# the lua c api through a gui, part 2

errors, how one crosses from c into lua, and what it costs the function it leaves behind

## what youll have by the end

every call in the gui module turning a failure into a clear lua error, buttons that take a style by name, a module that refuses the wrong dll at `require`, and a test script that checks the failures on purpose

same deal as part 1, gaps on purpose, each with **pseudo** steps and a **tip**

## where we left off

part 1 got a window on screen through our own module, and left two problems lying around

**a silent mistake** `gui.window("hello", -5, 300)` made a window anyway, bc we threw away the status that said no

**the same four lines** everywhere, `if (status != gui_ok) return luaL_error(...)`, pasted into every function

both of those are about errors, so thats this whole part

## the big picture

an error is lua **giving up on the current job** and jumping straight back to whoever said they would deal with it

nothing between the error and the catcher gets to finish, it is skipped, all of it

thats the one fact this part keeps coming back to

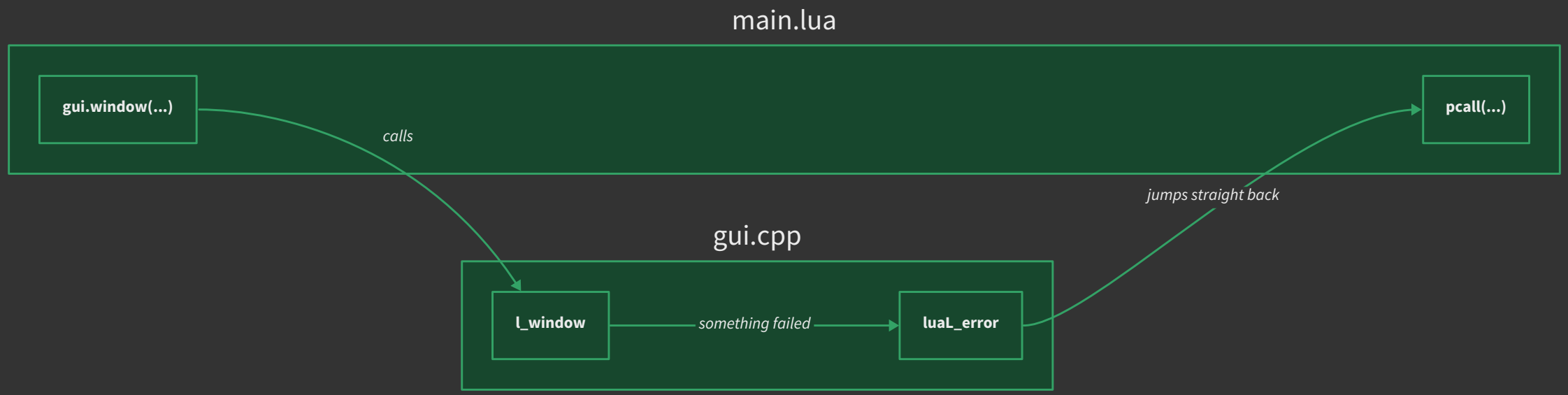
## back at the hatch

the hatch from part 1, but this time a plate comes back with a note on it, **no**

the note has to say what went wrong in words the other kitchen understands

and before it goes out, the kitchen has to put away anything it was halfway through, bc once the note is through the hatch, nobody is coming back to finish

# the map


main.lua
gui.window(...)
calls
pcall(...)
jumps straight back
gui.cpp
l_window
something failed
luaL_error

## section 1, errors from the lua side

## error, raising one

```
error("boom")
```

`error` stops the running function right there, and the one that called it, and so on up, until something catches it

the thing it carries up is the **error value**, here the string `"boom"`

if nothing catches it, the host gets it back from `luaL_dofile` and prints it, which is what our host from part 1 does

# pcall, catching one

```lua
print(pcall(function() error("boom") end))
print(pcall(function() return 1 + 2 end))
```

```
false   catch.lua:1: boom
true    3
```

`pcall` is **protected call**, it runs a function and catches any error inside it

it hands back `false` and the error value when it failed, or `true` and the functions results when it didnt

