The C Application Programming Interface

Abbreviations used in this document

To save space, the following abbreviations are used in function declarations; this table summarizes all abbreviations, including those defined later:

#define **L**	**lua_State *L**	Pointer to a Lua state (environment) to operate upon.
#define **LS**	**lua_State**	A Lua state.
#define **LN**	**lua_Number**	Number in Lua native format, whose actual type is defined by LUA_NUMBER at compile time (default: **double**).
#define **CF**	**lua_CFunction**	(Pointer to) a C function callable from Lua; see *C functions* below.
#define **LD**	**lua_Debug**	Structure containing debug information; see *debugging*.
#define **LB**	**luaL_Buffer**	Structure used by string buffer functions in auxiliary library; see *String buffers* in *auxiliary library*.
#define **CC**	**const char**	C type used for immutable characters or strings.
#define **SZ**	**size_t**	C type used for byte sizes (e.g. block lenghts).
#define **VL**	**va_list**	C type used to receive a variable number of arguments.

Required headers

extern "C" { ... }	required around #includes if Lua is compiled as C and linked to C++.
#include "lua.h"	required for the Lua core, link with **liblua** library.
#include "lualib.h"	required for the standard Lua libraries, link with **liblualib** library.
#include "lauxlib.h"	required for the auxiliary Lua library, link with **liblualib** library.

Initialization, termination, version information

LS ***lua_open** (void);	Creates and returns a Lua state; multiple states can coexist.
int **luaopen_base** (L);	Opens and initializes the basic library; returns 0.
int **luaopen_table** (L);	Opens and initializes the table library; returns 1 and pushes the “table” table on the Lua stack.
int **luaopen_math** (L);	Opens and initializes the math library; returns 1 and pushes the “math” table on the Lua stack.
int **luaopen_string** (L);	Opens and initializes the string library; returns 1 and pushes the “string” table on the Lua stack.
int **luaopen_io** (L);	Opens and initializes I/O and operating system libraries; returns 1 and pushes the “io” table on the Lua stack.
int **luaopen_debug** (L);	Opens and initializes the debug library; returns 1 and pushes the “debug” table on the Lua stack.
int **luaopen_loadlib** (L);	Opens and initializes the loadlib library. i.e. Lua **loadlib ()** function for dynamic loading (included in the *Basic library*); returns 0.
void **lua_openlibs** (L);	Opens all the above standard libraries.
void **lua_close** (L);	Closes the Lua state **L**, calls **__gc** metamethods (finalizers) for userdata (if any), releases all resources.

C functions

typedef int (*lua_CFunction) (L);	(pointer to) C function to be called by Lua.
#define **CF** **lua_CFunction**	Abbreviation used in this document.

C API: the Lua stack

Stack terms used in this document

size	The available stack space (maximum number of possible entries).
top	The number of elements currently in the stack.
stack[i]	Abbreviation for "the value found in the stack at position (index) **i**".
valid indexes	Stack indexes are valid if (1 <= abs(**i**) <= **top**): 1..**top** = absolute stack position (push order); -1..-**top** = offset from **top** + 1 (pop order); special pseudo-indexes (see *Pseudo-indexes* below); examples: [1] = first element; [-1] = **top** = last pushed element.
acceptable indexes	The valid indexes above plus (**top** < **i** <= **size**), containing no value. "Invalid indexes" must still be acceptable: Lua does no checking, unless **api_check ()** is enabled by removing the comments in the relevant line of **lapi.c**.
to push	To add an element on top of stack, increasing **top** by 1.
to pop	To remove an element from top of stack, decreasing **top** by 1.

Basic stack operations and information

LUA_MINSTACK	Initial stack size when Lua calls a C function; the user is responsible for avoiding stack overflow.
int **lua_checkstack** (L, int n);	Tries to grow stack size to top + **n** entries (cannot shrink it); returns 0 if not possible.
int **lua_gettop** (L);	Returns current top (0 = no elements in stack).
void **lua_settop** (L, int i);	Sets top to **i**; removes elements if new top is smaller than previous top, adds **nil** elements if larger.
void **lua_pushvalue** (L, int i);	Pushes a copy of the element at stack[**i**].
void **lua_insert** (L, int i);	Moves stack[top] to stack[**i**], shifting elements as needed.
void **lua_replace** (L, int i);	Moves stack[top] to stack[**i**], overwriting it (no shifting).
void **lua_remove** (L, int i);	Removes element from stack[**i**], shifting elements as needed.
void **lua_pop** (L, int n);	Pops and discards **n** elements.
void **lua_xmove** (LS *a, LS *b, int n);	Pops **n** values from the stack of Lua state (or thread) **a**, pushes them on the stack of Lua state (or thread) **b**.

