1 3.7 - Functions and Types

Here we list all functions and types from the C API in alphabetical order. Each function has an indicator like this: `[-o, +p, x]`
The first field, `o`, is how many elements the function pops from the stack. The second field, `p`, is how many elements the function pushes onto the stack. (Any function always pushes its results after popping its arguments.) A field in the form `x|y` means the function can push (or pop) `x` or `y` elements, depending on the situation; an interrogation mark '`?`' means that we cannot know how many elements the function pops/pushes by looking only at its arguments (e.g., they may depend on what is on the stack). The third field, `x`, tells whether the function may throw errors: '`-`' means the function never throws any error; '`m`' means the function may throw an error only due to not enough memory; '`e`' means the function may throw other kinds of errors; '`v`' means the function may throw an error on purpose.

1.1 `lua_Alloc`

```
typedef void * (*lua_Alloc) (void *ud,
                             void *ptr,
                             size_t osize,
                             size_t nsize);
```

The type of the memory-allocation function used by Lua states. The allocator function must provide a functionality similar to `realloc`, but not exactly the same. Its arguments are `ud`, an opaque pointer passed to `lua_newstate`; `ptr`, a pointer to the block being allocated/reallocated/freed; `osize`, the original size of the block; `nsize`, the new size of the block. `ptr` is `NULL` if and only if `osize` is zero. When `nsize` is zero, the allocator must return `NULL`; if `osize` is not zero, it should free the block pointed to by `ptr`. When `nsize` is not zero, the allocator returns `NULL` if and only if it cannot fill the request. When `nsize` is not zero and `osize` is zero, the allocator should behave like `malloc`. When `nsize` and `osize` are not zero, the allocator behaves like `realloc`. Lua assumes that the allocator never fails when `osize >= nsize`.
Here is a simple implementation for the allocator function. It is used in the auxiliary library by `luaL_newstate`.

```
static void *l_alloc (void *ud, void *ptr, size_t osize,
                                                  size_t nsize) {
       (void)ud;  (void)osize;  /* not used */
       if (nsize == 0) {
         free(ptr);
```

```
    return NULL;
  }
  else
    return realloc(ptr, nsize);
}
```

This code assumes that `free(NULL)` has no effect and that `realloc(NULL, size)` is equivalent to `malloc(size)`. ANSI C ensures both behaviors.

1.2 lua_atpanic

`[-0, +0, -]`

```
lua_CFunction lua_atpanic (lua_State *L, lua_CFunction panicf);
```

Sets a new panic function and returns the old one.
If an error happens outside any protected environment, Lua calls a *panic function* and then calls `exit(EXIT_FAILURE)`, thus exiting the host application. Your panic function can avoid this exit by never returning (e.g., doing a long jump).
The panic function can access the error message at the top of the stack.

1.3 lua_call

`[-(nargs + 1), +nresults, e]`

```
void lua_call (lua_State *L, int nargs, int nresults);
```

Calls a function.
To call a function you must use the following protocol: first, the function to be called is pushed onto the stack; then, the arguments to the function are pushed in direct order; that is, the first argument is pushed first. Finally you call `lua_call`; `nargs` is the number of arguments that you pushed onto the stack. All arguments and the function value are popped from the stack when the function is called. The function results are pushed onto the stack when the function returns. The number of results is adjusted to `nresults`, unless `nresults` is `LUA_MULTRET` . In this case, *all* results from the function are pushed. Lua takes care that the returned values fit into the stack space. The function results are pushed onto the stack in direct order (the first result is pushed first), so that after the call the last result is on the top of the stack.
Any error inside the called function is propagated upwards (with a `longjmp`).

The following example shows how the host program can do the equivalent to this Lua code:

```
a = f("how", t.x, 14)
```

Here it is in C:

```
lua_getfield(L, LUA_GLOBALSINDEX, "f"); /* function to be called */
     lua_pushstring(L, "how");                        /* 1st argument */
     lua_getfield(L, LUA_GLOBALSINDEX, "t");   /* table to be indexed */
     lua_getfield(L, -1, "x");        /* push result of t.x (2nd arg) */
     lua_remove(L, -2);                  /* remove 't' from the stack */
     lua_pushinteger(L, 14);                          /* 3rd argument */
     lua_call(L, 3, 1);     /* call 'f' with 3 arguments and 1 result */
     lua_setfield(L, LUA_GLOBALSINDEX, "a");        /* set global 'a' */
```

Note that the code above is "balanced": at its end, the stack is back to its original configuration. This is considered good programming practice.

1.4 `lua_CFunction`

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

Type for C functions.
In order to communicate properly with Lua, a C function must use the following protocol, which defines the way parameters and results are passed: a C function receives its arguments from Lua in its stack in direct order (the first argument is pushed first). So, when the function starts, `lua_gettop(L)` returns the number of arguments received by the function. The first argument (if any) is at index 1 and its last argument is at index `lua_gettop(L)`. To return values to Lua, a C function just pushes them onto the stack, in direct order (the first result is pushed first), and returns the number of results. Any other value in the stack below the results will be properly discarded by Lua. Like a Lua function, a C function called by Lua can also return many results.
As an example, the following function receives a variable number of numerical arguments and returns their average and sum:

```
static int foo (lua_State *L) {
       int n = lua_gettop(L);    /* number of arguments */
       lua_Number sum = 0;
       int i;
       for (i = 1; i <= n; i++) {
```

```
      if (!lua_isnumber(L, i)) {
        lua_pushstring(L, "incorrect argument");
        lua_error(L);
      }
      sum += lua_tonumber(L, i);
    }
    lua_pushnumber(L, sum/n);        /* first result */
    lua_pushnumber(L, sum);         /* second result */
    return 2;                   /* number of results */
  }
```

1.5 lua_checkstack

`[-0, +0, m]`

```
int lua_checkstack (lua_State *L, int extra);
```

Ensures that there are at least `extra` free stack slots in the stack. It returns false if it cannot grow the stack to that size. This function never shrinks the stack; if the stack is already larger than the new size, it is left unchanged.