## where the position comes from

that `catch.lua:1:` in front was added by `error`, it points at the line that raised

```
local function check_age(age)
        if type(age) ~= "number" then error("age must be a number", 2) end
end
print(pcall(function() check_age("old") end))
```

```
false   level.lua:4: age must be a number
```

the `2` is the **level**, 1 means blame this function, 2 means blame whoever called it, so the message points at the line with the mistake, not at the check

## an error can be anything

```lua
local ok, err = pcall(function() error({ code = 42 }) end)
print(ok, err.code)
```

```
false   42
```

the error value doesnt have to be a string, a table works fine and can carry as much as you like

strings are the norm tho, and everything in this series raises strings

## lua errors are the same thing

```
print(pcall(function() local t = nil; return t.x end))
```

```
false   nil.lua:1: attempt to index a nil value (local 't')
```

when lua itself hits a mistake it raises exactly the same kind of error, same value, same jump, same catch

so from the scripts side, an error from our c module is no different to any other

## exercise 1, catch one

write a script that tries to make a window with a width of `"wide"`, catches the error and prints it without the script dying

```
local gui = require("gui")
-- todo
print("still running")
```

### pseudo

1. wrap the bad `gui.window` call in a function
2. hand that function to `pcall`
3. print what pcall hands back

**tip** you should see `false` then `bad argument #2 to 'window' (number expected, got string)`, that message came from `luaL_checkinteger` in part 1 without us writing a word of it

## section 2, raising from c

## luaL_error, the c side of error

```
return luaL_error(L, "the window has closed");
```

this is `error` from c, it builds the message, printf style, adds the position the same way `error` does, and raises it

the position uses level 1, which from c means **the lua line that called us**, so the script sees the line of its own mistake

# it never returns

```
int l_version(lua_State *L)
{
        if (lua_gettop(L) > 0)
        {
                luaL_error(L, "version takes no arguments");
                printf("after the error\n");
        }
        ...
```

```
print(pcall(gui.version, 1))
```

```
false   version takes no arguments
```

the `printf` never prints, not once, `luaL_error` doesnt come back to finish the function

## then why return luaL_error

```
return luaL_error(L, "...");
```

`luaL_error` is declared to return an `int`, so it can sit behind a `return` and keep the compiler happy about a function that has to return one

it never actually returns anything, the `return` is there for **you**, it reads as this is the end of the function, which is true

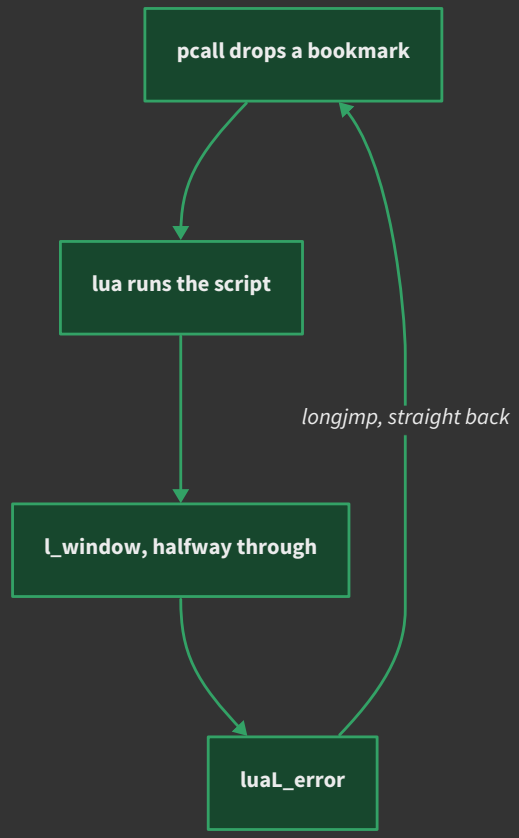
## how it jumps

a plain c lua doesnt use c++ exceptions, it uses `setjmp` and `longjmp`

**setjmp** a bookmark, `pcall` drops one before it runs your function

**longjmp** go straight back to that bookmark, from anywhere, every function in between is abandoned where it stands

# what gets skipped




`l_window` never finishes, whatever it was holding, whatever it meant to free next, all of it gets left

