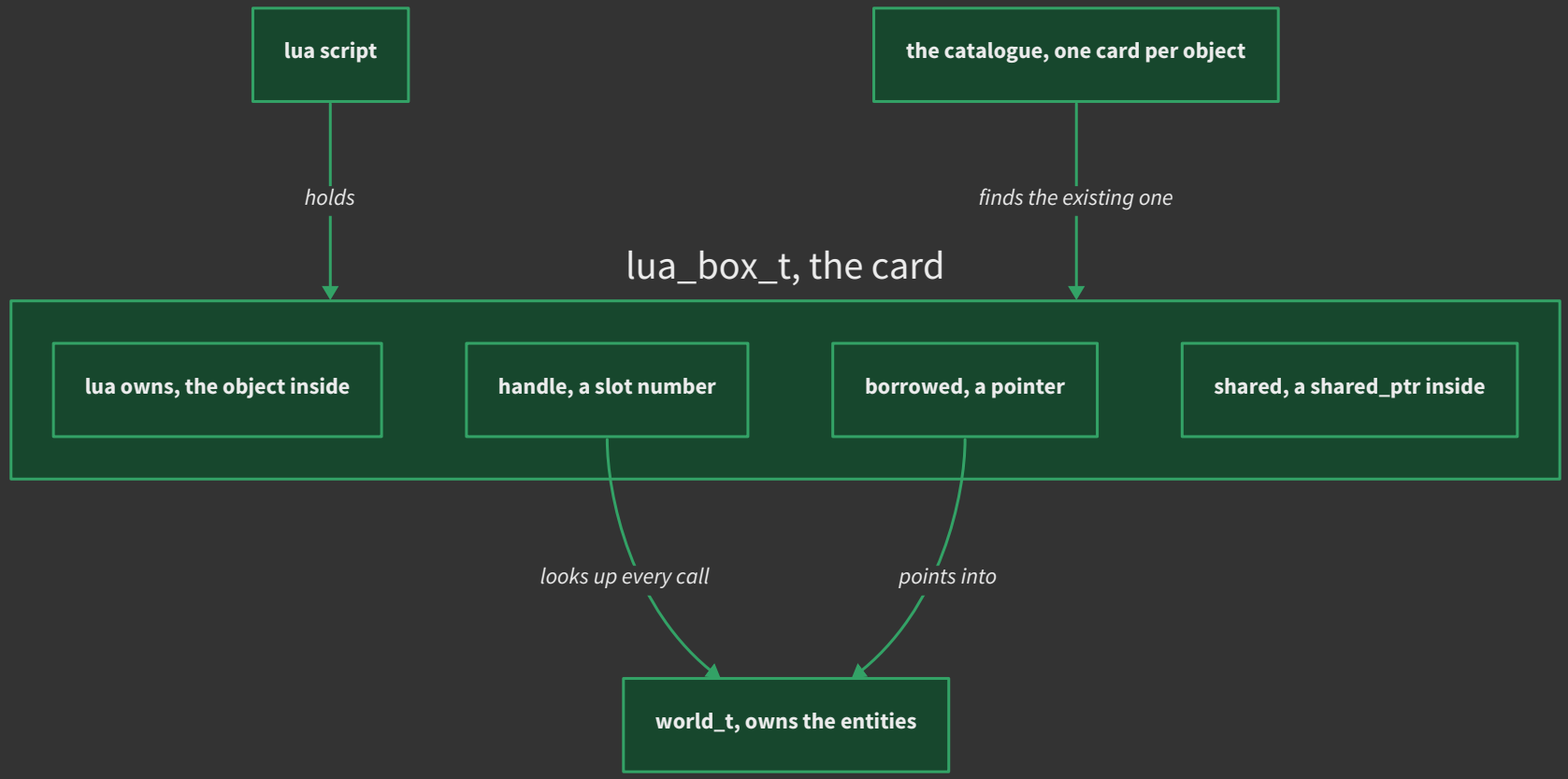
**yours to keep** you bought it at the sale, you decide when it goes

**shared** a book club copy, it stays until the last member hands it back

and the catalogue makes sure **one book is one card**, you never get two cards for the same copy on the same shelf

here the books are c++ objects, the cards are userdata, and the catalogue is a table we build in section 6

# the map


lua script
the catalogue, one card per object
holds
finds the existing one
lua_box_t, the card
lua owns, the object inside
handle, a slot number
borrowed, a pointer
shared, a shared_ptr inside
looks up every call
points into
world_t, owns the entities

## the plan, eight sections

1. **three owners** and the game layer we bind
2. **borrowed** a pointer, and the crash it can cause
3. **handles** a borrowed object that knows it is gone
4. **shared** both sides hold it
5. **one box** a header for all of them, and a fix for part 2
6. **identity** one object, one userdata
7. **member objects** `player.inventory`, tied to its player
8. **inheritance** an enemy where an entity is wanted

# section 1, three owners

## the owners, from part 1s last slide

part 1 ended on this list, and built only the first one

**lua owns it** the object lives inside the userdata, `__gc` destroys it, what `item.new` does

**c++ owns, lua borrows** lua holds a way to find the object, it must never destroy it

**shared** a `std::shared_ptr`, the object lives until both sides let go

this part builds the other two, and it splits borrowing in half, a **plain pointer** for things that outlive the script, and a **handle** for things that can die under it

# the game layer, entity_t

```
class entity_t
{
private:
        // which slot in the world, null until spawned
        handle_t          m_handle;

        // what it is called
        std::string               m_name;

        // 0 is dead
        int                               m_health = 0;

        // what it carries
        inventory_t               m_inventory;

public:
        virtual void take_damage(int _amount) { m_health = std::max(0, m_health - _amount); }
        virtual std::string describe() const { return m_name; }

        // ... handle(), name(), health(), inventory()
};
```