Pseudo-indexes

LUA_REGISTRYINDEX	Pseudo-index to access the registry table.
LUA_GLOBALSINDEX	Pseudo-index to access the global environment table.
int **lua_upvalueindex** (int n);	Returns a pseudo-index to access upvalue number **n** (from 1, in order of creation).

Type constants (also used for stack elements)

LUA_TNONE	No value: invalid (but acceptable) index.
LUA_TNIL	**nil**.
LUA_TBOOLEAN	Lua boolean (**true** or **false**).
LUA_TNUMBER	Lua number, actual type depends on LUA_NUMBER.
LUA_TSTRING	Lua string, may include embedded zeros.
LUA_TTABLE	Lua table.
LUA_TFUNCTION	Lua function or C function callable from Lua.
LUA_TUSERDATA	Full Lua userdata.
LUA_TLIGHTUSERDATA	Light Lua userdata (e.g. C pointer).
LUA_TTHREAD	Lua thread.

Checking stack elements

int **lua_type** (L, int i);	Returns the type of the value at stack[**i**], see *Type constants* above (LUA_TNONE if no value at **i**).
CC ***lua_typename** (L, int t);	Converts **t** returned by **lua_type ()** to a readable string.
int **lua_isnone** (L, int i);	Returns 1 if stack[**i**] has no value (LUA_TNONE), else 0.
int **lua_isnil** (L, int i);	Returns 1 if stack[**i**] is **nil**, else 0.
int **lua_isnoneornil** (L, int i);	Returns 1 if stack[**i**] has no value or is **nil**, else 0.
int **lua_isboolean** (L, int i);	Returns 1 if stack[**i**] is a boolean (**true** or **false**), else 0.
int **lua_isnumber** (L, int i);	Returns 1 if stack[**i**] is a number or a string representing a valid number (use **lua_type ()** to discriminate), else 0.
int **lua_isstring** (L, int i);	Returns 1 if stack[**i**] is a string or a number (use **lua_type ()** to discriminate), else 0.
int **lua_istable** (L, int i);	Returns 1 if stack[**i**] is a table, else 0.
int **lua_isfunction** (L, int i);	Returns 1 if stack[**i**] is a Lua function or a C function (use **lua_iscfunction ()** to discriminate), else 0.
int **lua_iscfunction** (L, int i);	Returns 1 if stack[**i**] is a C function, else 0.
int **lua_isuserdata** (L, int i);	Returns 1 if stack[**i**] is a full or a light userdata (use **lua_islightuserdata ()** to discriminate), else 0.
int **lua_islightuserdata** (L, int i)	Returns 1 if stack[**i**] is a light userdata, else 0.

See also: *Generic stack checking* in *auxiliary library*.

Reading values from stack elements

int **lua_toboolean** (L, int i);	Returns 0 if stack[**i**] is **false** or **nil** (also if **i** is invalid), 1 otherwise.
LN **lua_tonumber** (L, int i);	Returns stack[**i**] (number or string representing a valid number) as a number, 0 if invalid value or invalid **i**.
CC ***lua_tostring** (L, int i);	Returns stack[**i**] (string or number) as a zero-terminated string (may also contain embedded zeros), NULL if invalid value or invalid **i**; see note below. If element **i** is a number, it is changed to a string; this may confuse table traversal if done on keys.
SZ **lua_strlen** (L, int i);	Returns the actual length of string at stack[**i**], including embedded zeros (if any), 0 if invalid value or invalid **i**.
CF **lua_tocfunction** (L, int i);	Returns (a pointer to) a C function at stack[**i**], NULL if invalid value or invalid **i**.
void ***lua_touserdata** (L, int i);	Returns a pointer to the data block of full userdata at stack[**i**], the pointer itself for light userdata, NULL if invalid value or invalid **i**. See pointers note below.
LS ***lua_tothread** (L, int i);	Returns (a pointer to) a Lua thread (a Lua state) at stack[**i**], NULL if invalid value or invalid **i**. See pointers note below.
void ***lua_topointer** (L, int i);	Returns a pointer to a table, function, userdata or thread at stack[**i**], NULL if invalid value or invalid **i**. Mainly used for debugging. See pointers note below.

