# the lua c api through a gui, part 1

the stack, c functions lua can call, and a first window on screen from a lua script

## what youll have by the end

a c++ file lua can `require`, with a few functions in it, that opens a real window from a lua script and keeps it running until you close it

the slides leave gaps on purpose, every gap has **pseudo** steps and a **tip** pointing at the exact call, so youre filling in, not guessing

## where this sits

this is part 1 of 8, and every part teaches **one idea from the lua c api**

the thing we build along the way is a lua module over finns gui lib, so each idea gets learnt bc the next bit of the gui needs it

the gui is the reason, the c api is the lesson, so the gui only gets as many slides as it needs to make sense

## the big picture

lua is a small program that runs lua scripts

the **c api** is the set of functions that lets **your** c or c++ code and **lua** talk to each other, both ways, your code calls lua, lua calls your code

this whole series is about the second half, **lua calling your code**

## like a serving hatch

two kitchens, yours in c++ and lua next door, with one hatch in the wall between them

you cant reach into their kitchen and grab a pot, and they cant reach into yours, everything goes through the hatch, one plate at a time, stacked up

that stack of plates on the hatch is **the lua stack**, and it is the one idea this whole part rests on

# the map




the script talks to lua, lua talks to our module, our module talks to the gui dll, and nothing ever skips a step

## the gui, in one slide

finns gui lib is a c++ library that draws windows and controls, labels, buttons, lists and so on

we never touch the c++, we go through its **c doorway**, one header, `gui-lib/abi/window_abi.h`, a flat list of plain c functions built into `window.abi.dll`

it was made for a language with no compiler at all, so every function is dead simple to call from anywhere, which is exactly why it is nice to wrap in lua

# section 1, the stack

## lua_State, the whole world in one pointer

```
lua_State *L;
```

every single api call takes this as its first argument

it holds the globals, the loaded libraries, the garbage collector and the stack, all of it

you never look inside it, its opaque on purpose, meaning you only ever hold a pointer to it and pass that along

by convention its always called `L`, everyone does it, and finn-style lets it stay `L` too

## the stack, the one idea

a stack of values, numbered from the bottom up, 1 2 3 and so on

when lua calls your function it clears the hatch and puts the arguments on it, so **the arguments are just the stack**, nothing more

when youre done you push your results on top and tell lua how many you pushed

## stack indices, counting up from the bottom

say a script calls `gui.window("hello", 400, 300)`

```
index 3 ->  300         <- top
index 2 ->  400
index 1 ->  "hello"     <- bottom
```

so `1` is the first argument, `2` the second, and so on, positive indices count **up from the bottom**

## stack indices, counting down from the top

the same three values, addressed the other way

```
index -1 ->  300        <- top
index -2 ->  400
index -3 ->  "hello"    <- bottom
```

negative indices count **down from the top**, so `-1` always means whatever is on top right now

rule of thumb, positive for arguments bc their spot never moves, `-1` for the thing you just pushed bc you know its on top

## the three stack chores

**lua_gettop(L)** how many values are on the stack right now, and since the arguments are the stack, that is also how many arguments you got

**lua_settop(L, n)** make the stack exactly n tall, chopping values off or padding with nil

**lua_pop(L, n)** throw away the top n values

## pushing values on

one function per lua type, each one grows the stack by exactly one

```
lua_pushnil(L);
lua_pushboolean(L, 1);
lua_pushinteger(L, 42);
lua_pushnumber(L, 3.5);
lua_pushstring(L, "hello");
```

`lua_Integer` is a 64 bit int and `lua_Number` is a double, those are the only two number types lua has

## a note on strings

lua **copies** a string when you push it, it never keeps your pointer

```c
char name[32];
snprintf(name, sizeof name, "window %d", 3);
lua_pushstring(L, name);   // copied, name can die now
```

that matters bc lua owns its own memory and cleans it up itself, so it cant trust a pointer you might free later

## reading values off, two families

**lua_to*** the raw read, hands back a value or a zero, never complains

```
lua_Integer n = lua_tointeger(L, 1);   // 0 if it was not a number
```

**luaL_check*** the checked read, hands back the value or **stops with a lua error** that says what went wrong

```
lua_Integer n = luaL_checkinteger(L, 1);
```

