1 4 - The Auxiliary Library

The *auxiliary library* provides several convenient functions to interface C with Lua. While the basic API provides the primitive functions for all interactions between C and Lua, the auxiliary library provides higher-level functions for some common tasks.
All functions from the auxiliary library are defined in header file `lauxlib.h` and have a prefix `luaL_`.
All functions in the auxiliary library are built on top of the basic API, and so they provide nothing that cannot be done with this API.
Several functions in the auxiliary library are used to check C function arguments. Their names are always `luaL_check*` or `luaL_opt*`. All of these functions throw an error if the check is not satisfied. Because the error message is formatted for arguments (e.g., "`bad argument #1`"), you should not use these functions for other stack values.

1.1 4.1 - Functions and Types

Here we list all functions and types from the auxiliary library in alphabetical order.

1.1.1 `luaL_addchar`

`[-0, +0, m]`

```
void luaL_addchar (luaL_Buffer *B, char c);
```

Adds the character `c` to the buffer `B` (see `luaL_Buffer`).

1.1.2 `luaL_addlstring`

`[-0, +0, m]`

```
void luaL_addlstring (luaL_Buffer *B, const char *s, size_t l);
```

Adds the string pointed to by `s` with length `l` to the buffer `B` (see `luaL_Buffer`). The string may contain embedded zeros.

1.1.3 `luaL_addsize`

`[-0, +0, m]`

```
void luaL_addsize (luaL_Buffer *B, size_t n);
```

Adds to the buffer `B` (see `luaL_Buffer`) a string of length `n` previously copied to the buffer area (see `luaL_prepbuffer`).

1.1.4 `luaL_addstring`

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

```
void luaL_addstring (luaL_Buffer *B, const char *s);
```

Adds the zero-terminated string pointed to by `s` to the buffer `B` (see `luaL_Buffer`). The string may not contain embedded zeros.

1.1.5 `luaL_addvalue`

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

```
void luaL_addvalue (luaL_Buffer *B);
```

Adds the value at the top of the stack to the buffer `B` (see `luaL_Buffer`). Pops the value.
This is the only function on string buffers that can (and must) be called with an extra element on the stack, which is the value to be added to the buffer.

1.1.6 `luaL_argcheck`

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

```
void luaL_argcheck (lua_State *L,
                    int cond,
                    int narg,
                    const char *extramsg);
```

Checks whether `cond` is true. If not, raises an error with the following message, where `func` is retrieved from the call stack:

```
bad argument #<narg> to <func> (<extramsg>)
```

1.1.7 `luaL_argerror`

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

```
int luaL_argerror (lua_State *L, int narg, const char *extramsg);
```

Raises an error with the following message, where `func` is retrieved from the call stack:

```
bad argument #<narg> to <func> (<extramsg>)
```

This function never returns, but it is an idiom to use it in C functions as `return luaL_argerror(args)`.

1.1.8 `luaL_Buffer`

```
typedef struct luaL_Buffer luaL_Buffer;
```

Type for a *string buffer*.
A string buffer allows C code to build Lua strings piecemeal. Its pattern of use is as follows:

- First you declare a variable `b` of type `luaL_Buffer`.
- Then you initialize it with a call `luaL_buffinit(L, \&b)`.
- Then you add string pieces to the buffer calling any of the `luaL_add*` functions.
- You finish by calling `luaL_pushresult(\&b)`. This call leaves the final string on the top of the stack.

During its normal operation, a string buffer uses a variable number of stack slots. So, while using a buffer, you cannot assume that you know where the top of the stack is. You can use the stack between successive calls to buffer operations as long as that use is balanced; that is, when you call a buffer operation, the stack is at the same level it was immediately after the previous buffer operation. (The only exception to this rule is `luaL_addvalue`.) After calling `luaL_pushresult` the stack is back to its level when the buffer was initialized, plus the final string on its top.

1.1.9 `luaL_buffinit`

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

```
void luaL_buffinit (lua_State *L, luaL_Buffer *B);
```

Initializes a buffer `B`. This function does not allocate any space; the buffer must be declared as a variable (see `luaL_Buffer`).