Pointers note: Returned C pointers are valid while stack[**i**] remains in the stack; after that they could become invalid due to garbage collection.

See also: *Reading and checking values from stack elements* in *auxiliary library*.

Pushing elements on top of stack

void **lua_pushnil** (L);	Pushes a Lua **nil** value.
void **lua_pushboolean** (L, int b);	Pushes **b** as Lua boolean (0 becomes **false**, all other values become **true**).
void **lua_pushnumber** (L, LN n);	Pushes **n** as Lua number.
void **lua_pushstring** (L, CC *s);	Pushes a copy of zero-terminated string **s** as Lua string.
void **lua_pushliteral** (L, CC *s);	As **lua_pushstring ()** but **s** must be a literal string; slightly faster as it doesn't call **strlen ()**.
void **lua_pushlstring** (L, CC *s, SZ n);	Pushes a copy of **n** bytes of data block **s** as generic Lua string (may contain embedded zeros).
CC ***lua_pushfstring** (L, CC *fs, ...);	Pushes a Lua string built by replacing formatting directives in the string **fs** with the following args; behaves like **sprintf ()** but with no flags, width or precision and only allowing: "%s" = a zero-terminated string, "%f" = a lua_Number, "%d" = an integer, "%c" = a character passed as int, "%%" = a '%' symbol; takes care of allocation and deallocation; returns a pointer to the resulting string. See pointers note below.
CC ***lua_pushvfstring** (L, CC *fs, VL ap);	Same as **lua_pushfstring ()** above but receives a variable list of arguments as **vsprintf ()** does.
void **lua_pushcfunction** (L, CF cf);	Pushes a C function **cf** callable from Lua.
void **lua_pushcclosure**(L, CF cf, int n);	Pops **n** values and pushes a C function **cf** callable from Lua, with those values as upvalues.
void ***lua_newuserdata** (L, SZ n);	Allocates and pushes a **n**-byte memory block as full userdata (at garbage collection, a **__gc** metamethod will be called before deallocation); returns a pointer to the new data block. See pointers note below.
void **lua_pushlightuserdata** (L, void *p);	Pushes **p** as light userdata.

Pointers note: Returned C pointers are valid while stack[**i**] remains in the stack; after that they could become invalid due to garbage collection.

Comparing stack elements

int **lua_equal** (L, int i, int j);	Returns true (!= 0) only if stack[**i**] == stack[**j**] in Lua (possibly calling **__eq** metamethod) and indexes are valid.
int **lua_rawequal** (L, int i, int j);	Same as **lua_equal ()** above but does not call metamethod.
int **lua_lessthan** (L, int i, int j);	Returns true (!= 0) only if stack[**i**] < stack[**j**] in Lua (possibly calling **__lt** metamethod) and indexes are valid.

C API: tables, metatables, registry, environment

Tables and metatables

void **lua_newtable** (L);	Creates and pushes a new, empty table.
void **lua_settable** (L, int i);	Pops a key and a value, stores key-value into table at stack[**i**]; calls **__newindex** metamethod, if any, in case of new field assignment (the table stays at stack[**i**]).
void **lua_gettable** (L, int i);	Pops a key, reads and pushes its value from table at stack[**i**]; calls **__index** metamethod, if any, for non-existing field; pushes the read value, or **nil** (the table stays at stack[**i**]).
void **lua_rawset** (L, int i);	As **lua_settable ()** above, but does not call metamethod.
void **lua_rawget** (L, int i);	As **lua_gettable ()** above, but does not call metamethod.
void **lua_rawseti** (L, int i, int n);	Pops a value, stores it into numeric element **n** of table at stack[**i**] (the table stays at stack[**i**]).
void **lua_rawgeti** (L, int i, int n);	Reads a value from numeric element **n** of table at stack[**i**]; pushes the read value (the table stays at stack[**i**]).
int **lua_setmetatable** (L, int i);	Pops a table, sets it as metatable for object at stack[**i**]; returns 0 if stack[**i**] is not table or userdata, or **i** is invalid.
int **lua_getmetatable** (L, int i);	Reads metatable from object at stack[**i**]; pushes the metatable (if no error); returns 0 if stack[**i**] has no metatable or **i** is invalid.