## always reach for luaL_check

the `luaL_` prefix means the **auxiliary library**, `lauxlib.h`, a layer of handy helpers built on top of the raw api

use the checked family for arguments, every time, a script passing the wrong thing should get a clear error, not a silent zero

```
luaL_checkinteger(L, 1);
luaL_checknumber(L, 1);
luaL_checkstring(L, 1);              // const char *
luaL_checktype(L, 1, LUA_TTABLE);
```

part 2 is all about errors, for now just know these stop the function and tell the script why

## still with me ?

so far

- `L` is lua, you pass it to everything
- the stack is the hatch, arguments come in on it, results go out on it
- push to put things on, `luaL_check*` to read arguments off

thats the whole machine, everything from here is using it

## section 2, a c function lua can call

## the shape of every c function lua can call

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

one argument, the state, and it returns an int

no matter how many arguments the lua side passes, your function always looks like this, the arguments are on the stack, not in the signature

## the return value

the int you return is **how many results you pushed**

```
return 0;   // nothing back, the script gets nil if it asks
return 1;   // the value on top of the stack
return 2;   // the top two, first pushed is first returned
```

lua takes that many off the top of the stack and hands them to the script, anything left under them gets thrown away

## a whole function, worked

```
//
//      hands back which abi the loaded dll speaks
//
int l_version(lua_State *L)
{
        lua_pushinteger(L, gui_abi_version());
        return 1;
}
```

`gui_abi_version` is our first call into the dll, it answers with a plain int and it can never fail

push it, say one result, done

## the rhythm

every function in this series has the same beat

1. **read** the arguments off the stack
2. **work**, call the dll
3. **push** the results
4. **return** how many

`l_version` has no arguments so it skips step 1, the next one has one

## the l_ and the namespace

```
namespace
{
        int l_version(lua_State *L)
        {
                ...
        }
}
```

the `l_` prefix is the usual lua habit, it marks a function lua calls, so you can tell them from your own helpers at a glance

the **anonymous namespace**, a `namespace` with no name, means nothing outside this file can see what is inside it. finn-style uses this rather than `static` for file only things, lua still finds them through a table we build in a minute, not through the linker

## exercise 1, version

write `l_version`, takes nothing, hands back the abi version

```
int l_version(lua_State *L)
{
        // todo
}
```

### pseudo

1. ask the dll which version it is
2. push that as an integer
3. say you left one result

**tip** `gui_abi_version()` returns an `int32_t`, and `lua_pushinteger` takes a `lua_Integer`, the conversion just happens, no cast needed

## a function with an argument

```
//
//      sleeps for the given number of milliseconds
//
int l_sleep(lua_State *L)
{
        lua_Integer ms = luaL_checkinteger(L, 1);

        std::this_thread::sleep_for(std::chrono::milliseconds(ms));
        return 0;
}
```

read argument 1 as an integer, sleep that long, hand nothing back

`return 0` means no results, if the script writes `local x = gui.sleep(5)` it just gets nil

## why a gui wants sleep

nothing to do with the c api, a quick aside

the window loop at the end of this part runs round and round asking the window if anything happened, and without a rest it spins a cpu core flat out doing it

a 10 ms nap each lap is plenty for a small tool, and lua has no sleep built in, so we add one

# exercise 2, sleep

write `l_sleep`

```
int l_sleep(lua_State *L)
{
        // todo
}
```

## pseudo

1. read argument 1 as an integer
2. sleep that many milliseconds
3. say you left nothing

**tip** `std::this_thread::sleep_for` lives in `<thread>` and `std::chrono::milliseconds` in `<chrono>`, include both at the top

## section 3, a module lua can require

# luaL_Reg, the table of names

lua needs to know what to call your functions, so you hand it a list of name and function pairs

```
// what lua sees, name on the left, our function on the right
const luaL_Reg m_funcs[] =
{
        { "version",    l_version       },
        { "sleep",              l_sleep         },
        { nullptr,              nullptr         }
};
```

the left side is the name the script uses, `gui.version`, the right side is your function, they dont have to match

that `{ nullptr, nullptr }` row is how lua knows where the list ends, leave it off and it walks off the end of the array

## the finn-style bits on that slide

**m_funcs** it lives inside the anonymous namespace, and finn-style gives a namespace var that is internal to the file an `m_`, same as a class member

**the aligned columns** the names and functions are tabbed into columns so the list reads like a table, finn-style lines up anything that reads better lined up