1.1.10 `luaL_callmeta`

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

```
int luaL_callmeta (lua_State *L, int obj, const char *e);
```

Calls a metamethod.
If the object at index `obj` has a metatable and this metatable has a field `e`, this function calls this field and passes the object as its only argument. In this case this function returns 1 and pushes onto the stack the value returned by the call. If there is no metatable or no metamethod, this function returns 0 (without pushing any value on the stack).

1.1.11 `luaL_checkany`

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

```
void luaL_checkany (lua_State *L, int narg);
```

Checks whether the function has an argument of any type (including **nil**) at position `narg`.

1.1.12 `luaL_checkint`

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

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

Checks whether the function argument `narg` is a number and returns this number cast to an `int`.

1.1.13 `luaL_checkinteger`

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

```
lua_Integer luaL_checkinteger (lua_State *L, int narg);
```

Checks whether the function argument `narg` is a number and returns this number cast to a `lua_Integer`.

1.1.14 `luaL_checklong`

`[-0, +0, v]`

```
long luaL_checklong (lua_State *L, int narg);
```

Checks whether the function argument `narg` is a number and returns this number cast to a `long`.

1.1.15 `luaL_checklstring`

`[-0, +0, v]`

```
const char *luaL_checklstring (lua_State *L, int narg, size_t *l);
```

Checks whether the function argument `narg` is a string and returns this string; if `l` is not `NULL` fills `*l` with the string's length.
This function uses `lua_tolstring` to get its result, so all conversions and caveats of that function apply here.

1.1.16 `luaL_checknumber`

`[-0, +0, v]`

```
lua_Number luaL_checknumber (lua_State *L, int narg);
```

Checks whether the function argument `narg` is a number and returns this number.

1.1.17 `luaL_checkoption`

`[-0, +0, v]`

```
int luaL_checkoption (lua_State *L,
                      int narg,
                      const char *def,
                      const char *const lst[]);
```

Checks whether the function argument `narg` is a string and searches for this string in the array `lst` (which must be NULL-terminated). Returns the index in the array where the string was found. Raises an error if the argument is not a string or if the string cannot be found.
If `def` is not `NULL`, the function uses `def` as a default value when there is no argument `narg` or if this argument is **nil**.
This is a useful function for mapping strings to C enums. (The usual convention in Lua libraries is to use strings instead of numbers to select options.)

1.1.18 `luaL_checkstack`

`[-0, +0, v]`

```
void luaL_checkstack (lua_State *L, int sz, const char *msg);
```

Grows the stack size to `top + sz` elements, raising an error if the stack cannot grow to that size. `msg` is an additional text to go into the error message.

1.1.19 `luaL_checkstring`

`[-0, +0, v]`

```
const char *luaL_checkstring (lua_State *L, int narg);
```

Checks whether the function argument `narg` is a string and returns this string.
This function uses `lua_tolstring` to get its result, so all conversions and caveats of that function apply here.

1.1.20 `luaL_checktype`

`[-0, +0, v]`

```
void luaL_checktype (lua_State *L, int narg, int t);
```

Checks whether the function argument `narg` has type `t`. See `lua_type` for the encoding of types for `t`.

1.1.21 `luaL_checkudata`

`[-0, +0, v]`

```
void *luaL_checkudata (lua_State *L, int narg, const char *tname);
```

Checks whether the function argument `narg` is a userdata of the type `tname` (see `luaL_newmetatable`).

1.1.22 `luaL_dofile`

`[-0, +?, m]`

```
int luaL_dofile (lua_State *L, const char *filename);
```

Loads and runs the given file. It is defined as the following macro:

```
(luaL_loadfile(L, filename) || lua_pcall(L, 0, LUA_MULTRET, 0))
```

It returns 0 if there are no errors or 1 in case of errors.

1.1.23 `luaL_dostring`

`[-0, +?, m]`

```
int luaL_dostring (lua_State *L, const char *str);
```

Loads and runs the given string. It is defined as the following macro:

```
(luaL_loadstring(L, str) || lua_pcall(L, 0, LUA_MULTRET, 0))
```

It returns 0 if there are no errors or 1 in case of errors.

1.1.24 `luaL_error`

`[-0, +0, v]`

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

Raises an error. The error message format is given by `fmt` plus any extra arguments, following the same rules of `lua_pushfstring`. It also adds at the beginning of the message the file name and the line number where the error occurred, if this information is available.