1.6 lua_close

`[-0, +0, -]`

```
void lua_close (lua_State *L);
```

Destroys all objects in the given Lua state (calling the corresponding garbage-collection metamethods, if any) and frees all dynamic memory used by this state. On several platforms, you may not need to call this function, because all resources are naturally released when the host program ends. On the other hand, long-running programs, such as a daemon or a web server, might need to release states as soon as they are not needed, to avoid growing too large.

1.7 lua_concat

`[-n, +1, e]`

```
void lua_concat (lua_State *L, int n);
```

Concatenates the `n` values at the top of the stack, pops them, and leaves the result at the top. If `n` is 1, the result is the single value on the stack (that is, the function does nothing); if `n` is 0, the result is the empty string. Concatenation is performed following the usual semantics of Lua (see §2.5.4).

1.8 lua_cpcall

```
[-0, +(0|1), -]
```

```
int lua_cpcall (lua_State *L, lua_CFunction func, void *ud);
```

Calls the C function `func` in protected mode. `func` starts with only one element in its stack, a light userdata containing `ud`. In case of errors, `lua_cpcall` returns the same error codes as `lua_pcall`, plus the error object on the top of the stack; otherwise, it returns zero, and does not change the stack. All values returned by `func` are discarded.

1.9 lua_createtable

```
[-0, +1, m]
```

```
void lua_createtable (lua_State *L, int narr, int nrec);
```

Creates a new empty table and pushes it onto the stack. The new table has space pre-allocated for `narr` array elements and `nrec` non-array elements. This pre-allocation is useful when you know exactly how many elements the table will have. Otherwise you can use the function `lua_newtable`.

1.10 lua_dump

```
[-0, +0, m]
```

```
int lua_dump (lua_State *L, lua_Writer writer, void *data);
```

Dumps a function as a binary chunk. Receives a Lua function on the top of the stack and produces a binary chunk that, if loaded again, results in a function equivalent to the one dumped. As it produces parts of the chunk, `lua_dump` calls function `writer` (see `lua_Writer`) with the given `data` to write them.
The value returned is the error code returned by the last call to the writer; 0 means no errors.

This function does not pop the Lua function from the stack.

1.11 lua_equal

`[-0, +0, e]`

```
int lua_equal (lua_State *L, int index1, int index2);
```

Returns 1 if the two values in acceptable indices `index1` and `index2` are equal, following the semantics of the Lua `==` operator (that is, may call metamethods). Otherwise returns 0. Also returns 0 if any of the indices is non valid.

1.12 lua_error

`[-1, +0, v]`

```
int lua_error (lua_State *L);
```

Generates a Lua error. The error message (which can actually be a Lua value of any type) must be on the stack top. This function does a long jump, and therefore never returns. (see `luaL_error`).

1.13 lua_gc

`[-0, +0, e]`

```
int lua_gc (lua_State *L, int what, int data);
```

Controls the garbage collector.
This function performs several tasks, according to the value of the parameter `what`:

- `LUA_GCSTOP`: stops the garbage collector.
- `LUA_GCRESTART`: restarts the garbage collector.
- `LUA_GCCOLLECT`: performs a full garbage-collection cycle.
- `LUA_GCCOUNT`: returns the current amount of memory (in Kbytes) in use by Lua.
- `LUA_GCCOUNTB`: returns the remainder of dividing the current amount of bytes of memory in use by Lua by 1024.

- `LUA_GCSTEP`: performs an incremental step of garbage collection. The step "size" is controlled by `data` (larger values mean more steps) in a non-specified way. If you want to control the step size you must experimentally tune the value of `data`. The function returns 1 if the step finished a garbage-collection cycle.
- `LUA_GCSETPAUSE`: sets `data` as the new value for the *pause* of the collector (see §2.10). The function returns the previous value of the pause.
- `LUA_GCSETSTEPMUL`: sets `data` as the new value for the *step multiplier* of the collector (see §2.10). The function returns the previous value of the step multiplier.

1.14 lua_getallocf

`[-0, +0, -]`

```
lua_Alloc lua_getallocf (lua_State *L, void **ud);
```

Returns the memory-allocation function of a given state. If `ud` is not `NULL`, Lua stores in `*ud` the opaque pointer passed to `lua_newstate`.

1.15 lua_getfenv

`[-0, +1, -]`

```
void lua_getfenv (lua_State *L, int index);
```

Pushes onto the stack the environment table of the value at the given index.

1.16 lua_getfield

`[-0, +1, e]`

```
void lua_getfield (lua_State *L, int index, const char *k);
```

Pushes onto the stack the value `t[k]`, where `t` is the value at the given valid index. As in Lua, this function may trigger a metamethod for the "index" event (see §2.8).

1.17 lua_getglobal

`[-0, +1, e]`

```
void lua_getglobal (lua_State *L, const char *name);
```

Pushes onto the stack the value of the global `name`. It is defined as a macro:

```
##define lua_getglobal(L,s)  lua_getfield(L, LUA_GLOBALSINDEX, s)
```

1.18 lua_getmetatable

```
[-0, +(0|1), -]
```

```
int lua_getmetatable (lua_State *L, int index);
```

Pushes onto the stack the metatable of the value at the given acceptable index. If the index is not valid, or if the value does not have a metatable, the function returns 0 and pushes nothing on the stack.