See also: *Tables and metatables* in *auxiliary library*.

Useful operations on tables

void **lua_concat** (L, int n);	Pops **n** values, efficiently concatenates them into a single value (empty string if **n** is 0); numbers are converted to strings using Lua rules, for other types the **__concat** metamethod is called; pushes the resulting value.
int **lua_next** (L, int i);	Does an iteration step on table at stack[**i**]: pops a key (**nil** = start traversal), pushes the next key and its value (note: do not use **lua_tostring ()** on the key); returns 0 and pushes nothing if there are no more keys.

Registry table

LUA_REGISTRYINDEX	Pseudo-index to access the registry table.
void **lua_register** (L, CC *fn, CF cf);	Registers C function **cf** with Lua name **fn**.

See also: *Registry references* and *Library initialization* in *auxiliary library*.

Environment tables

LUA_GLOBALSINDEX	Pseudo-index to access the global environment table.
int **lua_setfenv** (L, int i);	Pops a table, sets it as environment table for Lua function at stack[**i**]; returns 0 if stack[**i**] is not a Lua function.
void **lua_getfenv** (L, int i);	Pushes the environment table of Lua function at stack[**i**], or the global environment if stack[**i**] is a C function.

C API: loading, saving, executing

Loading and saving chunks

typedef CC * (***lua_Chunkreader**) (L, void *d, SZ *n);	User-supplied reader function to read a block of **n** bytes into a local buffer; any needed state (e.g. a FILE*) can be passed using **d**; returns a pointer to a local buffer containing the data block, or NULL in case of error; also sets **n** to the number of bytes actually read.
typedef int (***lua_Chunkwriter**) (L, const void *p, SZ n, void *d);	User-supplied writer function to write a block of **n** bytes starting from address **p**; any needed state (e.g. a FILE*) can be passed using **d**; the returned value is currently unused (Lua 5.0.2).
int **lua_load** (L, lua_Chunkreader r, void *d, CC *s);	Loads and compiles (does not execute) a text or precompiled Lua chunk using user-supplied reader function **r** (**r** will also receive the user data argument **d**), uses **s** as name for the loaded chunk, pushes the compiled chunk as a function; returns 0 if OK, LUA_ERRSYNTAX if syntax error, LUA_ERRMEM if allocation error.
int **lua_dump** (L, lua_Chunkwriter w, void *d);	Saves (writes) the function from stack[top] as a binary precompiled chunk using user-supplied writer function **w** (**w** will also receive the user data argument **d**); cannot save functions with closures; returns 1 if OK, 0 if no valid function to save.

See also: *Chunk loading* in *auxiliary library* for simpler chunk loading from files and strings.

Executing chunks

void **lua_call** (L, int na, int nr);	Calls a (Lua or C) function; the function and **na** arguments must be pushed in direct order and will be removed from the stack; if **nr** is LUA_MULTRET all results will be pushed in direct order, else exactly **nr** results will be pushed; any error will be propagated to the caller.
int **lua_pcall** (L, int na, int nr, int i);	As **lua_call ()** but catches errors; in case of error, if **i** is 0 pushes an error message string, else calls the error function at stack[**i**], passing it the error message, then pushes the value it returns; returns 0 if OK, LUA_ERRRUN if runtime error, LUA_ERRMEM if allocation error (error function is not called), LUA_ERRERR if error while running the error handler function.
int **lua_cpcall** (L, CF cf, void *ud);	Pushes a light userdata containing **ud** and calls C function **cf**; in case of error pushes **ud**, else leaves the stack unchanged; returns 0 if OK, or error code as **lua_pcall ()** above.