# player_t and enemy_t

```
class player_t final : public entity_t
{
public:
        player_t() : entity_t("finn", 20) {}

        std::string describe() const override { return "the player"; }
};

class enemy_t final : public entity_t
{
private:
        // taken off every hit
        int               m_armour = 2;

public:
        enemy_t(std::string _name) : entity_t(std::move(_name), 10) {}

        // armour soaks the first few points
        void take_damage(int _amount) override { entity_t::take_damage(std::max(0, _amount - m_armour)); }
        std::string describe() const override { return name() + ", an enemy"; }
};
```



`final` nothing derives from these, it matters in section 8

# world_t, the owner

```
class world_t
{
private:
        struct slot_t
        {
                // empty when the slot is free
                std::unique_ptr<entity_t> entity;

                // bumped every time the slot is freed
                uint32_t                  generation       = 1;
        };

        std::vector<slot_t>               m_slots;

public:
        template <typename type_t, typename... args_t>
        type_t& spawn(args_t&&... _args);

        void kill(handle_t _handle);
        player_t& player();
};
```

## who owns what, in this part

**item_t** lua owns it, `item.new`, same as part 2

**the player and the enemies** the world owns them, lua borrows through a **handle**

**the loot** a `std::shared_ptr<item_t>`, the world and lua both hold it

**an inventory inside an entity** borrowed through its **parent**, it lives exactly as long as the entity does

one binding, four kinds of card, and a script cannot tell them apart, `x:method()` works the same on every one

## section 2, borrowed

## the simplest borrow, a pointer

c++ already has the player, so lua does not need a copy, just its address

```
static int l_player(lua_State* L)
{
        lua_exporter_t<player_t>::push_ref(L, world(L).player());
        return 1;
}
```

`push_ref` is new, it pushes a userdata holding a **pointer** to the object, the object stays where it is

and a borrowed userdata has **no destroy** in its `__gc`, lua did not make it, so lua must never end it

## the dangle

the script keeps the player, and c++ kills it

```
local p = world.player()
world.kill(p)
p:take_damage(1)
```

the userdata still holds the old address, the memory behind it is freed, and the call reads it anyway

```
exit 139
```

exit code 139 is a **segmentation fault**, the os killed it for reading memory it no longer owns, not even the first line of output made it out, the game just dies

## the same bug, under a sanitiser

a crash is the lucky case, freed memory often still holds something that looks fine, and the bug hides for months

build with `-fsanitize=address` and every read of freed memory gets caught on the spot

```
ERROR: AddressSanitizer: heap-use-after-free on address 0x607000000090
READ of size 8 at 0x607000000090 thread T0
    #0 in lua_guard<... call_method<&entity_t::take_damage(int), 0ul> ...> binding.hpp:77
freed by thread T0 here:
    #1 in world_module_t::l_kill(lua_State*) host.cpp:48
```

**heap use after free** the exact name for this bug, and it says where it was freed too

## why lua cannot know

a pointer is just a number, it says where, never whether

lua holds the number, c++ frees the memory, and nothing links the two, no one tells the userdata

so a plain pointer is only safe for things that **outlive the lua state**, the audio system, the renderer, the world itself

part 1 said exactly that, a borrow only for things that outlive the state, the player is not one of those

## exercise 1, the borrowed player

write the first version of `push_ref`, the raw pointer one, then make it crash

### pseudo

1. `lua_newuserdatauv(L, sizeof(type_t*), 0)` room for a pointer, not an object
2. store the address in it
3. tag it with the types metatable
4. in `__gc`, destroy nothing
5. run the dangle script above, then again with `-fsanitize=address`

**tip** every method reads the object through `check`, so `check` has to change too, it now reads a `type_t*` out of the userdata and follows it, and `l_new` must still build lua owned objects the old way, so the two layouts cannot share one `check` yet, section 5 fixes that

# section 3, handles

# handle_t, a numbered ticket

finns engine already has the answer, a **handle**, a ticket that stands in for the object

```
class handle_t
{
private:
        // which slot in the pool
        uint32_t m_index                    = 0;

        // which version of that slot, 0 is the null handle
        uint32_t m_generation      = 0;

        // ... index(), generation(), is_null()
};
```

**m_index** which slot, the peg number at a cloakroom

**m_generation** which version of that slot, every time a slot is freed its number goes up, so an old ticket stops matching

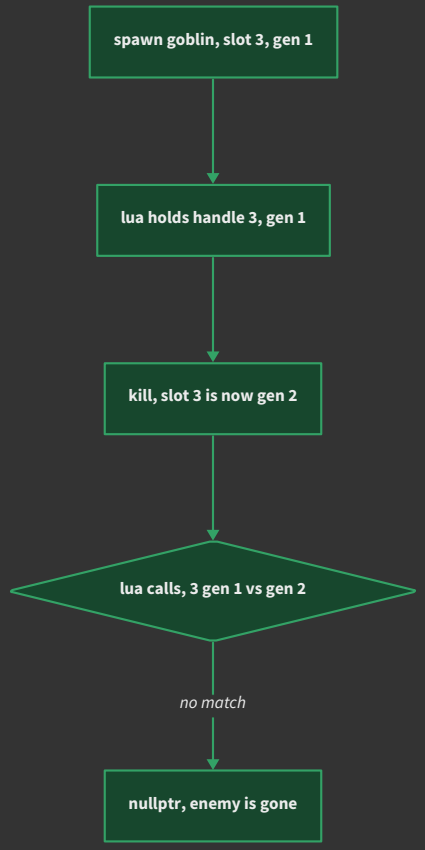
## the world checks the ticket

```
// null when the handle is stale, the slot was freed since
entity_t* get(handle_t _handle)
{
        if (_handle.is_null() || _handle.index() >= m_slots.size())
                return nullptr;

        slot_t& slot = m_slots[_handle.index()];
        return slot.generation == _handle.generation() ? slot.entity.get() : nullptr;
}
```

a **stale** handle is one whose slot has moved on, killed and maybe reused, and it gets `nullptr` back, never someone elses memory