1.19 lua_gettable

```
[-1, +1, e]
```

```
void lua_gettable (lua_State *L, int index);
```

Pushes onto the stack the value `t[k]`, where `t` is the value at the given valid index and `k` is the value at the top of the stack.
This function pops the key from the stack (putting the resulting value in its place). As in Lua, this function may trigger a metamethod for the "index" event (see §2.8).

1.20 lua_gettop

```
[-0, +0, -]
```

```
int lua_gettop (lua_State *L);
```

Returns the index of the top element in the stack. Because indices start at 1, this result is equal to the number of elements in the stack (and so 0 means an empty stack).

1.21 lua_insert

```
[-1, +1, -]
```

```
void lua_insert (lua_State *L, int index);
```

Moves the top element into the given valid index, shifting up the elements above this index to open space. Cannot be called with a pseudo-index, because a pseudo-index is not an actual stack position.

1.22 lua_Integer

```
typedef ptrdiff_t lua_Integer;
```

The type used by the Lua API to represent integral values.
By default it is a `ptrdiff_t`, which is usually the largest signed integral type the machine handles "comfortably".

1.23 lua_isboolean

```
[-0, +0, -]
```

```
int lua_isboolean (lua_State *L, int index);
```

Returns 1 if the value at the given acceptable index has type boolean, and 0 otherwise.

1.24 lua_iscfunction

```
[-0, +0, -]
```

```
int lua_iscfunction (lua_State *L, int index);
```

Returns 1 if the value at the given acceptable index is a C function, and 0 otherwise.

1.25 lua_isfunction

```
[-0, +0, -]
```

```
int lua_isfunction (lua_State *L, int index);
```

Returns 1 if the value at the given acceptable index is a function (either C or Lua), and 0 otherwise.

1.26 lua_islightuserdata

[-0, +0, -]

```
int lua_islightuserdata (lua_State *L, int index);
```

Returns 1 if the value at the given acceptable index is a light userdata, and 0 otherwise.

1.27 lua_isnil

[-0, +0, -]

```
int lua_isnil (lua_State *L, int index);
```

Returns 1 if the value at the given acceptable index is **nil**, and 0 otherwise.

1.28 lua_isnone

[-0, +0, -]

```
int lua_isnone (lua_State *L, int index);
```

Returns 1 if the given acceptable index is not valid (that is, it refers to an element outside the current stack), and 0 otherwise.

1.29 lua_isnoneornil

[-0, +0, -]

```
int lua_isnoneornil (lua_State *L, int index);
```

Returns 1 if the given acceptable index is not valid (that is, it refers to an element outside the current stack) or if the value at this index is **nil**, and 0 otherwise.

1.30 lua_isnumber

[-0, +0, -]

```
int lua_isnumber (lua_State *L, int index);
```

Returns 1 if the value at the given acceptable index is a number or a string convertible to a number, and 0 otherwise.

1.31 lua_isstring

```
[-0, +0, -]
```

```
int lua_isstring (lua_State *L, int index);
```

Returns 1 if the value at the given acceptable index is a string or a number (which is always convertible to a string), and 0 otherwise.

1.32 lua_istable

```
[-0, +0, -]
```

```
int lua_istable (lua_State *L, int index);
```

Returns 1 if the value at the given acceptable index is a table, and 0 otherwise.

1.33 lua_isthread

```
[-0, +0, -]
```

```
int lua_isthread (lua_State *L, int index);
```

Returns 1 if the value at the given acceptable index is a thread, and 0 otherwise.

1.34 lua_isuserdata

```
[-0, +0, -]
```

```
int lua_isuserdata (lua_State *L, int index);
```

Returns 1 if the value at the given acceptable index is a userdata (either full or light), and 0 otherwise.

1.35 lua_lessthan

```
[-0, +0, e]
```

```
int lua_lessthan (lua_State *L, int index1, int index2);
```

Returns 1 if the value at acceptable index `index1` is smaller than the value at acceptable index `index2`, following the semantics of the Lua `<` operator (that is, may call metamethods). Otherwise returns 0. Also returns 0 if any of the indices is non valid.

1.36 lua_load

`[-0, +1, -]`

```
int lua_load (lua_State *L,
              lua_Reader reader,
              void *data,
              const char *chunkname);
```

Loads a Lua chunk. If there are no errors, `lua_load` pushes the compiled chunk as a Lua function on top of the stack. Otherwise, it pushes an error message. The return values of `lua_load` are:

- **0:** no errors;
- `LUA_ERRSYNTAX` **:** syntax error during pre-compilation;
- `LUA_ERRMEM`**:** memory allocation error.

This function only loads a chunk; it does not run it.
`lua_load` automatically detects whether the chunk is text or binary, and loads it accordingly (see program `luac`).
The `lua_load` function uses a user-supplied `reader` function to read the chunk (see `lua_Reader`). The `data` argument is an opaque value passed to the reader function.
The `chunkname` argument gives a name to the chunk, which is used for error messages and in debug information (see §3.8).

1.37 lua_newstate

`[-0, +0, -]`

```
lua_State *lua_newstate (lua_Alloc f, void *ud);
```

Creates a new, independent state. Returns `NULL` if cannot create the state (due to lack of memory). The argument `f` is the allocator function; Lua does all memory

allocation for this state through this function. The second argument, `ud`, is an opaque pointer that Lua simply passes to the allocator in every call.

1.38 lua_newtable

`[-0, +1, m]`

```
void lua_newtable (lua_State *L);
```

Creates a new empty table and pushes it onto the stack. It is equivalent to `lua_createtable(L, 0, 0)`.

1.39 lua_newthread

`[-0, +1, m]`

```
lua_State *lua_newthread (lua_State *L);
```

Creates a new thread, pushes it on the stack, and returns a pointer to a `lua_State` that represents this new thread. The new state returned by this function shares with the original state all global objects (such as tables), but has an independent execution stack.
There is no explicit function to close or to destroy a thread. Threads are subject to garbage collection, like any Lua object.