This function never returns, but it is an idiom to use it in C functions as `return luaL_error(args)`.

1.1.25 `luaL_getmetafield`

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

```
int luaL_getmetafield (lua_State *L, int obj, const char *e);
```

Pushes onto the stack the field `e` from the metatable of the object at index `obj`. If the object does not have a metatable, or if the metatable does not have this field, returns 0 and pushes nothing.

1.1.26 `luaL_getmetatable`

`[-0, +1, -]`

```
void luaL_getmetatable (lua_State *L, const char *tname);
```

Pushes onto the stack the metatable associated with name `tname` in the registry (see `luaL_newmetatable`).

1.1.27 `luaL_gsub`

`[-0, +1, m]`

```
const char *luaL_gsub (lua_State *L,
                       const char *s,
                       const char *p,
                       const char *r);
```

Creates a copy of string `s` by replacing any occurrence of the string `p` with the string `r`. Pushes the resulting string on the stack and returns it.

1.1.28 `luaL_loadbuffer`

`[-0, +1, m]`

```
int luaL_loadbuffer (lua_State *L,
                     const char *buff,
                     size_t sz,
                     const char *name);
```

Loads a buffer as a Lua chunk. This function uses `lua_load` to load the chunk in the buffer pointed to by `buff` with size `sz`.
This function returns the same results as `lua_load`. `name` is the chunk name, used for debug information and error messages.

1.1.29 `luaL_loadfile`

[-0, +1, m]

```
int luaL_loadfile (lua_State *L, const char *filename);
```

Loads a file as a Lua chunk. This function uses `lua_load` to load the chunk in the file named `filename`. If `filename` is `NULL`, then it loads from the standard input. The first line in the file is ignored if it starts with a `#`.
This function returns the same results as `lua_load`, but it has an extra error code `LUA_ERRFILE` if it cannot open/read the file.
As `lua_load`, this function only loads the chunk; it does not run it.

1.1.30 `luaL_loadstring`

[-0, +1, m]

```
int luaL_loadstring (lua_State *L, const char *s);
```

Loads a string as a Lua chunk. This function uses `lua_load` to load the chunk in the zero-terminated string `s`.
This function returns the same results as `lua_load`.
Also as `lua_load`, this function only loads the chunk; it does not run it.

1.1.31 `luaL_newmetatable`

[-0, +1, m]

```
int luaL_newmetatable (lua_State *L, const char *tname);
```

If the registry already has the key `tname`, returns 0. Otherwise, creates a new table to be used as a metatable for userdata, adds it to the registry with key `tname`, and returns 1.
In both cases pushes onto the stack the final value associated with `tname` in the registry.

1.1.32 `luaL_newstate`

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

```
lua_State *luaL_newstate (void);
```

Creates a new Lua state. It calls `lua_newstate` with an allocator based on the standard C `realloc` function and then sets a panic function (see `lua_atpanic`) that prints an error message to the standard error output in case of fatal errors.
Returns the new state, or `NULL` if there is a memory allocation error.

1.1.33 `luaL_openlibs`

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

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

Opens all standard Lua libraries into the given state.

1.1.34 `luaL_optint`

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

```
int luaL_optint (lua_State *L, int narg, int d);
```

If the function argument `narg` is a number, returns this number cast to an `int`. If this argument is absent or is **nil**, returns `d`. Otherwise, raises an error.

1.1.35 `luaL_optinteger`

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

```
lua_Integer luaL_optinteger (lua_State *L,
                             int narg,
                             lua_Integer d);
```

If the function argument `narg` is a number, returns this number cast to a `lua_Integer`. If this argument is absent or is **nil**, returns `d`. Otherwise, raises an error.

1.1.36 `luaL_optlong`

`[-0, +0, v]`

```
long luaL_optlong (lua_State *L, int narg, long d);
```

If the function argument `narg` is a number, returns this number cast to a `long`. If this argument is absent or is **nil**, returns `d`. Otherwise, raises an error.

1.1.37 `luaL_optlstring`

`[-0, +0, v]`

```
const char *luaL_optlstring (lua_State *L,
                             int narg,
                             const char *d,
                             size_t *l);
```

If the function argument `narg` is a string, returns this string. If this argument is absent or is **nil**, returns `d`. Otherwise, raises an error.
If `l` is not `NULL`, fills the position `*l` with the results's length.