# a ticket going stale



## the def says how to find one

```
lua_exporter_t<enemy_t> enemy({
        .name              = "enemy",
        .parent            = inherits<enemy_t, entity_t>(),
        .find              = [&world](handle_t _handle) { return world.find<enemy_t>(_handle); },
        .create            = ctor<enemy_t, std::string>(),
});
```

`.find` a new field, handle in, pointer out, `nullptr` when stale, the lambda captures the world by reference

`[&world]` so the world must outlive every exporter, declare it first in `main`, the same rule as part 1, whatever lua calls into is declared before the host

ignore `.parent` for now, section 8

## which types become handles

```
// a type that knows its own handle gets pushed as one
template <typename type_t>
concept lua_handled = requires(const type_t& _obj)
{
        { _obj.handle() } -> std::same_as<handle_t>;
};
```

any type with a `handle()` method is pushed as a handle, `push_ref` asks the object for its ticket and stores that instead of the address

`-> std::same_as<handle_t>` inside a `requires`, it says the call must compile **and** give back exactly a `handle_t`

nothing to opt in, `entity_t` has `handle()`, so the player and every enemy pick it up for free

## the same script, with handles

```
local g = world.spawn("goblin")
g:take_damage(5)
print(g.health, g:describe())
world.kill(g)
g:take_damage(1)
```

```
7        goblin, an enemy
script:1: enemy is gone
```

a normal lua error with a line number, `pcall` catches it, the game keeps running

and a new enemy landing in the same slot does not fool it, kill an `imp`, spawn an `elf` into its slot, and the old `imp` card still says `enemy is gone`

## what a handle costs

every call looks the object up again, no caching the pointer

**an index** `m_slots[i]`, one step into a vector

**a compare** the generation, one number against another

that is the whole price, cheaper than the string lookup `check` already does on the tag, and it buys a crash turned into an error

## exercise 2, the handle check

make `check` turn a stale handle into an error

### pseudo

1. read the box, if its owner is `handle`, call the types `find` with the stored handle
2. `nullptr` back means stale
3. raise `luaL_error(L, "%s is gone", name)`
4. otherwise carry on with the pointer `find` gave you

**tip** use `luaL_error`, not `luaL_argerror`, the gone object is usually self, and `luaL_error` adds the scripts line for you, `script:1: enemy is gone`, which is the part that tells you which line to go and fix

## section 4, shared

## loot, held by both sides

the world drops some gold, a script picks it up, then the world clears its loot

```
std::shared_ptr<item_t> loot();
void clear_loot();
```

```
local l = world.loot()
world.clear_loot()
collectgarbage()
print(l, l.stack)
```

```
gold x50 50
```

the world let go, the gold is still alive, the script holds the last `shared_ptr` and it ends when that userdata is collected

# the shared_ptr lives in the box

```
// an object both sides hold, the shared_ptr lives in the box
static void push_shared(lua_State* L, std::shared_ptr<type_t> _obj)
{
        // ... nil for an empty one, the catalogue, section 6

        lua_box_t* box    = new_box(L, lua_box_size<std::shared_ptr<type_t>>, lua_owner_t::shared);
        auto* held                = std::construct_at(lua_payload<std::shared_ptr<type_t>>(box), std::move(_obj));

        box->obj = held->get();
        luaL_setmetatable(L, m_info.name.c_str());
        // ...
}
```

the userdata holds a `shared_ptr`, not the item, so lua counts as one more owner

and `__gc` destroys the `shared_ptr`, never the item, that just drops lua off the count, the item only ends when the count hits zero

## push only, and why

```
// push only, reading one back needs the exact type the box holds
template <typename type_t>
struct lua_traits_t<std::shared_ptr<type_t>>
{
        static void push(lua_State* L, const std::shared_ptr<type_t>& _value) { lua_exporter_t<type_t>::push_shared(L, _value); }
};
```

a phrasebook page with only `push`, like `string_view` in part 2

reading one back would need the **exact** `shared_ptr` type the box holds, an `enemy_t` box read as `shared_ptr<entity_t>` is a different type, so a function that wants the object just takes `item_t&`

## exercise 3, the shared layout

finish shared ownership

### pseudo

1. `push_shared`, from the slide, a `shared_ptr` built in the box
2. in `__gc`, when the owner is `shared`, `std::destroy_at` the `shared_ptr`
3. test it, `g_alive` stays up after `clear_loot`, and drops once the script lets go and the gc runs

**tip** the destroy has to name the exact type it built, `std::destroy_at(lua_payload<std::shared_ptr<type_t>>(box))`, destroying it as a `type_t` would run the items dtor on memory that holds a `shared_ptr`, and that one is a quiet disaster

## section 5, one box

## four cards, one reader

we now have four ways an object can sit behind a userdata, and every method goes through `check`

one `check` per layout would mean four copies of every rule, so every userdata gets the **same header** in front, and `check` reads the header first

```
enum class lua_owner_t : uint8_t
{
        lua,              // the object lives in the box, __gc ends it
        borrowed,         // a plain pointer, c++ owns it
        handle,           // a handle, looked up on every call
        shared,           // a shared_ptr in the box, both sides hold it
};
```

`enum class` a scoped enum, you write `lua_owner_t::lua`, and `: uint8_t` makes it one byte

# lua_box_t, the header

```
// in front of everything lua holds
struct lua_box_t
{
        // the object, unused by a handle
        void*                   obj             = nullptr;

        // only for the handle owner
        handle_t          handle;

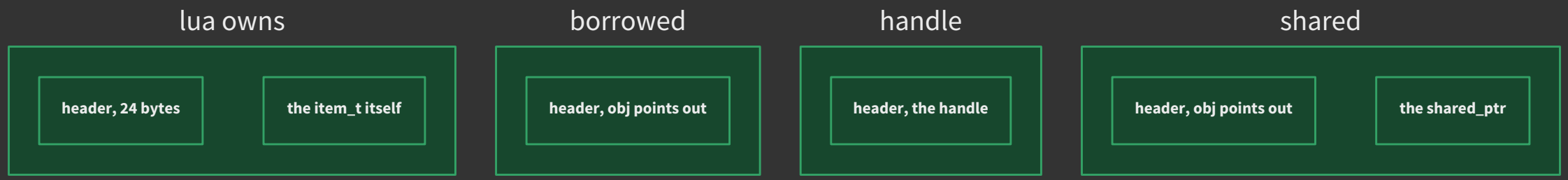
        // who ends it
        lua_owner_t             owner    = lua_owner_t::lua;

        // false once __close or __gc has run
        bool                    alive    = true;
};
```