1.40 lua_newuserdata

`[-0, +1, m]`

```
void *lua_newuserdata (lua_State *L, size_t size);
```

This function allocates a new block of memory with the given size, pushes onto the stack a new full userdata with the block address, and returns this address.
Userdata represent C values in Lua. A *full userdata* represents a block of memory. It is an object (like a table): you must create it, it can have its own metatable, and you can detect when it is being collected. A full userdata is only equal to itself (under raw equality).

When Lua collects a full userdata with a `gc` metamethod, Lua calls the metamethod and marks the userdata as finalized. When this userdata is collected again then Lua frees its corresponding memory.

1.41 lua_next

`[-1, +(2|0), e]`

```
int lua_next (lua_State *L, int index);
```

Pops a key from the stack, and pushes a key-value pair from the table at the given index (the "next" pair after the given key). If there are no more elements in the table, then `lua_next` returns 0 (and pushes nothing).
A typical traversal looks like this:

```
/* table is in the stack at index 't' */
     lua_pushnil(L);  /* first key */
     while (lua_next(L, t) != 0) {
       /* uses 'key' (at index -2) and 'value' (at index -1) */
       printf("%s - %s\n",
              lua_typename(L, lua_type(L, -2)),
              lua_typename(L, lua_type(L, -1)));
       /* removes 'value'; keeps 'key' for next iteration */
       lua_pop(L, 1);
     }
```

While traversing a table, do not call `lua_tolstring` directly on a key, unless you know that the key is actually a string. Recall that `lua_tolstring` *changes* the value at the given index; this confuses the next call to `lua_next`.

1.42 lua_Number

```
typedef double lua_Number;
```

The type of numbers in Lua. By default, it is double, but that can be changed in `luaconf.h`.
Through the configuration file you can change Lua to operate with another type for numbers (e.g., float or long).

1.43 lua_objlen

`[-0, +0, -]`

```
size_t lua_objlen (lua_State *L, int index);
```

Returns the "length" of the value at the given acceptable index: for strings, this is the string length; for tables, this is the result of the length operator ('#'); for userdata, this is the size of the block of memory allocated for the userdata; for other values, it is 0.

1.44 lua_pcall

```
[-(nargs + 1), +(nresults|1), -]
```

```
int lua_pcall (lua_State *L, int nargs, int nresults, int errfunc);
```

Calls a function in protected mode.
Both `nargs` and `nresults` have the same meaning as in `lua_call`. If there are no errors during the call, `lua_pcall` behaves exactly like `lua_call`. However, if there is any error, `lua_pcall` catches it, pushes a single value on the stack (the error message), and returns an error code. Like `lua_call`, `lua_pcall` always removes the function and its arguments from the stack.
If `errfunc` is 0, then the error message returned on the stack is exactly the original error message. Otherwise, `errfunc` is the stack index of an *error handler function*. (In the current implementation, this index cannot be a pseudo-index.) In case of runtime errors, this function will be called with the error message and its return value will be the message returned on the stack by `lua_pcall`.
Typically, the error handler function is used to add more debug information to the error message, such as a stack traceback. Such information cannot be gathered after the return of `lua_pcall`, since by then the stack has unwound.
The `lua_pcall` function returns 0 in case of success or one of the following error codes (defined in `lua.h`):

- `LUA_ERRRUN` : a runtime error.
- `LUA_ERRMEM` : memory allocation error. For such errors, Lua does not call the error handler function.
- `LUA_ERRERR` : error while running the error handler function.

1.45 lua_pop

```
[-n, +0, -]
```

```
void lua_pop (lua_State *L, int n);
```

Pops `n` elements from the stack.

1.46 lua_pushboolean

`[-0, +1, -]`

```
void lua_pushboolean (lua_State *L, int b);
```

Pushes a boolean value with value `b` onto the stack.

1.47 lua_pushcclosure

`[-n, +1, m]`

```
void lua_pushcclosure (lua_State *L, lua_CFunction fn, int n);
```

Pushes a new C closure onto the stack.
When a C function is created, it is possible to associate some values with it, thus creating a C closure (see §3.4); these values are then accessible to the function whenever it is called. To associate values with a C function, first these values should be pushed onto the stack (when there are multiple values, the first value is pushed first). Then `lua_pushcclosure` is called to create and push the C function onto the stack, with the argument `n` telling how many values should be associated with the function. `lua_pushcclosure` also pops these values from the stack.
The maximum value for `n` is 255.

1.48 lua_pushcfunction

`[-0, +1, m]`

```
void lua_pushcfunction (lua_State *L, lua_CFunction f);
```

Pushes a C function onto the stack. This function receives a pointer to a C function and pushes onto the stack a Lua value of type `function` that, when called, invokes the corresponding C function.
Any function to be registered in Lua must follow the correct protocol to receive its parameters and return its results (see `lua_CFunction`).
`lua_pushcfunction` is defined as a macro:

```
##define lua_pushcfunction(L,f)  lua_pushcclosure(L,f,0)
```

1.49 lua_pushfstring

`[-0, +1, m]`

```
const char *lua_pushfstring (lua_State *L, const char *fmt, ...);
```

Pushes onto the stack a formatted string and returns a pointer to this string. It is similar to the C function `sprintf`, but has some important differences:

- You do not have to allocate space for the result: the result is a Lua string and Lua takes care of memory allocation (and deallocation, through garbage collection).
- The conversion specifiers are quite restricted. There are no flags, widths, or precisions. The conversion specifiers can only be '`%%`' (inserts a '`%`' in the string), '`%s`' (inserts a zero-terminated string, with no size restrictions), '`%f`' (inserts a `lua_Number`), '`%p`' (inserts a pointer as a hexadecimal numeral), '`%d`' (inserts an `int`), and '`%c`' (inserts an `int` as a character).