**the array brace on its own line** an array literal opens on a new line with one entry per line

## the entry point

one function per module, named `luaopen_` plus the module name

```
//
//      builds the gui table, lua calls this once on the first require
//
extern "C" int luaopen_gui(lua_State *L)
{
        luaL_newlib(L, m_funcs);
        return 1;
}
```

lua calls it once, the first time the module is loaded, your job is to build the module table and leave it on the stack

it sits **outside** the anonymous namespace, bc the host program has to be able to see it

## what luaL_newlib actually does

its a macro, it expands to three things

```
#define luaL_newlib(L, l) \
        (luaL_checkversion(L), luaL_newlibtable(L, l), luaL_setfuncs(L, l, 0))
```

1. **checkversion** makes sure you compiled against the same lua youre running in
2. **newlibtable** makes a fresh empty table, sized for your list
3. **setfuncs** walks your `luaL_Reg` list and sets each name on that table

so afterwards there is one new table on top of the stack with all your functions in it, which is exactly what we return

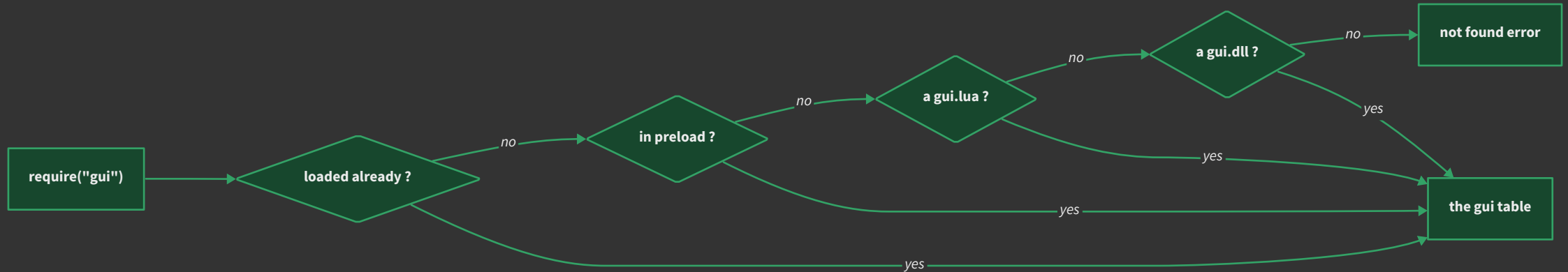
## extern "C", why

c++ quietly renames every function it compiles, it bakes the argument types into the name so overloads can share one, thats called **name mangling**

lua is written in c and knows nothing about it

`extern "C"` turns the renaming off for that one function, so its real name is plain `luaopen_gui`, which is the name anything outside c++ would look for

# how require finds a module



## the order, in words

**package.loaded** a module only ever loads once, the second `require` hands back the same table

**package.preload** a table of loader functions the host program filled in by hand, checked first of all

**package.path** looks for a `gui.lua` file

**package.cpath** looks for a `gui.dll` and calls the `luaopen_gui` inside it

## two ways to ship a c module

**a host exe** your own small program that runs lua, with our module compiled into it and put in `package.preload`

**a loadable dll** a `gui.dll` that the stock `lua.exe` finds on `package.cpath`

the dll road needs lua itself built as a dll, so the module and the exe share one copy of lua. finns lua is a static lib, so this series takes the **host exe** road

# the host, part one

```
// the modules entry point, it lives in gui.cpp
extern "C" int luaopen_gui(lua_State *L);

int main(int _argc, char **_argv)
{
        const char *path = _argc > 1 ? _argv[1] : "main.lua";

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

**luaL_newstate** makes a fresh lua, empty, not even `print` yet

**luaL_openlibs** loads the standard libraries into it, `print`, `string`, `table`, `package`, the lot

the `_` on `_argc` and `_argv` is finn-style for a function argument

# the host, part two

```
        // package.preload.gui = luaopen_gui, so require finds it
        lua_getglobal(L, "package");
        lua_getfield(L, -1, "preload");
        lua_pushcfunction(L, luaopen_gui);
        lua_setfield(L, -2, "gui");
        lua_pop(L, 2);
```

step by step, with the stack after each line

```
getglobal     package
getfield      package  preload
pushcfunction package  preload  luaopen_gui
setfield      package  preload            <- preload.gui = the function, popped
pop 2         (empty)
```

# the host, part three