`obj` a `void*`, bc the header does not know the type, the exporter does, and it casts on the way out

`alive` part 1s exercise 7 kept a `bool` in the byte after the object for `__close`, it lives in the header

# the box, per owner



## after the header, lined up

```
// the object, or the shared_ptr, sits after the header, lined up for its type
template <typename payload_t>
constexpr size_t lua_payload_at = (sizeof(lua_box_t) + alignof(payload_t) - 1) / alignof(payload_t) * alignof(payload_t);
```

the header is 24 bytes, and the thing after it has to start at an address its type allows, its **alignment**

this rounds 24 up to the next multiple of that, `+ align - 1` then divide then multiply, the classic round up in integers

`lua_payload<type_t>(box)` is just the box address plus that many bytes, and part 1s `static_assert` on over aligned types still guards the rest

## one path to the object

```
// where the object is right now, null when it is gone, never raises
inline void* lua_box_object(lua_State* L, int _idx, const lua_type_info_t& _info)
{
        auto* box = static_cast<lua_box_t*>(lua_touserdata(L, _idx));
        if (!box->alive)
                return nullptr;

        if (box->owner == lua_owner_t::handle)
                return _info.find(box->handle);

        // ... a borrowed member checks its parent, section 7

        return box->obj;
}
```

every owner comes out as a plain address or `nullptr`, so nothing after this line cares which card it was

**never raises** it is safe in pass 2 as well as pass 1, part 2s rule

## check, two questions in order

```
// pass 1, raises on the wrong type or a gone object
static type_t& check(lua_State* L, int _idx)
{
        const lua_type_info_t* info = lua_info_of(L, _idx);
        if (!lua_is_a(info, m_info))
                luaL_typeerror(L, _idx, m_info.name.c_str());

        void* obj = lua_box_object(L, _idx, *info);
        if (!obj)
                luaL_error(L, "%s is gone", info->name.c_str());

        return *static_cast<type_t*>(lua_upcast(obj, info, m_info));
}
```

**is it the right type** `lua_info_of` and `lua_is_a` replace `luaL_checkudata`, section 8 shows why, the message is the same `entity expected, got item`

**is it still there** the box, and `enemy is gone` if not

## part 2 had a hole

while wiring the box in, a missing ctor arg gave this, in part 2 as well

```
> item.new("sword")
script:1: bad argument #2 to 'new' (number expected, got userdata)
```

part 2s `l_new` pushed the new userdata **before** calling `create`, so a missing arg 2 was not missing, slot 2 **was** the userdata

the number is right, the reason is nonsense, and with a `std::optional` last arg it is worse, leaving it off is allowed and the call still fails, bc the slot is not empty

**the fix** check every arg **before** the box exists, part 2s own rule, check first, build second, one level up

# create gets an allocator

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

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

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

`create` used to be handed the memory, now it is handed **a way to get it**, and it decides when

the tag still goes on **last**, a half built object never sees `__gc`

## ctor, check then alloc

```
        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 box above them
                        lua_settop(L, sizeof...(args_t));
                        void* mem = _alloc(L);

                        // pass 2, built straight into the userdata
                        // ... lua_guard, std::construct_at(mem, read(L, I + 1)...)
                }(std::index_sequence_for<args_t...>{});
        };
```

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

```
bad argument #2 to 'new' (number expected, got no value)
```

# __gc and __close, one function

```
// __gc and __close, whichever comes first does the work
static int l_gc(lua_State* L)
{
        auto& exporter    = from_upvalue(L);
        auto* box                 = static_cast<lua_box_t*>(lua_touserdata(L, 1));

        if (!box->alive)
                return 0;

        box->alive = false;

        if (box->owner == lua_owner_t::lua)
                exporter.m_def.destroy(static_cast<type_t*>(box->obj));
        else if (box->owner == lua_owner_t::shared)
                std::destroy_at(lua_payload<std::shared_ptr<type_t>>(box));

        return 0;
}
```



borrowed and handle fall straight through, lua never ends what it did not make

## closed, then used

```
local keep
do
        local r <close> = enemy.new("slug")
        keep = r
end
print(keep.health)
```

```
script:1: enemy is gone
```

`<close>` destroyed the slug at the `end`, and `keep` still points at its box, the box says not alive, so `check` says gone

on a borrowed player `<close>` ends only the card, never the player, the next `world.player()` makes a fresh one

## exercise 4, lua_box_t

move every owner onto the header

### pseudo

1. `new_box(L, size, owner)` makes the userdata with **one user value**, builds a `lua_box_t` at the front, sets the owner
2. `l_new` and `push` set `obj` to the payload, `push_ref` to the borrowed address, `push_shared` to `held->get()`
3. `check` goes through `lua_box_object`
4. `l_gc` as above, and set it as both `__gc` and `__close` in `setup`

**tip** set one closure as both fields, push it, `lua_pushvalue(L, -1)` to copy it, then two `lua_setfield`s, each one pops a copy

# section 6, identity

## two cards for one player

```
print(world.player() == world.player())

local t = {}
t[world.player()] = "me"
print(t[world.player()])
```

```
false
nil
```