C API: threads, error handling, garbage collection

Threads

LS ***lua_newthread** (L);	Creates and pushes a new thread with a private stack; returns a pointer to a new Lua state.
int **lua_resume** (L, int na);	Starts or resumes a coroutine passing **na** pushed arguments; when returning, the stack will contain function return results, or **lua_yield ()** pushed return values, or an error message; returns 0 if OK, or error code as **lua_pcall ()** above.
int **lua_yield** (L, int nr);	Suspends coroutine execution passing **nr** return values to **lua_resume ()**; does not return to the calling C function; can only be called as C **return** expression;

Note: see **lua_xmove** in *Basic stack operations* for moving data between threads.

Error handling

int **lua_error** (L);	Raises an error, using error message from top of stack; does not return.
CF **lua_atpanic** (L, CF cf);	Registers C function **cf** to be called in case of unhandled error; the Lua state will be inconsistent when **cf** is called; if **cf** returns, calls **os.exit** (EXIT_FAILURE).

See also: *Error reporting* in *auxiliary library*.

Garbage collection

int **lua_gc** (L, int what, int data); Controls garbage collector.

- **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.
- **LUA_GCSETPAUSE:** sets data/100 as the new value for the *pause* of the collector. The function returns the previous value of the pause.
- **LUA_GCSETSTEPMUL:** sets data/100 as the new value for the *step multiplier* of the collector. The function returns the previous value of the step multiplier.

C API: debugging, hooks

Hooks

typedef void (***lua_Hook**) (L, LD *ar);	Function to be called by a hook (see above for **LD**).
int **lua_sethook** (L, lua_Hook hf, int m, int n);	Sets function **hf** as hook for the events given in mask **m**, a combination of one or more or-ed bitmasks: LUA_MASKCALL = function call, LUA_MASKRET = function return, LUA_MASKLINE = new code line, LUA_MASKCOUNT = every **n** instructions; removes the hook function if **m** is 0; returns 1.
lua_Hook **lua_gethook** (L);	Returns (a pointer to) the current hook function.
int **lua_gethookmask** (L);	Returns the current hook mask.
int **lua_gethookcount** (L);	Returns the current hook instruction count.

Debugging structure (activation record)

```
typedef struct lua_Debug {          /* Structure used by debugging functions */
  int event;
  CC *name;                         /* function name, or NULL if cannot get a name. */
  CC *namewhat;                     /* type of name: "global", "local", "method", "field", "" */
  CC *what;                         /* function type: "main", "Lua", "C" of "tail" (tail call) */
  CC *source;                       /* source as a string, or @filename */
  int currentline;                  /* line number, or -1 if not available */
  int nups;                         /* number of upvalues, 0 if none */
  int linedefined;                  /* line number where the function definition starts */
  char short_src[LUA_IDSIZE];       /* short, printable version of source */
  /* private part follows*/
} lua_Debug;
```

Debugging

#define **LD** **lua_Debug**	Abbreviation used in this document.
int **lua_getstack** (L, int n, LD *ar);	Makes **ar** refer to the function at calling level **n** [0 = current, 1 = caller]; returns 1 if OK, 0 if no such level.
int **lua_getinfo** (L, CC *w, LD *ar);	Fills fields of **ar** with information, according to one or more characters contained in the string **w**: 'n': fills **name** and **namewhat**. 'f': pushes the function referenced by **ar**. 'S': fills **what**, **source**, **short_src** and **linedefined**. 'l': fills **currentline**. 'u': fills **nups**. Requires a previous call to **lua_getstack ()** to refer **ar** to the desired function; returns 0 if error.
CC ***lua_getlocal** (L, const LD *ar, int n);	Pushes the value of **n**th local variable (from 1, in order of appearance); requires a previous call to **lua_getstack ()** to refer **ar** to the desired function; returns the name of the variable, or NULL if error.
CC ***lua_setlocal** (L, const LD *ar, int n);	Assigns value at stack[top] to the **n**th local variable (from 1, in order of appearance); requires a previous call to **lua_getstack ()** to refer **ar** to the desired function; returns the name of the variable, or NULL if error.
CC ***lua_getupvalue** (L, int i, int n);	Pushes the **n**th upvalue (from 1, in order of appearance) of thefunction at stack[**i**]; returns the name of the upvalue (empty string for C functions) or NULL if error.
CC ***lua_setupvalue** (L, int i, int n);	Pops and assign value to the **n**th upvalue (from 1, in order of appearance) of the function at stack[**i**]; returns the name of the upvalue (empty string for C functions) or NULL if error.