## destructors dont save you

in c++ you might think, fine, a destructor will clean up on the way out

in a plain c lua, `longjmp` skips them, on msvc whether they run depends on the `/EH` flag you built with

so the rule is simple, **never count on a destructor running when lua raises**, either way, clean up yourself before you raise, section 6 is exactly that

## section 3, one way to fail

## raise, or hand back nil

a lua library can report a failure two ways

**raise** `string.rep()` with no arguments, an error, the script stops unless it used `pcall`

**nil, msg** `io.open("missing.txt")` hands back `nil` and a message, the script carries on and checks

the rule of thumb, raise for **mistakes**, return nil for things that **just happen**, like a file not being there

## which for us

every status the dll hands back that isnt `gui_ok` is a mistake, a bad argument, a handle that isnt what the call wanted, or the dll itself in a bad way

the one thing a script should expect to happen, the user closing the window, isnt an error at all, `pump` already says it with `false`

so there is nothing left that nil would be for, **we raise, always**, and a script that wants to survive one uses `pcall`

## check

```
//
//      raises a lua error for the given status, does nothing when it is ok
//
void check(lua_State *L, int32_t _status)
{
        if (_status != gui_ok)
                luaL_error(L, "gui: %s", gui_err_text(_status));
}
```

the four lines from every function, said once

`gui_err_text` is the dll naming its own statuses, so we never keep a list of messages ourselves

the `gui:` in front tells a dll error apart from an argument error at a glance

## check in use

before, from part 1

```
int32_t status = gui_label_create(text, &label);

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

after

```
check(L, gui_label_create(text, &label));
```

one line, every call, no status left unchecked by accident

## what it looks like from lua

```
local gui = require("gui")
gui.show(gui.label("not a window"))
```

```
show.lua:2: gui: bad handle
```

the script line, the `gui:` so you know the dll said it, and the dlls own words

## exercise 2, check

write `check`, then put every dll call in the module through it, `l_label`, `l_set_root`, `l_show`, `l_pump`, plus the two from the part 1 have a go, `l_size` and `l_close`

```
void check(lua_State *L, int32_t _status)
{
        // todo
}
```

### pseudo

1. if the status is not `gui_ok`, raise with `gui_err_text` of it
2. swap each `status` and `if` pair for one `check(L, ...)` line

**tip** leave `l_window` alone for now, it has a trap in it that section 6 is about

## section 4, argument errors

## the checked reads, again

```
local gui = require("gui")
gui.window("test", "wide", 300)
```

```
wide.lua:2: bad argument #2 to 'window' (number expected, got string)
```

**wide.lua:2** the line that called us

**#2** which argument, counting from 1

**`'window'`** the name the script called

**(number expected, got string)** what was wrong with it

## luaL_argcheck, your own rule

a width can be a perfectly good integer and still be wrong

```
luaL_argcheck(L, width > 0, 2, "width must be above 0");
```

```
width.lua:2: bad argument #2 to 'window' (width must be above 0)
```

if the condition is false it raises, in the same shape as the checked reads, with your words in the brackets

## luaL_argerror, under it

```
if (!(width > 0))
        return luaL_argerror(L, 2, "width must be above 0");
```

thats all `luaL_argcheck` is, a condition wrapped around `luaL_argerror`

reach for `argerror` yourself when the check doesnt fit on one line

## optional arguments

```
lua_Integer ms = luaL_optinteger(L, 1, 10);
```

the `luaL_opt*` family reads an argument the same way the check family does, but when its **missing or nil** it hands back the default instead

so `gui.sleep()` naps 10 ms, and `gui.sleep(50)` naps 50, and `gui.sleep("x")` is still an error

## exercise 3, arguments

give `l_window` its two size checks and `l_sleep` a default and a check

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

### pseudo

1. sleep reads its argument with a default of 10
2. it refuses anything below 0
3. window refuses a width or height of 0 or less, before anything else happens

**tip** put the `luaL_argcheck` lines straight after the reads, right at the top, youll see why in section 6

# section 5, names not numbers

# a button has a style