```
        int status = luaL_dofile(L, path);

        if (status != LUA_OK)
                std::fprintf(stderr, "%s\n", lua_tostring(L, -1));

        lua_close(L);

        return status == LUA_OK ? 0 : 1;
}
```

**luaL_dofile** loads the script and runs it, `LUA_OK` if it went fine

if it didnt, the error message is sitting on top of the stack, so we print it

**lua_close** tears the whole lua down and frees everything in it

## building it, msvc

from the repo root, in a vs2022 developer prompt

```
cl /nologo /std:c++20 /EHsc ^
        /I lua-5.5.1\headers /I gui-lib\abi ^
        lualib\source\host.cpp lualib\source\gui.cpp ^
        <your lua>.lib gui-lib\bin\x64_Debug\window.abi.lib ^
        /Fe:gui-host.exe
```

swap `<your lua>.lib` for wherever your static lua build lands, then copy `window.abi.dll` next to `gui-host.exe` so windows can find it when it starts

`window.abi.lib` is the **import lib**, a small stub that tells the linker the functions live in the dll

## a word on how these slides were checked

every snippet here was compiled, and every output shown was pasted from a real run

that run used a **stand in** for `window.abi.dll`, a small file with the same functions, the same error messages and no window, bc the real dll is windows only. it lives at `lualib/tests/fake_abi.cpp`

a window actually showing up is the one thing it cant prove, that part you see on your own pc

## exercise 3, the module

write `m_funcs` and `luaopen_gui`, then the host, then a `main.lua` of

```
local gui = require("gui")
print("abi version", gui.version())
```

### pseudo

1. list `version` and `sleep` in a `luaL_Reg` array, end it with a null row
2. `luaopen_gui` builds the table from it and returns it
3. the host makes a state, opens the libs, sets `package.preload.gui`, runs the file

**tip** you should see `abi version  6`, if require says the module was not found, check the preload lines, the field name has to be exactly `"gui"`

# section 4, talking to the dll

## the abi, in one slide

every function in `window_abi.h` has the same shape

```
int32_t gui_desc_create(gui_desc_t **_out);
```

**it returns a status**, an `int32_t`, `gui_ok` (0) when it went fine and something else when it didnt

**the real answer comes back through an out pointer**, you pass the address of your variable and the dll writes into it

## out pointers, in plain c

```
gui_desc_t *desc = nullptr;
int32_t status = gui_desc_create(&desc);
```

`&desc` is the address of our variable, so the dll can reach in and write the new desc there

afterwards `status` says whether it worked and `desc` holds the answer

one shape for the whole dll, which is lovely, you always check the same thing

## handles

`gui_desc_t *`, `gui_host_t *`, `gui_control_t *`, these are **handles**, pointers to things that live inside the dll

theyre opaque, the dll never tells you whats inside, you only hold them and hand them back

a handle is the dll saying, here, hold this ticket, show it to me next time you want this thing

## a handle in lua

a script needs to hold a handle too, `local win = gui.window(...)` has to give it something

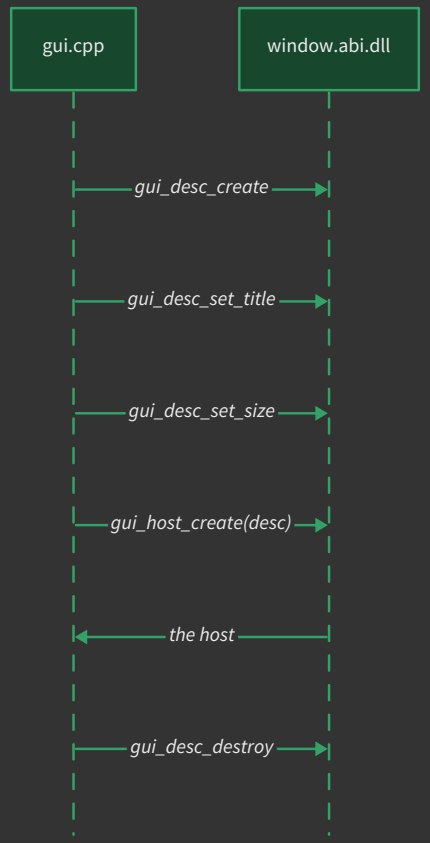
the simplest thing lua has for that is **light userdata**, a bare pointer lua can hold and pass around but never look inside