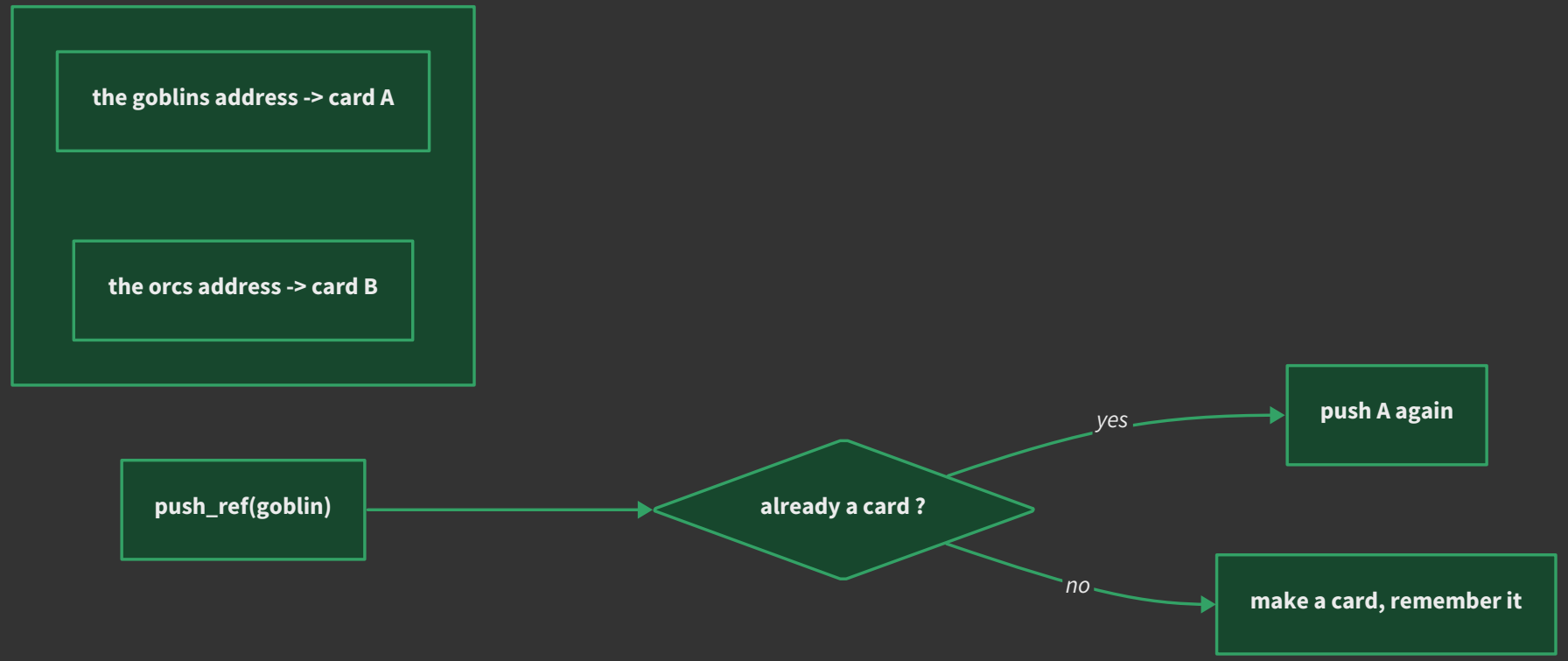
every `push_ref` made a brand new userdata, two userdata are two lua values, and lua compares userdata by **which one**, not by what is inside

so the same player is not equal to itself, and a table keyed by the player loses its entries

# the catalogue

one table per type, remembering which userdata already stands for which object

the enemy cache, weak values




the key is the c++ **address** as a light userdata, part 1s bare pointer, and it lives in the registry, keyed by the address of the types own info, so no two types share one

## weak, so it keeps nothing alive

```
// the identity cache, weak values so it never keeps an object alive
lua_newtable(L);
lua_createtable(L, 0, 1);
lua_pushstring(L, "v");
lua_setfield(L, -2, "__mode");
lua_setmetatable(L, -2);
lua_rawsetp(L, LUA_REGISTRYINDEX, &m_info);
```

a normal table keeps its values alive, so every card ever made would live forever

`__mode = "v"` makes the values **weak**, the table does not count as holding them, once no script holds a card the gc takes it and the entry just disappears

`lua_rawsetp` a raw set with a pointer as the key, made for exactly this

# push_cached

```
// pushes the userdata already standing for the object, false and nothing pushed when there is none
static bool push_cached(lua_State* L, void* _obj)
{
        lua_rawgetp(L, LUA_REGISTRYINDEX, &m_info);

        // a hit only counts if it still finds the same object
        if (lua_rawgetp(L, -1, _obj) == LUA_TUSERDATA && lua_box_object(L, -1, m_info) == _obj)
        {
                lua_remove(L, -2);
                return true;
        }

        lua_pop(L, 2);
        return false;
}
```

`push_ref` and `push_shared` call this first, and on a miss they make a card and store it with `cache`

## the address trap

an address is not an object, memory gets reused

kill the goblin, spawn an elf, and the allocator may well hand the elf **the same address** the goblin had

the cache still has the goblins card under that key, and a naive hit hands the script the dead goblins card for a live elf

so a hit is **checked**, the old card goes through `lua_box_object`, the goblins handle is stale so it gives `nullptr`, that is not the elfs address, so it counts as a miss and a new card overwrites the old entry

## identity, landed

```
> print(world.player() == world.player())
true
> local t = {} t[world.player()] = "me" print(t[world.player()])
me
```

the same card every time, so `==` works with no `__eq`, and so does a table key

one limit, the cache is **per type**, the same enemy pushed as an `entity_t&` gets an entity card, not its enemy card, keep a type returning the most derived reference it can

## exercise 5, the identity table

### pseudo

1. in `setup`, make the weak table and put it in the registry at `&m_info`
2. write `push_cached` and `cache`
3. `push_ref` and `push_shared` try `push_cached` first
4. test `world.player() == world.player()`, then kill and spawn in a loop and check an old card never answers for a new enemy

**tip** by value pushes never go in the cache, `item:split()` makes a new object every time so there is nothing to find, and caching them would only fill the table

## section 7, member objects

## the quiet bug, fixed

part 2 refused this, bc it pushed a copy

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

now a `T&` to a bound type goes out through `push_ref`, the **same** inventory, no copy

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

```
3       true
```

the potion lands in the real inventory, and the cache makes every read the same card