1.50 lua_pushinteger

`[-0, +1, -]`

```
void lua_pushinteger (lua_State *L, lua_Integer n);
```

Pushes a number with value `n` onto the stack.

1.51 lua_pushlightuserdata

`[-0, +1, -]`

```
void lua_pushlightuserdata (lua_State *L, void *p);
```

Pushes a light userdata onto the stack.
Userdata represent C values in Lua. A *light userdata* represents a pointer. It is a value (like a number): you do not create it, it has no individual metatable, and it

is not collected (as it was never created). A light userdata is equal to "any" light userdata with the same C address.

1.52 lua_pushliteral

`[-0, +1, m]`

```
void lua_pushliteral (lua_State *L, const char *s);
```

This macro is equivalent to lua_pushlstring, but can be used only when `s` is a literal string. In these cases, it automatically provides the string length.

1.53 lua_pushlstring

`[-0, +1, m]`

```
void lua_pushlstring (lua_State *L, const char *s, size_t len);
```

Pushes the string pointed to by `s` with size `len` onto the stack. Lua makes (or reuses) an internal copy of the given string, so the memory at `s` can be freed or reused immediately after the function returns. The string can contain embedded zeros.

1.54 lua_pushnil

`[-0, +1, -]`

```
void lua_pushnil (lua_State *L);
```

Pushes a nil value onto the stack.

1.55 lua_pushnumber

`[-0, +1, -]`

```
void lua_pushnumber (lua_State *L, lua_Number n);
```

Pushes a number with value `n` onto the stack.

1.56 lua_pushstring

`[-0, +1, m]`

```
void lua_pushstring (lua_State *L, const char *s);
```

Pushes the zero-terminated string pointed to by `s` onto the stack. Lua makes (or reuses) an internal copy of the given string, so the memory at `s` can be freed or reused immediately after the function returns. The string cannot contain embedded zeros; it is assumed to end at the first zero.

1.57 lua_pushthread

```
[-0, +1, -]
```

```
int lua_pushthread (lua_State *L);
```

Pushes the thread represented by `L` onto the stack. Returns 1 if this thread is the main thread of its state.

1.58 lua_pushvalue

```
[-0, +1, -]
```

```
void lua_pushvalue (lua_State *L, int index);
```

Pushes a copy of the element at the given valid index onto the stack.

1.59 lua_pushvfstring

```
[-0, +1, m]
```

```
const char *lua_pushvfstring (lua_State *L,
                              const char *fmt,
                              va_list argp);
```

Equivalent to lua_pushfstring, except that it receives a `va_list` instead of a variable number of arguments.

1.60 lua_rawequal

```
[-0, +0, -]
```

```
int lua_rawequal (lua_State *L, int index1, int index2);
```

Returns 1 if the two values in acceptable indices `index1` and `index2` are primitively equal (that is, without calling metamethods). Otherwise returns 0. Also returns 0 if any of the indices are non valid.

1.61 lua_rawget

```
[-1, +1, -]
```

```
void lua_rawget (lua_State *L, int index);
```

Similar to `lua_gettable`, but does a raw access (i.e., without metamethods).

1.62 lua_rawgeti

```
[-0, +1, -]
```

```
void lua_rawgeti (lua_State *L, int index, int n);
```

Pushes onto the stack the value `t[n]`, where `t` is the value at the given valid index. The access is raw; that is, it does not invoke metamethods.

1.63 lua_rawset

```
[-2, +0, m]
```

```
void lua_rawset (lua_State *L, int index);
```

Similar to `lua_settable`, but does a raw assignment (i.e., without metamethods).

1.64 lua_rawseti

```
[-1, +0, m]
```

```
void lua_rawseti (lua_State *L, int index, int n);
```

Does the equivalent of `t[n] = v`, where `t` is the value at the given valid index and `v` is the value at the top of the stack.

This function pops the value from the stack. The assignment is raw; that is, it does not invoke metamethods.

1.65 lua_Reader

```
typedef const char * (*lua_Reader) (lua_State *L,
                                    void *data,
                                    size_t *size);
```

The reader function used by lua_load. Every time it needs another piece of the chunk, lua_load calls the reader, passing along its data parameter. The reader must return a pointer to a block of memory with a new piece of the chunk and set size to the block size. The block must exist until the reader function is called again. To signal the end of the chunk, the reader must return NULL or set size to zero. The reader function may return pieces of any size greater than zero.

1.66 lua_register

```
[-0, +0, e]
```

```
void lua_register (lua_State *L,
                   const char *name,
                   lua_CFunction f);
```

Sets the C function f as the new value of global name. It is defined as a macro:

```
##define lua_register(L,n,f) \
            (lua_pushcfunction(L, f), lua_setglobal(L, n))
```

1.67 lua_remove

```
[-1, +0, -]
```

```
void lua_remove (lua_State *L, int index);
```

Removes the element at the given valid index, shifting down the elements above this index to fill the gap. Cannot be called with a pseudo-index, because a pseudo-index is not an actual stack position.

1.68 lua_replace

```
[-1, +0, -]
```

```
void lua_replace (lua_State *L, int index);
```

Moves the top element into the given position (and pops it), without shifting any element (therefore replacing the value at the given position).