```
lua_pushlightuserdata(L, host);                    // pointer in
gui_host_t *host = (gui_host_t *)lua_touserdata(L, 1);   // pointer out
```

`lua_touserdata` hands back a `void *`, so we cast it back to what we know it is, a plain c style cast, the house habit for api pointers

## section 5, a window

# what a window takes




the **desc** is a form you fill in, what to build, the **host** is the window it builds

## the desc lives for one call

a desc is only read by `gui_host_create`, once the window exists nobody needs it

so the script never sees it, `gui.window` makes it, fills it, uses it and throws it away, all inside the one c function

one less thing the script has to hold, and one less handle that could go missing

# l_window, the arguments

```
int l_window(lua_State *L)
{
        const char *title       = luaL_checkstring(L, 1);
        lua_Integer width       = luaL_checkinteger(L, 2);
        lua_Integer height      = luaL_checkinteger(L, 3);
```

step 1 of the rhythm, read, `gui.window("hello", 400, 300)` puts those three on the stack in that order

the `=` signs are lined up bc finn-style aligns a run of assignments

# l_window, the desc

```
        // the desc only lives for this call
        gui_desc_t *desc = nullptr;
        gui_desc_create(&desc);
        gui_desc_set_title(desc, title);
        gui_desc_set_size(desc, (int32_t)width, (int32_t)height);
```

make it, then fill it in, a setter per field

`title` is a `const char *` pointing into lua, the dll only borrows it for the length of the call, same as lua borrowing ours, so this is safe

yes, we ignore three statuses here, hold that thought till the end

# l_window, the host

```
        gui_host_t *host = nullptr;
        int32_t status = gui_host_create(desc, &host);
        gui_desc_destroy(desc);

        if (status != gui_ok)
                return luaL_error(L, "%s", gui_err_text(status));

        lua_pushlightuserdata(L, host);
        return 1;
}
```

build the window, bin the desc, then if the build failed stop with an error, otherwise push the host and return it

# luaL_error, a first look

```
return luaL_error(L, "%s", gui_err_text(status));
```

**gui_err_text** the dll turns any status into a short english message, `"bad handle"` and so on

**luaL_error** stops the function right there and hands the script an error with that message, printf style formatting

it never actually returns, the `return` in front is a habit that makes that obvious, part 2 is all about why

# exercise 4, window

write `l_window` and add it to `m_funcs`

```
int l_window(lua_State *L)
{
        // todo
}
```

## pseudo

1. read the title, width and height
2. make a desc, set the title and the size
3. make the host from it, destroy the desc
4. if the status was not ok, error with its text
5. push the host as light userdata, return 1

**tip** destroy the desc **before** the error check, so it goes whichever way the build went

# a label

```
int l_label(lua_State *L)
{
        const char *text = luaL_checkstring(L, 1);

        gui_control_t *label = nullptr;
        int32_t status = gui_label_create(text, &label);

        if (status != gui_ok)
                return luaL_error(L, "%s", gui_err_text(status));

        lua_pushlightuserdata(L, label);
        return 1;
}
```

the same rhythm again, read, call, check, push, return one

# set_root, two handles in

```
int l_set_root(lua_State *L)
{
        gui_host_t *host        = (gui_host_t *)lua_touserdata(L, 1);
        gui_control_t *root     = (gui_control_t *)lua_touserdata(L, 2);

        int32_t status = gui_host_set_root(host, root);

        if (status != gui_ok)
                return luaL_error(L, "%s", gui_err_text(status));

        return 0;
}
```

the **root** is the one control a window draws, everything else will hang off it later

the window only **borrows** it, the label stays the scripts, part 5 is the whole story on who owns what

## show

```
int l_show(lua_State *L)
{
        gui_host_t *host = (gui_host_t *)lua_touserdata(L, 1);

        int32_t status = gui_host_show(host, 1);

        if (status != gui_ok)
                return luaL_error(L, "%s", gui_err_text(status));

        return 0;
}
```

the `1` is a bool, the dll takes bools as `int32_t`, 0 is false and anything else is true

## exercise 5, label, set_root and show

write `l_label`, `l_set_root` and `l_show`, add all three to `m_funcs`

### pseudo

1. label, read the text, create it, check, push the handle
2. set_root, read two handles, set the root, check, return nothing
3. show, read one handle, show it, check, return nothing

**tip** youve now typed the same `if (status != gui_ok)` three times, notice that, part 2 turns it into one line

## section 6, the loop

# pump