```
typedef enum gui_btn_e
{
        gui_btn_primary = 0,
        gui_btn_secondary,
        gui_btn_success,
        gui_btn_warning,
        gui_btn_danger,
        gui_btn_text,
        ...
} gui_btn_t;
```

`gui_button_create(text, style, &out)` wants one of these, and to c its just an `int32_t`

so the easy version is `gui.button("ok", 0)`

## why not the number

**nobody can read it** `gui.button("ok", 4)` says nothing, `gui.button("ok", "danger")` says everything

**the numbers move** the abi is still young and its numbers are not frozen yet, finn has already slotted `warning` into the middle of this list once, which moved `danger` from 3 to 4

a script that used the name never noticed, a script that used the number would have turned every danger button into a warning one

# luaL_checkoption

```
// the button styles a script can ask for, by name, nullptr ends the list
const char *const m_style_names[] = { "primary", "secondary", "success", "warning", "danger", "text", nullptr };

int style = luaL_checkoption(L, 2, "secondary", m_style_names);
```

it reads argument 2 as a string, finds it in the list, and hands back **where** it found it, 0 for `"primary"`, 1 for `"secondary"`

the `"secondary"` is the default when the argument is missing, pass `nullptr` there instead to make it required

## the index is not the value

`luaL_checkoption` gives us a spot in **our** list, not a number from the dll

so we keep a second list beside it, in the same order, holding the real named values

```
// the abi value for each name above, in the same order
const int32_t m_style_values[] = { gui_btn_primary, gui_btn_secondary, gui_btn_success, gui_btn_warning, gui_btn_danger, gui_btn_text };

check(L, gui_button_create(text, m_style_values[style], &button));
```

the actual numbers only ever live in the compiler, if finn moves them, a rebuild picks it up and the script never knows

## what a bad name looks like

```
gui.button("ok", "big")
```

```
names.lua:2: bad argument #2 to 'button' (invalid option 'big')
```

```
gui.button("ok", 3)
```

```
number.lua:2: bad argument #2 to 'button' (invalid option '3')
```

and the number is no way round it either, lua turns `3` into the string `"3"`, which isnt on the list

# exercise 4, button

write `l_button`, text first, then a style name that defaults to `"secondary"`

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

## pseudo

1. read the text
2. read the style with `luaL_checkoption` against the names list
3. look the real value up in the values list
4. create it through `check`, push it, return 1

**tip** the two lists have to stay in the same order, keep them right next to each other so nobody can edit one without seeing the other

## section 6, tidy up before you raise

## the trap in l_window

with `check` everywhere, `l_window` would look like this

```
gui_desc_t *desc = nullptr;
check(L, gui_desc_create(&desc));
check(L, gui_desc_set_title(desc, title));
check(L, gui_desc_set_size(desc, (int32_t)width, (int32_t)height));

gui_host_t *host = nullptr;
check(L, gui_host_create(desc, &host));
gui_desc_destroy(desc);
```

tidy, and wrong, can you see it ?

## run it with a bad width