1.69 lua_resume

```
[-?, +?, -]
```

```
int lua_resume (lua_State *L, int narg);
```

Starts and resumes a coroutine in a given thread.
To start a coroutine, you first create a new thread (see lua_newthread); then you push onto its stack the main function plus any arguments; then you call lua_resume, with narg being the number of arguments. This call returns when the coroutine suspends or finishes its execution. When it returns, the stack contains all values passed to lua_yield, or all values returned by the body function. lua_resume returns LUA_YIELD if the coroutine yields, 0 if the coroutine finishes its execution without errors, or an error code in case of errors (see lua_pcall). In case of errors, the stack is not unwound, so you can use the debug API over it. The error message is on the top of the stack. To restart a coroutine, you put on its stack only the values to be passed as results from yield, and then call lua_resume.

1.70 lua_setallocf

```
[-0, +0, -]
```

```
void lua_setallocf (lua_State *L, lua_Alloc f, void *ud);
```

Changes the allocator function of a given state to f with user data ud.

1.71 lua_setfenv

```
[-1, +0, -]
```

```
int lua_setfenv (lua_State *L, int index);
```

Pops a table from the stack and sets it as the new environment for the value at the given index. If the value at the given index is neither a function nor a thread nor a userdata, lua_setfenv returns 0. Otherwise it returns 1.

1.72 lua_setfield

[-1, +0, e]

```
void lua_setfield (lua_State *L, int index, const char *k);
```

Does the equivalent to t[k] = v, where t is the value at the given valid index and v is the value at the top of the stack.
This function pops the value from the stack. As in Lua, this function may trigger a metamethod for the "newindex" event (see §2.8).

1.73 lua_setglobal

[-1, +0, e]

```
void lua_setglobal (lua_State *L, const char *name);
```

Pops a value from the stack and sets it as the new value of global name. It is defined as a macro:

```
##define lua_setglobal(L,s)   lua_setfield(L, LUA_GLOBALSINDEX, s)
```

1.74 lua_setmetatable

[-1, +0, -]

```
int lua_setmetatable (lua_State *L, int index);
```

Pops a table from the stack and sets it as the new metatable for the value at the given acceptable index.

1.75 lua_settable

[-2, +0, e]

```
void lua_settable (lua_State *L, int index);
```

Does the equivalent to `t[k] = v`, where `t` is the value at the given valid index, `v` is the value at the top of the stack, and `k` is the value just below the top.
This function pops both the key and the value from the stack. As in Lua, this function may trigger a metamethod for the "newindex" event (see §2.8).

1.76 lua_settop

`[-?, +?, -]`

```
void lua_settop (lua_State *L, int index);
```

Accepts any acceptable index, or 0, and sets the stack top to this index. If the new top is larger than the old one, then the new elements are filled with **nil**. If `index` is 0, then all stack elements are removed.

1.77 lua_State

```
typedef struct lua_State lua_State;
```

Opaque structure that keeps the whole state of a Lua interpreter. The Lua library is fully reentrant: it has no global variables. All information about a state is kept in this structure.
A pointer to this state must be passed as the first argument to every function in the library, except to `lua_newstate`, which creates a Lua state from scratch.

1.78 lua_status

`[-0, +0, -]`

```
int lua_status (lua_State *L);
```

Returns the status of the thread `L`.
The status can be 0 for a normal thread, an error code if the thread finished its execution with an error, or `LUA_YIELD` if the thread is suspended.

1.79 lua_toboolean

`[-0, +0, -]`

```
int lua_toboolean (lua_State *L, int index);
```

Converts the Lua value at the given acceptable index to a C boolean value (0 or 1). Like all tests in Lua, lua_toboolean returns 1 for any Lua value different from **false** and **nil**; otherwise it returns 0. It also returns 0 when called with a non-valid index. (If you want to accept only actual boolean values, use lua_isboolean to test the value's type.)

1.80 lua_tocfunction

[-0, +0, -]

```
lua_CFunction lua_tocfunction (lua_State *L, int index);
```

Converts a value at the given acceptable index to a C function. That value must be a C function; otherwise, returns NULL.

1.81 lua_tointeger

[-0, +0, -]

```
lua_Integer lua_tointeger (lua_State *L, int index);
```

Converts the Lua value at the given acceptable index to the signed integral type lua_Integer. The Lua value must be a number or a string convertible to a number (see §2.2.1); otherwise, lua_tointeger returns 0.
If the number is not an integer, it is truncated in some non-specified way.

1.82 lua_tolstring

[-0, +0, m]

```
const char *lua_tolstring (lua_State *L, int index, size_t *len);
```

Converts the Lua value at the given acceptable index to a C string. If len is not NULL, it also sets *len with the string length. The Lua value must be a string or a number; otherwise, the function returns NULL. If the value is a number, then lua_tolstring also *changes the actual value in the stack to a string*. (This change confuses lua_next when lua_tolstring is applied to keys during a table traversal.) lua_tolstring returns a fully aligned pointer to a string inside the Lua state. This string always has a zero ('\0') after its last character (as in C), but can contain other zeros in its body. Because Lua has garbage collection, there is no guarantee

that the pointer returned by lua_tolstring will be valid after the corresponding value is removed from the stack.

1.83 lua_tonumber

[-0, +0, -]

```
lua_Number lua_tonumber (lua_State *L, int index);
```

Converts the Lua value at the given acceptable index to the C type lua_Number (see lua_Number). The Lua value must be a number or a string convertible to a number (see §2.2.1); otherwise, lua_tonumber returns 0.

1.84 lua_topointer

[-0, +0, -]

```
const void *lua_topointer (lua_State *L, int index);
```

Converts the value at the given acceptable index to a generic C pointer (void*). The value can be a userdata, a table, a thread, or a function; otherwise, lua_topointer returns NULL. Different objects will give different pointers. There is no way to convert the pointer back to its original value.
Typically this function is used only for debug information.