## a prop from a getter

`inventory` is a method, not a field, and it still reads with a dot, `p.inventory`

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

        // std::invoke reads a field and calls a getter the same way
        class_t& self = lua_exporter_t<class_t>::check(L, 1);
        return push_result<decltype(std::invoke(member, self))>(L, std::invoke(member, self), 1);
}
```

**`std::invoke`** calls anything callable the same way, a field pointer gives the field, a method pointer calls the method

**`prop<&entity_t::inventory>`** and `prop<&entity_t::health>` both work, a getter method has no setter, so `p.health = 3` says `player has no field 'health' to set`

# push_result picks the card

```
// pushes one result as its type says, parent is the slot a borrowed one is kept to
template <typename result_t>
int push_result(lua_State* L, result_t&& _result, int _parent)
{
        if constexpr (lua_ref_return<result_t>)
        {
                lua_exporter_t<std::remove_reference_t<result_t>>::push_ref(L, _result, _parent);
                return 1;
        }
        else
                return push_results(L, _result);
}
```

`lua_ref_return` a non const `T&` where `T` has an object page, part 2s refusal turned into a route

`_parent` methods and getters pass `1`, self, a free function passes `0`, the next slide is why

## a member of a dying parent

```
local inv = enemy.new("bat").inventory
collectgarbage()
inv:add("wing", 2)
```

the bat is lua owned and nothing holds it, so the gc ends it, and its inventory with it, `inv` points into freed memory

```
ERROR: AddressSanitizer: heap-use-after-free on address 0x60e0000000c8
READ of size 8 at 0x60e0000000c8 thread T0
    #2 in call_method<&inventory_t::add(...), 0ul, 1ul> binding.hpp:830
```

the inventory is not a separate object, it is **inside** the bat, its life is the bats life

## the user value, a string back to the parent

every box is made with **one user value**, a lua value stored on the userdata itself

```
lua_box_t* box = new_box(L, sizeof(lua_box_t), lua_owner_t::borrowed);
box->obj = &_obj;

if (_parent)
{
        lua_pushvalue(L, _parent);
        lua_setiuservalue(L, -2, 1);
}
```

`lua_setiuservalue(L, -2, 1)` pops the parent and stores it in user value 1 of the new card

the gc sees the inventory card **holds** the bat card, so while a script holds the inventory the bat stays alive, and the same script now prints `2`

## a parent that can die in c++

the bat was lua owned, the gc was the danger, an **orc** from the world is killed by c++, and the gc has no say in that

```
local g = world.spawn("orc")
local inv = g.inventory
world.kill(g)
inv:add("x", 1)
```

```
script:1: inventory is gone
```

so `lua_box_object` asks the parent too, for a borrowed card it reads user value 1, and if that parent resolves to `nullptr` the member is gone as well

```
if (lua_getiuservalue(L, _idx, 1) == LUA_TUSERDATA)
{
        // ... resolve the parent the same way, nullptr means we are gone too
}
```

## exercise 6, member objects

### pseudo

1. `new_box` makes every userdata with `lua_newuserdatauv(L, size, 1)`, one user value
2. `push_ref` takes `_parent`, and stores it as user value 1 on a borrowed card
3. `lua_box_object`, for a borrowed card with a parent, resolve the parent first, `nullptr` if it is gone
4. test the bat, then the orc

**tip** make `_parent` absolute with `lua_absindex` **before** you push anything, a `-1` meant self when you were handed it, and it means your new box the moment the box is on the stack

# section 8, inheritance

## an enemy is an entity

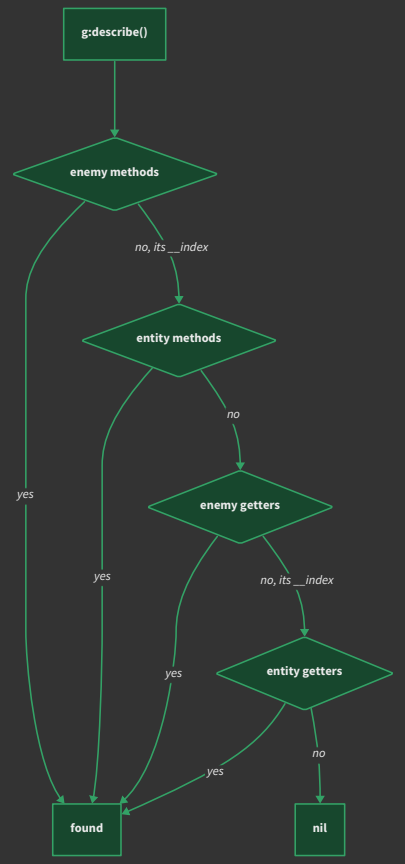
`take_damage`, `describe`, `name`, `health` and `inventory` all belong to `entity_t`, so they are bound **once**, on the entity

```
lua_exporter_t<player_t> player({
        .name              = "player",
        .parent            = inherits<player_t, entity_t>(),
        .find              = [&world](handle_t _handle) { return world.find<player_t>(_handle); },
});
```

the player def has **no methods of its own** and still does everything, the enemy the same

two things have to happen, lookups fall back to the entitys tables, and `check<entity_t>` has to accept a player

# lookups fall back




each table has its own metatable whose `__index` is the **parents** table, so lua walks the chain for us

# new_chained

```
// a table stored on the metatable as the given field, falling back to the parents one
void new_chained(lua_State* L, int _meta, const char* _field)
{
        lua_newtable(L);

        if (m_info.parent)
        {
                lua_createtable(L, 0, 1);
                luaL_getmetatable(L, m_info.parent->name.c_str());
                lua_getfield(L, -1, _field);
                lua_setfield(L, -3, "__index");
                lua_pop(L, 1);
                lua_setmetatable(L, -2);
        }

        lua_pushvalue(L, -1);
        lua_setfield(L, _meta, _field);
}
```



`setup` makes `__methods`, `__getters` and `__setters` this way, and stores each on the metatable, so a

## gettable, not rawget

part 2s `l_index` used `lua_rawget`, a lookup that **skips** metatables, fast, but it would never reach the parent