1.1.38 `luaL_optnumber`

`[-0, +0, v]`

```
lua_Number luaL_optnumber (lua_State *L, int narg, lua_Number d);
```

If the function argument `narg` is a number, returns this number. If this argument is absent or is **nil**, returns `d`. Otherwise, raises an error.

1.1.39 `luaL_optstring`

`[-0, +0, v]`

```
const char *luaL_optstring (lua_State *L,
                            int narg,
                            const char *d);
```

If the function argument `narg` is a string, returns this string. If this argument is absent or is **nil**, returns `d`. Otherwise, raises an error.

1.1.40 `luaL_prepbuffer`

`[-0, +0, -]`

```
char *luaL_prepbuffer (luaL_Buffer *B);
```

Returns an address to a space of size `LUAL_BUFFERSIZE` where you can copy a string to be added to buffer `B` (see `luaL_Buffer`). After copying the string into this space you must call `luaL_addsize` with the size of the string to actually add it to the buffer.

1.1.41 `luaL_pushresult`

`[-?, +1, m]`

```
void luaL_pushresult (luaL_Buffer *B);
```

Finishes the use of buffer `B` leaving the final string on the top of the stack.

1.1.42 `luaL_ref`

`[-1, +0, m]`

```
int luaL_ref (lua_State *L, int t);
```

Creates and returns a *reference*, in the table at index `t`, for the object at the top of the stack (and pops the object).
A reference is a unique integer key. As long as you do not manually add integer keys into table `t`, `luaL_ref` ensures the uniqueness of the key it returns. You can retrieve an object referred by reference `r` by calling `lua_rawgeti(L, t, r)`. Function `luaL_unref` frees a reference and its associated object.

If the object at the top of the stack is **nil**, `luaL_ref` returns the constant `LUA_REFNIL` . The constant `LUA_NOREF` is guaranteed to be different from any reference returned by `luaL_ref`.

1.1.43 `luaL_Reg`

```
typedef struct luaL_Reg {
  const char *name;
  lua_CFunction func;
} luaL_Reg;
```

Type for arrays of functions to be registered by `luaL_register`. `name` is the function name and `func` is a pointer to the function. Any array of `luaL_Reg` must end with an sentinel entry in which both `name` and `func` are `NULL`.

1.1.44 `luaL_register`

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

```
void luaL_register (lua_State *L,
                    const char *libname,
                    const luaL_Reg *l);
```

Opens a library.
When called with `libname` equal to `NULL`, it simply registers all functions in the list `l` (see `luaL_Reg`) into the table on the top of the stack.
When called with a non-null `libname`, `luaL_register` creates a new table `t`, sets it as the value of the global variable `libname`, sets it as the value of `package.loaded[libname]`, and registers on it all functions in the list `l`. If there is a table in `package.loaded[libname]` or in variable `libname`, reuses this table instead of creating a new one.
In any case the function leaves the table on the top of the stack.

1.1.45 `luaL_typename`

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

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

Returns the name of the type of the value at the given index.

1.1.46 `luaL_typerror`

`[-0, +0, v]`

```
int luaL_typerror (lua_State *L, int narg, const char *tname);
```

Generates an error with a message like the following:

```
location: bad argument narg to 'func' (tname expected, got rt)
```

where `location` is produced by `luaL_where`, `func` is the name of the current function, and `rt` is the type name of the actual argument.

1.1.47 `luaL_unref`

`[-0, +0, -]`

```
void luaL_unref (lua_State *L, int t, int ref);
```

Releases reference `ref` from the table at index `t` (see `luaL_ref`). The entry is removed from the table, so that the referred object can be collected. The reference `ref` is also freed to be used again.
If `ref` is `LUA_NOREF` or `LUA_REFNIL`, `luaL_unref` does nothing.

1.1.48 `luaL_where`

`[-0, +1, m]`

```
void luaL_where (lua_State *L, int lvl);
```

Pushes onto the stack a string identifying the current position of the control at level `lvl` in the call stack. Typically this string has the following format:

```
chunkname:currentline:
```

Level 0 is the running function, level 1 is the function that called the running function, etc.
This function is used to build a prefix for error messages.