1.85 lua_tostring

[-0, +0, m]

```
const char *lua_tostring (lua_State *L, int index);
```

Equivalent to lua_tolstring with len equal to NULL.

1.86 lua_tothread

[-0, +0, -]

```
lua_State *lua_tothread (lua_State *L, int index);
```

Converts the value at the given acceptable index to a Lua thread (represented as `lua_State*`). This value must be a thread; otherwise, the function returns `NULL`.

1.87 lua_touserdata

`[-0, +0, -]`

```
void *lua_touserdata (lua_State *L, int index);
```

If the value at the given acceptable index is a full userdata, returns its block address. If the value is a light userdata, returns its pointer. Otherwise, returns `NULL`.

1.88 lua_type

`[-0, +0, -]`

```
int lua_type (lua_State *L, int index);
```

Returns the type of the value in the given acceptable index, or `LUA_TNONE` for a non-valid index (that is, an index to an "empty" stack position). The types returned by lua_type are coded by the following constants defined in `lua.h`: `LUA_TNIL`, `LUA_TNUMBER`, `LUA_TBOOLEAN`, `LUA_TSTRING`, `LUA_TTABLE`, `LUA_TFUNCTION`, `LUA_TUSERDATA`, `LUA_TTHREAD`, and `LUA_TLIGHTUSERDATA`.

1.89 lua_typename

`[-0, +0, -]`

```
const char *lua_typename  (lua_State *L, int tp);
```

Returns the name of the type encoded by the value `tp`, which must be one the values returned by lua_type.

1.90 lua_Writer

```
typedef int (*lua_Writer) (lua_State *L,
                           const void* p,
                           size_t sz,
                           void* ud);
```

The type of the writer function used by lua_dump. Every time it produces another piece of chunk, lua_dump calls the writer, passing along the buffer to be written (p), its size (sz), and the data parameter supplied to lua_dump.
The writer returns an error code: 0 means no errors; any other value means an error and stops lua_dump from calling the writer again.

1.91 lua_xmove

[-?, +?, -]

```
void lua_xmove (lua_State *from, lua_State *to, int n);
```

Exchange values between different threads of the *same* global state.
This function pops n values from the stack from, and pushes them onto the stack to.

1.92 lua_yield

[-?, +?, -]

```
int lua_yield  (lua_State *L, int nresults);
```

Yields a coroutine.
This function should only be called as the return expression of a C function, as follows:

```
return lua_yield (L, nresults);
```

When a C function calls lua_yield in that way, the running coroutine suspends its execution, and the call to lua_resume that started this coroutine returns. The parameter nresults is the number of values from the stack that are passed as results to lua_resume.

2 3.8 - The Debug Interface

Lua has no built-in debugging facilities. Instead, it offers a special interface by means of functions and *hooks*. This interface allows the construction of different

kinds of debuggers, profilers, and other tools that need "inside information" from the interpreter.

2.1 lua_Debug

```
typedef struct lua_Debug {
  int event;
  const char *name;           /* (n) */
  const char *namewhat;       /* (n) */
  const char *what;           /* (S) */
  const char *source;         /* (S) */
  int currentline;            /* (l) */
  int nups;                   /* (u) number of upvalues */
  int linedefined;            /* (S) */
  int lastlinedefined;        /* (S) */
  char short_src[LUA_IDSIZE]; /* (S) */
  /* private part */
  {\em other fields}
} lua_Debug;
```

A structure used to carry different pieces of information about an active function. lua_getstack fills only the private part of this structure, for later use. To fill the other fields of lua_Debug with useful information, call lua_getinfo.
The fields of lua_Debug have the following meaning:

- source: If the function was defined in a string, then source is that string. If the function was defined in a file, then source starts with a '@' followed by the file name.
- short_src: a "printable" version of source, to be used in error messages.
- linedefined: the line number where the definition of the function starts.
- lastlinedefined: the line number where the definition of the function ends.
- what: the string "Lua" if the function is a Lua function, "C" if it is a C function, "main" if it is the main part of a chunk, and "tail" if it was a function that did a tail call. In the latter case, Lua has no other information about the function.
- currentline: the current line where the given function is executing. When no line information is available, currentline is set to -1.

- `name:` a reasonable name for the given function. Because functions in Lua are first-class values, they do not have a fixed name: some functions can be the value of multiple global variables, while others can be stored only in a table field. The `lua_getinfo` function checks how the function was called to find a suitable name. If it cannot find a name, then `name` is set to `NULL`.

- `namewhat:` explains the `name` field. The value of `namewhat` can be `"global"`, `"local"`, `"method"`, `"field"`, `"upvalue"`, or `""` (the empty string), according to how the function was called. (Lua uses the empty string when no other option seems to apply.)

- `nups:` the number of upvalues of the function.

2.2 lua_gethook

```
[-0, +0, -]
```

```
lua_Hook lua_gethook (lua_State *L);
```

Returns the current hook function.

2.3 lua_gethookcount

```
[-0, +0, -]
```

```
int lua_gethookcount (lua_State *L);
```

Returns the current hook count.

2.4 lua_gethookmask

```
[-0, +0, -]
```

```
int lua_gethookmask (lua_State *L);
```

Returns the current hook mask.