```
        // 1, a method, the table falls back to the parents
        lua_pushvalue(L, 2);
        if (lua_gettable(L, lua_upvalueindex(1)) != LUA_TNIL)
                return 1;
```

`lua_gettable` a normal lookup, metatables and all, so a miss walks up the chain

the price is one extra lookup per level on a miss, a hit on the types own table costs the same as before, part 6 measures it

# check, now for a family

`luaL_checkudata(L, 1, "entity")` compares one tag, and a player is tagged `"player"`, so it says no

so each metatable carries a pointer to a small c++ struct describing its type, including its parent

```
struct lua_type_info_t
{
        // the tag, the metatable name
        std::string                                     name;

        // the base type, null for a root
        const lua_type_info_t*                  parent   = nullptr;

        // this type to its parent, the address can move
        void*                                           (*upcast)(void*) = nullptr;

        // ... find, the handle lookup
};
```

`setup` stores `&m_info` on the metatable as `__info`, a light userdata, and `lua_info_of` reads it back with `luaL_getmetafield`

## is it one of the family

```
// is the given type the wanted one, or derived from it
inline bool lua_is_a(const lua_type_info_t* _info, const lua_type_info_t& _want)
{
        for (; _info; _info = _info->parent)
        {
                if (_info == &_want)
                        return true;
        }

        return false;
}
```

start at the objects real type and walk up, a player meets the entity one step up, an item never does

comparing **addresses** of infos, one per type, not strings, and a non userdata or someone elses userdata has no info, so it is `nullptr` and the answer is no

## the address can move

a `static_cast` from derived to base is **not always free**, with two bases, one of them cannot sit at the start

```
struct npc_t : ai_t, entity_t {};
```

```
the npc_t     0x16d702a38
as entity_t   0x16d702a48
from a void*  0x16d702a38
health, right 10, wrong 0
```

the entity part lives **16 bytes in**, a real cast adds 16, a cast through `void*` does not know to, and reads the `ai_t` instead, health `0`

`lua_box_t` stores a `void*`, so the step up has to be done by code that knows **both** types

# an upcast per link

```
// a derived type names its base, the upcast is written here where both types are known
template <typename type_t, typename base_t>
        requires std::derived_from<type_t, base_t>
lua_parent_t inherits()
{
        return { &lua_exporter_t<base_t>::info(), [](void* _obj) -> void* { return static_cast<base_t*>(static_cast<type_t*>(_obj)); } };
}
```

the lambda turns the `void*` back into a real `type_t*` **first**, then casts to the base, so the compiler adds whatever offset it needs

a lambda with no capture converts to a plain function pointer, so it fits `upcast` for free

`lua_upcast` walks the same chain as `lua_is_a`, calling `upcast` at each step

## the payoff, virtual

```
inline void deal_damage(entity_t& _target, int _amount) { _target.take_damage(_amount); }
```

```
local g = world.spawn("goblin")
g:take_damage(5)
world.deal_damage(g, 5)
print(g.health)
```

```
4
```

`take_damage` is bound **once**, on `entity_t`, and the goblins armour still soaked 2 of each hit, 10 minus 3 minus 3

that is c++ doing it, `check<entity_t>` handed back a real `entity_t&`, and a `virtual` call through it runs the override, the binding did nothing special

## why not lua owned through the base

a by value `entity_t` return **copies** into an `entity_t` sized box, and an enemy does not fit, so the enemy part gets cut off, **slicing**

```
entity_t copy_of(entity_t& _entity) { return _entity; }
```

```
goblin, an enemy  goblin
7         5
```

the copy forgot it was an enemy, no armour, no override, and no error

so the object page refuses a copy of a type that could be a base

```
        static void push(lua_State* L, const type_t& _value)
                requires (!std::is_polymorphic_v<type_t> || std::is_final_v<type_t>)
```

## the slicing refusal