```
local gui = require("gui")
gui.window("test", -5, 300)
```

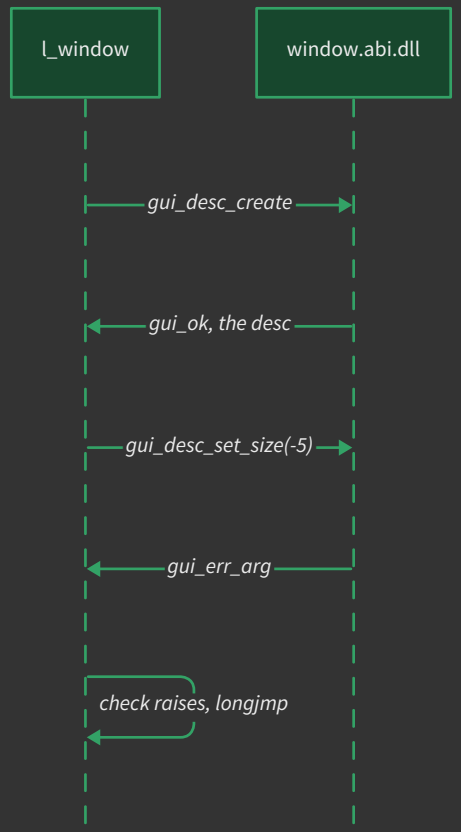
```
width.lua:2: gui: bad argument
```

the stand in dll counts what is still alive at the end

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

the error was right, but the desc never got destroyed

# why it leaked



## why it leaked, in words

`check` raised in the middle, the jump skipped every line after it, and `gui_desc_destroy` was one of them

the desc is ours from the moment `gui_desc_create` hands it over, and nothing else will ever free it

## fix one, check first

most mistakes can be caught before anything exists

```
lua_Integer width       = luaL_checkinteger(L, 2);
lua_Integer height      = luaL_checkinteger(L, 3);

// check everything we can before anything gets made
luaL_argcheck(L, width > 0, 2, "width must be above 0");
luaL_argcheck(L, height > 0, 3, "height must be above 0");
```

if these raise, there is nothing to leak, nothing has been made yet

the rule, **check, then allocate**

# fix two, tidy then raise

some failures you cant see coming, the dll might still say no, so between making the desc and destroying it, **nothing raises**

```
// the desc is ours now, so nothing raises until it is gone
int32_t status = gui_desc_set_title(desc, title);

if (status == gui_ok)
        status = gui_desc_set_size(desc, (int32_t)width, (int32_t)height);

gui_host_t *host = nullptr;

if (status == gui_ok)
        status = gui_host_create(desc, &host);

gui_desc_destroy(desc);
check(L, status);
```

keep the status, skip the rest once one fails, tidy up, and only then raise

# l_window, finished

```
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);

        // check everything we can before anything gets made
        luaL_argcheck(L, width > 0, 2, "width must be above 0");
        luaL_argcheck(L, height > 0, 3, "height must be above 0");

        gui_desc_t *desc = nullptr;
        check(L, gui_desc_create(&desc));

        // ... the tidy then raise block from the last slide

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

## exercise 5, the tidy window

rewrite `l_window` so it can never leak the desc, whatever fails

**pseudo**

1. read and check every argument first
2. make the desc through `check`, thats fine, if that one fails there is no desc yet to leak
3. from there, keep one status, and only run the next call while it is still ok
4. destroy the desc
5. now `check` the status, then push the host

**tip** a quick way to prove it, point it at `-5` with the argument checks commented out, the stand in should report 0 desc alive at exit

# section 7, failing early

## the wrong dll

our module was built against `window_abi.h` at abi version 6, `GUI_ABI_VERSION`

if an older or newer `window.abi.dll` sits next to the exe, every call might mean something slightly different, a moved number, a changed rule

better to find out at `require` than halfway through a script

# the version check

```
extern "C" int luaopen_gui(lua_State *L)
{
        // a dll from another abi would read every call wrong, so refuse it up front
        if (gui_abi_version() != GUI_ABI_VERSION)
                return luaL_error(L, "gui: the dll speaks abi %d, this module was built for %d", gui_abi_version(), GUI_ABI_VERSION);

        luaL_newlib(L, m_funcs);
        return 1;
}
```

```
gui: the dll speaks abi 5, this module was built for 6
```

`luaopen_gui` is a c function like any other, so it can raise like any other, and `require` hands the error straight to the script

## a real no from the dll

```
local gui = require("gui")
local win = gui.window("test", 400, 300)
gui.close(win)
while gui.pump(win) do gui.sleep() end
gui.show(win)
```

```
closed.lua:5: gui: the window has closed
```

the window is gone, so the dll refuses to show it, and `check` turns that into a lua error with the scripts own line on it

## exercise 6, the version check

add the version check to `luaopen_gui`

### pseudo

1. ask the dll its version
2. compare it with `GUI_ABI_VERSION` from the header
3. if they differ, raise with both numbers in the message