```
int l_pump(lua_State *L)
{
        gui_host_t *host = (gui_host_t *)lua_touserdata(L, 1);

        int32_t alive = 0;
        int32_t status = gui_host_pump(host, &alive);

        if (status != gui_ok)
                return luaL_error(L, "%s", gui_err_text(status));

        lua_pushboolean(L, alive);
        return 1;
}
```

## what pump does

`gui_host_pump` deals with whatever the window has waiting, clicks, redraws, moves, then hands straight back, it **never waits**

`alive` is the out pointer, it comes back 0 once the window has closed

`lua_pushboolean` turns it into a real lua `true` or `false`, so the script can loop on it

## the loop, in lua

```
while gui.pump(win) do
        gui.sleep(10)
end
```

pump, nap, pump, nap, until the user closes the window and pump says false

the script owns the loop, which means the script decides when to stop, when to check on things, when to do its own work between laps

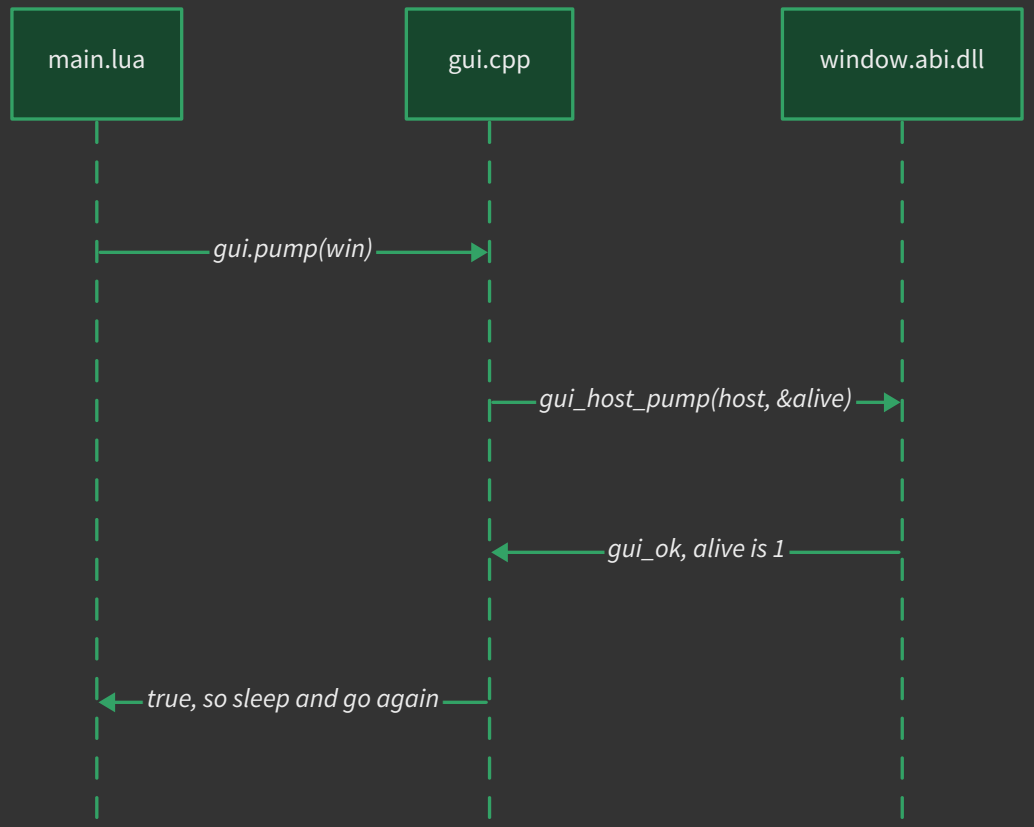
## why pump and not run

the dll also has `gui_host_run`, which loops for you until the window closes

sounds handier, but it **never hands back** till the end, so nothing in the script runs while the window is up, no timers, no checks, nothing

pump hands back every lap, so lua stays in charge, which is what every later part leans on

# how it flows




one lap, and on the lap after the window closes `alive` comes back 0, pump says `false` and the loop ends

# exercise 6, pump and the loop

write `l_pump`, add it, then finish `main.lua`