```
host.cpp:76:4: error: no matching function for call to 'function'
   76 |                          function<&copy_of>("copy_of"),
note: because 'typename fn_traits_t<entity_t (*)(entity_t &)>::result_t' (aka 'entity_t') does not satisfy 'lua_returnable'
note: and 'push_results(L, _result)' would be invalid: no matching function for call to 'push_results'
```

`std::is_polymorphic_v` true for any class with a `virtual`, it might be a base

`std::is_final_v` true for `enemy_t`, nothing can derive from it, so a copy is exactly that type and cannot slice

so `enemy.new("rat")` is fine, lua owning an enemy made as an enemy, what cannot happen is lua owning one through `entity_t`, polymorphic bases are borrowed or shared

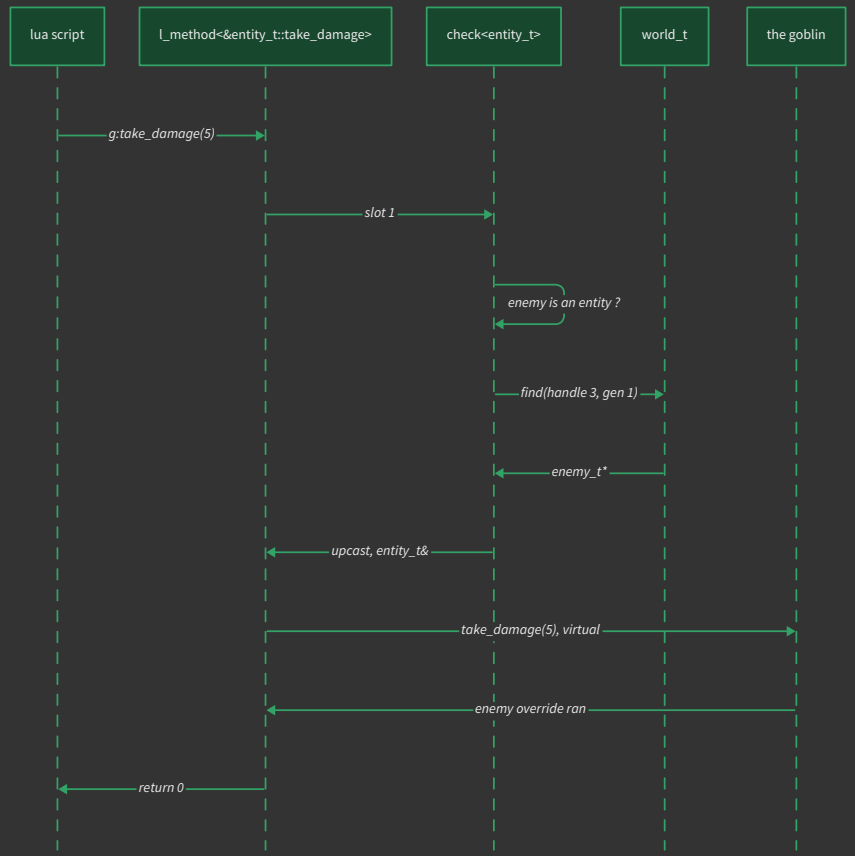
## exercise 7, the base chain

### pseudo

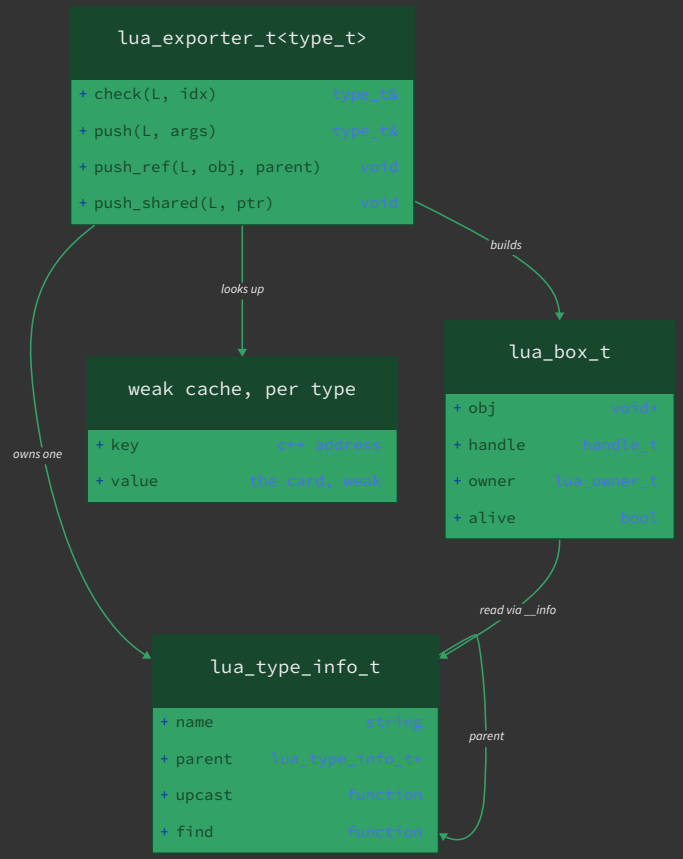
1. `lua_type_info_t` per exporter, a static member, filled from the def in the ctor
2. `setup` puts `&m_info` on the metatable as `__info`
3. `lua_is_a`, walk up from the real type
4. `lua_upcast`, walk up again calling `upcast` at each step
5. `check` uses both, the typeerror uses the **wanted** name, `entity expected, got item`

**tip** `luaL_getmetafield(L, idx, "__info")` pushes nothing and returns `LUA_TNIL` when there is no such field, and pushes the value otherwise, so only pop when it found something

# the flow, a base method on a handle



# the shape of it all



## who owns what, part 3 edition

**lua owns** `item.new`, `enemy.new`, a by value return, `__gc` or `__close` ends it

**borrowed** a plain pointer, only for things that outlive the state, or a member kept to its parent

**handle** the world owns it, every call looks it up, a dead one says `is gone`

**shared** a `shared_ptr` in the box, `__gc` drops lua off the count

**a polymorphic base** never copied, borrowed or shared only

## why its built this way

**one header, not four layouts** every rule lives once, `check` and `__gc` cannot drift apart per owner

**handles over pointers** a crash becomes a catchable error for the price of an index and a compare

**a checked, weak cache** identity without keeping anything alive, and without trusting a reused address

**the parent as a user value** lua already knows how to keep one value alive through another, so we let the gc do it

**an upcast per link** the compiler writes each cast where it knows both types, so multiple inheritance just works

## recap

- a userdata now starts with `lua_box_t`, the owner, the object, a handle, and the alive flag
- **borrowed** is a pointer, fine only for what outlives lua, a **handle** turns a dead object into `is gone`
- **shared** puts a `shared_ptr` in the box, `__gc` drops lua from the count
- one weak table per type keeps **one userdata per object**, and every hit is checked
- a `T&` result goes out borrowed, a member keeps its parent alive through a **user value**
- derived types fall back to the base tables, `check` walks the family with an **upcast per link**, and `virtual` just works
- `create` checks before it allocates now, part 2s missing arg message is fixed

## have a go, the capstone

one `deal_damage(entity_t&, int)`, and everything this part built at once

- a script that spawns two enemies, gets the player, and hits all three through `world.deal_damage`
- store all three in a table **keyed by the entity**, `hp[e] = e.health`, and find them again through a fresh `world.player()`
- kill one enemy, then hit it again inside `pcall`, and print the message, it should be `enemy is gone`
- the survivors keep their health and their inventories through all of it

**tip** run it under `-fsanitize=address` once you think it works, identity and handles both fail quietly, a bad cache hit or a missed parent is the sort of bug only the sanitiser shows you

## part 4, lua things held by c++

every arrow so far pointed one way, c++ objects handed to lua

part 4 turns it round, a lua **function** that c++ keeps and calls later, `events.on("hit", function(target, damage) end)`

which means holding a lua value past the end of a call, the registry and `luaL_ref`, and the same question this part asked, who keeps it alive, and what happens when it is gone