C API: auxiliary library

Generic stack checking

void **luaL_argcheck** (L, int c, int i, CC *m);	Raises a "bad argument" detailed error for stack[**i**] with message **m** if condition **c** is != 0.
void **luaL_checktype** (L, int i, t);	Raises a "bad argument" detailed error if stack[**i**] is not of type **t**, where **t** is a type constant (e.g. LUA_TTABLE).
void **luaL_checkany** (L, int i);	Raises a "value expected" error if there is no value (LUA_TNONE.) at stack[**i**].
void **luaL_checkstack** (L, int n, CC *m);	Tries to grow stack size to top + **n** entries (cannot shrink it), raises a "stack overflow" error including message **m** if growing is not possible.

Reading & checking values from stack elements

LN **luaL_checknumber** (L, int i);	Returns number (or string representing a valid number) from stack[**i**] if possible, else raises a "bad argument" error.
LN **LuaL_optnumber** (L, int i, LN d);	Returns default number **d** if stack[**i**] is **nil** or has no value (LUA_TNONE), else returns result from **luaL_checknumber (L**, **i)**.
int **luaL_checkint** (L, int i);	As **luaL_checknumber ()** but returns an **int**.
long **luaL_checklong** (L, int i);	As **luaL_checknumber ()** but returns a **long**.
int **luaL_optint** (L, int i, LN d);	As **luaL_checkoptnumber ()** but returns an **int**.
long **luaL_optlong** (L, int i, LN d);	As **luaL_checkoptnumber ()** but returns a **long**.
CC ***luaL_checklstring** (L, int i, SZ *n);	Returns string (or number) from stack[**i**] as a zero-terminated string (may also contain embedded zeros) if possible, else raises a "bad argument" error; also returns string length in ***n**, unless **n** is NULL. Note: if stack[**i**] is a number, it is changed to a string (this may confuse table traversal if done on keys).
CC ***LuaL_optlstring** (L, int i, CC *ds, SZ *n);	Returns default string **ds** if stack[**i**] is **nil** or has no value (LUA_TNONE), else returns result from **luaL_checklstring (L**, **i**, **n)**.
CC ***luaL_checkstring** (L, int i);	As **luaL_checklstring (L**, **i**, NULL**)**, used for normal C strings with no embedded zeros.
CC ***luaL_optstring** (L, int i, CC *ds);	As **luaL_optlstring (L**, **i**, **ds**, NULL**)**, used for normal C strings with no embedded zeros.

Note: the above functions are useful to get arguments in C functions called from Lua.

Tables and metatables

int **luaL_getn** (L, int i);	Returns the size of the table at stack[**i**]; works as **table.getn ()** in the Lua table library.
int **luaL_setn** (L, int i, int n);	Sets the size of the table at stack[**i**] to **n**; works as **table.setn ()** in the Lua table library.
int **luaL_newmetatable** (L, CC *tn);	Creates a new table (to be used as metatable), pushes it and creates a bidirectional registry association between that table and the name **tn**; returns 0 if **s** is already used.
void **luaL_getmetatable** (L, CC *tn);	Gets the metatable named **tn** from the registry and pushes it, or **nil** if none.
int **luaL_getmetafield** (L, int i, CC *fn);	Pushes field named **fn** (e.g. **__add**) of the metatable of the object at stack[**i**], if any; returns 1 if found and pushed, else 0.
int **luaL_callmeta** (L, int i, CC *fn);	Calls function in field named **fn** (e.g. **__tostring**) of the metatable of the object at stack[**i**], if any, passing the object itself and expecting one result; returns 1 if found and called, else 0.
void ***luaL_checkudata** (L, int i, CC *mn);	Checks if stack[**i**] is an userdata having a metatable named **mn**; returns its address, or NULL if the check fails.