```
local win = gui.window("hello", 400, 300)
gui.set_root(win, gui.label("hello from lua"))
gui.show(win)

-- todo, the loop
print("closed, bye")
```

## pseudo

1. pump reads the host, pumps it, checks, pushes alive as a boolean
2. the loop pumps until false, sleeping 10 ms a lap

**tip** `alive` is an `int32_t`, and `lua_pushboolean` takes an int, so it goes straight in

## the first run

```
abi version     6
closed, bye
```

on your pc a 400 by 300 window pops up between those two lines with **hello from lua** in it, close it and the script ends

thats a window from a lua script, through our own c module, in under 150 lines, comments and all

## section 7, what light userdata cant do

## it has no type

```
local gui = require("gui")
local label = gui.label("hi")
gui.show(label)
```

```
notype.lua:3: bad handle
```

a label is not a window, and lua had no way to know, light userdata is just a pointer, every pointer looks the same

it was the **dll** that noticed, it keeps a list of every handle it gave out and what kind it is. lua caught nothing

## it can be anything

```
local gui = require("gui")
gui.show(42)
```

```
anything.lua:2: bad handle
```

`lua_touserdata` on a number hands back a null pointer, and the dll refuses that too

we are leaning on the dll to catch our mistakes, a careful dll, but still, lua should be checking what it was given

## nothing frees it

the stand in dll counts whatever is still alive when the program ends, after the first run it says

```
still alive at exit, 0 desc, 1 host, 1 control
```

the window and the label were never destroyed, lua has no idea a light userdata points at something that needs freeing, so it never asks

the desc is gone though, it only ever lived inside `l_window`

## a silent mistake

```
local win = gui.window("hello", -5, 300)
print("made it", win)
```

```
made it userdata: 0x5654b610c940
```

no error at all, the window comes up at the default 480 by 320

`gui_desc_set_size` said no to a width of -5, and we threw that status away. every call that can fail needs checking, part 2 is exactly that

# section 8, wrapping up

## the new calls

| call | what it does |
|---|---|
| `lua_gettop` `lua_settop` `lua_pop` | read or change how tall the stack is |
| `lua_push*` | put a value on the stack |
| `luaL_check*` | read an argument, or error |
| `luaL_Reg` `luaL_newlib` | a module table from a list of functions |

## the new calls, the host and the handles

| call | what it does |
|---|---|
| `luaL_newstate` `luaL_openlibs` `lua_close` | make, fill and end a lua |
| `lua_getglobal` `lua_getfield` `lua_setfield` | read and write table fields through the stack |
| `luaL_dofile` | load and run a script |
| `lua_pushlightuserdata` `lua_touserdata` | a bare pointer in and out |
| `luaL_error` | stop with an error |

## why its built this way

- **the desc never reaches lua**, it only lives inside `gui.window`, so the script has less to hold
- **the script owns the loop**, pump over run, so lua stays in charge
- **a host exe over a loadable dll**, one copy of lua and nothing to get wrong with a static lua
- **light userdata for now**, the smallest thing that works, and its holes are what part 4 fixes
- **finn-style**, the anonymous namespace over `static`, `m_` on a file internal var, `_` on arguments, aligned `=`, and the three line header above every function

## recap

- the stack is the hatch, arguments in, results out, and the int you return says how many
- every c function lua calls has the same shape and the same rhythm, read, work, push, return
- a module is a `luaL_Reg` list turned into a table by `luaopen_gui`
- `require` checks `package.preload` first, which is how our host hands lua the module
- the dll answers with a status and an out pointer, every call, every time
- light userdata holds a handle, but has no type and never gets freed

## have a go

add two more functions

```
local w, h = gui.size(win)   -- 400  300
gui.close(win)               -- the loop ends on the next pump
```

### pseudo

1. size reads the host, calls `gui_host_logical` with two out pointers, pushes both, returns 2
2. close reads the host, calls `gui_host_close`, returns 0

**tip** a function can return more than one thing, push both, `return 2`, and lua hands them back in the order you pushed them

## whats next

part 2, **errors**

every one of those calls can fail, half of them we never checked, and the other half repeat the same four lines

next time we find out what `luaL_error` really does, how an error jumps across from c into lua, and what that jump costs a function halfway through its work