**tip** the stand in reads a `FAKE_VERSION` environment variable, set it to 5 to see the error without swapping any dlls

## section 8, testing the failures

# expect_error

```lua
-- runs the given function and checks it fails with the given text
local function expect_error(fn, text)
        local ok, err = pcall(fn)
        assert(not ok, "expected an error, got none")
        assert(string.find(err, text, 1, true), "wrong error, got " .. tostring(err))
end
```

**pcall** runs it and catches the error

**the first assert** it must have failed

**the second** with the words we expected, the `1, true` makes `string.find` match the text as is, no pattern tricks

## the test script

```
-- arguments
expect_error(function() gui.window("test", "wide", 300) end, "number expected, got string")
expect_error(function() gui.window("test", -5, 300) end, "width must be above 0")
expect_error(function() gui.button("ok", "big") end, "invalid option 'big'")
expect_error(function() gui.sleep(-1) end, "cannot sleep for less than 0 ms")

-- the dll saying no
expect_error(function() gui.show(gui.label("not a window")) end, "gui: bad handle")
```

```
all tests passed
```

a failure is behaviour too, and it is the behaviour that breaks quietly, so it gets tested as hard as the happy path

## exercise 7, grow the tests

the whole file is `lualib/tests/test.lua`, add a test for each of these

### pseudo

1. a height of 0
2. `gui.label()` with no text
3. showing a window after it closed

**tip** run it with `gui-host.exe lualib\tests\test.lua`, it only prints `all tests passed` once every assert holds, and any failure tells you which one and why

## section 9, wrapping up

# the new calls

| call | what it does |
|---|---|
| `error` `pcall` | raise and catch, from lua |
| `luaL_error` | raise from c, printf style, never returns |
| `luaL_argcheck` `luaL_argerror` | raise an argument error with your own words |
| `luaL_opt*` | read an argument, or a default when its missing |
| `luaL_checkoption` | a string argument from a fixed list, as its index |
| `gui_err_text` | the dll naming one of its statuses |
| `gui_abi_version` `GUI_ABI_VERSION` | which abi the dll speaks, and which one we built for |

## why its built this way

- **raise, never nil**, every status from this dll is a mistake, and the one expected event already has its own answer
- **one check helper**, so no call goes unchecked and every message has the same shape
- **names, never numbers**, the abi numbers are free to move and the script should never care
- **check then allocate, then tidy then raise**, bc a raise skips everything after it, destructors included
- **refuse the wrong dll at require**, the earliest place a mismatch can be caught
- **finn-style**, the given status in the comment on `check`, `m_` on the two lists, and the two lists side by side

## recap

- an error jumps straight back to the nearest `pcall`, everything in between is abandoned
- `luaL_error` is `error` from c, it never returns, and the position points at the scripts line
- raise for mistakes, nil for things that just happen, and for this dll thats always raise
- `luaL_argcheck` and `luaL_checkoption` make argument errors that read like luas own
- anything you made before a raise is yours to free, so check first, and tidy before you raise
- test the failures, `expect_error` makes it one line each

## have a go

let `gui.window` make a normal window with a title bar, not just a popup

```
local win = gui.window("tool", 400, 300, "chromed")   -- "popup" when missing
```

### pseudo

1. a names list, `"popup"` and `"chromed"`, and a values list of `gui_kind_popup` and `gui_kind_chromed`
2. read argument 4 with `luaL_checkoption`, default `"popup"`
3. set it on the desc with `gui_desc_set_kind`, inside the tidy block
4. a test for a bad kind name

**tip** a chromed window on the default present is refused, the dll answers `the window could not be made`, set `gui_desc_set_present(desc, gui_present_opaque)` as well when its chromed

## whats next

part 3, **tables**

`gui.window` already has four positional arguments, and a script reading `gui.window("tool", 400, 300, "chromed")` has to remember which number is which

next time the window takes a table, `gui.window{ title = "tool", width = 400 }`, and we learn to read one from c, walk it, and build one to hand back