Registry references

int **luaL_ref** (L, int i);	Pops a value and stores it into the table at stack[**i**] using a new, unique integer key as reference; typically used with **i** = LUA_REGISTRYINDEX to store a Lua value into the registry and make it accessible from C; returns the new integer key, or the unique value LUA_REFNIL if stack[**i**] is **nil**, or 0 if not done.
void **luaL_unref** (L, int i, int r);	Removes from the table at stack[**i**] the value stored into it by **luaL_ref ()** having reference **r**.
LUA_NOREF	Value representing "no reference", useful to mark references as invalid.

Library initialization

typedef struct luaL_reg {
  CC *name;
  CF cf;
} **luaL_reg**
	Structure used to declare an entry in a list of C functions to be registered by **luaL_openlib ()** below; **cf** is the function and **name** will be its Lua name.

int **luaL_openlib** (L, CC ln, const luaL_reg *fl, int n);	Creates (or reuses) a table named **ln** and fills it with the name-function pairs detailed in the **fl** list, terminated by a {NULL, NULL} pair; also pops **n** upvalues from the stack and sets them as common upvalues for all the functions in the table; typically used to create a Lua interface to a C library.

Chunk loading

int **luaL_loadfile**(L, CC *fn);	Loads and precompiles into a Lua chunk (does not execute) the contents of the file named **fn**; returns 0 if OK, LUA_ERRSYNTAX if syntax error, LUA_ERRMEM if allocation error, LUA_ERRFILE if error while reading **fn**.
int **luaL_loadbuffer** (L, CC *b, SZ n, CC *cn);	Loads and precompiles into a Lua chunk (does not execute) the contents of memory buffer (string) **b** for a length of **n** bytes, assigns **cn** as internal name for the loaded chunk; returns 0 if OK, LUA_ERRSYNTAX if syntax error, LUA_ERRMEM if allocation error.

Error reporting

int **luaL_error** (L, CC *fs, ...);	Builds a Lua string by replacing formatting directives in the string **fs** with the following args, as **lua_pushfstring ()** does (see *Pushing elements on top of stack*), pushes the resulting message and calls **lua_error ()**; does not return.
int **luaL_argerror** (L, int i, CC *m);	Unconditionally raises a "bad argument" detailed error for stack[**i**], including message **m**; also works from within methods having a **self** argument; does not return.
int **luaL_typerror** (L, int i, CC *tn);	Unconditionally raises a "bad argument" detailed error for stack[**i**], including expected type name **tn** and actual type name; does not return.
void **luaL_where** (L, int n);	Pushes a string with the current source line and number at level **n** [0 = current, 1 = caller].

String buffers

#define **LB** **luaL_Buffer**	Abbreviation used in this document.
void **luaL_buffinit** (L, LB *b);	Initializes the buffer **b**.
void **luaL_putchar** (int b, int c);	Adds character **c** to the buffer **b**.
void **luaL_addlstring** (LB *b, CC *s, int n);	Adds a copy of memory block (generic string) **s** of length **n** to the buffer **b**.
void **luaL_addstring** (LB *b, CC *s);	Adds a copy of zero-terminated string **s** to the buffer **b**.
void **luaL_addvalue** (LB *b);	Pops a value (string or number) and adds it to the buffer **b**; does not violate the balanced stack usage requirement when using buffers.
void **luaL_pushresult** (LB *b);	Pushes the contents of buffer **b** as a single string, empties the buffer.
char ***luaL_prepbuffer** (LB *b);	Returns the address of a memory block where up to LUAL_BUFFERSIZE bytes can be written (the user is responsible for avoiding overflow); **luaL_addsize ()** should be called afterwards to add those bytes to the buffer **b**.
void **luaL_addsize** (LB *b, int n);	Adds **n** bytes (n <= LUAL_BUFFERSIZE) to the buffer **b**; the bytes should have previously been written into memory at the address returned by **luaL_prepbuffer ()**; no other buffer functions should be called between **luaL_prepbuffer ()** and **luaL_addsize ()**.

Notes: string buffering uses the stack as temporary space and has no size limit; the (system-dependent) constant LUAL_BUFFERSIZE is only used for direct manipulation via **luaL_prepbuffer ()** and **luaL_addsize ()**;
stack usage must be balanced between calls to buffering functions, with the exception of **luaL_addvalue ()**.