2.5 lua_getinfo

```
[-(0|1), +(0|1|2), m]
```

```
int lua_getinfo (lua_State *L, const char *what, lua_Debug *ar);
```

Returns information about a specific function or function invocation.
To get information about a function invocation, the parameter `ar` must be a valid activation record that was filled by a previous call to `lua_getstack` or given as argument to a hook (see `lua_Hook`).
To get information about a function you push it onto the stack and start the `what` string with the character '>'. (In that case, `lua_getinfo` pops the function in the top of the stack.) For instance, to know in which line a function `f` was defined, you can write the following code:

```
lua_Debug ar;
     lua_getfield(L, LUA_GLOBALSINDEX, "f");  /* get global 'f' */
     lua_getinfo(L, ">S", \&ar);
     printf("%d\n", ar.linedefined);
```

Each character in the string `what` selects some fields of the structure `ar` to be filled or a value to be pushed on the stack:

- '`n`': fills in the field `name` and `namewhat`;
- '`S`': fills in the fields `source`, `short_src`, `linedefined`, `lastlinedefined`, and `what`;
- '`l`': fills in the field `currentline`;
- '`u`': fills in the field `nups`;
- '`f`': pushes onto the stack the function that is running at the given level;
- '`L`': pushes onto the stack a table whose indices are the numbers of the lines that are valid on the function. (A *valid line* is a line with some associated code, that is, a line where you can put a break point. Non-valid lines include empty lines and comments.)

This function returns 0 on error (for instance, an invalid option in `what`).

2.6 lua_getlocal

```
[-0, +(0|1), -]
```

```
const char *lua_getlocal (lua_State *L, lua_Debug *ar, int n);
```

Gets information about a local variable of a given activation record. The parameter `ar` must be a valid activation record that was filled by a previous call to `lua_getstack` or given as argument to a hook (see `lua_Hook`). The index `n` selects which local variable to inspect (1 is the first parameter or active local variable, and so on, until the last active local variable). `lua_getlocal` pushes the variable's value onto the stack and returns its name.
Variable names starting with '`(`' (open parentheses) represent internal variables (loop control variables, temporaries, and C function locals).
Returns `NULL` (and pushes nothing) when the index is greater than the number of active local variables.

2.7 lua_getstack

`[-0, +0, -]`

```
int lua_getstack (lua_State *L, int level, lua_Debug *ar);
```

Get information about the interpreter runtime stack.
This function fills parts of a `lua_Debug` structure with an identification of the *activation record* of the function executing at a given level. Level 0 is the current running function, whereas level *n+1* is the function that has called level *n*. When there are no errors, `lua_getstack` returns 1; when called with a level greater than the stack depth, it returns 0.

2.8 lua_getupvalue

`[-0, +(0|1), -]`

```
const char *lua_getupvalue (lua_State *L, int funcindex, int n);
```

Gets information about a closure's upvalue. (For Lua functions, upvalues are the external local variables that the function uses, and that are consequently included in its closure.) `lua_getupvalue` gets the index `n` of an upvalue, pushes the upvalue's value onto the stack, and returns its name. `funcindex` points to the closure in the stack. (Upvalues have no particular order, as they are active through the whole function. So, they are numbered in an arbitrary order.)

Returns NULL (and pushes nothing) when the index is greater than the number of upvalues. For C functions, this function uses the empty string "" as a name for all upvalues.

2.9 lua_Hook

```
typedef void (*lua_Hook) (lua_State *L, lua_Debug *ar);
```

Type for debugging hook functions.
Whenever a hook is called, its ar argument has its field event set to the specific event that triggered the hook. Lua identifies these events with the following constants: LUA_HOOKCALL , LUA_HOOKRET , LUA_HOOKTAILRET , LUA_HOOKLINE , and LUA_HOOKCOUNT . Moreover, for line events, the field currentline is also set. To get the value of any other field in ar, the hook must call lua_getinfo. For return events, event can be LUA_HOOKRET, the normal value, or LUA_HOOKTAILRET. In the latter case, Lua is simulating a return from a function that did a tail call; in this case, it is useless to call lua_getinfo.
While Lua is running a hook, it disables other calls to hooks. Therefore, if a hook calls back Lua to execute a function or a chunk, this execution occurs without any calls to hooks.

2.10 lua_sethook

```
[-0, +0, -]
```

```
int lua_sethook (lua_State *L, lua_Hook f, int mask, int count);
```

Sets the debugging hook function.
Argument f is the hook function. mask specifies on which events the hook will be called: it is formed by a bitwise or of the constants LUA_MASKCALL , LUA_MASKRET , LUA_MASKLINE , and LUA_MASKCOUNT . The count argument is only meaningful when the mask includes LUA_MASKCOUNT. For each event, the hook is called as explained below:

- **The call hook:** is called when the interpreter calls a function. The hook is called just after Lua enters the new function, before the function gets its arguments.
- **The return hook:** is called when the interpreter returns from a function. The hook is called just before Lua leaves the function. You have no access to the values to be returned by the function.

- **The line hook:** is called when the interpreter is about to start the execution of a new line of code, or when it jumps back in the code (even to the same line). (This event only happens while Lua is executing a Lua function.)

- **The count hook:** is called after the interpreter executes every `count` instructions. (This event only happens while Lua is executing a Lua function.)

A hook is disabled by setting `mask` to zero.

2.11 lua_setlocal

`[-(0|1), +0, -]`

```
const char *lua_setlocal (lua_State *L, lua_Debug *ar, int n);
```

Sets the value of a local variable of a given activation record. Parameters `ar` and `n` are as in `lua_getlocal` (see `lua_getlocal`). `lua_setlocal` assigns the value at the top of the stack to the variable and returns its name. It also pops the value from the stack.
Returns `NULL` (and pops nothing) when the index is greater than the number of active local variables.

2.12 lua_setupvalue

`[-(0|1), +0, -]`

```
const char *lua_setupvalue (lua_State *L, int funcindex, int n);
```

Sets the value of a closure's upvalue. It assigns the value at the top of the stack to the upvalue and returns its name. It also pops the value from the stack. Parameters `funcindex` and `n` are as in the `lua_getupvalue` (see `lua_getupvalue`).
Returns `NULL` (and pops nothing) when the index is greater than the number of upvalues.